In this chapter we will first study the super-important algorithm—the Fast Fourier Transform—based on the divide and conquer design technique, and then we will look at a fairly simple example of divide and conquer that we will use in the next chapter.

A Brief Look at the Fast Fourier Transform

In this chapter we will look at what Levitin says may be the most important algorithm ever, for its practical applications, namely the fast Fourier transform (FFT).

The FFT has many practical applications, but we will focus on just one, namely *polynomial interpolation* over the complex numbers. To help learn the main ideas, we will also study a toy version of FFT—interpolation over Z_p (where p is a very carefully chosen prime number).

The Polynomial Interpolation Problem

The *polynomial interpolation problem* is the problem of taking some given data points and finding a polynomial that passes through them.

> You have spent a fair amount of time in your life doing this for the very special case of taking two data points and finding the line—a polynomial of degree 1—that passes through them.

More precisely, for any $n > 0$, let (x_j, y_j) for $0 \leq j < n$ be n given data points, and let

$$P(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}$$

be the desired polynomial. Then our goal is to find c_0, ..., c_{n-1} such that for $0 \leq j < n$,

$$P(x_j) = y_j.$$

The Method of Undetermined Coefficients (MUC)

The obvious way to solve the interpolation problem is to simply write down the given facts and express them in the form of a matrix equation.

For each j, we want $P(x_j) = y_j$, which means we want

$$c_0 + c_1 x_j + c_2 x_j^2 + \cdots c_{n-1} x_j^{n-1} = y_j,$$

which is a linear equation with unknowns c_0, ..., c_{n-1}.

We can express these linear equations in matrix form:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ & & \vdots & & \\ 1 & x_j & x_j^2 & \cdots & x_j^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_j \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_j \\ \vdots \\ y_{n-1} \end{bmatrix}$$

or

$$Ac = y,$$

where we call the big square matrix A (known as a *Vandermonde matrix*), the vector of unknown coefficients c, and the vector of y_j values y.

Note that all the values in the coefficient matrix A are known, since all the x_j are given numbers, and similarly all the values in the right-hand side y are known, since all the y_j are given numbers. So, we can solve this, say by some form of Gaussian elimination, for the unknown c_j values—the coefficients of the desired polynomial.

Example of Polynomial Interpolation by MUC

Suppose we are given data points $(1, 3)$, $(2, 2)$, $(3, 7)$, $(4, 24)$, and we want to find the cubic polynomial that passes through them. From the previous work, we know that we want to solve the linear system:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ 1 & 4 & 16 & 64 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 7 \\ 24 \end{bmatrix}$$

Using the algorithms that you should have learned in MTH 2140 or the equivalent, we form the augmented matrix, do row operations to zero out the augmented coefficient matrix below the diagonal, and then do back substitution to get the solution to the system of equations:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 3 \\ 1 & 2 & 4 & 8 & 2 \\ 1 & 3 & 9 & 27 & 7 \\ 1 & 4 & 16 & 64 & 24 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 3 \\ 0 & 1 & 3 & 7 & -1 \\ 0 & 2 & 8 & 26 & 4 \\ 0 & 3 & 15 & 63 & 21 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 3 \\ 0 & 1 & 3 & 7 & -1 \\ 0 & 0 & 2 & 12 & 6 \\ 0 & 0 & 6 & 42 & 24 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 3 \\ 0 & 1 & 3 & 7 & -1 \\ 0 & 0 & 2 & 12 & 6 \\ 0 & 0 & 0 & 6 & 6 \end{bmatrix} \begin{array}{c} c_0 = 3 - c_1 - c_2 - c_3 = 3 - 1 + 3 - 1 = 4 \\ c_1 = -1 - 3c_2 - 7c_3 = -1 + 9 - 7 = 1 \\ 2c_2 = 6 - 12c_3 = 6 - 12 = -6 \rightarrow c2 = -3 \\ 6c_3 = 6 \rightarrow c_3 = 1 \end{array}$$

where the solution is obtained by first figuring out c_3, then c_2, and so on.

Thus, we have found the desired unknown coefficients for the interpolating polynomial, which turns out to be

$$P(x) = 4 + x - 3x^2 + x^3.$$

It is easy to check that this polynomial does hit the given data points.

> In this example, we solved the system of linear equations $Ac = y$ by forming the augmented matrix (adding y as the last column) and doing row operations, but note that this is just a somewhat more efficient approach than computing the inverse of the coefficient matrix and multiplying it by the right-hand side vector, that is, to compute
>
> $$c = A^{-1}y.$$

In the usual case, where all the x_j, y_j, and c_j are ordinary real numbers, MUC turns out to be a fairly terrible algorithm for interpolation, because it is very inefficient compared to other approaches, and, worse, is very subject to rounding error. If you take a numerical analysis class such as MTH 4480/4490, you might learn several superior algorithms for polynomial interpolation, but that's not our interest here.

Polynomial Interpolation in Some Special Cases

We are interested in two special cases where instead of real numbers, we use "numbers" in a field other than the real numbers. Specifically, we will use Z_p, for p prime, as our toy example to play with, and then move on to the field of *complex numbers* which is what is used in actual practical applications of the Fast Fourier Transform.

The really special thing we will have, in both these cases (which isn't true for the real numbers), is that we will have a "number" α such that $\alpha^n = 1$. In all of the following algebraic and algorithmic work to develop the basic ideas of the FFT, we won't even need to say which field we are using, as long as it satisfies this special property!

In our special situation, we will only interpolate data points that have

$$x_j = \alpha^j,$$

for $0 \leq j < n$.

> This is kind of like taking equally-spaced x_j along the number line in the real number case, so it's not that weird. In fact, in the actual application, using complex numbers, doing this makes the x_j equally-spaced in frequency space, which is great for taking a bunch of analog data obtained by sampling an analog signal at various frequencies and determining the coefficients of the interpolating polynomial, so the coefficients can be transmitted and used by the receiver to reproduce an approximation to the original signal.

Typically we also pick α such that none of these x_j values for $j > 0$ are equal to 1. Such a value α is known as a *primitive nth root of unity.*

MUC in the Special Situation

Earlier we might have noticed that row k of the coefficient matrix for MUC consists of the successive powers of x_k, so the element in row j, column k, is

$$x_j^k.$$

In the special situation, with our clever choice of $x_j = \alpha^j$, the coefficient matrix, which we will name F_n, has a very nice structure.

The element in row j, column k, is

$$(F_n)_{j,k} = x_j^k = (\alpha^j)^k = \alpha^{jk}.$$

> We will use $n = 8$ to derive the big ideas of the FFT. We will assume that we have some value α such that $\alpha^8 = 1$ and worry later about how to arrange that.
>
> We won't bother to derive the FFT for the general case, relying on our example for insight. It should be pretty obvious that everything we are doing would work for any n that is a power of 2, for any α such that $\alpha^n = 1$.

First, let's write out F_8 and get ready to notice the crucial algebraic facts that allow us to "divide and conquer."

$$F_8 = \begin{bmatrix} \alpha^{0\cdot0} & \alpha^{0\cdot1} & \alpha^{0\cdot2} & \alpha^{0\cdot3} & \alpha^{0\cdot4} & \alpha^{0\cdot5} & \alpha^{0\cdot6} & \alpha^{0\cdot7} \\ \alpha^{1\cdot0} & \alpha^{1\cdot1} & \alpha^{1\cdot2} & \alpha^{1\cdot3} & \alpha^{1\cdot4} & \alpha^{1\cdot5} & \alpha^{1\cdot6} & \alpha^{1\cdot7} \\ \alpha^{2\cdot0} & \alpha^{2\cdot1} & \alpha^{2\cdot2} & \alpha^{2\cdot3} & \alpha^{2\cdot4} & \alpha^{2\cdot5} & \alpha^{2\cdot6} & \alpha^{2\cdot7} \\ \alpha^{3\cdot0} & \alpha^{3\cdot1} & \alpha^{3\cdot2} & \alpha^{3\cdot3} & \alpha^{3\cdot4} & \alpha^{3\cdot5} & \alpha^{3\cdot6} & \alpha^{3\cdot7} \\ \alpha^{4\cdot0} & \alpha^{4\cdot1} & \alpha^{4\cdot2} & \alpha^{4\cdot3} & \alpha^{4\cdot4} & \alpha^{4\cdot5} & \alpha^{4\cdot6} & \alpha^{4\cdot7} \\ \alpha^{5\cdot0} & \alpha^{5\cdot1} & \alpha^{5\cdot2} & \alpha^{5\cdot3} & \alpha^{5\cdot4} & \alpha^{5\cdot5} & \alpha^{5\cdot6} & \alpha^{5\cdot7} \\ \alpha^{6\cdot0} & \alpha^{6\cdot1} & \alpha^{6\cdot2} & \alpha^{6\cdot3} & \alpha^{6\cdot4} & \alpha^{6\cdot5} & \alpha^{6\cdot6} & \alpha^{6\cdot7} \\ \alpha^{7\cdot0} & \alpha^{7\cdot1} & \alpha^{7\cdot2} & \alpha^{7\cdot3} & \alpha^{7\cdot4} & \alpha^{7\cdot5} & \alpha^{7\cdot6} & \alpha^{7\cdot7} \end{bmatrix}$$

$$= \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 & \alpha^4 & \alpha^5 & \alpha^6 & \alpha^7 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^8 & \alpha^{10} & \alpha^{12} & \alpha^{14} \\ \alpha^0 & \alpha^3 & \alpha^6 & \alpha^9 & \alpha^{12} & \alpha^{15} & \alpha^{18} & \alpha^{21} \\ \alpha^0 & \alpha^4 & \alpha^8 & \alpha^{12} & \alpha^{16} & \alpha^{20} & \alpha^{24} & \alpha^{28} \\ \alpha^0 & \alpha^5 & \alpha^{10} & \alpha^{15} & \alpha^{20} & \alpha^{25} & \alpha^{30} & \alpha^{35} \\ \alpha^0 & \alpha^6 & \alpha^{12} & \alpha^{18} & \alpha^{24} & \alpha^{30} & \alpha^{36} & \alpha^{42} \\ \alpha^0 & \alpha^7 & \alpha^{14} & \alpha^{21} & \alpha^{28} & \alpha^{35} & \alpha^{42} & \alpha^{49} \end{bmatrix}$$

Now we do the crucial, super clever thing that makes divide-and-conquer work—we rearrange the columns of F_8 to put the elements with even j first, then the odd ones, obtaining

this system of linear equations that is equivalent to the original one:

$$\begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^1 & \alpha^3 & \alpha^5 & \alpha^7 \\ \alpha^0 & \alpha^4 & \alpha^8 & \alpha^{12} & \alpha^2 & \alpha^6 & \alpha^{10} & \alpha^{14} \\ \alpha^0 & \alpha^6 & \alpha^{12} & \alpha^{18} & \alpha^3 & \alpha^9 & \alpha^{15} & \alpha^{21} \\ \alpha^0 & \alpha^8 & \alpha^{16} & \alpha^{24} & \alpha^4 & \alpha^{12} & \alpha^{20} & \alpha^{28} \\ \alpha^0 & \alpha^{10} & \alpha^{20} & \alpha^{30} & \alpha^5 & \alpha^{15} & \alpha^{25} & \alpha^{35} \\ \alpha^0 & \alpha^{12} & \alpha^{24} & \alpha^{36} & \alpha^6 & \alpha^{18} & \alpha^{30} & \alpha^{42} \\ \alpha^0 & \alpha^{14} & \alpha^{28} & \alpha^{42} & \alpha^7 & \alpha^{21} & \alpha^{35} & \alpha^{49} \end{bmatrix} \begin{bmatrix} c_0 \\ c_2 \\ c_4 \\ c_6 \\ c_1 \\ c_3 \\ c_5 \\ c_7 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}.$$

Note that when we rearrange the columns of a linear system, we also have to rearrange the unknowns in the same way in order to be saying the same thing.

Now we make the first crucial observation: if we split each row into two halves, the elements in the second half are a fixed multiple of the corresponding elements in the first half. For example, the fifth row has $[\alpha^0\ \alpha^{10}\ \alpha^{20}\ \alpha^{30}]$ in the first half, and

$$[\alpha^5\ \alpha^{15}\ \alpha^{25}\ \alpha^{35}] = \alpha^5\ [\alpha^0\ \alpha^{10}\ \alpha^{20}\ \alpha^{30}]$$

in the second half.

The second crucial observation is obvious when we bring in the fact that $\alpha^8 = 1$. When we use this fact everywhere, we obtain the system:

$$\begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^1 & \alpha^3 & \alpha^5 & \alpha^7 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^2 & \alpha^6 & \alpha^2 & \alpha^6 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^3 & \alpha^1 & \alpha^7 & \alpha^5 \\ \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^4 & \alpha^4 & \alpha^4 & \alpha^4 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^5 & \alpha^7 & \alpha^1 & \alpha^3 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^6 & \alpha^2 & \alpha^6 & \alpha^2 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^7 & \alpha^5 & \alpha^3 & \alpha^1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_2 \\ c_4 \\ c_6 \\ c_1 \\ c_3 \\ c_5 \\ c_7 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}.$$

Note that the first observation still holds, because we haven't changed any elements—we've just written them differently—but it's harder to see now.

The second crucial observation is obvious when we partition the coefficient matrix into halves in rows and columns:

$$\left[\begin{array}{cccc|cccc} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^1 & \alpha^3 & \alpha^5 & \alpha^7 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^2 & \alpha^6 & \alpha^2 & \alpha^6 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^3 & \alpha^1 & \alpha^7 & \alpha^5 \\ \hline \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^4 & \alpha^4 & \alpha^4 & \alpha^4 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^5 & \alpha^7 & \alpha^1 & \alpha^3 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^6 & \alpha^2 & \alpha^6 & \alpha^2 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^7 & \alpha^5 & \alpha^3 & \alpha^1 \end{array}\right] \left[\begin{array}{c} c_0 \\ c_2 \\ c_4 \\ c_6 \\ \hline c_1 \\ c_3 \\ c_5 \\ c_7 \end{array}\right] = \left[\begin{array}{c} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \hline y_4 \\ y_5 \\ y_6 \\ y_7 \end{array}\right].$$

We see that the half-size coefficient matrix in the upper-left corner is identical to the one in the lower-left corner.

Using the first observation (which isn't so obvious now that we've reduced things using $\alpha^8 = 1$), we can see that the system is

$$\left[\begin{array}{cccc|cccc} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0\alpha^0 & \alpha^0\alpha^0 & \alpha^0\alpha^0 & \alpha^0\alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^1\alpha^0 & \alpha^1\alpha^2 & \alpha^1\alpha^4 & \alpha^1\alpha^6 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^2\alpha^0 & \alpha^2\alpha^4 & \alpha^2\alpha^0 & \alpha^2\alpha^4 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^3\alpha^0 & \alpha^3\alpha^6 & \alpha^3\alpha^4 & \alpha^3\alpha^2 \\ \hline \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^4\alpha^0 & \alpha^4\alpha^0 & \alpha^4\alpha^0 & \alpha^4\alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^5\alpha^0 & \alpha^5\alpha^2 & \alpha^5\alpha^4 & \alpha^5\alpha^6 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^6\alpha^0 & \alpha^6\alpha^4 & \alpha^6\alpha^0 & \alpha^6\alpha^4 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^7\alpha^0 & \alpha^7\alpha^6 & \alpha^7\alpha^4 & \alpha^7\alpha^2 \end{array}\right] \left[\begin{array}{c} c_0 \\ c_2 \\ c_4 \\ c_6 \\ \hline c_1 \\ c_3 \\ 5 \\ c_7 \end{array}\right] = \left[\begin{array}{c} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \hline y_4 \\ y_5 \\ y_6 \\ y_7 \end{array}\right]$$

We can see that there is just *one* half-size coefficient matrix in action here. So, if we let

$$F_4 = \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 \end{bmatrix},$$

and let $v = \begin{bmatrix} c_0 \\ c_2 \\ c_4 \\ c_6 \end{bmatrix}$ and $w = \begin{bmatrix} c_1 \\ c_3 \\ c_5 \\ c_7 \end{bmatrix}$, we see that the system can be written as

$$F_4 v + \begin{bmatrix} \alpha^0 & 0 & 0 & 0 \\ 0 & \alpha^1 & 0 & 0 \\ 0 & 0 & \alpha^2 & 0 \\ 0 & 0 & 0 & \alpha^3 \end{bmatrix} F_4 w = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix},$$

and

$$F_4 v + \begin{bmatrix} \alpha^4 & 0 & 0 & 0 \\ 0 & \alpha^5 & 0 & 0 \\ 0 & 0 & \alpha^6 & 0 \\ 0 & 0 & 0 & \alpha^7 \end{bmatrix} F_4 w = \begin{bmatrix} y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}.$$

The point of all this is that we can compute $F_8 c$ by doing two half-size matrix-vector multiplications, namely $F_4 v$ and $F_4 w$, and then do just 8 scalar multiplications to multiply the elements of $F_4 w$ by the suitable α^k value.

Note that the preceding derivation expressed the upper-right corner coefficient matrix as

$$\begin{bmatrix} \alpha^0 & 0 & 0 & 0 \\ 0 & \alpha^1 & 0 & 0 \\ 0 & 0 & \alpha^2 & 0 \\ 0 & 0 & 0 & \alpha^3 \end{bmatrix} F_4,$$

and the lower-right corner coefficient matrix as

$$\begin{bmatrix} \alpha^4 & 0 & 0 & 0 \\ 0 & \alpha^5 & 0 & 0 \\ 0 & 0 & \alpha^6 & 0 \\ 0 & 0 & 0 & \alpha^7 \end{bmatrix} F_4,$$

which makes it look like we need to multiply two $n/2$ by $n/2$ matrices together, at a cost of $\Theta(n^2)$, but it is crucial that we instead first compute $F_4 w$ just once, and then multiply that vector by the diagonal matrices, but just by multiplying the first component by α^0, the second by α^1, and so on, for a total cost of just $\Theta(n)$.

In terms of designing a divide-and-conquer algorithm for polynomial interpolation, we have made great progress, but a few issues remain.

Note that we have only studied the $n = 8$ case. However, it is pretty easy to see that the same things happen for any even n, so we won't waste time going through it.

The obvious big issue is that we have to be able to apply these ideas recursively. In general, the matrix $F_{\frac{n}{2}}$ turns out to be the correct matrix for α^2. In our $n = 8$ example, if we let $\gamma = \alpha^2$, we can easily check that

$$F_4 = \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 \end{bmatrix} = \begin{bmatrix} \gamma^0 & \gamma^0 & \gamma^0 & \gamma^0 \\ \gamma^0 & \gamma^1 & \gamma^2 & \gamma^3 \\ \gamma^0 & \gamma^2 & \gamma^0 & \gamma^2 \\ \gamma^0 & \gamma^3 & \gamma^2 & \gamma^1 \end{bmatrix}$$

where this last matrix is exactly what we would call F_4, using γ in place of α. Note that γ has the desired property for F_4, namely that

$$\gamma^4 = (\alpha^2)^4 = \alpha^8 = 1.$$

Finally, if we switch the even and odd columns on F_4, we obtain

$$\begin{bmatrix} \gamma^0 & \gamma^0 & \gamma^0 & \gamma^0 \\ \gamma^0 & \gamma^2 & \gamma^1 & \gamma^3 \\ \gamma^0 & \gamma^0 & \gamma^2 & \gamma^2 \\ \gamma^0 & \gamma^2 & \gamma^3 & \gamma^1 \end{bmatrix},$$

so when we divide this matrix, we obtain

$$F_2 = \begin{bmatrix} \gamma^0 & \gamma^0 \\ \gamma^0 & \gamma^2 \end{bmatrix} = \begin{bmatrix} \delta^0 & \delta^0 \\ \delta^0 & \delta^1 \end{bmatrix},$$

where $\delta = \gamma^2$, and

$$\delta^2 = (\gamma^2)^2 = \gamma^4 = 1$$

Example of FFT Multiplication

⇒ Let's go through these ideas again using $n = 6$, with "numbers" in Z_7, and $\alpha = 5$. Since n is not a power of 2, we will only be able to demonstrate the first step of the divide and conquer process.

For convenience as we form F_6, here are the powers of α:

k	0	1	2	3	4	5	6
9^k	1	5	4	6	2	3	1

Let's use $c = \begin{bmatrix} 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 5 \end{bmatrix}$. We'll form F_6, compute $F_6 c$ by the usual matrix-vector multiplication algorithm, switch columns and components of c, and then do the two half-size multiplications (using ordinary matrix-vector multiplication—this is where if n were a power of 2, we could apply the FFT idea recursively, but here we can't), and combine them to get the answer, to be checked against the usual algorithm.

Exercise 11 [10 points] (target due date: Monday, September 14)

Your job on this Exercise is to work through all the previous stuff with a concrete example. You should basically do exactly what I did in the video, but for $n = 8$, using numbers in Z_{17}, $\alpha = 9$, and

$$c = \begin{bmatrix} 1 \\ 14 \\ 13 \\ 11 \\ 6 \\ 10 \\ 16 \\ 7 \end{bmatrix}.$$

Plus, instead of computing $F_4 v$ and $F_4 w$ by the usual matrix-vector multiplication, you must apply the FFT ideas to do each of them separately using usual matrix-vector multiplication for F_2 times vectors of length 2. In other words, you are to start with $n = 8$ and recurse until you get to the $n = 2$ level, and then do those multiplications the usual way (rather than recursing one more level to do 1 by 1 matrix vector multiplication).

You can check your work by doing the original problem $F_8 c$ using the usual matrix-vector multiplication (or you can turn it in and risk having me say, nope, please fix!).

Your write-up of your work should follow how I did the example in the video (except you have to do the recursive steps, whereas I did the two $n = 3$ level matrix-vector multiplications the usual way).

In case it is helpful, here are some small multiples of 17, all of which are equal to 0 in Z_{17}:

$$0, 17, 34, 51, 68, 85, 102, 119, 136, 153, 170, 187, 204, 221, 234, 255, 272, 289$$

And, here is the full multiplication table for Z_{17}, so you can look up products instead of figuring them:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	0	2	4	6	8	10	12	14	16	1	3	5	7	9	11	13	15
3	0	3	6	9	12	15	1	4	7	10	13	16	2	5	8	11	14
4	0	4	8	12	16	3	7	11	15	2	6	10	14	1	5	9	13
5	0	5	10	15	3	8	13	1	6	11	16	4	9	14	2	7	12
6	0	6	12	1	7	13	2	8	14	3	9	15	4	10	16	5	11
7	0	7	14	4	11	1	8	15	5	12	2	9	16	6	13	3	10
8	0	8	16	7	15	6	14	5	13	4	12	3	11	2	10	1	9
9	0	9	1	10	2	11	3	12	4	13	5	14	6	15	7	16	8
10	0	10	3	13	6	16	9	2	12	5	15	8	1	11	4	14	7
11	0	11	5	16	10	4	15	9	3	14	8	2	13	7	1	12	6
12	0	12	7	2	14	9	4	16	11	6	1	13	8	3	15	10	5
13	0	13	9	5	1	14	10	6	2	15	11	7	3	16	12	8	4
14	0	14	11	8	5	2	16	13	10	7	4	1	15	12	9	6	3
15	0	15	13	11	9	7	5	3	1	16	14	12	10	8	6	4	2
16	0	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Finishing Up Our Brief Study of FFT

First, let's do efficiency analysis for FFT. Let $M(n)$ be the number of multiplications (we are going to ignore additions because they are not important in this analysis) needed to compute $F_n c$ for $n = 2^m$ and c any n vector. We have seen that the divide-and-conquer idea lets us compute this matrix-vector product using just two half-size matrix vector products—to form $F_{n/2}v$ and $F_{n/2}w$—and n multiplications to scale the components of $F_{n/2}w$. So, we have

$$M(n) = 2M(n/2) + n.$$

Applying our usual techniques, we have

$$\begin{aligned} M(n) = M(2^m) &= 2M(2^{m-1}) + 2^m \\ &= 2[2M(2^{m-2}) + 2^{m-1}] + 2^m \\ &= 2^2 M(2^{m-2}) + 2^m + 2^m \\ &\vdots \\ &= 2^m M(1) + m2^m \\ &= nM(1) + \log n \cdot n \\ &\in \Theta(n \log n). \end{aligned}$$

This is the exciting moment: the FFT technique can do matrix-vector multiplication in $\Theta(n \log n)$ operations, whereas the usual algorithm clearly takes $\Theta(n^2)$.

In practice, n can easily be in the thousands. For example, I just googled `audible sound frequencies` and found that "humans can detect sounds in a frequency range from about 20 Hz to 20 kHz" (i.e., from 20 to 20,000). If we have an analogue sound signal made up of physically recorded real-world sounds, we can imagine easily wanting, for a high-quality digital capture of the sound, wanting to sample at least some thousands of different frequencies (say 1024, or 2048, or 4096, since we want to restrict ourselves—mostly for convenience (there are versions of FFT that don't need n to be a power of 2)—to powers of 2). For n of this size, the time difference between n^2 and $n \log n$ is very significant.

One concern pops up, though. It takes n^2 multiplications just to form F_n, so doesn't that ruin our savings? It does not, for several reasons. First, even if we wanted to or had to actually form all the F_n matrices (for n, $n/2$, etc. downward), we would only have to form them once. In the previous discussion, if we decided we wanted to sample at say $n = 2048$ frequencies, we could do this $\Theta(n^2)$ work once, and then use these matrices, stored in memory, a zillion times. Note that what is really happening is that at equally-spaced time intervals your phone is sampling its analog signal from the microphone, performing the FFT to get the coefficients, transmitting those coefficients to the cell tower, where they are sent to another phone and used to reconstruct an approximation to the original signal. This is happening many, many times, all with the same matrices.

Perhaps more importantly (especially since storing a 2048 by 2048 matrix would be a fair amount of memory), if we were to think more deeply about the algorithm (or have to program it up!), we would realize that we don't ever need to form or use the matrices F_n. The product $F_8 c$, for example, is computed by rearranging the components of c and multiplying the even ones by F_4 and the odd ones by F_4, with the odd results being scaled and everything added up. Similarly, we didn't really need to form F_4, because its product with the correct 4-vectors can be computed using F_2 products, and so on.

Now for the last concern, which actually looks like it might be a real problem (but isn't).

We started this chapter talking about interpolation, which meant solving $F_n c = y$, with y known, to find the polynomial coefficients c, but for notational convenience we've been talking as if we knew c and needed to compute $F_n c$ to get y. Really we wanted to multiply y by F_n^{-1} to get c—the unknown coefficients of the interpolating polynomial. Fortunately, getting F_n^{-1} is easy.

Fourier Matrices Have Easy Inverses

If we generate F_n from some α, it turns out that if we generate following the same formulas a matrix G_n from $\beta = \alpha^{-1}$, then $F_n \cdot G_n$ is the identity matrix multiplied by n.

> So, F_n^{-1} is simply the Fourier matrix formed from β, multiplied by $1/n$. So, really everything we were doing with α should have been done with β, and we should have been using y, not c, to actually solve the original interpolation problem.
>
> This is very lucky, because computing the inverse of a matrix in general is in $\Theta(n^3)$, which would ruin everything.

To prove this for any n, assume that $\alpha^n = 1$, and let $\beta = \alpha^{-1}$. Then row j of F_n is

$$[\alpha^{j\cdot 0}\ \alpha^{j\cdot 1}\ \ldots \alpha^{jm}\ \ldots\ \alpha^{j(n-1)}],$$

and column k of G_n is

$$\begin{bmatrix} \beta^{0\cdot k} \\ \vdots \\ \beta^{mk} \\ \vdots \\ \beta^{(n-1)j} \end{bmatrix},$$

so the element of $F_n G_n$ in row j, column k is the usual matrix product of these two matrices, which is

$$\alpha^{j\cdot 0}\beta^{0\cdot k} + \alpha^{j\cdot 1}\beta^{1\cdot k} + \cdots + \alpha^{jm}\beta^{mk} + \cdots + \alpha^{j(n-1)}\beta^{(n-1)k}$$
$$= (\alpha^j\beta^k)^0 + (\alpha^j\beta^k)^1 + \cdots + (\alpha^j\beta^k)^m + \cdots + (\alpha^j\beta^k)^{n-1}.$$

Now, if we let $r = \alpha^j\beta^k$, we see that this element of the matrix is a geometric series, namely

$$1 + r + \cdots + r^{n-1}.$$

Now, for elements of the product array not on the diagonal, that is, $k \neq j$, assuming that α and β are *primitive* roots of unity (meaning no $k < n$ makes α^k or β^k equal to 1), it is easy to see that $r \neq 1$ (just pair up α's and β's and cancel them until we get either some left-over α's or some left-over β's, with fewer than n, so can't get to 1) so (by the formula for the sum of a geometric series) the element is

$$\frac{r^n - 1}{r - 1}$$

which is equal to 0 since

$$r^n = (\alpha^j \beta^k)^n = \alpha^{jn} \beta^{kn} = (\alpha^n)^j (\beta^n)^k = 1^j 1^k = 1.$$

For elements of the product array on the diagonal, $j = k$, so

$$r = \alpha^j \beta^j = \alpha^j (\alpha^{-1})^j = (\alpha \alpha^{-1})^j = (1)^j = 1,$$

so the elements add up to n.

Example of Inverse of Fourier Matrix

⇒ To verify concretely that the previous discussion is correct, let's take our earlier F_6 matrix obtained by using $\alpha = 5$, and form G_6 using $\beta = \alpha^{-1} = 3$, and then multiply the two together and see what we get.

The Fast Fourier Transform Using Complex Numbers

Note: the following material on complex numbers is just here in case you find it interesting. If you already know about complex numbers, it might be a good review. Or, if you have always wanted to know a little about complex numbers, it might be useful.

> In the previous work, we used the field Z_p, with p prime, to obtain a toy situation in which we could find α with the crucial property that $\alpha^n = 1$, while maintaining all our comfortable algebra facts. In real world applications of Fast Fourier multiplication, people use the field of complex numbers.

A Super-Brief Look at Complex Numbers

A complex number is any entity of the form $a+bi$, where a and b are ordinary real numbers and i is the square root of -1, in the sense that $i^2 = -1$.

To perform arithmetic on complex numbers, you just behave totally as usual, except i is mixed in there, with the $i^2 = -1$ fact used appropriately.

⇒ Compute $(2 + 3i) + (4 - 7i)$ and $(2 + 3i)(4 - 7i)$.

To divide complex numbers, we have to be a little clever:

$$\frac{a+bi}{c+di} = \frac{a+bi}{c+di} \cdot \frac{c-di}{c-di} = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i.$$

It turns out that complex numbers extend the real numbers and in some situations are essential. For example, it is true that any polynomial of degree n with complex coefficients has n complex roots—the analogous property is not true with the real numbers (there are, for example, quadratics that have no real roots).

If we picture $a + bi$ as being plotted at the point (a, b), then the real numbers lie on the x axis, purely imaginary numbers lie on the y axis, and the vast majority of complex numbers have both real and imaginary parts and fill in the whole plane.

The size of a complex number $a + bi$, denoted by $|a + bi|$, is simply the distance from the origin to the complex number in the ordinary sense, that is,

$$|a + bi| = \sqrt{a^2 + b^2}.$$

The Complex Exponential Function

Consider the quantity $e^{i\theta}$, where θ is a real number. To make sense of this, let's use the Taylor series for $e^{i\theta}$ (blithely assuming that it still makes sense for complex numbers, which it turns out it does). We obtain a famous result:

$$\mathrm{e}^{i\theta} = 1 + \frac{(i\theta)^1}{1!} + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \cdots$$

$$= 1 + \frac{\theta^1}{1!}i - \frac{\theta^2}{2!} - \frac{\theta^3}{3!}i + \frac{\theta^4}{4!} + \frac{\theta^5}{5!}i + \cdots$$

$$= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} + \cdots\right) + \left(\frac{\theta^1}{1!} - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} + \cdots\right) i$$

$$= \cos(\theta) + \sin(\theta)i.$$

One consequence of this is that a complex number of the form $e^{\theta i}$ is on the unit circle in the complex plane, at an angle of θ counterclockwise from the positive x axis.

Now we want to study how complex numbers of the form $e^{i\theta}$ behave when we multiply them by each other. We easily see, using algebra and some trig identities, that

$$\mathrm{e}^{i\alpha} \cdot \mathrm{e}^{i\beta} = (\cos\alpha + \sin\alpha i) \cdot (\cos\beta + \sin\beta i)$$

$$= (\cos\alpha\cos\beta - \sin\alpha\sin\beta) + (\cos\alpha\sin\beta + \sin\alpha\cos\beta)i$$

$$= \cos(\alpha+\beta) + \sin(\alpha+\beta)i.$$

Thus, two complex numbers that lie on the unit circle multiply together by simply adding their angles.

And, it is easy to see that $\mathrm{e}^{i\alpha} \cdot \mathrm{e}^{-i\alpha} = 1$.

We are finally ready for the main thing that we need to understand how complex numbers can be used for polynomial interpolation, namely that for any positive integer n, the complex number $\alpha = \mathrm{e}^{\frac{2\pi}{n}i}$, which is located on the unit circle one nth of the way around the circle, is known as an nth root of 1, because the complex numbers α, α^2, α^3, ..., α^{n-1}, $\alpha^n = 1$ are equally spaced around the unit circle at angles of $\frac{2\pi}{n}$, and when you raise any of these to the nth power, you get 1.

So, we have seen that for any n, we can use $\alpha = \mathrm{e}^{\frac{2\pi}{n}i}$ to form F_n, and $\beta = \mathrm{e}^{-\frac{2\pi}{n}i}$ to form G_n, and do the FFT stuff.

Better yet, we have just seen that complex numbers of the form $\mathrm{e}^{i\theta}$ actually involve trig functions, so they are good at producing functions that are combinations of various sine and cosine functions with different periods and amplitudes.

This is not the appropriate place to get into the actual uses of the Fourier transform problem, but it is in fact very useful in lots of different applications, such as signal processing and image compression. For these applications n is typically large, so the improvement from $\Theta(n^2)$ for the obvious algorithm to $\Theta(n \log n)$ for the FFT is crucial, making it feasible to perform the Fourier transform quickly enough to be practically useful.

For example, if you are talking on your phone, the analog function produced by your talking at some moment in time is sampled at a bunch of frequencies from the microphone, obtaining the y vector, which is then multiplied by $\frac{1}{n}G_n$ by the FFT technique, obtaining the coefficients c. These coefficients c are then digitized and sent over the wires/airwaves. On the other end, the y is recovered by doing $F_n c$ by the FFT, and then y can be used to

produce a sound through the speaker. The point is that the whole process needs to take place in real time, at frequent time intervals, without irritating the speaker and listener by either inadequate sampling or taking too long to compute everything.

Efficient Modular Exponentiation

We will finish our study of the divide and conquer algorithm design technique by studying a fairly simple problem known as *modular exponentiation*. This algorithm is crucial for RSA encryption, which we will study in the next chapter.

The Exponentiation Problem

Working in any reasonable world (standard real numbers, Z_n, complex numbers, etc.), it makes sense to ask how we can compute a^k, where a is some "number" and k is some positive integer, perhaps huge.

Here is the obvious and simple algorithm:

```
public int exp( int a, int k )
{
  int r = 1;
  for( int j=1; j<=k; j++ )
    r = r * a;
  return r;
}
```

This algorithm is hopelessly inefficient! Of course, to say an algorithm for a problem is inefficient, we have to come up with a better one.

Let's invent the key idea. Suppose we want to compute a^{133}. The previous algorithm would compute

$$a^1, a^2, a^3, a^4, a^5, \ldots, a^{133},$$

obtaining each a^k by $a^{k-} \cdot a$. Note that we get a^{133} after 132 multiplications.

Here is a much better idea—compute

$$a^1, a^2, a^4, a^8, a^{16}, a^{32}, a^{64}, a^{128}$$

by *squaring* each successive value to get the next value of a raised to a power of 2. Now, we adjust slightly, computing

$$a^{133} = a^{128+5} = a^{128} \cdot a^4 \cdot a^1$$

This approach only takes 7 multiplications to get a^{128}, and then two more to get a^{133}, for a total of 9, much better than 132.

Modular Exponentiation

Before going on, we will restrict ourselves to situations where a is in Z_n for some integer n, and we will perform all operations in Z_n. Computing a^k in Z_n is known as *modular exponentation*.

> Doing exponentiation over ordinary real numbers instead of Z_n isn't that much fun, because the numbers get huge quite quickly. Over Z_n, nothing we compute is ever outside of the range from 0 to $n-1$. And, of course, modular exponentiation is what we want for RSA encryption, it turns out.

Let's invent the algorithm from a divide and conquer perspective (even though what we did before is the big idea).

Suppose for simplicity that $k = 2^m$. Then to compute $a^k = a^{2^m}$ in Z_n, we can use the divide-and-conquer approach.

We realize that if we compute $a^{k/2} = a^{2^{m-1}}$ in Z_n, then we can simply square that value to compute the desired value, since $a^{k/2}a^{k/2} = a^k$. If we do this recursively all the way down, we have an algorithm that takes only m operations to compute the value, rather than 2^m.

In this case, it is easier to implement the recursion iteratively—we compute a^1, then square that to get a^2, square that to get a^4, and so on, until we reach $a^{2^m} = a^k$.

And, of course, at each step we reduce the result mod n so that the numbers never get larger than n^2.

It is obvious that this algorithm requires m steps, and since $m = \log_2(k) \leq \log_2(n)$, the algorithm takes $O(\log_2(n))$ steps. And, each step involves multiplying together two integers of size at most n and reducing the result mod n, which is manageable.

Example of Efficient Modular Exponentiation Algorithm by Hand

As a first example, let's compute a^e in Z_n where $a = 73$, $e = 29$, and $n = 151$. The main idea is that since $29 = 16 + 8 + 4 + 1$,

$$73^{29} = 73^{16} \cdot 73^8 \cdot 73^4 \cdot 73^1,$$

and we note that $a^{2m} = (a^m)^2$, so we can compute all the values of a raised to a power of 2 by repeated squaring, so in Z_{151} we compute

73^1			$= 73$
73^2		$= 5329$	$= 44$
73^4	$= 44^2$	$= 1936$	$= 124$
73^8	$= 124^2$	$= 15376$	$= 125$
73^{16}	$= 125^2$	$= 15625$	$= 72$

Now that we have all the values of a raised to the necessary powers of 2, we can multiply them together to obtain the answer, which is

$$73^{16} \cdot 73^8 \cdot 73^4 \cdot 73^1 = 72 \cdot 125 \cdot 124 \cdot 73 = 27.$$

The Final Slick Modular Exponentiation Algorithm

All we have to do now is remember how to convert a given decimal integer into binary and integrate that into our successive squares algorithm.

The idea of converting a given integer k into binary is to note that if k is even, its last (or least significant or rightmost) bit is 0, and if k is odd, that bit is 1. Then we divide k by 2 and use the same idea to get the rightmost bit of $k/2$, which is the 2's bit of k, and so on.

So, our final slick algorithm is demonstrated here for the same problem as was done previously. We can make a chart where the first column contains the successive squares of a, the second column holds the repeated halvings of e, using ordinary integer arithmetic, and the third column contains the cumulative products of the items in the first column where the second column contains an odd number.

Doing this gives us:

```
 73    29     73
 44    14
124     7    143
125     3     57
 72     1     27
```

Exercise 12 [10 points] (target due date: Monday, September 14)

Demonstrate how to perform the slick, efficient modular exponentiation algorithm using a calculator for $a = 29613$, $e = 8075$, working in Z_{31861}.

Remember that to compute, say, a^2 in Z_n, you first do

$$a^2 = 29613 \cdot 29613 = 876929769,$$

and then reduce it mod n by dividing by $n = 31861$, obtaining

$$876929676/31861 = 27523.610966385235868,$$

then subtract the integer part, obtaining

$$0.6109663852385868$$

and then multiply by 31861 to get the remainder:

$$0.6109663852385868 \cdot 31861 = 19466.$$

The folder `Code/RSA` at the course web site in my OneDrive contains a class `ModExp` that does this algorithm, but you should only use it to check your work—on the test you will be asked questions in such a way as to assess your understanding, not just whether you can generate the numbers this program generates.
