(M. Dietzfelbinger, 2018-12-05)

Remarks on

2.6 The Fast Fourier Transform

Note: You should refer to the presentation in the book as well.

2.6.1 Multiplying polynomials and an alternative representation

The purpose of Section 2.6 in the book is to develop a fast algorithm for multiplying polynomials with one variable. What is a polynomial? What does it mean to *multiply* polynomials?

Examples for polynomials:

$$A(x) = -3 + 4x^2 + 5x^3 \quad \text{and} \quad B(x) = -3 - 3x^2 + 7x^5.$$

Goal: *Multiply* the polynomials, i.e., calculate:

$$\begin{aligned} A(x) \cdot B(x) &= (-3) \cdot (-3) + (4 \cdot (-3) + (-3) \cdot (-3))x^2 + (5 \cdot (-3))x^3 \\ &\qquad + (4 \cdot (-3))x^4 + (5 \cdot (-3) + ((-3) \cdot 7)x^5 + (4 \cdot 7)x^7 + (5 \cdot 7)x^8 \\ &= 9 - 3x^2 - 15x^3 - 12x^4 - 36x^5 + 28x^7 + 35x^8. \end{aligned}$$

The rule behind the calculation: Multiply out all terms, and collect coefficients with the same power x^k of x.

More generally, a polynomial $A(x)$ (over $\mathbb{Z}$, or $\mathbb{Q}$, or $\mathbb{R}$, or $\mathbb{C}$) is a sum of finitely many terms $a_i x^i$ with *coefficients* a_i (from $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$, respectively) and powers x^i of the *variable* x (which is just a symbol). In general formulas polynomial multiplication reads as follows.

Given are two polynomials

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d \quad \text{and} \quad B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d,$$

by their sequences $(a_0, \ldots, a_d)$ and $(b_0, \ldots, b_d)$ of coefficients.[1] The product $A(x) \cdot B(x)$ is the polynomial

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d},$$

where the coefficients $c_0, \ldots, c_{2d}$ are given by

$$c_k = a_0 b_k + a_1 b_{k-1} + a_2 b_{k-2} + \cdots + a_k b_0 = \sum_{\substack{0 \le i,j \le d \\ i+j=k}} a_i b_j, \text{ for } 0 \le k \le 2d. \tag{2.1}$$

(This is the sum of all products $a_i b_j$ of coefficients where the corresponding product $x^i x^j$ equals x^k.)

The task of *polynomial multiplication* is to calculate $(c_0, \ldots, c_{2d})$ as in (2.1) from $(a_0, \ldots, a_d)$ and $(b_0, \ldots, b_d)$.

Note that in this formulation no variable is mentioned anymore. One just transforms two sequences of numbers into a new one. The sequence $(c_0, \ldots, c_{2d})$ is called the *convolution* of $(a_0, \ldots, a_d)$ and $(b_0, \ldots, b_d)$.

Please read page 59 in the book. There it is explained why polynomial multiplication is extremely important in *signal processing*, an elementary and ubiquitous task in engineering, and why d may be quite large, many thousands, in such engineering applications.

By the very definition of the polynomial product, it can be calculated in time[2] $O(d^2)$. For polynomials with many thousands of coefficients this is too slow, if the operation is to be carried out frequently.

Goal: We wish to find an algorithm that multiplies polynomials in time $O(d \log d)$.

There is a fundamental trick to be used here, which tells us we should "*reduce*" *multiplication* to *evaluation* and *interpolation*.

As a basis for this approach, one has to understand that a polynomial $A(x)$ can be described not only by its coefficients, but equally well by its values $A(x_i)$ at sufficiently many given points x_i.

[1] One should not worry if the degrees of $A(x)$ and $B(x)$, i.e., the highest powers x^i with a nonzero coefficient, are not the same in $A(x)$ and $B(x)$. Just add zero coefficients at the right to get two coefficient sequences of equal length.

[2] We will always count an arithmetic operation on coefficients as one operation, and "time" essentially means the number of these operations.

Example: The value of $A(x) = -3+4x^2+5x^3$ at $x_0 = 2$: $A(2) = -3+4\cdot 2^2+5\cdot 2^3 = 53$.

Definition: If z is a number, then the *value* $A(z)$ of the polynomial $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_dx^d$ at z is the number $a_0 + a_1z + a_2z^2 + \cdots + a_dz^d$.

Recall the following: If $A(x) = ax + b$ is a linear polynomial, x_0 and x_1 are different, and $r_0 = A(x_0)$ and $r_1 = A(x_1)$ are known, we can calculate the coefficients: $a = \frac{r_1-r_0}{x_1-x_0}$ and $b = r_0 - ax_0$. Also, if $ax^2 + bx + c$ is a quadratic polynomial and we have values $r_0 = A(x_0)$, $r_1 = A(x_1)$, $r_2 = A(x_2)$, we can calculate the coefficients a, b, c.

The idea is that a polynomial of arbitrary degree is determined if its values at sufficiently many points are known.

Fact: The following are two *equivalent* representations for a polynomial $A(x)$ of degree at most d, given $d+1$ distinct real numbers $x_0 \ldots, x_d$ (often called "sample points"):

(i) The vector $(a_0, \ldots, a_d)$ of coefficients.

(ii) $d+1$ values $r_0 = A(x_0)$, $r_1 = A(x_1)$, $\ldots$, $r_d = A(x_d)$.

The fact means that if one knows $x_0 \ldots, x_d$ and $a_0, \ldots, a_d$, then one can calculate $A(x_0)$, $A(x_1)$, $\ldots$, $A(x_d)$ (this is clear), and if one knows $x_0 \ldots, x_d$ and $r_0, \ldots, r_d$ with $r_i = A(x_i)$ for $0 \le i \le d$, then the coefficients $a_0, \ldots, a_d$ are uniquely determined and one can even calculate them (this is maybe not quite so clear).

Idea of proof of claim: Given $x_0, \ldots, x_d$, one looks at the *Vandermonde matrix* $V(x_0, \ldots, x_d)$, which is a $(d+1) \times (d+1)$-matrix:

$$V(x_0, \ldots, x_d) = \begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 & \ldots & x_0^d \\ 1 & x_1 & x_1^2 & x_1^3 & \ldots & x_1^d \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & x_d^3 & \ldots & x_d^d \end{pmatrix}. \tag{2.2}$$

Note that if $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ and $r_i = A(x_i)$ for $i = 0, 1, \ldots, d$, then

$$V(x_0, \ldots, x_d) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_d \end{pmatrix}. \tag{2.3}$$

It is well known, and taught in calculus and linear algebra classes that the determinant of the Vandermonde matrix is $\det(V(x_0, \ldots, x_d)) = \prod_{0 \le i < j \le d} (x_j - x_i)$, which is nonzero if the numbers $x_0, \ldots, x_d$ are different. Hence $V(x_0, \ldots, x_d)$ is regular, i.e., (2.3) can be used in both directions: If $a_0, \ldots, a_d$ are given, we can calculate the values $r_0, \ldots, r_d$; if $r_0, \ldots, r_d$ are given, we can calculate $a_0, \ldots, a_d$ by solving a linear system.

The two computational tasks involved in this back-and-forth transformation are:

Evaluation: Calculate (ii) from (i).
Interpolation: Calculate (i) from (ii).

If we multiply two polynomials of degree at most d, the product will have degree at most $2d$. For doing interpolation with such a polynomial we need at least $2d+1$ points. We will choose some $n > 2d$ and expand coefficient sequences and value sequences of all concerned polynomials to length exactly n, by adding zeros as coefficients. The degree, i.e., the highest power of x that appears, will be at most $n-1$. It will be convenient to assume n is a power of 2. This can be achieved with some $n \le 4d$.

Example: If $A(x) = -3 + 4x^2 + 5x^3$ and $B(x) = -3 - 3x^2 + 7x^5$, we have $d \ge 5$, hence $2d \ge 10$, and we would choose $n = 16$ and coefficient sequences $(-3, 0, 4, 5, 0, \ldots, 0)$ and $(-3, 0, -3, 0, 0, 7, 0, \ldots, 0)$ of length 16, and 16 points $x_0, \ldots, x_{15}$.

If given algorithms for evaluation and interpolation, we can solve polynomial multiplication as follows.

(1) Fix some $n > 2d$ and suitable (distinct) arguments $x_0, \ldots, x_{n-1}$.

(2) Use **evaluation** to find $y_0 = A(x_0)$, ..., $y_{n-1} = A(x_{n-1})$, and $z_0 = B(x_0)$, ..., $z_{n-1} = B(x_{n-1})$.

(3) Calculate $u_0 = y_0 \cdot z_0$, $u_1 = y_1 \cdot z_1$, ..., $u_{n-1} = y_{n-1} \cdot z_{n-1}$.
// ($O(n) = O(d)$ time.) Then: $C(x_0) = u_0, C(x_1) = u_1, \ldots, C(x_{n-1}) = u_{n-1}$.

(4) Use **interpolation** to calculate the coefficients $c_0, c_1, \ldots, c_{n-1}$ of $C(x)$ from $u_0, u_1, \ldots, u_{n-1}$.

Example: $A(x) = x + 1$, $B(x) = x + 2$.

(1) $n = 4$, and $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$.

(2) $y_0 = 1$, $y_1 = 2$, $y_2 = 3$, $y_3 = 4$, and $z_0 = 2$, $z_1 = 3$, $z_2 = 4$, $z_3 = 5$.

(3) $u_0 = 1 \cdot 2 = 2$, $u_1 = 2 \cdot 3 = 6$, $u_2 = 3 \cdot 4 = 12$, $u_3 = 4 \cdot 5 = 20$.

(4) Solve

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \\ 12 \\ 20 \end{pmatrix} \tag{2.4}$$

to find the solution

$$c_0 = 2, c_1 = 3, c_2 = 1, c_3 = 0.$$

(Check that this vector solves the equation, and that x^2+3x+2 is the product.)

We will develop a fast algorithm for evaluation, for cleverly chosen points $x_0, \ldots, x_{n-1}$. (Actually, these will not be real numbers, but complex numbers.) This algorithm is called the "fast Fourier transform (FFT)". The fast algorithm for interpolation will be an easy variation of the FFT.

2.6.2 Evaluation of a polynomial at several points: FFT

(Multiple) **Evaluation** of a polynomial means the following: We are given a polynomial

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

over the reals or the complex numbers (by its coefficient vector $(a_0, a_1, \ldots, a_{n-2}, a_{n-1})$), and want to evaluate it at n points $x_0, x_1, \ldots, x_{n-1}$, i.e., we want to calculate

$$A(x_0), A(x_1), \ldots, A(x_{n-1}).$$

The vector $(A(x_0), A(x_1), \ldots, A(x_{n-1}))$ is called the *value vector* for argument vector $(x_0, x_1, \ldots, x_{n-1})$.

Note that $(A(x_0), A(x_1), \ldots, A(x_{n-1}))^\mathsf{T}$ is the product $V(x_0, \ldots, x_{n-1}) \cdot (a_0, a_1, \ldots, a_{n-1})^\mathsf{T}$ for the Vandermonde matrix from (2.2).

The naive way to calculate these n values by the *Horner scheme*

$$A(y) = ((\cdots(((a_{n-1} \cdot y + a_{n-2}) \cdot y + a_{n-3}) \cdot y + a_{n-4}) \cdots) \cdot y + a_1) \cdot y + a_0,$$

for $y = x_0, \ldots, x_{n-1}$ one after the other, would involve $n(n-1)$ multiplications and n^2 additions, and would require $\Theta(n^2)$ time. The algorithm we shall develop will do it much, much faster, if the points $x_0, x_1, \ldots, x_{n-1}$ are cleverly chosen. We shall choose these points later. With these particular points the transformation

$$(a_0, a_1, \ldots, a_{n-2}, a_{n-1}) \mapsto (A(x_0), A(x_1), \ldots, A(x_{n-1}))$$

is called the "discrete Fourier transform". The algorithm we are going to develop is called "fast Fourier transform" or "FFT". It follows the Divide-and-Conquer paradigm.

For simplicity, we assume from here on that $n = 2^k$ for some integer $k \geq 0$.
(We can always add some coefficients with value 0, so that there are n coefficients, for n a power of 2.)

Divide-and-Conquer says: (0) If the input is **small**, solve the problem directly. Otherwise: (1) **split**, (2) use **recursion**, and (3) **combine**. Next we discuss the details.

(0) Triviality test: If $n = 1$, then $A(x) = a_0$ is a constant polynomial and there is only one input x_0, so the result is $A(x_0) = a_0$. The time for this is $O(1)$.

From here on assume $n > 1$.

(1) Splitting: The input is the polynomial $A(x)$, given by its n coefficients $a_0, \ldots, a_{n-1}$. Also, we have $x_0, x_1, \ldots, x_{n-1}$.

We want to form two subinstances of half the size. This means: Two polynomials with degree $\frac{n}{2} - 1$ each, and $\frac{n}{2}$ inputs $x'_0, \ldots, x'_{n/2-1}$.

For forming two polynomials there are many possibilities. We choose the following specific way. ("e" is for "even", and "o" is for "odd".)

$$\begin{aligned} A_{\mathrm{e}}(x) &= a_0 + a_2 x + \cdots + a_{n-4} x^{\frac{n}{2}-2} + a_{n-2} x^{\frac{n}{2}-1} \text{ , and} \\ A_{\mathrm{o}}(x) &= a_1 + a_3 x + \cdots + a_{n-3} x^{\frac{n}{2}-2} + a_{n-1} x^{\frac{n}{2}-1}. \end{aligned} \tag{2.5}$$

(Note that the exponent with the x is halved in each term.) Does it split the input? Yes. If the original coefficient vector is $(a_0, a_1, \ldots, a_{n-2}, a_{n-1})$, we get two polynomials with coefficient vectors

$$\begin{aligned} &(a_0, a_2, \ldots, a_{n-4}, a_{n-2}) \quad \text{(even-numbered positions) and} \\ &(a_1, a_3, \ldots, a_{n-3}, a_{n-1}) \quad \text{(odd-numbered positions).} \end{aligned} \tag{2.6}$$

Both polynomials have (about) half the degree of $A(x)$, and coefficient vectors of length $n/2$.

Example: Let $A(x) = 3 + 5x + x^2 + 6x^3 - 4x^4 - 2x^6 + 9x^7$. The coefficient vector is $(3, 5, 1, 6, -4, 0, -2, 9)$. Then

$$A_{\mathrm{e}}(x) = 3 + x - 4x^2 - 2x^3 \quad \text{and} \quad A_{\mathrm{o}}(x) = 5 + 6x + 9x^3.$$

The two coefficient vectors for the recursion are $(3, 1, -4, -2)$ and $(5, 6, 0, 9)$.

The following fundamental formula connects $A(x)$, $A_{\mathrm{e}}(x)$, and $A_{\mathrm{o}}(x)$:

$$\boxed{A(x) = A_{\mathrm{e}}(x^2) + x \cdot A_{\mathrm{o}}(x^2).} \tag{2.7}$$

One should check how this works. By substituting x^2 in $A_{\mathrm{e}}(x)$ we get the even-numbered coefficients with the correct powers of x; by substituting x^2 in $A_{\mathrm{o}}(x)$ and multiplying by x the same happens for the odd-numbered coefficients. In the example: $A_{\mathrm{e}}(x^2) = 3 + x^2 - 4x^4 - 2x^6$ and $x \cdot A_{\mathrm{o}}(x^2) = x \cdot (5 + 6x^2 + 9x^6)$. If we multiply out and add, we get

$$3 + x^2 - 4x^4 - 2x^6 + 5x + 6x^3 + 9x^7 = A(x).$$

If we can manage to calculate $A_{\mathrm{e}}(x_j^2)$ and $A_{\mathrm{o}}(x_j^2)$ by recursion, for $0 \le j < n$, we are fine, because then we can use (2.7) to calculate $A(x_0), A(x_1), \ldots, A(x_{n-1})$ in time $O(n)$. This looks as if we have to evaluate A_{e} and A_{o} at n points each. This doesn't fit, since by recursion we can handle only $n/2$ points. It seems we are stuck.

However, if we can arrange things so that $x_0^2, x_1^2, \ldots, x_{n-1}^2$ are not n different points, but only $n/2$ many, it will work. So here's big **trick number one** of FFT: We will make sure that the n different points $x_0, x_1, \ldots, x_{n-1}$ come in "plus-minus pairs":

$$x_{n/2} = -x_0, x_{n/2+1} = -x_1, \ldots, x_{n-1} = -x_{n/2-1}. \tag{2.8}$$

Then

$$x_{n/2}^2 = x_0^2, x_{n/2+1}^2 = x_1^2, \ldots, x_{n-1}^2 = x_{n/2-1}^2,$$

and $x_0^2, x_1^2, \ldots, x_{n-1}^2$ are only $n/2$ different numbers!

Example: The eight input points could be $(1, 2, 3, 4, -1, -2, -3, -4)$. The vector of squares is $(1, 4, 9, 16, 1, 4, 9, 16)$, containing only four different numbers.

So we assume for the moment that the n input numbers are arranged in plus-minus pairs as in (2.8).

Then splitting, recursion, and combining works as follows:

Splitting: Form coefficient vectors for $A_{\mathrm{e}}(x)$ and $A_{\mathrm{o}}(x)$, as given in (2.6), and form sample point vector $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Recursion: Solve the FFT problem on the inputs $A_{\mathrm{e}}(x)$ with $(x'_0, x'_1, \ldots, x'_{n/2-1})$ and $A_{\mathrm{o}}(x)$ with $(x'_0, x'_1, \ldots, x'_{n/2-1})$ recursively. This gives two value vectors

$$\begin{aligned} &(s_0, \ldots, s_{n/2-1}) = (A_{\mathrm{e}}(x_0^2), A_{\mathrm{e}}(x_1^2), \ldots, A_{\mathrm{e}}(x_{n/2-1}^2)) \quad \text{and} \\ &(t_0, \ldots, t_{n/2-1}) = (A_{\mathrm{o}}(x_0^2), A_{\mathrm{o}}(x_1^2), \ldots, A_{\mathrm{o}}(x_{n/2-1}^2)). \end{aligned} \tag{2.9}$$

Combining: Using (2.7) and (2.8) we can determine the result from (2.9) as follows:

$$\begin{aligned} A(x_j) &= A_{\mathrm{e}}(x_j^2) + x_j \cdot A_{\mathrm{o}}(x_j^2) = s_j + x_j t_j, \text{ for } 0 \le j < n/2; \\ A(x_{n/2+j}) &= A_{\mathrm{e}}(x_j^2) - x_j \cdot A_{\mathrm{o}}(x_j^2) = s_j - x_j t_j, \text{ for } 0 \le j < n/2. \end{aligned}$$

Note that only $n/2$ multiplications and n additions are needed in the combining step.

The time analysis is simple, using the master theorem. If $T(n)$ is the running time for inputs of size n, we have the recurrence

$$T(n) = \begin{cases} 1 & \text{, if } n = 1; \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n) & \text{, if } n > 1. \end{cases}$$

The master theorem, second case, gives the solution $T(n) = O(n \log n)$. (The parameters: $a = b = 2$, $d = 1$.)

A problem yet to be solved is how we can make sure that we always have plus-minus pairs, on each level of the recursion. For this, we have to choose the sample points $x_0, x_1, \ldots, x_{n-1}$ in a very clever way. But how? If, say, on the outermost level, we start out with sample points $1, 2, \ldots, n/2-1, -1, -2, \ldots, -(n/2-1)$, then on the next level of recursion we have the $n/2-1$ numbers $1, 2, 4, 9, \ldots, (n/2-1)^2$, all positive, and no plus-minus pairs anymore. It is quite clear that with real numbers as arguments we can never succeed, since squares of nonzero numbers are positive, and can never form plus-minus pairs. So we have to look beyond real numbers. Here's big **trick number two** of FFT: use *complex numbers*.

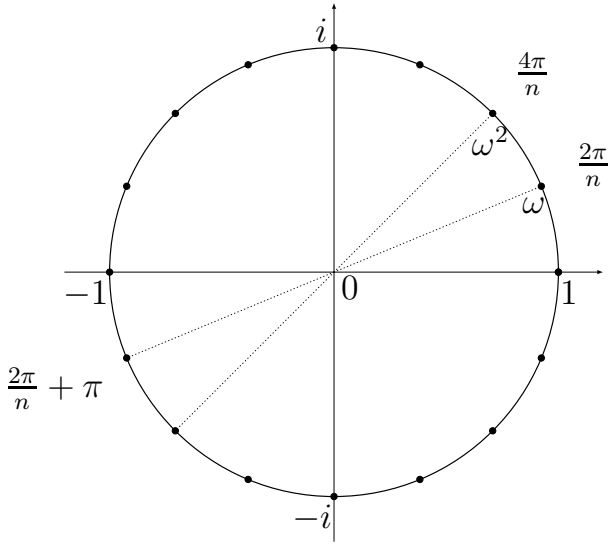
A brief introduction into the aspects of the geometry and arithmetic of complex numbers that we need is given in the book (page 63) and not repeated here.

The central concept is that of an ***n*th root of unity**. This is a complex number z with the property that $z^n = 1$. One can show that there are exactly n such numbers, and that they all have absolute value 1, which means that in the complex plane they sit on the unit circle. Actually, one can describe them very precisely. Given $n = 2^k$, we define (with i the imaginary unit)[3]:

$$\omega := e^{2\pi i/n} = \cos(2\pi/n) + i \cdot \sin(2\pi/n).$$

Geometrically seen, this is the point in the complex plane that one reaches if one starts from 1 and walks counterclockwise along the unit circle for an angle of $2\pi/n$ (which is a fraction of $1/n$ of the full circle). This is an nth root of unity, since $(e^{2\pi i/n})^n = e^{2\pi i} = \cos(2\pi) + i \cdot \sin(2\pi) = 1$.

[3]The Greek letter "ω" is read as "omega".



Our sample points are the n powers of ω:

$$x_0 = \omega^0 = 1, x_1 = \omega^1 = \omega, x_2 = \omega^2, \ldots, x_{n-1} = \omega^{n-1}.$$

By the properties of multiplication in the complex numbers ("multiply absolute values, add angles") these points sit on the unit circle, at regular distances (see picture). Arithmetically, we calculate $(\omega^j)^n = (\omega^n)^j = 1^j = 1$ to see that these numbers are exactly the nth roots of unity.[4]

We observe that these sample points do form plus-minus pairs. The reason is that

$$\omega^{n/2} = e^{(2\pi i/n)\cdot n/2} = e^{\pi i} = \cos(\pi) + i \cdot \sin(\pi) = -1,$$

the point an angle of π (i.e., halfway) around the unit circle. For $0 \leq j < n/2$ we get

$$x_{n/2+j} = \omega^{n/2+j} = \omega^{n/2} \cdot \omega^j = -\omega^j = -x_j,$$

the point exactly opposite $x_j = \omega^j$ on the unit circle.

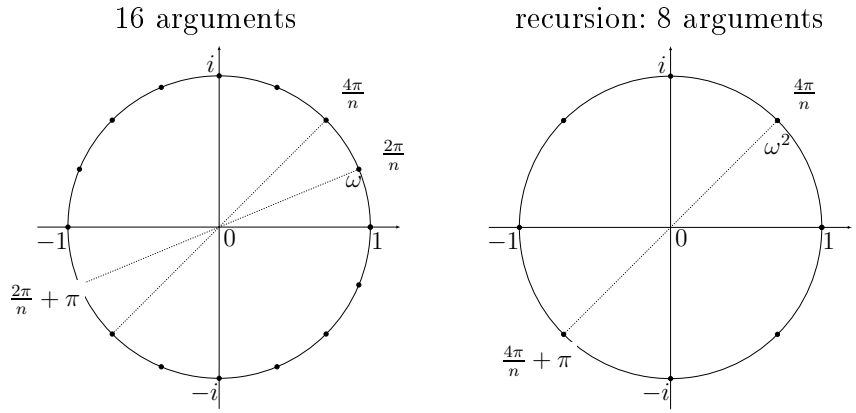
What else do we have to check? If we form the sequence $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$ of sample points in the splitting step, this should again be a sequence of roots of unity. Fortunately, this is the case. The point $\omega' = \omega^2 = e^{(2\pi i/n)\cdot 2} =$

[4] An nth root of unity z with the property that $z^0 = 1, z^1 = z, z^2, \ldots, z^{n-1}$ are the n different roots of unity is called a *primitive* nth root of unity. One can show that $z = \omega^j$ is a primitive nth root of unity if and only if j is odd.

$e^{2\pi i/(n/2)}$ is the $(n/2)$th root of unity we need to generate all the other ones, and

$$x'_j = x_j^2 = (\omega^j)^2 = (\omega^2)^j = \omega'^j$$

is its jth power, for $0 \le j < n/2$. So the shorter sequence of sample points formed in the splitting step is exactly of the type we need to get the recursion going. Note that it picks every second point from the sequence $(x_0, x_1, \ldots, x_{n-1})$.



Now everything fits. In each level of the recursion the vector of sample points is a sequence of roots of unity, which come in plus-minus pairs.

In a very streamlined version of the algorithm we carry along the coefficient vector and only the relevant primitive root of unity. The sample points used in the combining step are calculated only when they are needed. (In the book this algorithm is given in Fig. 2.9 on page 68.)

Algorithm 2.1 (FFT—Fast Fourier Transform)**.**
Input: $(a_0, \ldots, a_{n-1})$: coefficients; ω: primitive nth root of unity;
// Evaluate polynomial $A(x)$ with coefficients $(a_0, \ldots, a_{n-1})$
at points $1, \omega, \omega^2, \ldots, \omega^{n-1}$
Method: // Divide-and-Conquer
// Triviality test:
1 **if** $n = 1$ **then return** (a_0);
// Split in two pieces and use recursion:
2 $(s_0, \ldots, s_{n/2-1}) := \mathrm{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;
3 $(t_0, \ldots, t_{n/2-1}) := \mathrm{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;
// Combine results:
4 $y := 1$; // y runs through values $1, \omega, \omega^2, \ldots, \omega^{n/2-1}$
5 **for** j **from** 0 **by** 1 **to** $n/2 - 1$ **do**
6 $t := y \cdot t_j$;
7 $r_j := s_j + t$;
8 $r_{n/2+j} := s_j - t$;
9 $y := y \cdot \omega$;
10 **return** $(r_0, \ldots, r_{n-1})$ // value vector

From the pseudocode one sees that in the splitting and the combining steps only very simple operations are needed. Splitting is just picking out coefficients in a certain order; combining needs n multiplications (including the computation of the powers of ω) and n additions/subtractions.

Exercise: Given a coefficient vector $(a_0, \ldots, a_{n-1})$, show how to rearrange this sequence so that throughout the recursion the divide step is always just cutting the vector in the middle. (Example: If $n = 8$, the order should be $(a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)$.) Can you explain why this is called the "bit reversal ordering"?

2.6.3 Interpolation

Interpolation is the inverse of multiple evaluation. Given is a sequence $(r_0, r_1, \ldots, r_{n-1})$ of values a polynomial $A(x)$ takes at input points $x_0, x_1, \ldots, x_{n-1}$; we want to find the coefficient vector $(a_0, a_1, \ldots, a_{n-1})$ of $A(x)$.

Recall from Section 2.6.1 the relation between coefficient vectors and value vectors for sample points $x_0, \ldots, x_{n-1}$ in terms of the Vandermonde matrix

$$V(x_0, \ldots, x_{n-1}) = \begin{pmatrix} x_0^0 & x_0^1 & \ldots & x_0^{n-1} \\ x_1^0 & x_1^1 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1}^0 & x_{n-1}^1 & \ldots & x_{n-1}^{n-1} \end{pmatrix},$$

namely

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

As said in Section 2.6.1, the matrix $V(x_0, \ldots, x_{n-1})$ is regular. So for general sample points $x_0, \ldots, x_{n-1}$ evaluation is matrix-vector multiplication and interpolation amounts to solving a linear system. Matrix-vector multiplication in general takes $\Theta(n^2)$ operations, as does solving the linear system for this particular matrix. In numerical analysis, one gets to know formulas that solve the interpolation problem in time $O(n^2)$ without calculating the inverse matrix.

Luckily, for our special vector $(x_0, x_1, x_2, \ldots, x_{n-1}) = (1, \omega, \omega^2, \ldots, \omega^{n-1})$ for ω a primitive nth root of unity these things can be done faster. Let $M(\omega)$ be the Vandermonde matrix $V(1, \omega, \omega^2, \ldots, \omega^{n-1})$ of this special sequence. In general:

$$M(\omega) = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \ldots & \omega^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \ldots & \omega^{(n-1)(n-1)} \end{pmatrix},$$

the entry in row i, column j being $(\omega^i)^j = \omega^{ij}$. Since $(\omega^i)^j = (\omega^j)^i$, this is a symmetric matrix.

We get that in our special case $\mathrm{FFT}(a_0, \ldots, a_{n-1}) = (r_0, \ldots, r_{n-1})$ can be written as follows:

$$\begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix} = M(\omega) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}. \tag{2.10}$$

So FFT carries out matrix-vector multiplication *for this particular matrix* in time $O(n \log n)$.

We next note that the inverse $M(\omega)^{-1}$ of $M(\omega)$ can be written down immediately. For this, we consider $\bar{\omega} := \omega^{n-1} = \omega^{-1}$, which is both the multiplicative inverse and the complex conjugate of ω. Geometrically, $\bar{\omega}$ is the point in the complex plane reached when one starts at 1 and walks *clockwise* on the unit circle for an angle of $2\pi/n$. We consider the product $M(\omega) \cdot M(\bar{\omega})$. For the entry in row i, column j of this matrix there are two possibilities:

If $i = j$, then this entry is

$$\sum_{0 \le k < n} \omega^{ik} \cdot \bar{\omega}^{ki} = \sum_{0 \le k < n} \omega^{ik} \cdot \omega^{-ki} = \sum_{0 \le k < n} 1 = n.$$

If $i \neq j$, then this entry is

$$\sum_{0 \le k < n} \omega^{ik} \cdot \bar{\omega}^{kj} = \sum_{0 \le k < n} \omega^{ik} \cdot \omega^{-kj} = \sum_{0 \le k < n} (\omega^{i-j})^k = \frac{(\omega^{i-j})^n - 1}{\omega^{i-j} - 1} = 0.$$

(We used the formula for geometric series, and the fact that $\omega^{i-j} \neq 1$, but $(\omega^{i-j})^n = (\omega^n)^{i-j} = 1$.) So $M(\omega) \cdot M(\bar{\omega})$ is a diagonal matrix with n's on the diagonal.

This implies: $M(\omega)^{-1} = (1/n) \cdot M(\bar{\omega})$, and (2.10) turns into

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = (1/n) \cdot M(\bar{\omega}) \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}. \tag{2.11}$$

If we manage to multiply $M(\bar{\omega})$ fast with $(r_0, r_1, \ldots, r_{n-1})^\mathsf{T}$, we can solve interpolation fast. Now (2.10) says that FFT is just a way to carry out matrix-vector multiplication for the special matrix $M(\omega)$. If one checks what makes FFT work for this special matrix, one sees that one does not use more information about ω than the following:

(i) $\omega^n = 1$.

(ii) $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are all distinct.

Everything else follows from (i) and (ii): $\omega^{n/2} = -1$, and $\omega' = \omega^2$ has properties (i) and (ii) for parameter $n/2$. Conditions (i) and (ii) are also satisfied by $\bar{\omega}$. This means that a call $\mathrm{FFT}((r_0, r_1, \ldots, r_{n-1}), \bar{\omega})$ will calculate $M(\bar{\omega}) \cdot (r_0, r_1, \ldots, r_{n-1})^{\mathsf{T}}$, and we have

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \cdot \mathrm{FFT}((r_0, r_1, \ldots, r_{n-1}), \bar{\omega}).$$

So we can apply the FFT algorithm with a tiny modification for interpolation as well, and just as FFT with ω this takes time $O(n \log n)$. Using FFT for doing interpolation is **trick number three** needed for doing polynomial multiplication in time $O(n \log n)$.

Remark. If we now go back to Section 2.6.1, where we described polynomial multiplication by evaluation and interpolation, we find that we can carry out all steps in $O(n \log n)$ time. Step (1) (fixing $x_0, \ldots, x_{n-1}$) takes virtually no time. Steps (2) and (4) take time $O(n \log n)$. Step (3) (pointwise multiplication) takes linear time.

Theorem 2.1. *Two polynomials $A(x)$ and $B(x)$ of degree up to d, given by their coefficient vectors, can be multiplied in time $O(d \log d)$.*