The rest of the course will be loosely organized around different algorithm design techniques, each accompanied by some examples of its use. Where appropriate, you will be asked to use the techniques to design new algorithms for given problems.

Actually we have already seen one technique: brute force.

Algorithms designed using our second technique—divide and conquer—end up being recursive and hence requiring efficiency analysis like in Project 7 (from now on we will refer to this analysis technique as "the substitution method"). The divide and conquer technique can lead to algorithms that are more efficient than other, usually simpler, algorithms.

> Most of the other design techniques to lead to algorithms whose efficiency is either easily analyzed, or extremely difficult to analyze.

The Divide and Conquer Algorithm Design Technique

We begin with the divide and conquer technique, which says that to solve a problem, we find a way to view the problem as several sub-problems, such that we can solve the original problem by using the solutions to the sub-problems.

These algorithms can, if properly designed, be more efficient than one might expect.

You have probably already seen this technique in action in CS 2050, say in the the merge sort and quick sort algorithms.

Now we will apply the divide and conquer design technique to a number of interesting problems. Some of these are of practical use, and some are just interesting as examples of divide-and-conquer.

Note: for this material, there is no pre-reading, because I want you to kind of invent the algorithms for yourselves, in your groups, without having seen anything except the basic "divide and conquer" idea.

⇒ Project 9 [routine] Karatsuba's Method for Multiplying Integers

For a new problem/algorithm to tackle by the divide and conquer technique, consider the problem of multiplying two integers symbolically, using ordinary "by hand" techniques.

Most people (I hope!) had to learn in elementary school an algorithm for multiplying two integers by hand. One version of this algorithm is demonstrated here—each row is produced by multiplying the first number by one digit of the second number, suitably shifted, and then the columns have to be added up:

```
          3 1 7 8
      ×   5 6 9 4
  ---------------
        1 2 7 1 2
      2 8 6 0 2
    1 9 0 6 8
  1 5 8 9 0
  ---------------
  1 8 0 9 5 5 3 2
```

If we decide to use as our representative operation—the one we count—the action of multiplying two digits—it is easy to see that the traditional algorithm has efficiency in $\Theta(n^2)$.

According to Levitin, in 1960 Anatoly Karatsuba developed a more efficient algorithm for this elementary school problem! We will now develop this algorithm using the divide and conquer technique.

Karatsuba used the divide and conquer idea of thinking about this multiplication not as a bunch of digit by digit products, but as a number of half-size problems. He approached the problem of multiplying two n-digit numbers by the strategy of doing some $n/2$ multiplication problems and then adding.

Specifically, suppose that x and y are n-digit numbers, and you want to form xy (the multiplication operator is often omitted as usual in algebra). So, we break x into its first $n/2$ digits, a, and its last $n/2$ digits b (assuming that n is even for convenience), then $x = 10^{n/2}a + b$, and similarly $y = 10^{n/2}c + d$, where all of a, b, c, and d are $n/2$ digits long.

For our earlier example, this idea means to compute

$$(31 \cdot 100 + 78) \cdot (56 \cdot 100 + 94)$$

Now, by simple algebra,

$$xy = (10^{n/2}a + b) \cdot (10^{n/2/}c + d) = ac10^n + (ad + bc)10^{n/2} + bd,$$

which shows that xy can be formed by doing four multiplications of size $n/2$, and then doing a bunch of shifting and adding. Note that we will ignore adding because adding two n digit numbers takes time proportional to n, whereas multiplication takes time proportional to n^2, so the work of adding is negligible, probably. Shifting is also cheaper than adding.

We can analyze the efficiency of this algorithm by letting $T(n)$ be the number of single digit multiplications needed to multiply together two n digit integers, and noting that

$$T(n) = 4T(n/2)$$

for the algorithm we have just designed.

In your group, use Project 7 to say what efficiency category this algorithm is in.

But, Karatsuba went on to notice something clever, namely that he could form the product of two n digits numbers by doing only *three* half-size multiplications. Instead of computing the 4 products ac, ad, bc, and bd, we can instead begin by computing $p_1 = (a + b) \cdot (c + d)$ (an apparently mildly insane action). Then we go ahead and compute $p_2 = ac$ and $p_3 = bd$, and, after only 3 size $n/2$ multiplications, we have enough information to compute xy.

$\Rightarrow$ Here is the crucial idea for Karatsuba's method: multiply out the quantity $p_1 = (a+b)(c+d)$ and figure out how you can use p_1, p_2, and p_3 to obtain the 4 products needed for the traditional algorithm (at a cost of only 3 products, and some extra additions and shiftings).

Work out our first example using this idea—just show how you can compute $3178 \cdot 5694$ using just three two-digit by two-digit multiplications, combined with some shifting and adding (or subtracting).

Now we want to analyze the efficiency of this algorithm. Of course, the complete divide and conquer algorithm uses the main idea recursively. For example, to multiply two 8-digit numbers, we use the main idea to form this product using just three 4-digit by 4-digit multiplications. But, then to do each of those 4-digit by 4-digit multiplications, we use the same idea to do each of them using just three 2-digit by 2-digit multiplications. And, finally, we do all of the 2-digit by 2-digit multiplications using the same main idea to require just three 1-digit by 1-digit multiplications. So, how many total 1-digit by 1-digit multiplications does that make? It's not so easy to see, which is why we use our standard analysis technique, as follows.

So, we let $S(n)$ be the number of 1-digit multiplications required to multiply two n-digit numbers, assume as usual that $n = 2^m$, and observe that

$$S(n) = S(2^m) = 3S(2^{m-1}) + a2^m + b,$$

where a and b are smallish positive integers, with these terms accounting for extra work being done to add, subtract, and shift various numbers of length 2^m, and any constant overhead.

In our usual way, you can show that

$$S(n) \in \Theta(3^{\log_2(n)}).$$

Prove the handy result that $3^{\log_2(n)} = n^{\log_2(3)}$.

> Show that the desired result is equivalent to an obviously true result by taking $\log_2()$ of both sides and noting that this operation is reversible—in other words, $\log_2(x) = \log_2(y)$ if and only if $x = y$ for x and y positive.

So, Karatsuba's algorithm takes $n^{\log_2(3)} \approx n^{1.5849625}$. For example with $n = 512$, Karatsuba's algorithm takes $3^{\log_2(512)} = 3^9 = 19683$ single-digit multiplications, whereas the traditional algorithm takes $512^2 = 262144$ single-digit multiplications.

> According to the documentation for the Java `BigInteger` class, Karatsuba's method is actually used for the `multiply` method.

$\Rightarrow$ As the only thing I will actually check for this Project, show how you can use Karatsuba's method to compute

$$3157624897154208 \cdot 1342796457403786$$

Assume you are using a calculator that can only compute products up to 4 digits times 4 digits, and do all additions, subtractions, and shifts, of any size, "by hand." Note that this will require you to do Karatsuba's idea multiple times, first for computing some 8 digit by 8 digit products, and then doing each of those by some 4 digit by 4 digit products (on the calculator), and then putting it all together by hand.

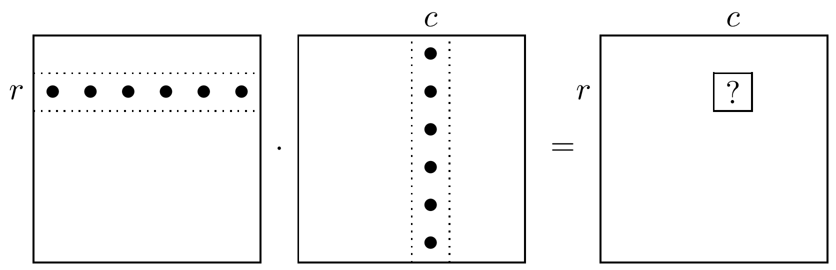
Strassen's Algorithm for Multiplying Square Matrices

In the next Project you'll learn how to design, by a divide-and-conquer—an algorithm that can multiply two square matrices (as usual with size that is a power of 2) more efficiently than the standard algorithm.

Matrix Multiplication

Recall from matrix algebra the standard algorithm for multiplying two matrices. To compute $C = AB$, where A is an m by p matrix and B is a p by n matrix, each element C_{rc} is computed as row r of A "times" column c of B, where a "row times a column" is done by multiplying all corresponding spots in the two lists and adding them up.

It is easy to see that for n by n matrices, a "row times a column" takes n multiplications and $n-1$ additions, which is in $\Theta(n)$, and there are n^2 elements of C to be computed, so traditional matrix multiplication is in $\Theta(n^3)$.



⇒ Project 10 [routine] Deriving and Demonstrating Strassen's Algorithm

For large n, $\Theta(n^3)$ can be a lot of work (though we are mostly studying an improved method for practice using divide and conquer, not because Strassen's algorithm is so useful in practice).

Consider multiplication of two 2 by 2 matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}.$$

This algorithm takes 8 multiplications and 4 additions. Strassen's algorithm is based on the following clever way to get the same result at a cost of 7 multiplications and a bunch of additions, which, when we do the time analysis, turns out to be better asymptotically.

> Of course, to be more efficient, Strassen's algorithm applies the same idea that we will be deriving recursively. For example, if you wanted to multiply two 1024 by 1024 matrices, a through h would each be 512 by 512 matrices, and Strassen's algorithm would be applied to compute each 512 by 512 product. To avoid going mildly insane, we will only study the clever algebra for Strassen's algorithm when a through e are single numbers.

The idea is to get the 8 scalars such as ae, bg, and so on, by multiplying sums of terms. For example, $(a+b)(e+h) = ae + ah + be + bh$, so at a cost of one multiplication we can get two terms that we want, namely ae and bh. Of course, we need those terms separately, not added together, and not added to the other terms.

To understand Strassen's idea, we use a 4 by 4 chart to visualize the terms obtained by a product of this sort. Here, for example, is the chart for $(a+b)(e+h) = ae + ah + be + bh$:

	e	f	g	h
a	$+$			$+$
b	$+$			$+$
c				
d				

Strassen figured out that if we use these quantities involving the four original numbers:

$$m_1 = (a+d)(e+h),$$

$$m_2 = (c-a)(e+f),$$

$$m_3 = (b-d)(g+h),$$

then we can form $m_1 + m_3$ and $m_1 + m_2$ and then find four terms with one original number times the sum of two others, like $(a+b)h$, and can finally form the 4 desired elements of the matrix product at a cost of just 7 multiplications.

⇒ Do all the following work to finish creating Strassen's algorithm (or some minor variation of Strassen's algorithm). Please do not look anything up on the internet—that won't help you with whatever variation of the following work I might decide to put on Test 1!

The "deliverable" to be checked for this Project will be your formulas that produce the four expressions needed for 2 by 2 matrix multiplication in only 7 multiplications, along with your demonstration of using the formulas, checked against the traditional approach.

Using the tables provided below, you can, with a little cleverness, figure out the details of Strassen's Algorithm.

Fill in the $+$ and $-$ symbols in these charts for the 4 *target* expressions in the matrix product—the ones we need for our final answer:

$ae + bg$:

	e	f	g	h
a				
b				
c				
d				

$af + bh$:

	e	f	g	h
a				
b				
c				
d				

$ce + dg$:

	e	f	g	h
a				
b				
c				
d				

$cf + dh$:

	e	f	g	h
a				
b				
c				
d				

Now fill in the symbols for the three key products figured out by Strassen, namely m_1, m_2, and m_3:

$m_1 = (a + d)(e + h)$:

	e	f	g	h
a				
b				
c				
d				

$m_2 = (c - a)(e + f)$:

	e	f	g	h
a				
b				
c				
d				

$m_3 = (b - d)(g + h)$:

	e	f	g	h
a				
b				
c				
d				

Fill in the symbols for the two suggested sums:

$m_1 + m_2$:

	e	f	g	h
a				
b				
c				
d				

$m_1 + m_3$:

	e	f	g	h
a				
b				
c				
d				

Here are some charts for figuring out the remaining 4 products we are allowed (with some extra for trial and error):

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

	e	f	g	h
a				
b				
c				
d				

⇒ Now, write down (somewhere you can cherish them forever, or at least until Test 1 is over) formulas for the four target expressions using sums and differences of just the 7 quantities you have found using 7 multiplications.

⇒ Also, verify that your formulas use 18 additions/subtractions of matrices of size $n/2$ along with 7 multiplications, which shows that the recurrence relation given in Project 7 for the number of additions in Strassen's algorithm is correct.

⇒ Demonstrate your formulas to compute

$$\begin{bmatrix} 5 & 7 \\ 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 3 & 6 \end{bmatrix}.$$

Be sure to check your result against the traditional algorithm so you can catch any errors in your formulas or process.

⇒ Project 11 [routine] The Integers Mod N

This Project has nothing to do with divide-and-conquer, but presents material that you will need to understand for upcoming algorithms, including several that use divide-and-conquer.

Define Z_n to be the integers where arithmetic is done using this simple rule:

> do the operation (addition, subtraction, and multiplication) as usual, and then reduce the result by doing integer division by n and taking the remainder as the result. This operation is known as "reducing a number 'mod n'. "

You may or may not have seen this in your Discrete Math course. We consider two integers as being "the same" if they differ by a multiple of n. Note that all integers belong to Z_n, but we prefer 0 through $n-1$ as representatives. When reduced mod n, all integers are seen to be equal to one of these *canonical* values. This is a lot like fractions—we prefer $\frac{1}{2}$ to the infinitely many other ways to express this same number as a ratio of two integers, like, say, $\frac{17}{34}$.

For example, Z_5 has the canonical values 0 through 4, with these addition and multiplication tables:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

⇒ Verify some of the values in this table. For any prime number n, it turns out that Z_n is a *field*, which is an algebraic entity that satisfies all the algebraic properties of the real numbers that don't involve comparisons. Note that for every number x, there is a number y such that $x+y=0$, and for every non-zero number x, there is a number z that that $xz=1$. Also, $x+0=x$ and $x \cdot 1 = x$ for every x, so 0 and 1 are additive and multiplicative identity elements. Since arithmetic is done as usual, followed by reduction mod n, both addition and multiplication satisfy the associative, commutative, and distributive properties.

⇒ Play with these ideas a little more by computing computing, say, $8 \cdot 15$ in Z_{17} (some calculators are fancy enough to reduce a number mod 17 as a built-in operation, but if not, you can divide the number by 17, hit =, subtract the whole number part, hit =, and then multiply by 17 to get the remainder. Also, when working in Z_{17}, it can be useful to start by quickly writing down a bunch of multiples of 17, namely 17, 34, 51, 68, ..., 170. These values are all 0 in Z_{17}, which let's you quickly reduce numbers by 17 in your head, like $40 = 6$ because $40 - 34 = 6$)

Find the number in Z_{17} you can add to 8 to give 0 (this number can be referred to as -8, but we want it as one of the canonical numbers, i.e, from 0 through 16).

Find the number in Z_{17} you can multiply by 8 to give 1. Note that this is considerably harder—later we'll learn an efficient algorithm for doing it.

Does 13 have a square root in Z_{17} (a number x such that $x^2 = 13$)? If so, find it.

It will turn out that there is a very useful pattern for the powers of a given value in Z_n. To explore this, fill in this table of the various powers of $\alpha = 3$, in Z_{17}. Be sure to do this efficiently by noting that $\alpha^k = \alpha^{k-1} \cdot \alpha$ for $k \geq 1$—do not compute α^k in Z_{17} using your calculator and then reduce the result mod 17 (this approach will, later, quickly overwhelm your calculator, even for relatively small values of n):

k	0	1	2	3	4	5	6	7	8	9	10
3^k											

k	11	12	13	14	15	16	17	18	19	20
3^k										

The Fast Fourier Transform

Now 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 instead study a toy version of FFT—interpolation over Z_p.

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 $0 \le j < n$ 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 $0 \le j < n$,

$$P(x_j) = y_j.$$

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:

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

Note that all the values in the coefficient matrix (the big square one) are known, since all the x_j are given numbers, and similarly all the values in the right-hand side 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

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:

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

Either by hand (the problem has been rigged so the row operations don't get too crazy) or by a computer/calculator, we do row operations to find that $c_0 = 4$, $c_1 = 1$, $c_2 = -3$, and $c_3 = 1$, so the desired polynomial is

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

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

$\Rightarrow$ **Embedded Exercise 10**

Make sure that you understand how the coefficient matrix was created, and check that the claimed solution actually solves the linear system and gives the correct polynomial.

In the usual case, where all the x_j, y_j, and c_j are ordinary real numbers, MUC turns out to be a terrible algorithm! It is very inefficient, and 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 mention the field of *complex numbers* as the actual point 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 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!

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

$$x_j = \alpha^j,$$

for $0 \le j < n$.

> 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 $n = 8$ as an introductory example of the big ideas of the FFT.

> Note, though, that we can't use Z_9 as our field for these values, because 9 is not prime and it turns out that no α in Z_9 has the desired properties. The fields Z_5, with $n = 4$, and Z_{17}, with $n = 16$ would be better, but one is too small to see the algebraic properties, and the other is kind of too big, so we'll stick with $n = 8$, realizing that really this example would only work over the complex number field.

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

$$F_8 = \begin{bmatrix} \alpha^{0\cdot 0} & \alpha^{0\cdot 1} & \alpha^{0\cdot 2} & \alpha^{0\cdot 3} & \alpha^{0\cdot 4} & \alpha^{0\cdot 5} & \alpha^{0\cdot 6} & \alpha^{0\cdot 7} \\ \alpha^{1\cdot 0} & \alpha^{1\cdot 1} & \alpha^{1\cdot 2} & \alpha^{1\cdot 3} & \alpha^{1\cdot 4} & \alpha^{1\cdot 5} & \alpha^{1\cdot 6} & \alpha^{1\cdot 7} \\ \alpha^{2\cdot 0} & \alpha^{2\cdot 1} & \alpha^{2\cdot 2} & \alpha^{2\cdot 3} & \alpha^{2\cdot 4} & \alpha^{2\cdot 5} & \alpha^{2\cdot 6} & \alpha^{2\cdot 7} \\ \alpha^{3\cdot 0} & \alpha^{3\cdot 1} & \alpha^{3\cdot 2} & \alpha^{3\cdot 3} & \alpha^{3\cdot 4} & \alpha^{3\cdot 5} & \alpha^{3\cdot 6} & \alpha^{3\cdot 7} \\ \alpha^{4\cdot 0} & \alpha^{4\cdot 1} & \alpha^{4\cdot 2} & \alpha^{4\cdot 3} & \alpha^{4\cdot 4} & \alpha^{4\cdot 5} & \alpha^{4\cdot 6} & \alpha^{4\cdot 7} \\ \alpha^{5\cdot 0} & \alpha^{5\cdot 1} & \alpha^{5\cdot 2} & \alpha^{5\cdot 3} & \alpha^{5\cdot 4} & \alpha^{5\cdot 5} & \alpha^{5\cdot 6} & \alpha^{5\cdot 7} \\ \alpha^{6\cdot 0} & \alpha^{6\cdot 1} & \alpha^{6\cdot 2} & \alpha^{6\cdot 3} & \alpha^{6\cdot 4} & \alpha^{6\cdot 5} & \alpha^{6\cdot 6} & \alpha^{6\cdot 7} \\ \alpha^{7\cdot 0} & \alpha^{7\cdot 1} & \alpha^{7\cdot 2} & \alpha^{7\cdot 3} & \alpha^{7\cdot 4} & \alpha^{7\cdot 5} & \alpha^{7\cdot 6} & \alpha^{7\cdot 7} \end{bmatrix}$$

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

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 unknows 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 have 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:

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

Checking the first observation (which isn't so obvious now that we've reduced things), we can see that the system is

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

Now, 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 (we don't actually do the matrix multiplication shown, because it can be done much more efficiently in an obvious way).

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

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

⇒ Project 12 [routine] Exploring FFT

Your job on this Project is to work through all the previous stuff with a concrete example. Specifically, use Z_{17} as the field for all the numbers, and use $n = 8$. To do this (which I realized after already printing the earlier comment about F_8), we can use $\alpha = 9$.

For your convenience with this exercise, 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}, to save lots of work later:

	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

Write here the powers of 9 in Z_{17} (you'll need this in the next work):

k	0	1	2	3	4	5	6	7	8
9^k									

Note that all of the following steps were described more abstractly in the previous discussion—all you have to do is go through with concrete numbers in Z_{17} and make sure that you understand everything.

Define F_8 by

$$(F_8)_{j,k} = 9^{jk}.$$

(using $\alpha = 9$, which works because $9^8 = 1$ in Z_{17}, and no smaller power of 9 gives 1)

Using the coefficient matrix given on the bottom of page 36 (where $\alpha^8 = 1$ has been used, and the columns and values of c_j have been switched around), fill in all the values in the modified coefficient matrix (noting that all the values in this matrix are available in your chart of the powers of 9 in Z_{17}):

$$\begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^1 & \alpha^3 & \alpha^5 & \alpha^7 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^2 & \alpha^6 & \alpha^2 & \alpha^6 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^3 & \alpha^1 & \alpha^7 & \alpha^5 \\ \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 & \alpha^4 & \alpha^4 & \alpha^4 & \alpha^4 \\ \alpha^0 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^5 & \alpha^7 & \alpha^1 & \alpha^3 \\ \alpha^0 & \alpha^4 & \alpha^0 & \alpha^4 & \alpha^6 & \alpha^2 & \alpha^6 & \alpha^2 \\ \alpha^0 & \alpha^6 & \alpha^4 & \alpha^2 & \alpha^7 & \alpha^5 & \alpha^3 & \alpha^1 \end{bmatrix} = \begin{array}{c|c|c|c|c|c|c|c|c|} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & & & & & & & & \\ \hline 1 & & & & & & & & \\ \hline 2 & & & & & & & & \\ \hline 3 & & & & & & & & \\ \hline 4 & & & & & & & & \\ \hline 5 & & & & & & & & \\ \hline 6 & & & & & & & & \\ \hline 7 & & & & & & & & \\ \hline \end{array}$$

Now you need to show how to compute F_8 times a given vector c, using the Fast Fourier multiplication idea, using the example (chosen at random):

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

Let's digress to think about efficiency a little here. First, how much work is it to form a matrix such as this? There are n^2 components, and computing each component takes $\Theta(1)$ work, so it takes $\Theta(n^2)$ work to form an n by n matrix like this. This might seem prohibitively high, because in a minute we are going to get all happy over the fact that the Fast Fourier multiplication idea drops the work of multiplying a square matrix times a vector from $\Theta(n^2)$ to $\Theta(n \log n)$, so doesn't the fact that it takes $\Theta(n^2)$ to form the matrix in the first place make this improvement negligible?

It would if we had to form the matrix every time we wanted to do a matrix-vector multiplication, but note that once we pick α and n (basically choosing how many uniformly distributed frequencies we want to sample), the coefficient matrix can be formed once and used for any number of different c vectors. It's like each time you talk for a few seconds, the analog data from that talking gives a c, which is then quickly multiplied by the n by n matrix, which was formed once and hard-coded into your cell phone, essentially.

In concrete terms for this Project, it would take 64 multiplications (along with the corresponding additions) to compute the matrix-vector product F_8c. As you proceed with this Project, note that you will be showing how to accomplish the same thing using just $n \log n = 8 \cdot 3 = 24$ multiplications.

By the way, note that we started talking about interpolation, which meant solving $F_nc = y$, with y known, to find the polynomial coefficients c, but now for convenience we're considering how to multiply F_nc to get y. We'll fix this later.

Now write out the top-level step, as follows.

First, looking at the 4 sub-matrices (the dividing lines have been conveniently provided), note that you really have just one 4 by 4 matrix—appearing in the upper left section—with some multiplications.

Note that the upper left 4 by 4 matrix, which we will denote by F_4, also occurs in the lower left 4 by 4 section.

Carefully write out the details showing that the upper right 4 by 4 matrix can be obtained by taking F_4 and multiplying each row, in order, by α^0, α^1, and so on. Similarly, show/verify that the lower right 4 by 4 section can be obtained by multiplying the rows by α^4, α^5, and so on.

Now, be careful to rearrange the rows of c in the same way that the columns of F_8 were rearranged. Then, write out what calculations need to be done, noting that only two matrix-vector multiplications are needed.

Back on page 37 we conveniently express the multiplying of the rows of F_4 by various powers of α using diagonal matrices, but note that if you have any square matrix A, and vector x, and any diagonal matrix D, then forming DAx can be done in $\Theta(n^2)$ time, even though general multiplication of two n by n matrices takes $\Theta(n^3)$. This is because we can do $DAx = D(Ax)$, and note that a diagonal matrix times a vector can be done in $\Theta(n)$ time.

At this point, if we were willing to perform these two half-size matrix-vector multiplications, we could quickly be done. But, to get the full idea of Fast Fourier multiplication, you should now repeat the entire process for both products, doing to F_4 everything that had to be done with F_8.

Figure out the 4 by 4 matrix you need to multiply by two 4-vectors. Then compute the powers of 13 and note that this 4 by 4 matrix, with columns swapped, is F_4 according to the formula, with $\alpha = 9^2 = 13$.

Now apply the formulas for doing the two desired 4 by 4 matrix times 4-vectors efficiently. When you get down to wanting 2 by 2 matrices times 2-vectors, just go ahead and use the traditional slow algorithm (we could sub-divide once more to get down to 1 by 1 matrices times 1-vectors, but let's not).

Put it all together and consider your final answer for $F_8 c$. Compare this result to what you get by doing the "64 multiplications" traditional slow algorithm, and fix any problems that have occurred.

⇒ Project 13 [routine] Computing the Inverse of a Fourier Coefficient Matrix

We actually need to be able to compute $F_8^{-1} y$ to accomplish our original goal (interpolation) and we've been talking in all the previous about computing $F_8 c$. Fortunately, with these special matrices computing the inverse is super-easy (and takes just $\Theta(n^2)$ time, versus $\Theta(n^3)$ for a general matrix, say using Gaussian elimination).

It turns out that F_n^{-1} can be trivially computed by the simple formula

$$F^{-1} = \frac{1}{n}\overline{F_n},$$

where $\overline{F_n}$ is obtained by taking the inverse in the field of each element in F_n.

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 $\overline{F_n}$ is

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

so the element of $F_n\overline{F_n}$ 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 the element 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, 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

$$r = \alpha^j \beta^j = \alpha^j \overline{\alpha}^j = (\alpha \overline{\alpha})^j = (|\alpha|^2)^j = 1^j = 1$$

so the element adds up to n.

$\Rightarrow$ To verify concretely that the previous discussion is correct, write down the component-wise inverse (obtained by taking the multiplicative inverse of each component) of the matrix F_8 from Project 12, and then multiply that matrix times F_8 to see that things really work!

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.

$\Rightarrow$ 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:

$$\begin{aligned}
\mathrm{e}^{i\theta} &= 1 + \frac{(i\theta)^1}{1!} + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \cdots \\
&= 1 + \frac{\theta^1}{1!}i - \frac{\theta^2}{2!} - \frac{\theta^3}{3!}i + \frac{\theta^4}{4!} + \frac{\theta^5}{5!}i + \cdots \\
&= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} + \cdots\right) + \left(\frac{\theta^1}{1!} - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} + \cdots\right) i \\
&= \cos(\theta) + \sin(\theta) i.
\end{aligned}$$

One consequence of this is that a complex number of the form $e^{\theta i}$ is on the unit circle in the complex plane.

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 $\alpha = \mathrm{e}^{\frac{2\pi}{n}i}$ 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 is sampled at some data points, and then interpolated using the FFT. The coefficients that generate an approximation to the analog function are digitized and sent over the wires/airwaves, with those coefficients being used, again with the FFT, to reproduce that approximation on the other end. The point is that the whole process needs to take place in real time without irritating the speaker and listener.
