# THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS

KARL L. STRATOS

ABSTRACT. If $n \in \mathbb{Z}$ denotes the size of input polynomials, the conventional way to compute their product takes the runtime of $\Theta(n^2)$. In this paper, we show how the Fast Fourier Transform, or FFT, can reduce this runtime to $\Theta(n \log n)$. We shall also investigate its applications in integer multiplication and signal processing.

## 1. Introduction

Throughout history, methods for fast computation have been developed in order to accomplish engineering tasks in a more efficient manner. Indeed, it has been a long progress that is too interesting to skip over. So before we start, as a warm-up let us try to comprehend the inner workings of some methods for trivial calculation.

Consider simple addition, which can be geographically interpreted as
*"If a and b are the lengths of two sticks, then if we place the sticks one after the other, the length of the stick thus formed will be a + b."*[4]

For adding two numbers, probably most if not all employ the technique:

(1) Align their right-hand ends.
(2) Compute from right to left a single addition digit by digit.
(3) Maintain the overflow as a carry.

For example, adding two integers 2135 and 632 we will have

$$\begin{array}{r} 1 \quad (carry) \\ 2535 \\ +\ 632 \\ ---\ \\ 3167 \end{array}$$

Such a systemetic procedure is called an *algorithm*. More formally, an **algorithm** is a method for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.

Now, why does this algorithm work? In order to answer the question, we need the following proposition:

**Proposition 1.1.** *In any base $b \geq 2$, where $b \in \mathbb{Z}$, the sum of any three single-digit numbers is at most two digits long.*

*Proof.* For base case $b = 2$, this is obviously true, for $1 + 1 + 1 = 11$ has two digits. Assume that, for some $b \in \mathbb{Z}_{\geq 2}$, the sum of three single-digit numbers in base $b$ is at most two digits long. This means that the sum of three $b - 1$ is less than or

equal to $(b-1) \cdot b^1 + (b-1) \cdot b^0$. Thus $3 \cdot (b-1) = 3b - 3 \leq b^2 - 1$, or $3b \leq b^2 + 2$. Now, consider the base $b+1$. The sum of numbers in this base can at most be $3b$. Denote this sum as $s$. By the inductive hypothesis,

$$s \leq 3b \leq b^2 + 2 \leq b^2 + b = b \cdot b^1 + b \cdot b^0$$

since $b \geq 2$. Hence $s$ is two digits long. □

Notice that whenever two individual numbers are added, their sum is at most two-digit long. *Hence the carry is always a single digit*! And, by the above proposition, the sum of the carry and two numbers is also at most two-digit long. This is why we can execute the usual addition algorithm without worrying if it might fail.[1]

How about multiplication? Just as in addition, most of us use the standard technique: To multiply two numbers $x$ and $y$, we create an array of intermediate sums, each representing the product of x by a single digit of y. These values are appropriately left-shifted and then added up. For example, 27 times 82 looks something like

$$\begin{array}{rl} 27 & \\ \times 82 & \\ --- & \\ 54 & (27 \times 2) \\ 216\phantom{0} & (27 \times 8) \\ ---- & \\ 2214 & (54 + 2160) \end{array}$$

But some people have come up with a different method: one that does not even require the multiplication table, only the ability to multiply and divide by 2, and to add. Ancient Egyptians seem to have used it, as shown in Rhind Mathematical Papyri written in the seventeenth century B.C. by the scribe Ahmes.[4] This is also known as *Russian Peasant Multiplication.* According to this method, to add two numbers $A$ and $B$,

(1) Align $A$ and $B$ side-by-side, each at the head of a column.
(2) Divide $A$ by 2, flooring the quotient, until there is nothing left to divide. Write the series of results under A.
(3) Keep doubling $B$ until you have doubled it as many times as you divided $A$. Write the series of results under B.
(4) Add up all the numbers in the B-column that are next to an odd number in the A-column.

Let us do the same example to check if it works. To multiply 27 times 82:

| | | |
|---|---|---|
| 27 | 82 | (+) |
| 13 | 164 | (+) |
| 6 | 328 | (*ignore*) |
| 3 | 656 | (+) |
| 1 | 1312 | (+) |
| − − − − − − − − − − − − − − − − − | | |
| | 2214 | (1312 + 656 + 164 + 82) |

So it works! Can you see why it works? Notice that we are basically converting the first number 27 from decimal to binary: 11011. Thus we actually had:

$$\begin{aligned} 82 \cdot 27 &= 82 \cdot ((1) \cdot 2^4 + (1) \cdot 2^3 + (0) \cdot 2^2 + (1) \cdot 2^1 + (1) \cdot 2^0) \\ &= 82 \cdot (16 + 8 + 2 + 1) \\ &= 1312 + 656 + 164 + 82 \\ &= 2214. \end{aligned}$$

The decomposition into a sum of powers of two was probably not intended as a change from base ten to base two; the Egyptians then were unaware of such concepts and had to resort to much simpler methods. The ancient Egyptians had laid out tables of a great number of powers of two so as not to be obliged to recalculate them each time. The decomposition of a number thus consists of finding the powers of two which make it up. The Egyptians knew empirically that a given power of two would only appear once in a number. For the decomposition, they proceeded methodically; they would initially find the largest power of two less than or equal to the number in question, subtract it out and repeat until nothing remained. (The Egyptians did not make use of the number zero in mathematics).[4]

An acute reader probably have noticed that this is a very natural construction to be written into a programming language. Another way to package this algorithm (to multiply two numbers $x, y \in \mathbb{Z}$) is

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{if y is even} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if y is odd.} \end{cases}$$

We are now ready to write a simple pseudo-code. A **pseudo-code** is a description of how a certain algorithm works that can be readily implemented on any computer language. Essentially, it is distills and collects the core working of the ideas. We will employ the following convention for writing a psuedo-code. The first bolded line indicates the name of the function (here, **MULTIPLY**). Next, its input and output are described (here, two numbers and their product). When calling a function, we say the name and put a parenthesis to the right to denote what the input values are (e.g. multiply(3,4) means call the multiply function with the input of 3 and 4). We write a **return** statement to indicate that the function has fulfilled its purpose and is returning the output value. Usually, the function terminates at this point. Given this explanation, we can write a pseudo-code for multiplying two integers using the above method as follows:

**MULTIPLY**$(x, y)$
INPUT: Two integers $x, y$, where $y \geq 0$
OUTPUT: their product
if ($y == 0$): return 0
$z = \text{MULTIPLY}(x, \lfloor y/2 \rfloor)$
if ($y$ is even):
return $2z$
else:
return $x + 2z$

As a final note, this method of *breaking a problem into smaller subproblems, recursively solving these subproblems, and accordingly combining their answers* to obtain the final result is called the **divide-and-conquer** strategy. The above **MULTIPLY** is, strictly speaking, not an example of the divide-and-conquer tactics since it does not divide the problem into subproblems, but rather just reduce it to smaller size. So it is more like a plain recursive algorithm. But we will be using the divide-and-conquer approach to deal with the FFT later.

## 2. Basic Definitions

Now that we have seen some instances of handling computational method, let us turn to the task at hand: operations on polynomials. But first, we need some basic definitions.

In order to be able to measure the efficiency of an algorithm independent of the platform (may that be a computer or your head) on which it is run, we need a way to describe how long it takes given some amount of input. In other words, we must define the *runtime* of the algorithm based on its *input size*, often denoted as $n$. We shall need the following notations to be more precise:

**Definition 2.1.** Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ ("big-oh of g") if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$. We say $f = \Omega(g)$ ("omega of g") if $g = O(f)$. If $f = O(g)$ and $f = \Omega(g)$, then we say $f = \Theta(g)$ ("theta of g").[1]

The functions of our interest will be of the form $f : S \mapsto T$, where $S$ and $T$ denote input size and time, respectively.

Notice that saying $f = \Theta(g)$ is a very loose analog of "$f = g$." These notations allow us to disregard constants and focus on the big picture. For example, let us consider two algorithms for a particular computational task. One takes $f_1(n) = n^2$ steps to complete it, while the other takes $f_2(n) = 2n + 20$ steps. We can declare that $f_2$ has a *better runtime* than $f_1$, or that $f_1$ *grows faster* than $f_2$, since $f_1 = O(f_2)$. That is, intuitively, regardless of what happens with a small input size, $f_2(n)$ eventually gives fewer steps than $f_1(n)$ as the input size $n$ grows to a large number. More concretely, although $f_1(n) \leq f_2(n)$ for $n < 6$, $f_2(n) \leq f_1(n)$ for all $n \geq 6$. We say the runtime is *linear* if it is $\Theta(n)$. We shall employ these platform-free notations rather loosely throughout the paper to compare the runtimes of various algorithms, since they are not the main subject in themselves.

In particular, the runtime of a divide-and-conquer algorithm can be captured by a *recurrence relation*. We write a recurrence relation for a recursive divide-and-conquer algorithm as follows ($T(n)$ means the time it takes to solve a problem of size $n$):

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

where $n$ is the size of the problem, $a$ is the number of subproblems resulting from each recursive call, $n/b$ is the size of each subproblem, and $O(n^d)$ is the time it takes to combine the answers $(a, b, d > 0)$.

We have a very convenient theorem called the *Master Theorem* that gives us right away a closed-form solution to this general recurrence.

**Theorem 2.2** (Mater Theorem). *If* $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *for some constants* $a > 0, b > 0,$ *and* $d \geq 0$*, then*

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

*Proof.* Let us assume that $n$ is a power of $b$ for convenience's sake. This will not influence the final bound in any important way since $n$ is at most a multiplicative factor of $b$ away from some power of $b$. The size of the subproblems decreases by a factor of $b$ with each level of recursion, so it reaches the base case after $\log_b n$ levels. The "branching" factor is $a$, so the $k$th level of the tree is made up of $a^k$ subproblems, each of size $n/b^k$. Therefore, the total work done at *this* ($k$th) level is

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k.$$

Notice that as $k$ goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio $a/b^d$. This ratio breaks down to three cases, each of which corresponds to the assertion of the Master Theorem:

(1) The ratio is less than 1. Then the series is decreasing, so its sum is just given by its first term, $O(n^d)$ (remember we're dealing with Big-O's).
(2) The ratio is greater than 1. Then the series is increasing, so its sum is given by the last term,

$$n^d (\frac{a}{b^d})^{log_b n} = n^d \frac{a^{log_b n}}{(b^{log_b n})^d} = a^{log_b n} = a^{(log_a n)(log_b a)} = n^{log_b a}.$$

(3) The ratio is equal to 1. Then all $O(\log n)$ terms of the series are equal to $O(n^d)$.[1]

□

We shall use this theorem to obtain the runtime of the FFT algorithm.

**Definition 2.3.** A **polynomial** in the variable x over an algebraic field $\mathbb{F}$ is a representation of a funnction $A(x)$ as a formal sum:[2]

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Note that the exponent $j$ must be non-negagive whole number. That means $x^3 - 5x + 1$ is a polynomial while $x^3 - \frac{5}{x} + 2x^{(1/2)}$ is not, since the latter contains a negative and a rational exponent.

We call the values $a_0, a_1, \ldots, a_{n-1}$ the **coefficients** of the polynomial. The coefficients are drawn from a field $\mathbb{F}$, typcially the set $\mathbb{C}$ of complex numbers.

A polynomial $A(x)$ has **degree** $k$ if its highest nonzero coefficient is $a_k$ (i.e. $k$ is the highest exponent). Any $n \in \mathbb{N}$ such that $n > k$ is called a **degree-bound** of that polynomial. For example, if a polynomial has a degree-bound 5, then its degree may be any integer between 0 and 4, inclusive.

We are trying to investigate an efficient way of performing operations on polynomials, so we define these operations rigorously. For **polynomial addition**, if $A(x)$ and $B(x)$ are polynomials of degree-bound $n$, we say that their **sum** is a polynomial $C(x)$, also of degree-bound $n$, such that

$$C(x) = A(x) + B(x)$$

for all $x \in \mathbb{F}$. [1] In other words, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,$$

where $c_j = a_j + b_j$ for all $j \in [0, n-1]$. For example, if we have the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^2 + 4x - 5$, we shall have $C(x) = 6x^3 + 5x^2 - 6x + 4$.

Likewise, we define **polynomial multiplication** as follows: if $A(x)$ and $B(x)$ are polynomials of degree-bound $n$, we say their **product** $C(x)$ is a polynomial of degree-bound $2n - 1$ such that

$$C(x) = A(x)B(x)$$

for all $x \in \mathbb{F}$. Another way to express the product $C(x)$ is to use the well-known *Cauchy Product*, which is a discrete convolution of two sequences (in this case, the coefficients of the two polynomials):

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j,$$

[1] Notice we are using the concept of degree-bound so that we are not limited only to the operations on polynomials of the same degree.

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}.$$

Thus degree($C$) = degree($A$) + degree($B$), which means if $A$ is a polynomial of degree-bound $n$ and $B$ is a polynomial of degree-bound $m$, then $C$ is a polynomial of degree-bound $n + m - 1$. For notational simplicity, because a polynomial of degree-bound $k - 1$ is also a polynomial of degre-bound $k$, we shall say that $C$ has a degree-bound $n + m$.

## 3. Statement of the Problem

Consider any two polynomials $A(x)$ and $B(x)$ of degree-bound $n$. Using the conventional method for multiplication, we can multiply $A$ and $B$ as follows:

(1) Align their right-hand ends.
(2) Multiply every term of $A$ by one term of $B$.
(3) Do *step*(2) for every term of $B$.
(4) Sum the results, adding coefficients of the same degree of $x$.

For example, if $A(x) = 2x^3 + x^2 + 3x + 1$ and $B(x) = 3x^3 + 6x + 1$, then we will compute:

$$\begin{aligned}
&1 \cdot 2x^3 + 1 \cdot x^2 + 1 \cdot 3x + 1 \cdot 1 \\
&6x \cdot 2x^3 + 6x \cdot x^2 + 6x \cdot 3x + 6x \cdot 1 \\
&0 \cdot 2x^3 + 0 \cdot x^2 + 0 \cdot 3x + 0 \cdot 1 \\
&3x^3 \cdot 2x^3 + 3x^3 \cdot x^2 + 3x^3 \cdot 3x + 3x^3 \cdot 1
\end{aligned}$$

Adding them together, we have our result

$$A(x) \cdot B(x) = 6x^6 + 3x^5 + 21x^4 + 11x^3 + 19x^2 + 9x + 1.$$

The final addition is quite easy (as to be shown shortly) and takes linear time. However, the multiplication part is more cumbersome, for we are multiplying two numbers $n$ times (*step*(2)), *and* we are doing this process again for $n$ times (*step*(3)). Because this procedure takes $n \cdot n = n^2$ steps, it clearly has the runtime of $\Theta(n^2)$, and so does the entire algorithm. To illustrate, the above example cost us $4^2 = 16$ multiplications.

Here, we are employing the usual coefficient representation of polynomials to do the operations.

**Definition 3.1.** A **coefficient representation** of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x_j$ of degree bound $n$ is a vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$.

This coefficient representation works efficiently in certain operations on polynomials. For instance, if we want to *evaluate* the polynomial $A(x)$ at a given point $x_0$, all we have to do is to plug in $x_0$ and compute each term. It boils down to the runtime of $\Theta(n)$ using **Horner's rule**:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\ldots))$$

(we are simply pulling the variable $x$ out so that we can add and multiply only once each time).

Also, if we want to *add* two polynomials represented by the coefficient vectors $a = (a_0, a_1, \dots, a_{n-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$ then we just produce the coefficent vector $c = (c_0, c_1, \dots, c_{n-1})$ by computing $c_j = a_j + b_j$ for $j \in [0, n-1]$. This is clearly a linear-time procedure, as asserted above.

We have already seen that the multiplication of two degree-bound $n$ polynomials $A(x)$ and $B(x)$ does not work quite as satisfactorily in coefficient representation, since its runtime is $\Theta(n^2)$. Calculating the convolution of the input vectors $a$ and $b$ for each resulting vector $c$ seems to be a lot of work. This is why we might wish to look at a different way of expressing polynomials.

## 4. Solution

Our game plan of efficiently multiplying two polynomials $A$ and $B$ is now:

(1) Convert $A$ and $B$ from the disappointing coefficient representation to some better form F. ($\Theta(n \log n)$)
(2) Carry out the multiplication while $A$ and $B$ are in the form F to obtain their product (in the form F). ($\Theta(n)$)
(3) Convert this product from the form F back to the coefficient representation as our answer. ($\Theta(n \log n)$)

Therefore, if this plan succeeds, we will have achieved a method of polynomial multiplication that runs in time $\Theta(n \log n)$.
It turns out that the "better form" in step(1) is something called the *point-value representation*.

**Definition 4.1.** A **point-value representation** of a polynomial $A(x)$ of degree-bound $n$ is a set of $n$ **point-value pairs**

$$(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$$

such that all of the $x_k$ are distinct and

$$y_k = A(x_k)$$

for $k = 0, 1, \cdots, n-1$.[2]

Note that a polynomial has many different point-value representations, since any set of $n$ distinct points $x_0, x_1, \cdots, x_{n-1}$ can be used as a basis for the representation.

Given a polynomial represented in coefficient form, it is straightforward to compute its point-value representation (step(1)). All we have to do is select any $n$ distict points $x_0, x_1, \cdots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \cdots, n-1$. With Horner's method, this $n$-point evalution takes $\Theta(n^2)$ (do you see why?), and will ruin our whole plan. However, we shall see later that if we choose the $x_k$ wisely, we can reduce this runtime to $\Theta(n \log n)$.

Also, the inverse of evaluation - determining the coefficient form from a point-value representation (step(3)) - is called **interpolation**. How do we know that the coefficient form that we derive from a point-value form represents a unique polynomial? Well, the intuitive approach is that when you have two distinct points on a plane, then *there is only one line that you can draw in such a way that it goes*

*through both of the points.* We will generalize this idea, and prove that interpolation is well defined when the desired interpolating polynomial has a degree-bound equal to the given number of point-values pairs.

**Theorem 4.2** (Uniqueness of an interpolating polynomial)**.** *For any set $(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})$ of $n$ point-value pairs such that all the $x_k$ values are distinct, there is a unique polynomial $A(x)$ of degree-bound $n$ such that*

$$y_k = A(x_k) \tag{1}$$

*for $k = 0, 1, \cdots, n-1$.*

The proof of this theorem depends on the invertibility of a certain matrix, thus we will need some definitions and theorems.

**Definition 4.3.** A **determinant** is a special number associated to any square matrix. More precisely, the determinant of an $n \times n$ matrix $A$ having entries from a field $\mathbb{F}$ is a scalar in $\mathbb{F}$, denoted by $\det(A)$ or $|A|$, and can be computed in the following manner:

1. If A is $1 \times 1$, then $\det(A) = A_{11}$, the single entry of A.
2. If A is $n \times n$ for $n > 1$, then

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} A_{ij} \cdot \det(\tilde{A}_{ij}) \tag{2}$$

where $\tilde{A}_{ij}$ denotes the $(n-1) \times (n-1)$ matrix obtained from $A$ by deleting row $i$ and column $j$.[5]

We also define the **cofactor** $c_{ij}$ of the entry of $A$ in row $i$, column $j$, by

$$c_{ij} = (-1)^{i+j} \det(\tilde{A}_{ij}).$$

Then we can express the formula for the determinant of $A$ as

$$\det(A) = A_{i1}c_{i1} + A_{i2}c_{i2} + ... + A_{in}cin$$

for any $1 \leq i \leq n$.

A square matrix $A$ of size $n \times n$ is said to be **invertible** if there exists an $n \times n$ matrix $A^{-1}$ such that $AA^{-1} = A^{-1}A = I$, where $I$ denotes the identity matrix whose entries are defined by $I_{ij} = 1$ if $i = j$ and $I_{ij} = 0$ if $i \neq j$.

**Theorem 4.4.** *If a square matrix $A$ has an inverse $B$, then this inverse is unique.*

*Proof.* Suppose $C$ is another inverse of the matrix $A$. Then we have $AB = BA = I$ and also $AC = CA = I$. But $C = IC = BAC = BI = B$. □

Now, we show that a matrix is invertible if and only if its determinant is nonzero.

**Theorem 4.5.** *An $n \times n$ matrix* $A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix}$.

*is invertible if and only if* $\det(A) \neq 0$.

*Sketch Proof.* We note from the determinant formula (2) that whenever $i+j$ is even, the term is positive, and whenever $i+j$ is odd, the term is negative. It is a matter of simple calculation to show

$$A_{i1}c_{j1} + A_{i2}c_{j2} + ... + A_{in}c_{jn} = 0$$

for $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$. (It is the determinant of the matrix you get when in matrix $A$ you replace row $i$ by row $j$. The matrix then has two equal rows, so its determinant must be 0.) Now, we claim that the inverse of $A$ is
$A^{-1} = \begin{bmatrix} \frac{c_{11}}{\mathtt{det}(A)} & \frac{c_{21}}{\mathtt{det}(A)} & \cdots & \frac{c_{n1}}{\mathtt{det}(A)} \\ \frac{c_{12}}{\mathtt{det}(A)} & \frac{c_{22}}{\mathtt{det}(A)} & \cdots & \frac{c_{n2}}{\mathtt{det}(A)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{c_{1n}}{\mathtt{det}(A)} & \frac{c_{2n}}{\mathtt{det}(A)} & \cdots & \frac{c_{nn}}{\mathtt{det}(A)} \end{bmatrix}$.
This is also a matter of checking if $AA^{-1} = A^{-1}A = I$, which is left to the reader. This computation of the inverse of A clearly can't work when $\det(A) = 0$. So we have the desired result.
The fact that a matrix with determinant 0 really can't have an inverse can be seen from the following: *We know that, when det(A) = 0, then row n is a linear combination of rows* $1, 2, 3, \cdots, n-1$. Now suppose that matrix $B$ is the inverse of matrix $A$, so $AB = I$. Note that the $n$th row of $AB$ is now a linear combination of the rows $1, 2, 3, ..., n-1$ of $AB$ too. The elements in the $n$th column of rows $1, 2, 3, ..., n-1$ of $I$ are all zero, so that means that the $n$th element of the $n$th row of $AB$ should be zero, too, being a linear combination of zeroes. But the $n$th element of the $n$th row of $I$ is one, and we have a contradiction.[6] □

**Definition 4.6.** A **Vandermonde matrix** a matrix with the terms of a geometric progression in each row ($V_{i,j} = \alpha_i^{j-1}$ for all indices $i$ and $j$).[4] In other words, it is an $m$ by $n$ matrix
$V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \dots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{bmatrix}$.

This particular type of matrix has a special property.

**Theorem 4.7.** *An $n$ by $n$ Vandermonde matrix $V$ has the following determinant:*

$$\det(V) = \prod_{1 \leq i < j \leq n} (a_j - a_i). \quad (3)$$

*Proof.* For $n = 2$ the determinant of the Vandermonde matrix is

$$\det \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} = x_2 - x_1.$$

So the property holds.
Let

$$V_n = \begin{vmatrix} a_1^{n-1} & a_1^{n-2} & \cdots & a_1 & 1 \\ a_2^{n-1} & a_2^{n-2} & \cdots & a_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_n^{n-1} & a_n^{n-2} & \cdots & a_n & 1 \end{vmatrix}$$

(It's written that way round to make the proof come out easier).
For all $n \in \mathbb{N}$, let $P(n)$ be the proposition that $V_n = \prod_{1 \leqslant i < j \leqslant n} (a_i - a_j)$. We have already proved the basis with the 2 by 2 matrix. Now we need to show that, if $P(k)$ is true, where $k \geqslant 2$, then it follows that $P(k+1)$ is true. So this is our induction hypothesis:

$$V_k = \prod_{1 \leqslant i < j \leqslant k} (a_i - a_j).$$

Take the determinant $V_{k+1} = \begin{vmatrix} x^k & x^{k-1} & \cdots & x^2 & x & 1 \\ a_2^k & a_2^{k-1} & \cdots & a_2^2 & a_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{k+1}^k & a_{k+1}^{k-1} & \cdots & a_{k+1}^2 & a_{k+1} & 1 \end{vmatrix}$.

If you expand it in terms of the first row, you can see it is a polynomial in $x$ whose degree is no greater than $k$. Call that polynomial $f(x)$. If you substitute any $a_r$ for $x$ in the determinant, two of its rows will be the same. So the value of such a determinant will be 0. Such a substitution in the determinant is equivalent to substituting $a_r$ for $x$ in $f(x)$. Thus it follows that $f(a_2) = f(a_3) = \ldots = f(a_{k+1}) = 0$ as well. So $f(x)$ is divisible by each of the factors $x - a_2, x - a_3, \ldots, x - a_{k+1}$. All these factors are distinct otherwise the original determinant is zero. So $f(x) = C(x - a_2)(x - a_3) \cdots (x - a_k)(x - a_{k+1})$. As the degree of $f(x)$ is no greater than $k$, it follows that $C$ is independent of $x$. From expansion, we can see that the coefficient of $x^k$ is

$$\begin{vmatrix} a_2^{k-1} & \cdots & a_2^2 & a_2 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{k+1}^{k-1} & \cdots & a_{k+1}^2 & a_{k+1} & 1 \end{vmatrix}.$$

By the induction hypothesis, this is equal to $\prod_{2 \leqslant i < j \leqslant k+1} (a_i - a_j)$. So this has to be our value of $C$. So we have $f(x) = C(x - a_2)(x - a_3) \cdots (x - a_k)(x - a_{k+1})$ $\prod_{2 \leqslant i < j \leqslant k+1} (a_i - a_j)$. Substituting $a_1$ for $x$, we retrieve the proposition $P(k+1)$. So $P(k) \implies P(k+1)$. Therefore $V_n = \prod_{1 \leqslant i < j \leqslant n} (a_i - a_j)$. Or, in our original formulation, this is equivalent to

$$\det(V) = \prod_{1 \leq i < j \leq n} (a_j - a_i).[7]$$

□

Finally, let us turn to the proof of the first theorem.

*Proof of the Uniqueness of an interpolating polynomial.* Note that the equation (1) is equivalent to the matrix equation

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

We also note that the left matrix is a Vandermonde matrix, and denote it by $V(x_0, x_1, \cdots, x_{n-1})$. By **Theorem 4.7**, this matrix has determinant

$\prod_{1 \leq j < k \leq n} (a_k - a_j)$,

and therefore, the determinant is nonzero if the $x_k$ are distinct. Moreover, in that case, the matrix is invertible by **Theorem 4.5**, and the inverse is unique by **Theorem 4.4**. Thus, the coefficients $a_j$ can be uniquely solved for, given the point-value representation:

$$a = V(x_0, x_1, \cdots, x_{n-1})^{-1} y \tag{4}$$

or

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

□

Thanks to the uniqueness of an interpolating polynomial, we do not have to worry about getting our polynomial back to its orinial form in coefficient representation when we turn it into point-value representation.

The point-value representation allows us to multiply polynomials in time $\Theta(n)$, much less than the time required to multiply polynomials in coefficient form. To see this, let $C(x) = A(x)B(x)$. Then $C(x_k) = A(x_k)B(x_k)$ for any point $x_k$, and we can pointwise multiply a point-value representation for $A$ by a point-value representation for $B$ to obtain a point-value representation for C. There is an annoying restrction, however, that degree$(C)$ = degree$(A)$ + degree$(B)$; if $A$ and $B$ are of degree-bound $n$, then $C$ is of degree-bound $2n$. Since $A$ and $B$ will give us $n$ point-value pairs when multiplied, and since we need $2n$ pairs to interpolate a unique polynomial $C$ of degree-bound $2n$, we must *begin* with "extended" point-value representations for $A$ and $B$ consisting of $2n$ point-value pairs each. More concretely, given an extended point-value represenation for $A$

$$(x_0, y_0), (x_1, y_1), \cdots, (x_{2n-1}, y_{2n-1}),$$

and a corresponding extended point-value representation for $B$

$$(x_0, y'_0), (x_1, y'_1), \cdots, (x_{2n-1}, y'_{2n-1})$$

then we can get a point-value representation for $C$ as

$$(x_0, y_0 y'_0), (x_1, y_1 y'_1), \cdots, (x_{2n-1}, y_{2n-1} y'_{2n-1}).$$

We know that once we have our polynomials in point-value form, multiplying them is a piece of cake. So now the key to our task is our ability to convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice-versa

(interpolate).

Recall that evaluation takes the runtime of $\Theta(n^2)$ if we simple-mindedly choose $n$ different points to evaluate a polynomial of degree-bound $n$. But we shall show that if we choose "complex roots of unity" as the evaluation points, we can produce a point-value representation by taking the Discrete Fourier Transform (DFT - to be defined later) of a coefficient vector, reducing the runtime to $\Theta(n \log n)$. The inverse operation, interpolation, can be performed by taking the "inverse DFT" of point-value pairs, yielding a coefficient vector.

**The Discrete Fourier Transform and Fast Fourier Transform**

First, we need a definition of complex roots of unity. And we also need to clarify a series of their properties to exploit.

**Definition 4.8.** A **complex nth root of unity** is a complex number $w$ such that

$$w^n = 1.$$

To visualize the roots, we shall use the **Euler's formula**:

**Theorem 4.9** (Euler's formula)**.** *For any $x \in \mathbb{R}$,*

$$e^i x = cos(x) + isin(x).$$

Since this is a very elegant and insightful statement, we shall take a leisurely detour and prove it in two different ways: one using Taylor series and the other calculus.

*First proof.* We express the functions $e^x$, $cosx$, and $sinx$ as Taylor expansions around zero:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \\ cosx &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \\ sinx &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots . \end{aligned}$$

We can replace the real variable $x$ with $u \in \mathbb{C}$ since each series has an infinite radius of convergence. Furthermore, since each series is absolutely convergent, the rearrangement of terms does not change the limit value:

$$\begin{aligned} e^{iu} &= 1 + iu + \frac{(iu)^2}{2!} + \frac{(iu)^3}{3!} + \frac{(iu)^4}{4!} + \frac{(iu)^5}{5!} + \frac{(iu)^6}{6!} + \frac{(iu)^7}{7!} + \frac{(iu)^8}{8!} + \cdots \\ &= 1 + iu - \frac{u^2}{2!} - \frac{iu^3}{3!} + \frac{u^4}{4!} + \frac{iu^5}{5!} - \frac{u^6}{6!} - \frac{iu^7}{7!} + \frac{u^8}{8!} + \cdots \\ &= \left(1 - \frac{u^2}{2!} + \frac{u^4}{4!} - \frac{u^6}{6!} + \frac{u^8}{8!} - \cdots\right) + i\left(u - \frac{u^3}{3!} + \frac{u^5}{5!} - \frac{u^7}{7!} + \cdots\right) \\ &= \cos u + i \sin u \end{aligned}$$

□

*Second proof.* Define a function $f(x)$ as

$$f(x) = (\cos x + i \sin x) \cdot e^{-ix}.$$

By the product rule, the derivative of $f$ is given by:

$$\begin{aligned}
\frac{d}{dx}f(x) &= (\cos x + i\sin x) \cdot \frac{d}{dx}e^{-ix} + \frac{d}{dx}(\cos x + i \sin x) \cdot e^{-ix} \\
&= (\cos x + i \sin x)(-ie^{-ix}) + (-\sin x + i \cos x) \cdot e^{-ix} \\
&= (-i \cos x - i^2 \sin x) \cdot e^{-ix} + (-\sin x + i\cos x) \cdot e^{-ix} && (i^2 = -1) \\
&= (-i\cos x + \sin x - \sin x + i \cos x) \cdot e^{-ix} \\
&= 0.
\end{aligned}$$

This means that (x) is a constant function. Thus

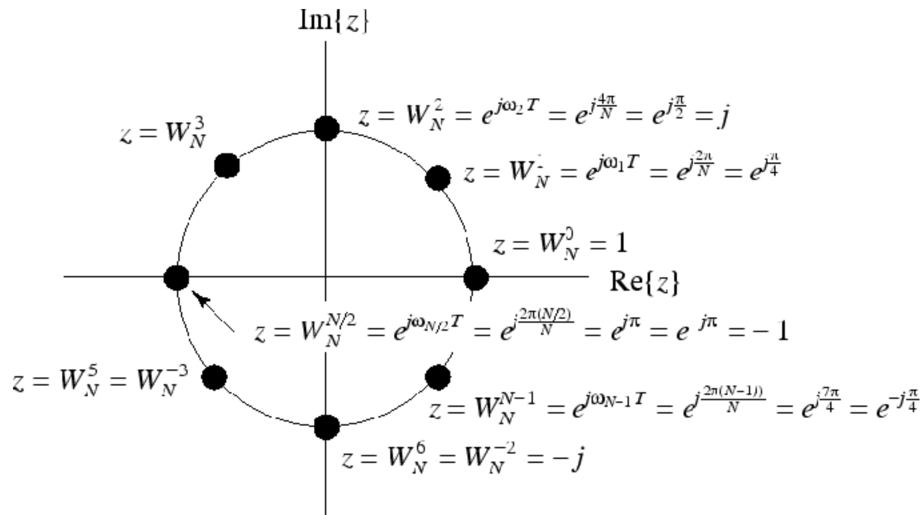
$$(\cos x + i \sin x) \cdot e^{-ix} = f(x) = f(0) = (\cos 0 + i \sin 0) \cdot e^0 = 1.$$

Multiplying both sides by $e^{ix}$, we obtain

$$\cos x + i \sin x \; = e^{ix}.$$

□

There are exactly $n$ complex $n$th roots of unity: $e^{\frac{2\pi ik}{n}}$ for $k = 0, 1, \cdots, n-1$. Euler's formula allows us to visualize the roots of unity on a complex plane. For instance, the following figure displays the 8th roots of unity




Several things to notice here. First, the $n$ roots are equally spaced around the circle of unit radius centered at the origin of the complex plane. The value

$$w_n = e^{\frac{2\pi i}{n}}$$

is called the **principal nth root of unity** (note that all other roots of unity are powers of $w_n$). So in the figure above, $w_8 = e^{\frac{2\pi i}{8}}$ is the principal 8th root of unity. Second, when $n$ is even, there will be a point at $z = -1$, while if $n$ is odd, there is no point at $z = -1$. Third, the $n$ complex $n$th roots of unity

$$w_n^0, w_n^1, \cdots, w_n^{n-1},$$

form a group under multiplication. This group has the same structure as the additive group $(\mathbb{Z}_n, +)$ modulo $n$, since $w_n^n = w_n^0 = 1$ implies that $w_n^j w_n^k = w_n^{j+k} = w_n^{(j+k) \bmod n}$. Similarly, $w_n^{-1} = w_n^{n-1}$.

The following lemmas will help us understand the essential properties of the complex $n$th roots of unity.

**Lemma 4.10** (Cancellation lemma)**.** *For any integers $n \geq 0$, $k \geq 0$, and $d \geq 0$,*

$$w_{dn}^{dk} = w_n^k.$$

*Proof.* This lemma follows directly from the definition of the roots since

$$\begin{aligned} w_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= w_n^k. \end{aligned}$$

□

**Corollary 4.11.** *For any even integer $n > 0$,*

$$w_n^{n/2} = w_2 = -1.$$

*Proof.*

$$\begin{aligned} w_n^{n/2} &= w_1^{1/2} \\ &= w_2 \\ &= e^{\pi i} \\ &= cos(\pi) + isin(\pi) \\ &= -1. \end{aligned}$$

□

**Lemma 4.12** (Halving lemma)**.** *If $n > 0$ is even, then the squares of the $n$ complex $n$th roots of unity are the $n/2$ complex $(n/2)$th roots of unity.*

*Proof.* By the canellation lemma, we have $(w_n^k)^2 = w_{n/2}^k$, for any nonnegative integer $k$. Note that if we square all of the $n$th roots of unity, then each $(n/2)$th root of unity is obtained exactly twice, since

$$\begin{aligned} (w_n^{k+n/2})^2 &= w_n^{2k+n} \\ &= w_n^{2k} w_n^n \\ &= w_n^{2k} \\ &= (w_n^k)^2. \end{aligned}$$

Thus, $w_n^k$ and $w_n^{k+n/2}$ have the same square. □

This halving lemma is crucial to our divide-and-conquer approach for converting between coefficient and point-value representations of polynomials, since it guarantees that the recursive subproblems are only half as large.

**Lemma 4.13** (Summation lemma). *For any integer $n \geq 1$ and nonzero integer $k$ not divisible by $n$,*

$$\sum_{j=0}^{n-1} (w_n^k)^j = 0.$$

*Proof.* The closed form of summation applies to complex values as well as to reals, and thus we have

$$\begin{aligned}\sum_{j=0}^{n-1} (w_n^k)^j &= \frac{(w_n^k)^n - 1}{w_n^k - 1} \\ &= \frac{(w_n^n)^k - 1}{w_n^k - 1} \\ &= \frac{(1)^k - 1}{w_n^k - 1} \\ &= 0\end{aligned}$$

The restriction on $k$ that it is not divisible by $n$ makes sure that the denominator is never 0, since $w_n^k = 1$ only when $k$ is divisible by $n$. □

**The DFT**

Our job is to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound $n$ at $n$ different points. We shall use the $n$ complex $n$th roots of unity

$$w_n^0, w_n^1, w_n^2, \cdots, w_n^{n-1}$$

as our evaluation points.

One minor detail to remember: this length $n$ is actually $2n$ (as discussed) since we double the degree-bound of the given polynomials prior to evaluation. Recall that the product of two polynomials of degree-bound $n$ is a polynomial of degree-bound $2n$ so that we must double their degree-bounds to $2n$. This is easy, since all we have to do is to add $n$ high-order coefficients of 0. Therefore, in fact we are working with complex $(2n)$th roots of unity. But for simplicity, let us indulge the rather erroneous notation of $n$. Furthermore, let us assume that $n$ is a power of 2 (why not, we can always add new high-order coefficients of 0 as necessary).

We assume that $A$ is given in coefficient form:

$$a = (a_0, a_1, \cdots, a_{n-1}).$$

**Definition 4.14.** Let us denote the results $y_k$, for $k = 0, 1, \cdots, n-1$, by

$$\begin{aligned} y_k &= A(w_n^k) \\ &= \sum_{j=0}^{n-1} a_j w_n^{kj}. \end{aligned}$$

Then the vector $y = (y_0, y_1, \cdots, y_{n-1})$ is the **Discrete Fourier Transform (DFT)** of the coefficient vector $a = (a_0, a_1, \cdots, a_{n-1})$. We write $y = \mathrm{DFT}_n(a)$ to inform the length.

**The FFT**

The **Fast Fourier Transform (FFT)** is a method that takes advantage of the special properties of the complex roots of unity to compute $\mathrm{DFT}_n(a)$ in time $\Theta(nlogn)$, as opposed to the $\Theta(n^2)$ time of the straightforward method.

The FFT method employs a divide-and-conquer strategy. The motivation is from the halving lemma, which implies that, for even $n$, we might need only *half* of the points for evaluation.

Here is the strategy. Use the even-index and odd-index coefficients of $A(x)$ separately to define the *two new polynomials* $A'(x)$ and $A''(x)$ of degree-bound $n/2$:

$$\begin{aligned} A'(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}, \\ A''(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}. \end{aligned}$$

Note that $A'$ contains all the even-index coefficients of $A$ (the binary representation of the index ends in 0) and $A''$ contains all the odd-index coefficients (the binary representation of the index ends in 1). You should check the following fact:

$$A(x) = A'(x^2) + xA''(x^2). \tag{5}$$

Now, the problem of evaluating $A(x)$ at $w_n^0, w_n^1, \cdots, w_n^{n-1}$ reduces to the following divide-and-conquer scheme

(1) Evaluate the degree-bound $n/2$ polynomials $A'(x)$ and $A''(x)$ at the points

$$(w_n^0)^2, (w_n^1)^2, \cdots, (w_n^{n-1})^2,$$

(2) Combine the results according to equation (5).

Now that we have the square forms of the complex roots, can you see how the halving lemma comes in? The $n$ values in step (1)

$$(w_n^0)^2, (w_n^1)^2, \cdots, (w_n^{n-1})^2$$

are nothing but $n/2$ distinct values of the complex $(n/2)$th roots of unity, with each root occuring exacly twice

$$(w_{n/2}^0), (w_{n/2}^1), \cdots, (w_{n/2}^{n/2-1})$$

Therefore, the polynomials $A'$ and $A''$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity. The recursion is possible because the subproblems have exactly the same form as the original problem, but only are half the size. i.e. We have successfully decomposed an $n$-element $\text{DFT}_n$ computation into two $n/2$-element $\text{DFT}_{n/2}$ computations. Note that our requirement for n to be a power of 2 guarantees a smooth termination of the algorithm (because we're dividing in half). We are now ready to write an algorithm. We will employ the same convention that we used for the **MULTIPLY** pseudo-code; that is, the first line **RECURSIVE-FFT** denotes the name of the function, the input and output are explained next, and **return** statements are used to signify the end of the current fuction. We will investigate how the code works line by line.

**RECURSIVE-FFT**$(a)$
INPUT: $A(x)$ in the coefficient vector $a = (a_0, a_1, \cdots, a_{n-1})$
OUTPUT: the DFT of $a$
1. $n \leftarrow length[a]$
2. **if** $(n = 1)$
3. **then return** $a$ $\nabla$*base case*
4. $w_n \leftarrow e^{2\pi i/n}$
5. $w \leftarrow 1$
6. $a' \leftarrow (a_0, a_2, \cdots, a_{n-2})$
7. $a'' \leftarrow (a_1, a_3, \cdots, a_{n-1})$
8. $y' \leftarrow$ **RECURSIVE-FFT**$(a')$
9. $y'' \leftarrow$ **RECURSIVE-FFT**$(a'')$
10. **for**$k \leftarrow 0$ **to** $n/2 - 1$
11. $y_k \leftarrow y'_k + wy''_k$
12. $y_{k+(n/2)} \leftarrow y'_k - wy''_k$
13. $w \leftarrow ww_n$
14. **return** $y$

Let us comprehend how each line works. Lines 2-3 represent the basis of the recursion. To see why it is so, consider the DFT of one element

$$\begin{aligned} y_0 &= a_0 w_1^0 \\ &= a_0 \cdot 1 \\ &= a_0, \end{aligned}$$

which is the original element itself.

Lines 6-7 define the coefficient vectors for the polynomials $A'$ and $A''$. Lines 4, 5, and 13 guarantee that $w$ is updated properly so that whenever lines 12-13 are executed, we have $w = w_n^k$.

Lines 8-9 perform the two recursive $\text{DFT}_{n/2}$ computations and set, for $k = 0, 1, \cdots, n/2 - 1$,

$$\begin{aligned} y'_k &= a'(w_{n/2}^k), \\ y''_k &= a''(w_{n/2}^k), \end{aligned}$$

By the cancellation lemma (since $w_{n/2}^k = w_n^2 k$), this is equivalent to

$$\begin{aligned} y'_k &= a'(w_n^2 k), \\ y''_k &= a''(w_n^2 k), \end{aligned}$$

Lines 12-13 combine the results of the recursive $\mathrm{DFT}_{n/2}$ calculations. For $y_0, y_1, \cdots, y_{n/2-1}$, line 11 yields

$$\begin{aligned} y_k &= y'_k + w_n^k y''_k \\ &= a'(w_n^2 k) + w_n^k a'(w_n^2 k) \\ &= a(w_n^k) \end{aligned}$$

For $y_{n/2}, y_{n/2+1}, \cdots, y_n$ (letting $k = 0, 1, \cdots, n/2 - 1$), line 12 yields

$$\begin{aligned} y_{k+(n/2)} &= y'_k - w_n^k y''_k \\ &= y'_k + w_n^{k+(n/2)} y''_k && \text{(since } w_n^{k+(n/2)} = -w_n^k) \\ &= a'(w_n^2 k) + w_n^{k+(n/2)} a'(w_n^2 k) \\ &= a'(w_n^{2k+n}) + w_n^{k+(n/2)} a'(w_n^2 k) && \text{(since } w_n^{2k+n} = w_n^2 k) \\ &= a(w_n^{k+(n/2)}) \end{aligned}$$

Therefore, the vector $y$ returned by **RECURSIVE-FFT** is indeed the DFT of the input vector $a$!

Now, let us verify the runtime of this algorithm. Since each recursive call creates 2 subproblems of size $n/2$, and the time to combine the results is $\Theta(n)$, where $n$ is the length of the input vector, we have the recurrence $T(n) = 2T(n/2) + \Theta(n)$. By the Master Theorem (since $1 = \log_2 2$), the closed form of this relation is

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n). \end{aligned}$$

Thus we have achieved our first goal: we can evaluate a polynomial of degree-bound $n$ at the complex $n$th roots of unity in time $\Theta(n \log n)$ using the Fast Fourier Transform. The only missing link now is to quickly recover the coefficient vector from the DFT.

**Interpolation at the complex roots of unity**

We shall interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

Recall the matrix representation of the formula $y = A(x)$

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

From this, we can write the DFT as the matrix product $y = V_n a$, where $V_n$ is a Vandermonde matrix containing the appropriate powers of $w_n$:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & w_n^3 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & w_n^6 & \dots & w_n^{2(n-1)} \\ 1 & w_n^3 & w_n^6 & w_n^9 & \dots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \dots & w_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

The $(k, j)$ entry of $V_n$ is $w_n^{kj}$, for $j, k = 0, 1, \cdots, n-1$, and the exponenets of the entries of $V_n$ form a multiplication table.

For the inverse operation, which we write as $a = \text{DFT}_n^{-1}(y)$, we multiply $y$ by the matrix $V_n^{-1}$, the inverse of $V_n$ (which we know exists from previous exercises).

**Theorem 4.15.** *For $j, k = 0, 1, \cdots, n-1$, the $(j, k)$ entry of $V_n^{-1}$ is $w_n^{-kj}/n$.*

*Proof.* We shall show that $V_n^{-1} V_n = I_n$. Consider the $(j, j')$ entry of $V_n^{-1} V_n$:

$$\begin{aligned} [V_n^{-1} V_n]_{jj'} &= \sum_{k=0}^{n-1} (w_n^{-kj}/n)(w_n^{kj'}) \\ &= \sum_{k=0}^{n-1} w_n^{k(j'-j)}/n. \end{aligned}$$

This summation equals 1 if $j' = j$, and it is 0 otherwise by the summation lemma. So the resulting matrix of $V_n^{-1} V_n$ will be precisely equal to the identity matrix. □

Given the inverse matrix $V_n^{-1}$, $\text{DFT}_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k w_n^{-kj}$$

for $j = 0, 1, \cdots, n-1$.

Compare this to the equation that we used for evaluation:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj} \text{for } k = 0, 1, \cdots, n-1.$$

Thus we can simply modify the FFT algorithm to compute the inverse DFT: just switch the roles of $a$ and $y$, replace $w_n$ by $w_n^{-1}$, and divide each element of the result by $n$.

**RECURSIVE-INVERSE-FFT**$(y)$
INPUT: $A(x)$ in the DFT vector form $y = (y_0, y_1, \cdots, y_{n-1})$
OUTPUT: the coefficient vector form $a$
$n \leftarrow length[y]$
**if** $(n = 1)$
**then return** $y$ $\nabla$*base case*
$w_n^{-1} \leftarrow e^{2\pi i/n}$
$w \leftarrow 1$
$y' \leftarrow (y_0, y_2, \cdots, y_{n-2})$
$y'' \leftarrow (y_1, y_3, \cdots, y_{n-1})$
$a' \leftarrow$ **RECURSIVE-INVERSE-FFT**$(y')$
$a'' \leftarrow$ **RECURSIVE-INVERSE-FFT**$(y'')$
**for**$k \leftarrow 0$ **to** $n/2 - 1$
$a_k \leftarrow \frac{1}{n}(a'_k + wa''_k)$
$a_{k+(n/2)} \leftarrow \frac{1}{n}(a'_k - wa''_k)$
$w \leftarrow ww_n^{-1}$
**return** $a$

Since this is structurally the same as the original FFT algorithm, it also has the runtime of $\Theta(n \log n)$.

**Summary**

We have accomplished our task of multiplying polynomials in time $\Theta(n \log n)$. Remember our game plan to multiply polynomials $A$ and $B$?

(1) Convert $A$ and $B$ from the disappointing coefficient representation to some better form F.
(2) Carry out the multiplication while $A$ and $B$ are in the form F to obtain their product (in the form F).
(3) Convert this product from the form F back to the coefficient representation as our answer.

Step(1), converting the coefficient representation to the point-value form, can be done in $\Theta(n \log n)$ thanks to the special properties of the complex $n$th roots of unity.

Step(2), multiplication, can be easily done by multiplying the value of $A$ and $B$ at $2n$ points. This is a $\Theta(n)$ operation.

Step(3), converting the point-value form back to the coefficient representation, can be done in $\Theta(n \log n)$ by using a similar algorithm in Step(1).

Therefore, the runtime of the entire process is $\Theta(n \log n)$, and we have broken the limitation of $\Theta(n^2)$ runtime in the conventional method. In the context of polynomial multiplication, we have shown the following.

**Theorem 4.16.** *For any two vectors $a$ and $b$ of length $n$, where $n$ is a power of 2,*

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)),$$

*where the vectors $a$ and $b$ are supplemented by 0's (zero coefficients) to length $2n$ and $\cdot$ denotes the componentwise product of two $2n$-element vectors.*

## 5. Applications

A breakthrough in this sort of abstract, broad form naturally leads to many beneficial consequences in various fields. For one thing, it turns out that the fastest algorithms we have for multiplying integers rely heavily on polynomial multiplication (after all, polynomials and integers are quite similar - just replace the variable $x$ by the base and watch out for carries).

**Integer Multiplication**

Recall the multiplication methods that we reviewed in the beginning of this paper. Either the conventional "left-shifting-and-adding" or the Russian Peasant multiplication takes $O(n^2)$ runtime. However, the FFT-based algorithm can reduce it to (surprise!) $O(n \log n)$ time. We shall quickly glace over how this is done.

Two large integers $X$ and $Y$ of size at most $n-1$ can be written in the form

$$X = P(B), Y = Q(B)$$

where $B$ is the base (usually $B = 10$ or a power of 10) $P$ and $Q$ two polynomials

$$P(z) = \sum_{j=0}^{n-1} x_j z^j, Q(z) = \sum_{j=0}^{n-1} y_j z^j.$$

If $R(z)$ denotes the polynomial obtained by the product of $P(z)$ and $Q(z)$, then $XY = R(B)$, and a final rearrangement on the coefficients of $R(z)$ permits to obtain the product $XY$. Thus, we have transformed the problem into multiplication of two polynomials of degree $< n$. And we now know how to do this task quickly, don't we? But nonetheless let's look at a formal presentation of the algorithm to multiply big numbers with FFT (the method is called *Strassen multiplication* when it is used with floating complex numbers):[8]

Let $n$ be a power of two (remember we can always add zero coefficients). To compute $Z = XY$ in time $O(n \log n)$, perform the following steps :

(1) Compute the Fourier transform (DFT) $X^*$ of size $2n$ of the sequence $(x_j)$. In other words, compute

$$X^* = (x_0^*, x_1^*, \cdots, x_{2n-1}^*) = \mathrm{DFT}_{2n}(x_0, x_1, \cdots, x_{n-1}, 0, \cdots, 0).$$

(2) Do the same for the sequence $(y_j)$ and obtain the Fourier transform $Y^*$:

$$Y^* = (y_0^*, y_1^*, \cdots, y_{2n-1}^*) = \mathrm{DFT}_{2n}(y_0, y_1, \cdots, y_{n-1}, 0, \cdots, 0).$$

(3) Multiply term by term of $X^*$ by $Y^*$ and thus obtain $Z^*$:

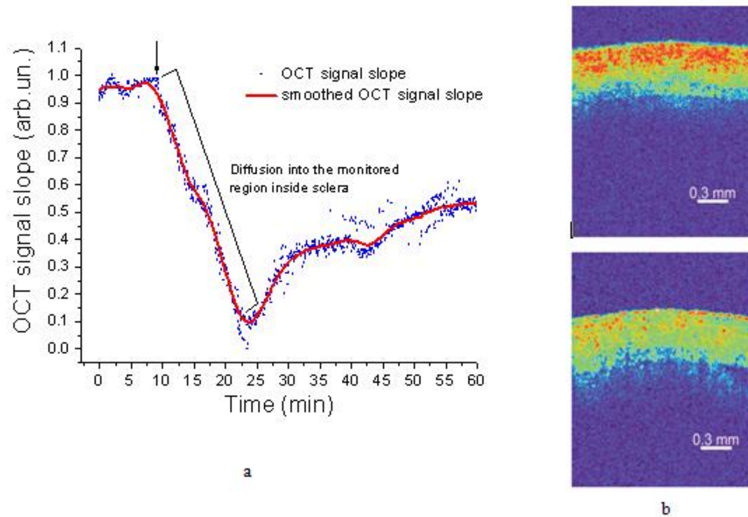
$$Z^* = (z_0^*, z_1^*, \cdots, z_{2n-1}^*), \text{where } z_i^* = x_i^* y_i^*.$$

(4) Compute the inverse Fourier transform ($\mathrm{DFT}^{-1}$) of $Z^*$ to obtain the final answer $Z$:

$$Z^* = (z_0, z_1, \cdots, z_{2n-1}), \text{where } z_i = \frac{1}{2n} \sum_{k=0}^{2n-1} z_k^* w_{2n}^{-ki}.$$

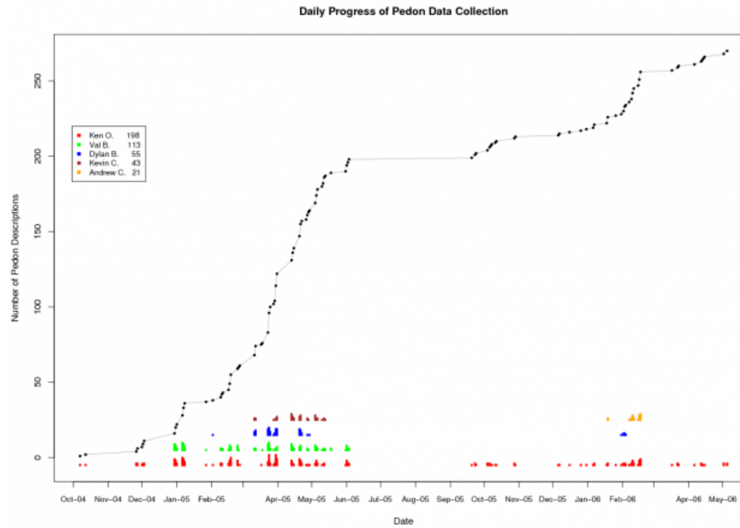
**Signal Processing**

But perhaps a more important application of the FFT is in signal processing. A signal is any quantity that is a function of time. An example might be a graph such as




It has many uses. It might, for instance, capture a human voice by measuring fluctuations in air pressure close to the speaker's mouth. Or it might trace the pattern of stars in the night sky by measuring brightness as a function of angle.

In order to extract information from a signal, we need to first turn it from *analog* to *digital* by sampling. That is, to get values at particular points as shown in

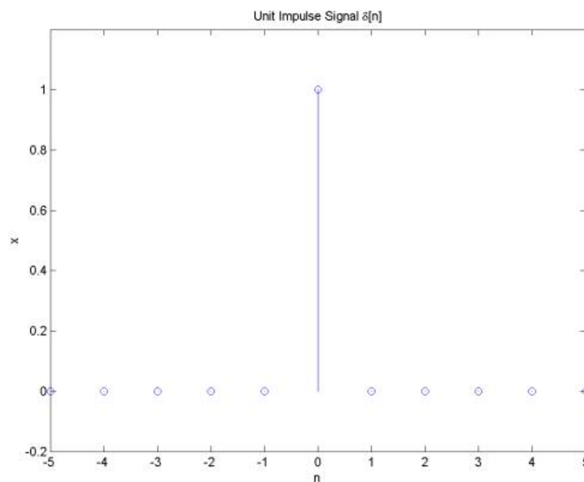



Then, we need to put it through a system that will transform it in some way. The output is called the response of the system:

$$\text{signal} \to \text{system} \to \text{response}.$$

An important class of systems are those that are *linear* (the response to the sum of two signals is just the sum of their individual responses) and *time invariant* (shifting the input signal by time $t$ produces the same output, also shifted by $t$).

Any system with these properties is completely characterized by its response to the *unit impulse* $\delta(t)$, which is the simplest possible input signal, consisting solely of a "jerk" (that is, the 3rd derivative of the function) at $t = 0$



To see why, consider the close relative $\delta(t-i)$, a shifted impulse in which the "jerk" occurs at time $i$. Let $a(t)$ be any signal. Then $a(t)$ can be expressed as a *linear combination* (letting $\delta(t-i)$ pick out its behavior at time $i$)

$$a(t) = \sum_{i=0}^{T-1} a(i)\delta(t-i).$$

if $T$ is the number of samples.

By linearity, the system response to input $a(t)$ is determined by the responses to the various $\delta(t-i)$. And by time invariance, these are in turn just shifted copies of the *impulse response* $b(t)$, the response to $\delta(t)$.

In other words, the output of the system at time $k$ is

$$c(k) = \sum_{i=0}^{k} a(i)b(k-i).$$

*But this is exactly the formula for polynomial multiplication!* [1]

We will not go over the details of implementation, however. But perhaps we can appreciate how this rather unexpected relation between the FFT (polynomial multiplication) and signal processing has come about.

*Acknowledgements.* I would like to appreciate my brother Eung Sun, my father and my mother for being them. Also, I want to thank Professor Daniel Stevankovic for teaching me about the FFT, and also Professor Salur for giving me the opportunity to write my very first paper on a mathematical topic.

## References

[1] Dasgupta, S., Papadimitrious, C., and Vazirani, U., *Algorithms*, McGraw-Hill Higher Education (2008), 11–22.
[2] Cormen T. H., Leiserson C. E., Rivest R. L., and Stein, C., *Introduction to Algorithms*, The MIT Press (2001), 822–844.
[3] Bartle, R. G. and Sherbert, D. R., *Introduction to Real Analysis*, John Wiley & Sons, Inc. (2000), 661–693.
[4] *Wikipedia, the free encyclopedia*, Retrieved November 8th, 2009, from http://en.wikipedia.org.
[5] Friedberg, S. H., Insel, A. J., and Spence, L. E., *Linear Algebra*, Pearson Education, Inc. (2003), 199–244.
[6] *The Math Forem at Drexel University*, Retrieved November 9th, 2009, from http://mathforum.org/library/drmath/view/55499.html
[7] *ProofWiki*, Retrieved November 24th, 2009, from http://www.proofwiki.org/
[8] *Numbers and Computation*, Retrieved November 28th, 2009, from http://numbers.computation.free.fr/Constants/Algorithms/fft.html

University of Rochester, NY, 14627, US / Gangnam-gu, Seoul, Korea
*E-mail address*: jlee164@u.rochester.edu