LARGE INTEGER MULTIPLICATION: A FRIENDLY SURVEY OF ALGORITHMS

JT MILLER

Abstract. How do computers multiply integers larger than the number of bits within dedicated hardware? This paper provides a survey of several large integer multiplication algorithms that answer this question. The aim of the paper is to provide a friendly introduction with many examples such that undergraduate students can look past the complexity of these algorithms, grasp the fundamentals, and hopefully understand some of the beauty and wonder in the algorithms.

Contents

Date: December 8, 2023.

1. Introduction

Large integer multiplication is pervasive in our lives in the form of cryptography. One need only look at the innards of any cryptography library to find a BigInt library or API behind the scenes. Within those libraries/APIs are the algorithms discussed within this paper. There are other less pervasive uses, such as scientific and algorithm research, large-scale data analysis, and of course computational mathematics.

How do computers multiply integers larger than the number of bits within dedicated hardware? This paper provides a survey of several large integer multiplication algorithms that answer this question. The aim of the paper is to provide a friendly introduction with many examples such that undergraduate students can look past the complexity of these algorithms, grasp the fundamentals, and hopefully understand some of the beauty in the algorithms.

The title of the paper is a tribute to Dr. Silverman's wonderful book, *A Friendly Introduction to Number Theory* [13]. We will start with grade school multiplication, then cover a simple but useful extension that allows multiplication in "chunks". We then cover the first advance in nearly 4000 years [3], Karatsuba's algorithm. Next we will discuss the Toom-Cook algorithm that is a generalization of Karatsuba's algorithm. Next we cover the fundamentals of the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT) to understand how these algorithms are used to multiply large integers. Finally, we cover the Schönhage-Strassen multiplication algorithm, which gets us to $\mathcal{O}(n \log n \log\log n)$ time complexity.

1.1. Prerequisites.

There are surprisingly few prerequisites to understand this material. The most important prerequisites are some familiarity with mathematical notation and the desire to understand large integer multiplication. Grade school arithmetic and trigonometry through complex numbers and the unit circle is all that is required. The more advanced algorithms would benefit from familiarity with matrix manipulation, Linear Algebra, and Calculus, but these are not strict prerequisites. As this is a friendly introduction, we have tried to make the algorithms available to the widest audience possible, and hence avoided discussions that require more advanced mathematics. Suggestions to make the material available to an even wider audience are welcome.

1.2. Notation and Volume Challenges.

The density and abstract symbolic nature of mathematical notation makes understanding the underlying concepts expressed by the notation extremely challenging. Human beings are primarily visual learners, and mathematical notation purposefully condenses and symbolizes ideas into abstract notation that does not readily translate to an intuitive understanding of the underlying concepts. Being able to understand the notation takes time

and practice. Developing an intuitive understanding of what the notation represents even moreso. We have striven to provide a non-trivial example for each algorithm/transform/concept, but because we are unpacking concise notation, the sheer volume of arithemtic produced can prove challenging on its own. Do not despair - if you find yourself lost in the details, remind yourself of what we are trying to compute and the underlying concepts behind the computations and notation. After all, we're just multiplying integers, how complex can it be?

1.3. **Software.** The companion software for this paper can be found on GitHub at *github.com/xiangyazi24/Integer_Mul_REU*. All of the examples shown are solved in software, as well as all algorithms illustrated.

2. Grade School Multiplication

Multiplication dates back to at least 2000 B.C. [3] and of course we all learn how to multiply integers in grade school. A review of this algorithm gives us a grasp of the terminology and notation we will use throughout the rest of this paper to discuss multiplication algorithms. We start with a representation for integers, which will be useful for later algorithms.

2.1. **Integer Representation** We can express any positive integer x as a summation of coefficients raised to a base power [13]:

$$x = \sum_{i=0}^{n-1} X_i B^i.$$

Given a base system and coefficients, we can express x as a coefficient vector with n entries:

$$\mathbf{X} = \{X_0, X_1, X_2, ..., X_{n-1}\}.$$

E.g. we can represent the number 40342 as

$$\mathbf{X} = \{2, 4, 3, 0, 4\},$$

and in base 10 we would have

$$2 \cdot 10^0 + 4 \cdot 10^1 + 3 \cdot 10^2 + 0 \cdot 10^3 + 4 \cdot 10^4 = 40342.$$

Note that our coefficients for powers appear "backward" in the set/array, as lower-powered coefficients appear first, but we naturally write higher-power to lower-power coefficients left to right when expressing numbers. Even from this simple idea, we can make a few observations that will prove useful later. Given two positive integers x and y, with one integer having fewer coefficients, we can still express the integers as vectors of the same length by zero-padding the smaller integer. E.g. let $x = 389$ and $y = 4$, then we can simply say $\mathbf{X} = \{9, 8, 3\}$ and $\mathbf{Y} = \{4, 0, 0\}$. If the larger integer has n coefficients, we now have two integers with n coefficients.

How many coefficients are necessary to store the result of two n-digit positive integers?

Lemma 2.1. *Multiplying two n-digit positive integers requires no more than $2n$ coefficients.*

Proof. Let x be an n-digit positive integer in positive base B.

$$x \leq B^n - 1$$
$$x^2 \leq (B^n - 1)^2$$
$$x^2 \leq B^{2n} - 2B^n + 1$$

$2B^n > 1$ for all $B > 0$, so we may simplify the right side of equation **??** to

$$x^2 < B^{2n}$$

□

Let's work an example multiplying $9999 \times 9999 = 99980001$ in base 10. We have $B = 10$ and $n = 4$. In this case, we are using the maximum value (9999) that can be represented in base 10 with 4 digits. We can simply use equation **??** directly and compute x^2.

$$x^2 = B^{2n} - 2B^n + 1$$
$$x^2 = 10^8 - 2 \cdot 10^4 + 1$$
$$x^2 = 99980001$$

Let us now discuss our (modified) grade school multiplication algorithm.

2.2. **Algorithm** For grade school multiplication, let us use the values $x = 1234$ and $y = 6789$ to illustrate our algorithm. Instead of simply multiplying the ones' digits 9 and 4 and writing the new ones' digit 6 and carrying the 3, we will write the individual products in each "column" corresponding to the base power, as shown in Figure 1. After we have computed all partial products, we add the partial products within each "column" or base power. Finally, we carry any overflow of partial products to the next column/base power, and we have our summation of final products yielding the answer 8377626.

Expressed mathematically, we are performing the following,

$$xy = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} X_j Y_k B^{j+k}. \tag{2.2}$$

Now, let us determine an algorithm for performing our grade school multiplication.

Algorithm 1 Grade School Multiplication

for $j \in \{0, 1, ..., n-1\}$ **do**
  for $k \in \{0, 1, ..., n-1\}$ **do**
    $W[j+k] += X_j Y_k B^{j+k}$
  end for
end for
$c = 0$
for $j \in \{0, 1, ..., 2n-2\}$ **do**
  $c = \lfloor W[j]/B \rfloor$
  $W[j] = W[j] \pmod B$
  $W[j+1] += c$
end for
return W

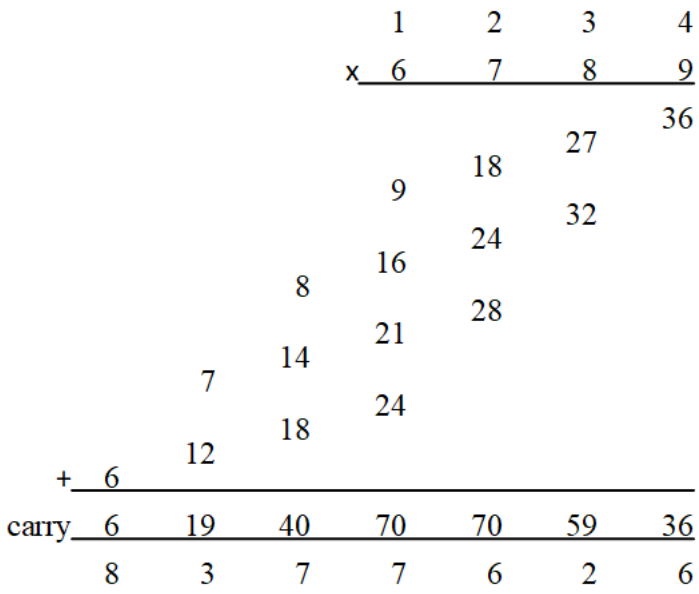

FIGURE 1. Grade School Multiplication of 1234×6789

2.3. **Time Complexity Analysis** We will not belabor the time-complexity analysis to determine exact counts of the operations involved. In general additions are easier for computers, so we will typically only care about the number of necessary multiplications. From Figure 1 we can see that for n digits, we have $n \times n$ partial products to compute. Accordingly, our grade school multiplication algorithm yields $\mathcal{O}(n^2)$.

3. Chunky Grade School Multiplication

We can make a simple modification to our grade school multiplication that allows us to increase our multiplication speed by a factor of our machine's bit-length. Because this technique is generic, the exact bit-lengths involved in integer hardware multiplication aren't necessarily important. The increase comes by recognizing that instead of multiplying a single digit at a time, we can multiply chunks of digits. We will use $123\,456 \times 987\,654 = 121\,931\,812\,224$ to illustrate[1] our chunky grade school multiplication algorithm improvement.

Figure 2 shows how we can group our digits into chunks. These chunks allow fewer partial products even though we now have more total digits. The chunks "skip" base powers by grouping digits.

[1]Note that the spaces in numbers such as these are strictly for readability purposes. Numbers formatted with spaces like those in the previous multiplication are still a single contiguous number.

		123	456	
	x	987	654	
				298224
		80442		
		450072		
	+	121401		
carry		121401	530514	298224
121		931	812	224

FIGURE 2. Chunky Grade School Multiplication of $123\,456 \times 987\,654$

If we group 3 digits then we are using a chunk size of 3 base powers, which we then keep track of throughout the algorithm. In actual large integer multiplication algorithms, we must be careful to guarantee that the summation of partial products for a given base power/column fits within the machine word size/bit width. Let us use the positive integer h for chunk size, then we have just a small modification from equation 2.2,

$$xy = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} X_j Y_k B^{h(j+k)}. \tag{3.1}$$

Algorithm 2 Chunkify

Pick a chunk size $h > 1$
Pad X, Y to kh where k is a positive integer
$n = len(X)/h$
$X_{chunky} = \{0, 0, ..., 0\}$
$Y_{chunky} = \{0, 0, ..., 0\}$
for $j \in \{0, 1, ..., nh - 1\}$ **do**
    $X_{chunky}[\lfloor j/h \rfloor] += X[j] * 10^{j \pmod h}$
    $Y_{chunky}[\lfloor j/h \rfloor] += Y[j] * 10^{j \pmod h}$
end for
return X_{chunky}, Y_{chunky}

3.1. **Algorithm**

3.2. **Time Complexity Analysis** From Figure 2 we can see that we have reduced from $n \times n$ operations to $n/h \times n/h$ operations, which can be a significant real reduction, but still results in a run time of $\mathcal{O}(n^2)$.

4. KARATSUBA

We now turn our attention to Karatsuba's algorithm. In 1960, Andrey Kolmogorov conjectured that $\mathcal{O}(n^2)$ was as fast as multiplication could be achieved. Kolmogorov held a seminar at Moscow University where he presented this and other

Algorithm 3 Chunky Grade School Multiplication

for $j \in \{0, 1, ..., n-1\}$ **do**
  for $k \in \{0, 1, ..., n-1\}$ **do**
    $W[j+k] += X_j Y_k B^{j+k}$
  end for
end for
$c = 0$
for $j \in \{0, 1, ..., 2n-2\}$ **do**
  $c = \lfloor W[j]/B^h \rfloor$
  $W[j] = W[j] \pmod{B^h}$
  $W[j+1] += c$
end for
return W

conjectures. Anatoly Karatsuba, a 23-year-old student, found a divide and conquer deconstruction method that multiplies integers in $\mathcal{O}(n^{\log_2 3})$ that he presented at the next seminar, which Kolmogorov then disbanded [15]. Kolmogorov was so impressed with Karatsuba's algorithm that he wrote a paper for him, which Karatsuba only became aware of after the edits were mailed back to him [10]. Perhaps the best way to motivate some is to say that something is impossible.

4.1. **Algorithm** Karatsuba's algorithm involves manipulating the partial products involved in multiplication, a divide and conquer approach. Let the tens' and ones' digits of x be ab, or $X_1 = a$, and $X_0 = b$ from our previous notation. Similarly, let the tens and ones' digits of y be cd, or $Y_1 = c$ and $Y_0 = d$. Figure 3 shows this simple grade school multiplication.

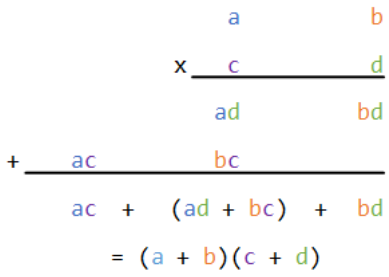

FIGURE 3. Grade School *abcd* Multiplication

Note that we are neglecting the base. Figure 4 shows the same multiplication with our base z included.

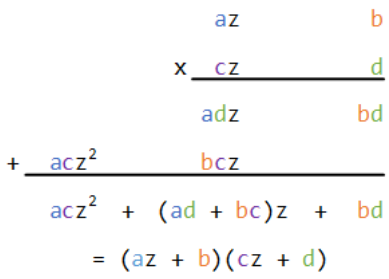

FIGURE 4. Grade School *abcdz* Multiplication (includes base z)

Karatsuba recognized that the multiplication of partial products, $(a+b)(c+d) = ac + (ad + bc) + bd$ could be manipulated to yield three multiplications instead of four. ac is one multiplication, bd is the second, and then we can use the fact that $(ad + bc) = (a + b)(c + d) - (ac + bd)$ for the third multiplication. We now have all the partial products necessary, having deconstructed the addition of partial products. At the cost of a few more additions and subtractions, we can remove one multiplication. Figure 5 shows this simple coefficient manipulation with the multiplications numbered at the bottom of equations and the "free" multiplication shown. The letters *A*-*E* denote the arithmetic quantities we will compute during the algorithm.

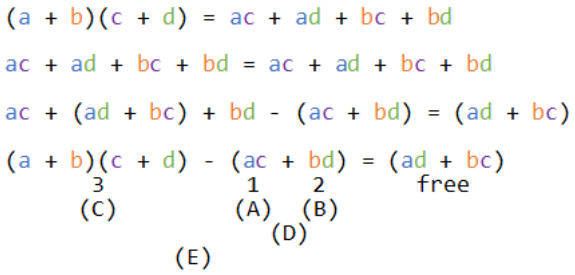

FIGURE 5. Karatsuba's Manipulation of Coefficients

Figure 6 shows an example for $25\,786\,109 \times 72\,166\,948 = 1\,860\,904\,787\,325\,332$ with the same coloring used to denote *abcd* as the previous explanations. Note that we have combined the algorithm with chunking (typically this is referred to as blocking, but the term chunking is more fun) that combines several coefficients.

```
          25786109
        x72166948
 ---------------
 1860904787325332

ac = (2578)(7216) = 18602848

bd = (6109)(6948) = 42445332

(a + b)(c + d) = (2578 + 6109)(7216 + 6948) = (8687)(14164) = 123042668

(ac + bd) = (18602848 + 42445332) = 61048180

(a + b)(c + d) - (ac + bd) = 123042668 - 61048180 = 61994488

acz^2 = 1860284800000000
((a + b)(c + d) - (ac + bd))z = 619944880000

bd = 42445332

1860284800000000 + 619944880000 + 42445332
= 1860904787325332
```

FIGURE 6. Karatsuba Example

Karatsuba's three coefficient products, ac, bd, and $(a+b)(c+d)$, can be used to recursively subdivide the problem further. Figure 7 shows a larger Karatsuba example where we recursively apply Karatsuba to coefficients.[2] The equations in the upper right are those as before. The numbers 1-3 show the three multiplications and the letters A-E denote the arithmetic quantities computed during the algorithm. We shows these A-E quantities under each stage, including the final summation with base powers multiplied into computed quantities. Note that we dispense with printing this final summation after the first stage as the large quantities become cumbersome to print and we just print the A-E quantities and final result. [15] provides an excellent graphic with another example and a different representation of the computations involved.

Algorithm 4 Karatsuba's Multiplication

Input: Given 2 numbers as ab and cd with a chunk in base z, calculate their product $ab \cdot cd$.
$\alpha = ac$
$\beta = bd$
$\delta = (a+b)(c+d)$
while α, β, $\delta >$ machine number of digits **do**
Split each of the α, β, δ multiplications into ab, cd and apply Karatsuba.
end while
$\gamma = \delta - (\alpha + \beta) = (ad + bc)$
return $\alpha z^2 + \gamma z + \beta$

[2]The recursion is for illustrative/instructive purposes and unnecessary. We could have used more coefficients and a larger base power without the recursive stages.

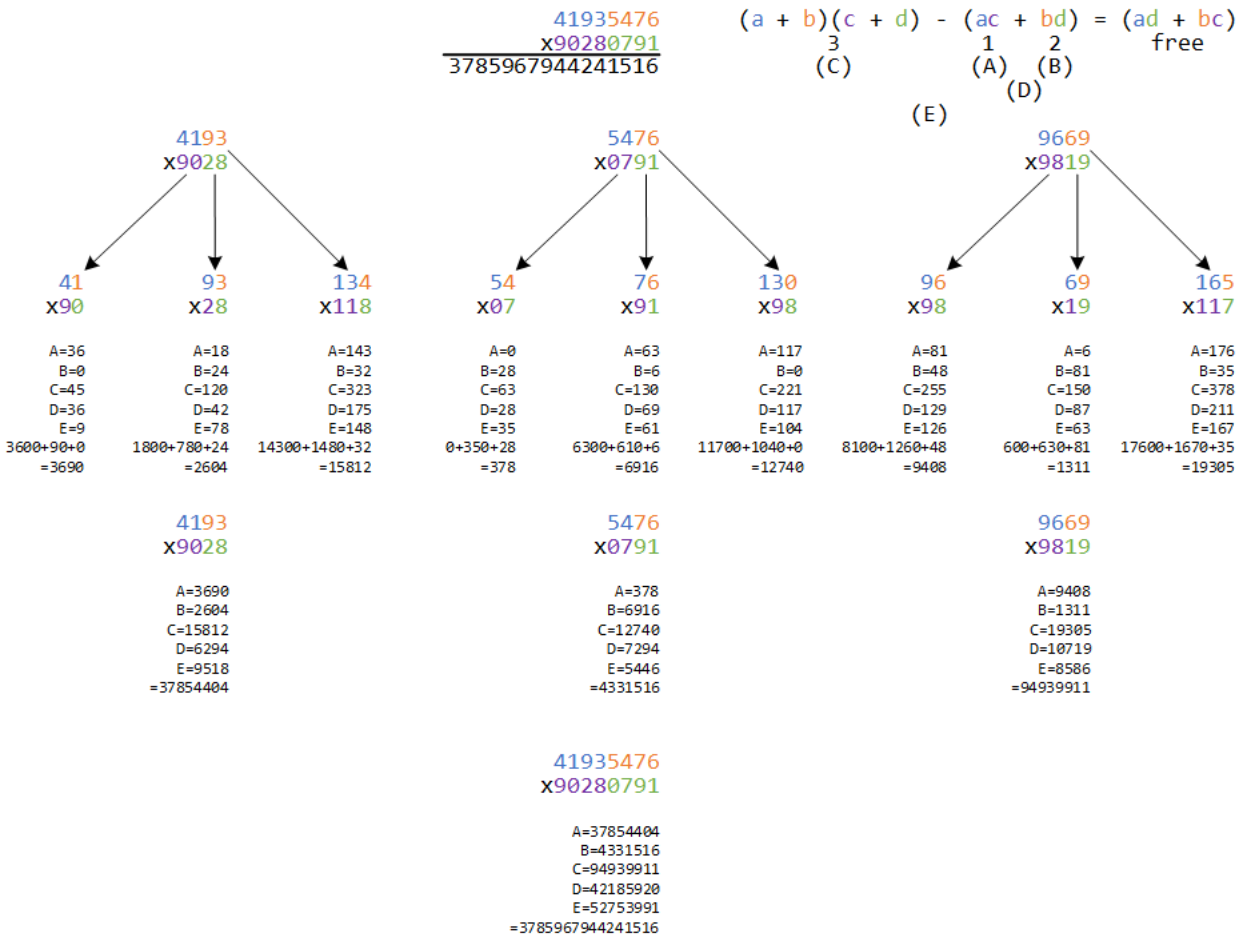

FIGURE 7. Recursive Karatsuba Example

4.2. **Time Complexity Analysis** As Algorithm 4 and Figure 7 show, Karatsuba recursively subdivides the problem in half using three coefficient multiplications per subdivision. The runtime is then $\mathcal{O}(n^{\log_2 3})$. $\log_2 3 \approxeq 1.585$, with the runtime complexity of Karatsuba's algorithm commonly referred to as $\mathcal{O}(n^{1.585})$.

5. TOOM-COOK

The Toom-Cook algorithm generalizes the divide and conquer approach used in Karatsuba. Instead of splitting the multiplicands into two parts, Toom-Cook splits the multiplication into r parts. Toom-Cook then uses the splits as coefficients in a polynomial, as we have done before. The clever part of Toom-Cook is to use these splits parts in the X and Y coefficients to represent polynomials $x(t)$ and $y(t)$, then multiply those polynomials together. Because the $x(t)$ and $y(t)$ polynomials will be of degree $r-1$, their product will be of degree $2r-2$, at most. We can evaluate the multiplication of $x(t)$ and $y(t)$ at $2r-1$ points to recover the product $w(t)$, of

degree $2r - 2$ (at most). Hence we are now multiplying together polynomials to recover integers!

Judicious selection of r and the points at which we multiply the polynomials yields the best results. Toom-Cook multiplication of n-word numbers takes, at best, $\mathcal{O}(n^{1+1/\sqrt{\log n}})$ [2].

Consider that we pad the multiplicands so that we can split the multiplicand into three coefficients of equal length. We can then write our multiplicands as polynomials

$$x(t) = x_2t^2 + x_1t + x_0$$
$$y(t) = y_2t^2 + y_1t + y_0.$$

We multiply these two polynomials to produce $w(t)$:

$$w(t) = x(t) * y(t).$$

Because we know the maximum degree of $w(t)$, we can write the polynomial's representation as

$$w(t) = w_4t^4 + w_3t^3 + w_2t^2 + w_1t + w_0.$$

We should note that t is equivalent to our base, which of course may be "chunks". Hence $t = B^h$, and $t^k = B^{kh}$ where k is some positive integer. Obviously, $w_4 = x_2y_2$, and so on:

$$w_4 = x_2y_2$$
$$w_3 = x_2y_1 + x_1y_2$$
$$w_2 = x_2y_0 + x_1y_1 + x_0y_2$$
$$w_1 = x_1y_0 + x_0y_1$$
$$w_0 = x_0y_0.$$

For the Toom-Cook algorithm split into 3 parts, a popular choice for the points is 0, 1, -1, -2, and ∞. The point at infinity simply reduces to the highest level coefficient. Our equations reduce nicely using these points:

$$\begin{aligned}
x(0) &= x_2(0)^2 + x_1(0) + x_0 &&= x_0\\
x(1) &= x_2(1)^2 + x_1(1) + x_0 &&= x_2 + x_1 + x_0\\
x(-1) &= x_2(-1)^2 + x_1(-1) + x_0 &&= x_2 - x_1 + x_0\\
x(-2) &= x_2(-2)^2 + x_1(-2) + x_0 &&= 4x_2 - 2x_1 + x_0\\
x(\infty) &= x_2(\infty)^2 + x_1(\infty) + x_0 &&= x_2.
\end{aligned}$$

We can further optimize the above arithmetic computations by saving some partially computed results[3] and applying to subsequent arithmetic computations:

$$\gamma = x_0 + x_2 \tag{5.1}$$
$$x(0) = x_0 \tag{5.2}$$
$$x(1) = \gamma + x_1 \tag{5.3}$$
$$x(-1) = \gamma - x_1 \tag{5.4}$$
$$x(-2) = 2(x(-1) + x_2) - x_0 \tag{5.5}$$
$$x(\infty) = x_2. \tag{5.6}$$

The same computations are used for $y(t)$. Now we can compute $w(t) = x(t)y(t)$. At this point, we will introduce some linear algebra that compactly shows the $2r-1$ equations we are solving and how we can obtain $w(t)$ (and the Ws we are after) by inverting this matrix. For those unfamiliar with linear algebra, you can skip down below to where we restate $w(t)$. You can solve the simultaneous set of equations to arrive at the same point.

$$\begin{pmatrix} w(0) \\ w(1) \\ w(-1) \\ w(-2) \\ w(\infty) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ (-2)^0 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}.$$

Because we were judicious in the choice of our evaluation points, the matrix above is invertible, and we may simply invert the matrix to solve for our coefficients directly.

$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} w(0) \\ w(1) \\ w(-1) \\ w(-2) \\ w(\infty) \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1/2 & 1/3 & -1 & 1/6 & -2 \\ -1 & 1/2 & 1/2 & 0 & -1 \\ -1/2 & 1/6 & 1/2 & -1/6 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w(0) \\ w(1) \\ w(-1) \\ w(-2) \\ w(\infty) \end{pmatrix}$$

[3]Note that γ is simply a variable we use for the partial computation of $x_0 + x_2$, there is no other significance.

For our friends without linear algebra rejoining us, this is

$$\begin{aligned}
w_0 &= w(0) \\
w_1 &= \frac{1}{2}w(0) + \frac{1}{3}w(1) - w(-1) + \frac{1}{6}w(-2) - 2w(\infty) \\
w_2 &= -w(0) + \frac{1}{2}w(1) + \frac{1}{2}w(-1) - w(\infty) \\
w_3 &= -\frac{1}{2}w(0) + \frac{1}{6}w(1) + \frac{1}{2}w(-1) - \frac{1}{6}w(-2) + 2w(\infty) \\
w_4 &= w(\infty)
\end{aligned}$$

Note that we could optimize this by computing partial summations and then reusing results as we did in Equation 5.1-5.6. We leave this as an exercise for the reader. Let us compute a Toom-3 example with $698\,310\,488\,572\,646\,777\,019\,184 \times 144\,585\,992\,498\,882\,884\,065\,634 = 100\,965\,915\,062\,655\,948\,833\,325\,499\,910\,140\,535\,809\,533\,122\,656$. Here,

$$\begin{aligned}
x_2 &= 69\,831\,048 \\
x_1 &= 85\,726\,467 \\
x_0 &= 77\,019\,184 \\
y_2 &= 14\,458\,599 \\
y_1 &= 24\,988\,828 \\
y_0 &= 84\,065\,634.
\end{aligned}$$

Now we can compute our points for our x and y polynomials:

$$\begin{aligned}
x(0) &= x_0 && = 77\,019\,184 \\
x(1) &= x_2 + x_1 + x_0 && = 69\,831\,048 + 85\,726\,467 + 77\,019\,184 && = 232\,576\,699 \\
x(-1) &= x_2 - x_1 + x_0 && = 69\,831\,048 - 85\,726\,467 + 77\,019\,184 && = 61\,123\,765 \\
x(-2) &= 4x_2 - 2x_1 + x_0 && = 4 \cdot 69\,831\,048 - 2 \cdot 85\,726\,467 + 77\,019\,184 && = 184\,890\,442 \\
x(\infty) &= x_2 && = 85\,726\,467
\end{aligned}$$

$$\begin{aligned}
y(0) &= y_0 && = 84\,065\,634 \\
y(1) &= y_2 + y_1 + y_0 && = 14\,458\,599 + 24\,988\,828 + 84\,065\,634 && = 123\,513\,061 \\
y(-1) &= y_2 - y_1 + y_0 && = 14\,458\,599 - 24\,988\,828 + 84\,065\,634 && = 73\,535\,405 \\
y(-2) &= 4y_2 - yx_1 + y_0 && = 4 \cdot 14\,458\,599 - 2 \cdot 24\,988\,828 + 84\,065\,634 && = 91\,922\,374 \\
y(\infty) &= y_2 && = 14\,458\,599.
\end{aligned}$$

And then multiply the two polynomials points' together to yield points for $w(t)$:

$$\begin{aligned}
w(0) &= x(0)y(0) && = 77\,019\,184 \cdot 123\,513\,061 && = 6\,474\,666\,533\,122\,656 \\
w(1) &= x(1)y(1) && = 232\,576\,699 \cdot 73\,535\,405 && = 28\,726\,260\,010\,765\,639 \\
w(-1) &= x(-1)y(-1) && = 61\,123\,765 \cdot 84\,065\,634 && = 4\,494\,760\,814\,399\,825 \\
w(-2) &= x(-2)y(-2) && = 77\,019\,184 \cdot 91\,922\,374 && = 16\,995\,568\,358\,549\,308 \\
w(\infty) &= x(\infty)y(\infty) && = 184\,890\,442 \cdot 14\,458\,599 && = 1\,009\,659\,120\,781\,752.
\end{aligned}$$

We then multiply our inverse matrix by $w(t)$'s points to yield the coefficients:

$$w_0 = 6\,474\,666\,533\,122\,656$$
$$w_1 = 9\,131\,268\,940\,611\,430$$
$$w_2 = 9\,126\,184\,758\,678\,324$$
$$w_3 = 2\,984\,480\,657\,571\,477$$
$$w_4 = 1\,009\,659\,120\,781\,752.$$

Now we perform the carries based on our chunk size. We used a chunk of 8, hence we carry beyond 10^8 digits of each chunk to the next chunk. As an example, for w_0, the lower 8 digits are 33 122 656 and we carry and add 64 746 665 to the lower 8 digits of the next chunk, 40 611 430, giving us $64\,746\,665 + 40\,611\,430 = 105\,358\,095$. Hence the lower 16 digits of our product are 535 809 533 122 656. We continue in this manner to find our final answer of $698\,310\,488\,572\,646\,777\,019\,184 \times 144\,585\,992\,498\,882\,884\,065\,634 =$
$100\,965\,915\,062\,655\,948\,833\,325\,499\,910\,140\,535\,809\,533\,122\,656$.

5.1. **Algorithm** The algorithm for Toom-3 is straightforward and was demonstrated in the previous example. For completeness, the algorithm above, and reproduced from [6], follows.

Algorithm 5 Toom-3

$r_0 = x_0 - 2x_1 + 4x_2$
$r_1 = x_0 - x_1 + x_2$
$r_2 = x_0$
$r_3 = x_0 + x_1 + x_2$
$r_4 = x_0 + 2x_1 + 4x_2$
$x_0 = y_0 - 2y_1 + 4y_2$
$s_1 = y_0 - y_1 + y_2$
$s_2 = y_0$
$s_3 = y_0 + y_1 + y_2$
$s_4 = y_0 + 2y_1 + 4y_2$
for $0 \leq j < 5$ **do**
    $t_j = r_j s_j$
end for
$z_0 = t_2$
$z_1 = t_0/12 - 2t_1/3 + 5t_3/3 - t_4/12$
$z_2 = -t_0/24 + 2t_1/3 - 5t_2/4 + 2t_3/3 - t_4/24$
$z_3 = -t_0/12 + t_1/6 + t_4/12$
$z_4 = t_0/24 - t_1/6 + t_2/4 - t_3/6 + t_4/24$
Perform carries
return $(z_0, z_1, z_2, z_3, z_4, carry)$

5.2. **Time Complexity Analysis** The basic form of Toom-Cook k-way is to split the operands into k partial products. For Toom-3 as we demonstrated, we split into 3 partial products but have 5 multiplies at each stage, so the complexity is $\mathcal{O}(n^{\log_3 5})$. Note how quickly the number of coefficient terms for our polynomial grows as we partition our multiplicands into larger numbers. In most cases, this limits the practical use of Toom-Cook to three or four partitions.

6. Convolution, the DFT, and the FFT

The Toom-Cook algorithm showed an alternate method to carry out multiplication rather than just optimizing the mechanics of our familiar grade school multiplication process. In the case of Toom-Cook, that was using the multiplicands as coefficients for polynomials, multiplying enough points to represent a resultant polynomial, then interpolating back into the coefficients that represent our desired product. Such alternate methods are often referred to as transforms. We continue on this path of transforms now with the Discrete Fourier Transform (DFT). The Discrete Fourier Transform (DFT) is typically used to take a number of time-domain samples and transform those samples to show the frequency domain content of the signal represented by the time-domain samples. The Fast Fourier Transform (FFT) is a novel method to compute the DFT quickly with the restriction that the number of time domain samples must be a binary power, i.e. 2^m for some positive integer m. We will utilize the DFT and FFT algorithms that transform a time-domain signal to a frequency domain signal to transform our multiplication process. Before we get to these transforms, we will review the DFT and FFT algorithms.

6.1. **The Discrete Fourier Transform.**

The DFT formula is

$$X_m = \sum_{n=0}^{N-1} x_n e^{-i2\pi nm/N}.$$

where $e^{-i2\pi nm/N}$ is a complex number that represents $cos(2\pi k/n) - i\sin(2\pi k/n)$. The interesting thing to note, from a number theoretic transform viewpoint, is that $e^{i2\pi k/n}$ raised to any positive integer always has a magnitude of 1. That is, $|e^{-i2\pi nm/N}| = 1$. A *root of unity* is a number with the property that when raised to a positive integer power, the result is 1. This will be important in our use of the DFT/FFT to compute the multiplication of large integers. The DFT provides the spectral content of the signal, and the larger the number of points used, N, the finer the resolution of the spectral content, and the more calculations that must be carried out to attain the DFT.

The root of unity quantity $e^{-i2\pi nm/N}$ is often called the twiddle factor in the DFT and FFT. We use a shorthand to denote twiddle factors:

$$W_N = e^{-i2\pi/N}. \tag{6.1}$$

The DFT then becomes:

$$X_m = \sum_{n=0}^{N-1} x_n W_N^{nm}.$$

Why are we bothering to introduce the DFT and how to transform a signal from the time domain to the frequency domain? Because convolution in the time domain of coefficients is the same as multiplication. The issue with convolution in the time domain is that it is slow, the same as our $\mathcal{O}(n^2)$ from grade school multiplication. But convolution in the time domain is the same as multiplication in the frequency domain. The Fast part of the FFT means we can perform our transform in $\mathcal{O}(n \log n)$. We can quickly transform our signal made up of polynomial coefficients (our time domain signal) using the FFT, do a point-wise multiplication in the frequency domain, then transform back into the time domain by computing an inverse FFT. We are now transforming and multiplying transforms to complete our large integer multiplication!

Let's first demonstrate that convolution of polynomial coefficients is the same as multiplication. Let's use 1234×6789 as our example. In this case, we have

$$x = \{4, 3, 2, 1\}$$
$$y = \{9, 8, 7, 6\}.$$

The convolution is defined as

$$(x * y)(n) = \sum_{k=-\infty}^{\infty} x(k)y(n-k). \tag{6.2}$$

If we are working with coefficients for a polynomial with n coefficients, we can change the summation limits to reflect the actual limits of the summation. Remember that our product can have up to $2n$ coefficients if our multiplicands each have n bits. Because we're starting our numbering at 0, we set the upper limit at $2n-1$. While we're at it, let's use a new variable for the convolution result and define $h = x * y$, then:

$$h(n) = \sum_{k=0}^{2n-1} x(k)y(n-k). \tag{6.3}$$

Let's work through our example of $x = 1234 \times y = 6789$:

$$\begin{aligned}
h_0 &= x_0y_0 && = 4 \cdot 9 && = 36 \\
h_1 &= x_0y_1 + x_1y_0 && = 4 \cdot 8 + 3 \cdot 9 && = 59 \\
h_2 &= x_0y_2 + x_1y_1 + x_2y_0 && = 4 \cdot 7 + 3 \cdot 8 + 2 \cdot 9 && = 70 \\
h_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 && = 4 \cdot 6 + 3 \cdot 7 + 2 \cdot 8 + 1 \cdot 9 && = 70 \\
h_4 &= x_1y_3 + x_2y_2 + x_3y_1 && = 3 \cdot 6 + 2 \cdot 7 + 1 \cdot 8 && = 40 \\
h_5 &= x_2y_3 + x_3y_1 && = 2 \cdot 6 + 1 \cdot 7 && = 19 \\
h_6 &= x_3y_3 && = 6
\end{aligned}$$

Of course, we then carry at each power:

$$\begin{aligned}
h_0 &= 6 && \to 6 \\
h_1 &= 59 + 3 && \to 2 \\
h_2 &= 70 + 6 && \to 6 \\
h_3 &= 70 + 7 && \to 7 \\
h_4 &= 40 + 7 && \to 7 \\
h_5 &= 19 + 4 && \to 3 \\
h_6 &= 6 + 2 && \to 8
\end{aligned}$$

So our final answer is $1234 \times 6789 = 8\,377\,626$. This operation should look extremely familiar. This is the same as our initial grade school example from section 2.2. We reproduce Figure 1 here for convenience.

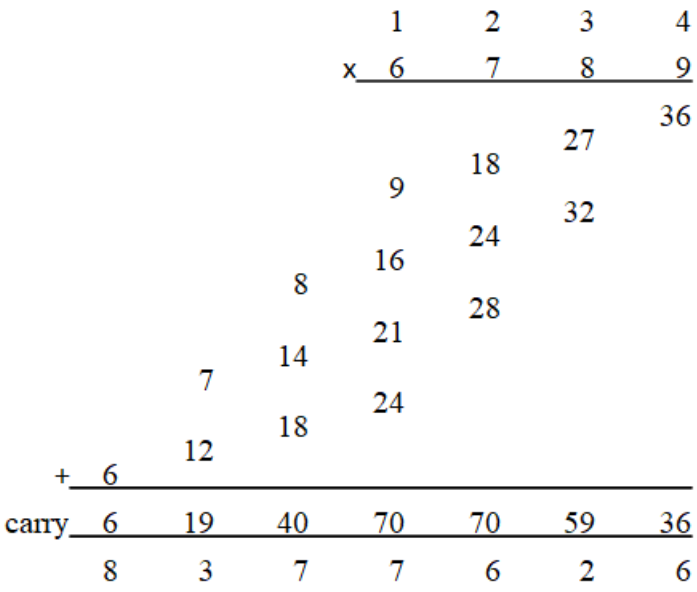

FIGURE 8. Grade School Multiplication of 1234×6789 - again

To provide a visual, many texts discuss the convolution operation in terms of reflecting one function about the y-axis and sliding that function, $x(n)$ in this case, across the other. When we do this we will see the same grade school multiplication operation that we are used to. Figure 9 below illustrates the operation. Note that we have provided zeroes for coefficients outside our populated coefficients. We dispense with these zeroes to show the "sliding" of $x(n)$ across $y(n)$. The box drawn around the coefficients shows us which terms we are multiplying (vertically) and summing (horizontally) to compute each $h(n)$ coefficient.

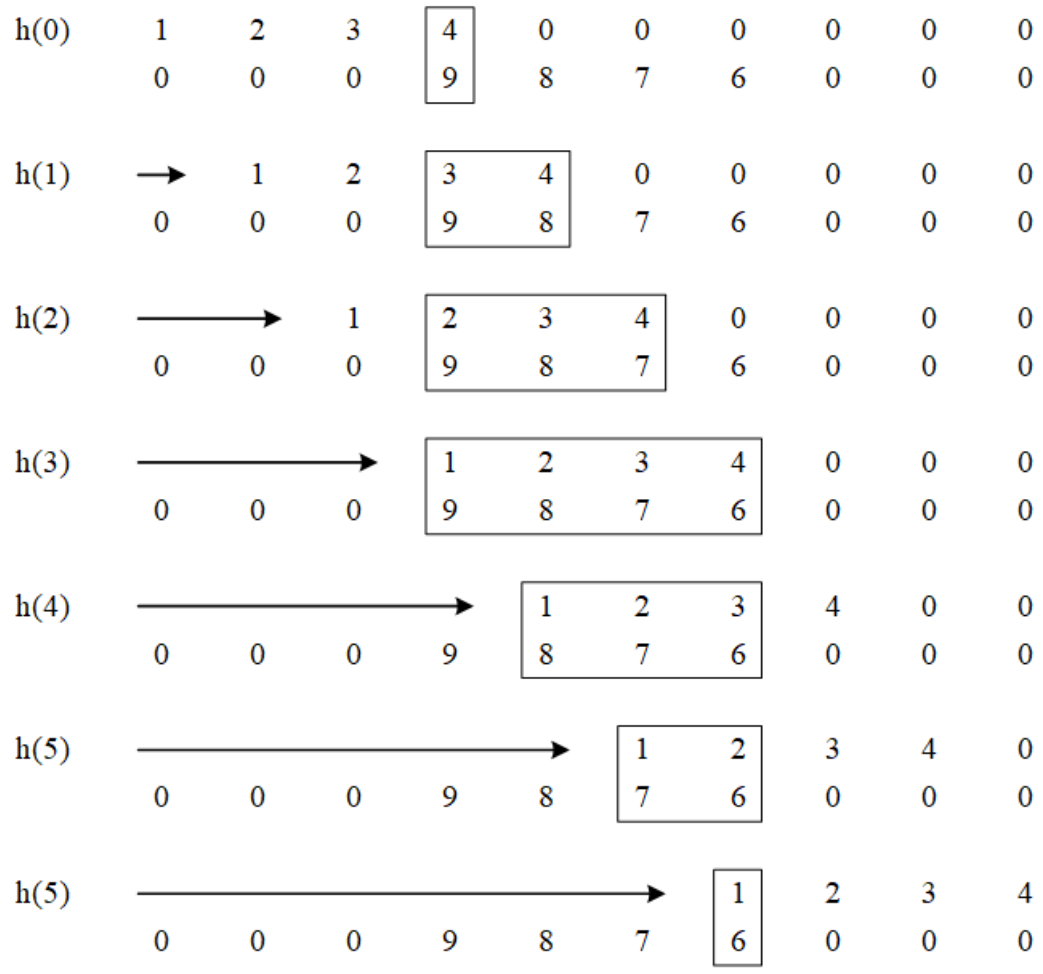

FIGURE 9. Multiplication via Convolution of $1234 \times 6789 = 8\,377\,626$

Comparing the partial products summed for each coefficient to the grade school multiplication columns in Figure 8, we find that the partial products summed for each coefficient are exactly the same. Convolution of polynomial coefficients for base power expression of two numbers is exactly the same as multiplication[4].

Obviously, from here and our examples before, the time complexity of the convolution operation is $\mathcal{O}(n^2)$. Let us now turn to the Fast Fourier Transform and how we may quickly compute the DFT using the FFT.

6.2. **The Fast Fourier Transform.** The Fast Fourier Transform algorithm created by Cooley and Tukey in 1965 [4] creates a divide-and-conquer method to

[4]Note that convolution of two functions/signals is not the same as multiplication - we have cleverly expressed our multiplicands as polynomials using base powers and coefficients and for that specific case, convolution is the same as multiplication. However, in general, that is not the case.

compute the DFT and take advantage of the symmetry and rotation properties of the complex exponentials used in the DFT. Recall the equation for the DFT:

$$X(m) = \sum_{n=0}^{N-1} x(n) e^{-i2\pi nm/N}.$$

And the twiddle factor expression:

$$W_N = e^{-i2\pi/N}.$$

The DFT with the twiddle factor then becomes:

$$X(m) = \sum_{n=0}^{N-1} x(n) W_N^{mn}.$$

And finally, the FFT will make heavy use of the symmetry and periodicity properties of complex exponentials [12]:

$$\textbf{Symmetry property: } W_N^{m+N/2} = -W_N^m$$
$$\textbf{Periodicity property: } W_N^{m+N} = W_N^m$$

The FFT begins by dividing the DFT summation into the even ($2n$) and odd ($2n$+1) terms:

$$\begin{aligned} X(m) &= \sum_{n=0}^{N-1} x(n) W_N^{mn} \\ &= \sum_{n=0}^{N/2-1} x(2n) W_N^{(2n)m} + \sum_{n=0}^{N/2-1} x(2n+1) W_N^{(2n+1)m} \\ &= \sum_{n=0}^{N/2-1} x(2n) W_N^{2nm} + W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_N^{2nm} \end{aligned}$$

where we have simply factored out the constant W_N^m from the $(2n+1)$ odd term in the second summation. We can use the periodicity property to simplify the expression further. Because

$$W_N^{2m} = e^{-i2\pi(2m)/N} = e^{-i2\pi m/(N/2)} = W_{N/2}^m$$

we can write

$$X(m) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} + W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm} \tag{6.4}$$

where m is in the range of 0 to $N/2-1$. We can substitute $(m+N/2)$ for m to obtain the other half of the equation:

$$X(m+N/2) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{n(m+N/2)} + W_N^{(m+N/2)} \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{n(m+N/2)}. \tag{6.5}$$

As Lyons notes in [11], this looks worse, but we can use our complex exponential properties to reduce the complexity of the equations.

$$W_{N/2}^{n(m+N/2)} = W_{N/2}^{nm} W_{N/2}^{n(N/2)} = W_{N/2}^{nm} e^{-i2\pi n(N/2)/(N/2)} \tag{6.6}$$

$$= W_{N/2}^{nm} e^{-i2\pi n} = W_{N/2}^{nm}(1) = -W_{N/2}^{nm} \tag{6.7}$$

as $e^{-i2\pi n} = 1$ for any integer n. Similarly, we may reduce the twiddle factor in front of the second summation:

$$W_N^{(m+N/2)} = W_N^m W_N^{N/2} = W_N^m e^{-i2\pi(N/2)/N} \tag{6.8}$$

$$= W_N^m e^{-i\pi} = W_N^m(-1) = -W_N^m. \tag{6.9}$$

We may now substitute 6.6 and 6.8 into 6.5 to simplify the second half of the FFT:

$$X(m+N/2) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} - W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm}. \tag{6.10}$$

Finally, let's write the equations for the first $N/2$ points from equation 6.4 and second $N/2$ points from equation 6.10 together:

$$X(m) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} + W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm} \tag{6.11}$$

$$X(m+N/2) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} - W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm}. \tag{6.12}$$

We can now see how similar these two equations are, and start to see how this divide and conquer vastly reduces our computations. In practice, the number of bits for a particular multiply is fixed, and this allows us to compute the twiddle factors ahead of time and store them in a lookup table. Let's continue the divide-and-conquer approach. We can further split the even and odd summations:

$$\sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} = \sum_{n=0}^{N/4-1} x(4n) W_{N/2}^{2nm} + \sum_{n=0}^{N/4-1} x(4n+2) W_{N/2}^{(2n+1)m}.$$

We can use $W_{N/2}^{2nm} = W_{N/4}^{nm}$ to simplify these two $N/4$-point DFTs:

$$\sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} = \sum_{n=0}^{N/4-1} x(4n) W_{N/4}^{nm} + W_{N/2}^m \sum_{n=0}^{N/4-1} x(4n+2) W_{N/4}^{nm}.$$

Similarly,

$$\sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm} = \sum_{n=0}^{N/4-1} x(4n+1) W_{N/4}^{nm} + W_{N/2}^m \sum_{n=0}^{N/4-1} x(4n+3) W_{N/4}^{nm}.$$

At this point a graphic may be instructive. We can think of the FFT as a set of recursive equations where we work backward through the DFT summation, halving the computed sum at each step. Figure 10 shows how this would look for an 8-point FFT. Decimation in time refers to the fact that we are halving the computed time series at each step - how we separate the summation into $(2n)$ and $(2n+1)$ samples. If we separated the DFT output series into even and odd samples then we would

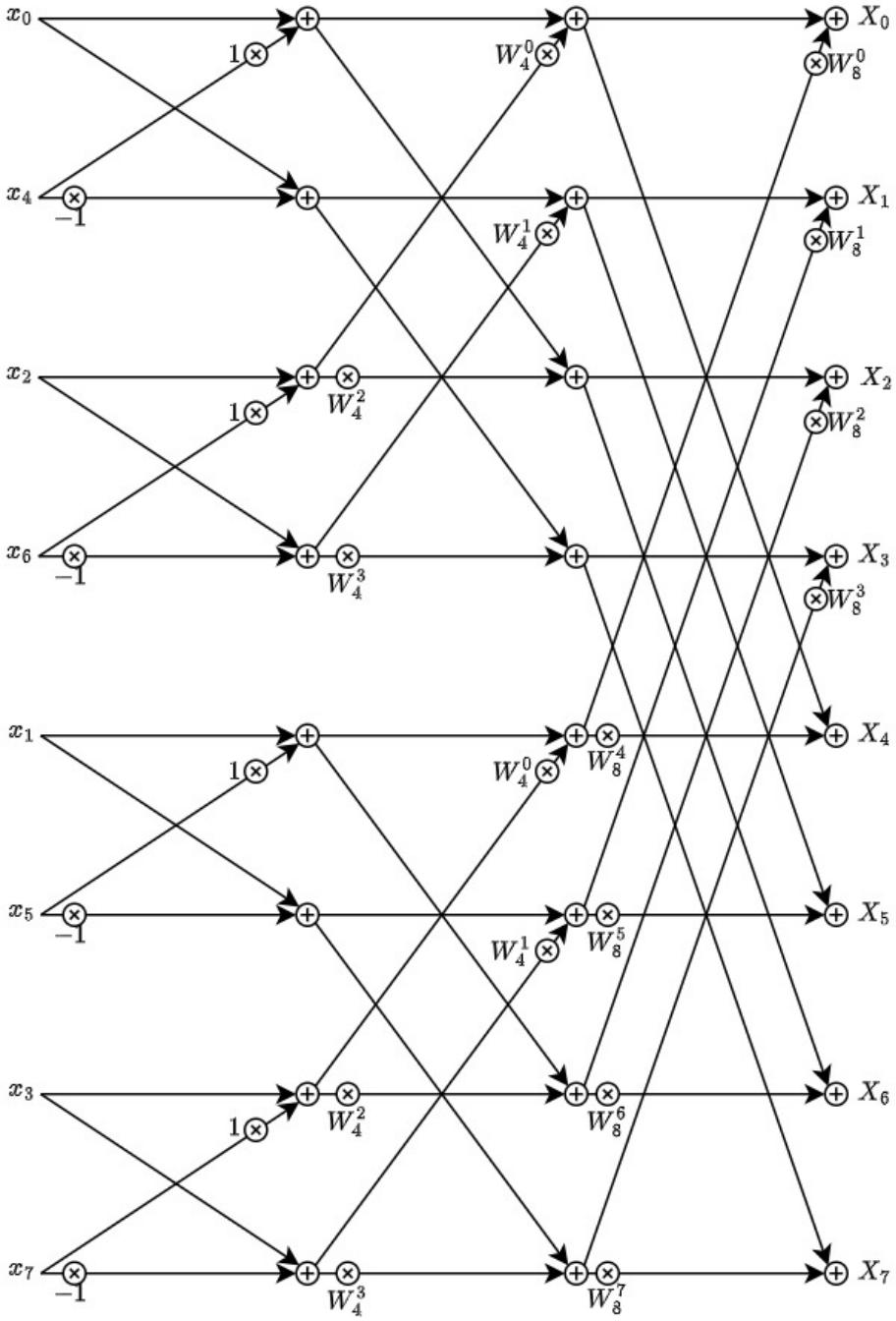

FIGURE 10. 8-point Decimation in Time FFT

be creating a decimation-in-frequency DFT. Both approaches are used in practice and yield the same result, aside from quantization error.

Figure 10 shows an 8-point DFT with its various butterfly stages. Reconciling the decimation in time FFT butterfly stages with the equations is not necessarily intuitive. In a moment, we will enumerate the various butterflies at each stage to show how the computations break down. For now, what is intuitive is the $\mathcal{O}(n)$ multiplications at each stage, and that there are $log_2 n$ total stages to the radix-2 decimation-in-time FFT. Hence the FFT is a $\mathcal{O}(n \log n)$ algorithm.

Now let's turn to computing the FFT at each stage. We start with the output, which we know is simply

$$X(m) = \sum_{n=0}^{N-1} x(n) W^{nm}.$$

We split this into the two equations shown into a summation of even and odd time samples (decimated in time), shown below for reference:

$$X(m) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} + W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm}$$

$$X(m+N/2) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nm} - W_N^m \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nm}.$$

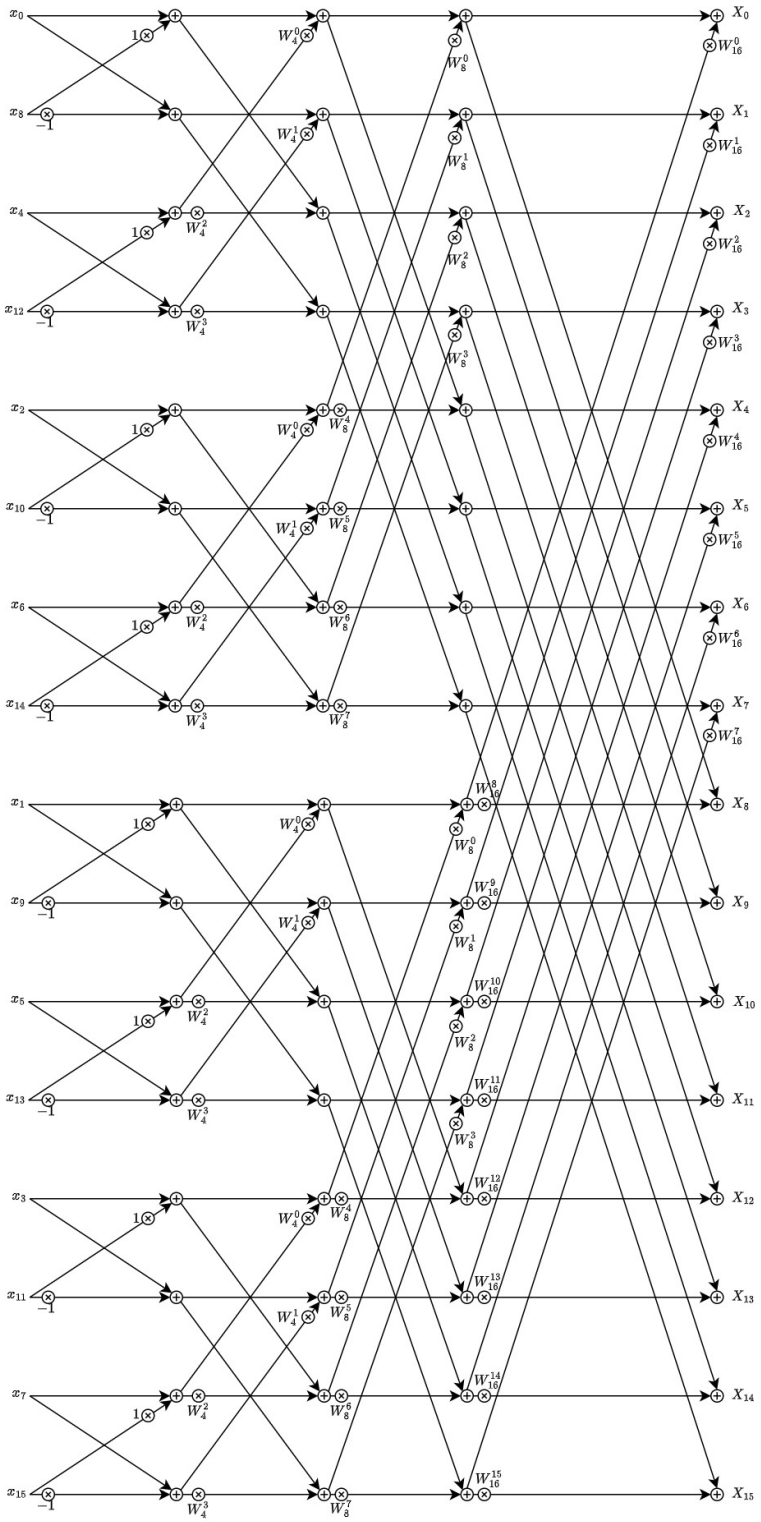

FIGURE 11. 16-point Decimation in Time FFT

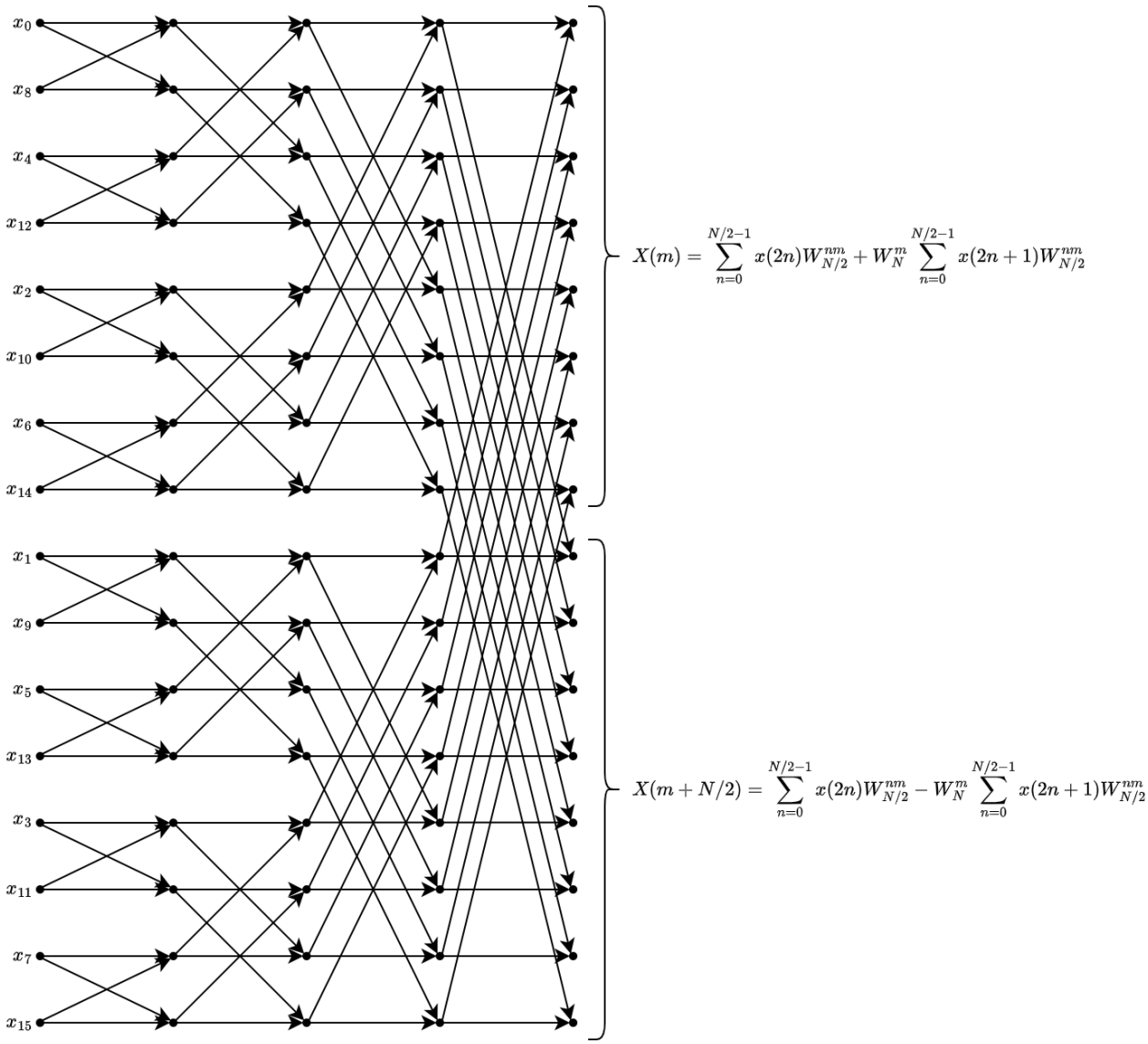

FIGURE 12. 16-point Decimation in Time FFT Stage 4

Remember, we go to the left to halve our summations again. This brings us to the third stage in our 16-point FFT, where we have divided the total points, 16 in this case, by four ($N/4$). Figure 13 shows these four summations.

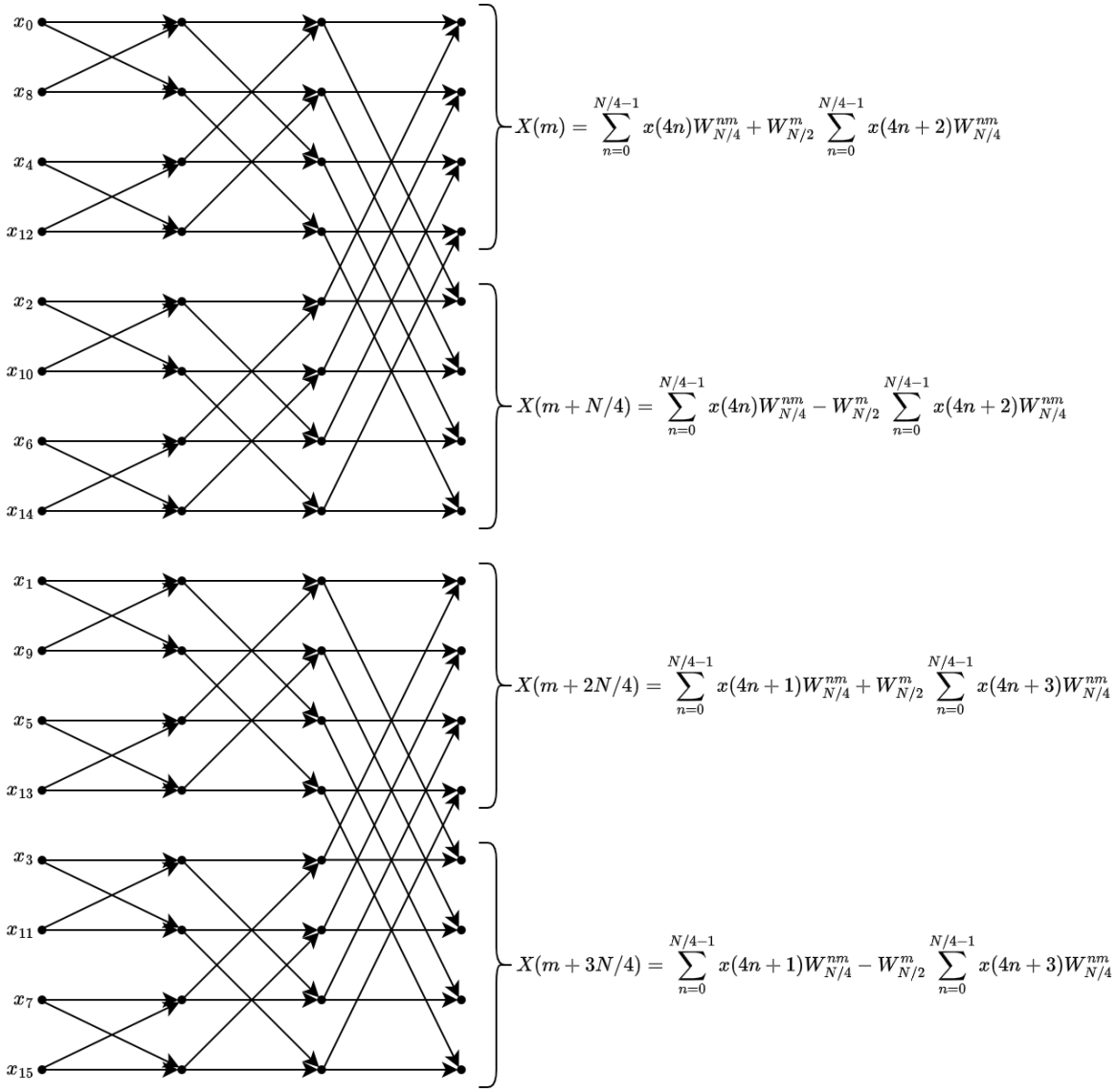

FIGURE 13. 16-point Decimation in Time FFT Stage 3

Dividing our sums again we move one more stage to the left to the second stage in our 16-point FFT. Not that here we have eight summations. Figure 14 shows the summations for the second stage of our 16-point FFT.

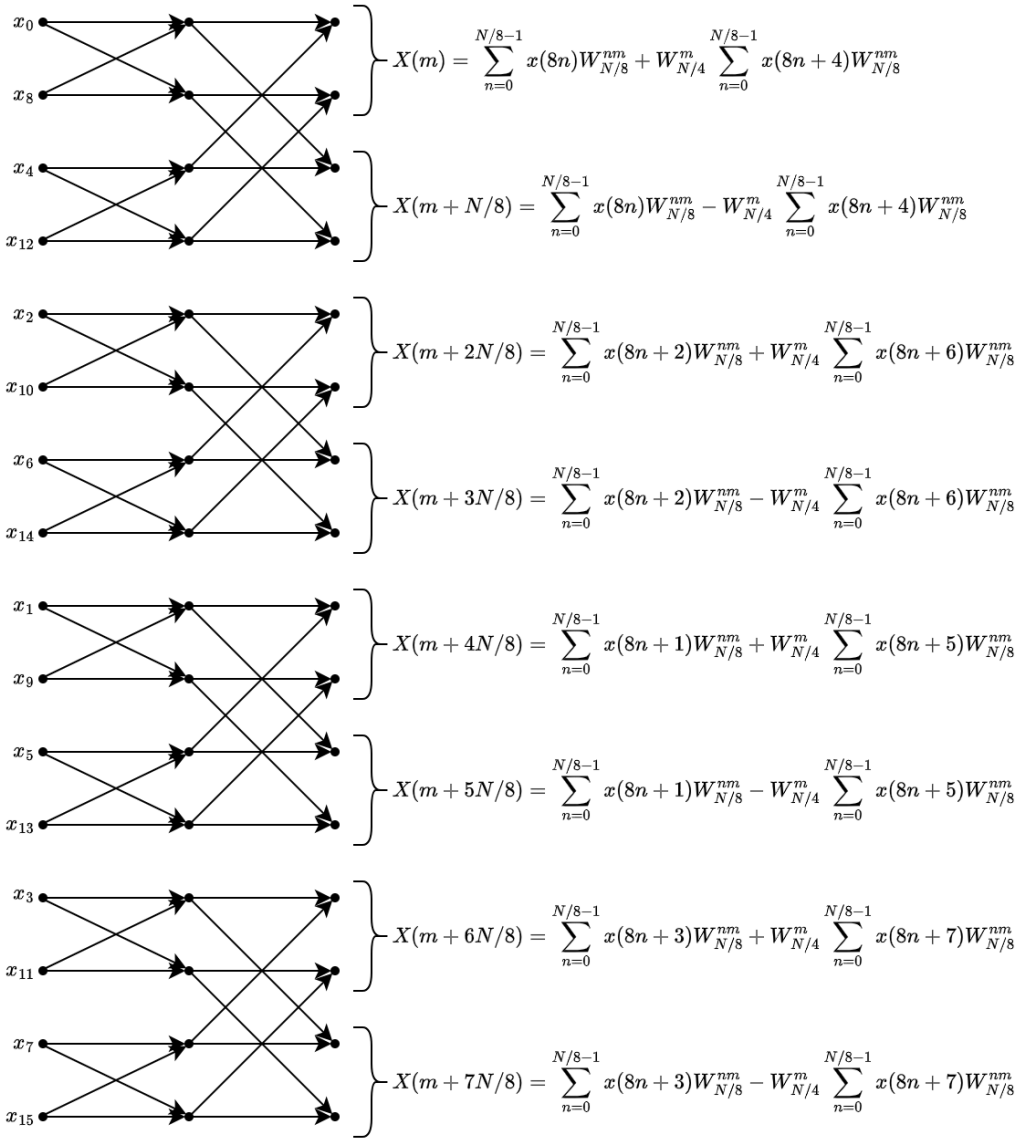

FIGURE 14. 16-point Decimation in Time FFT Stage 2

Finally, we move once again to the left, arriving at the first stage of our 16-point FFT, shown in Figure 15. Although we have kept the sums general, it is trivial to derive the sum for this first stage. For instance, looking at the first equation $X(m)$, we have

$$X(m) = \sum_{n=0}^{N/16-1} x(16n)W_{N/16}^{nm} + W_{N/8}^{m} \sum_{n=0}^{N/16-1} x(16n+8)W_{N/16}^{nm}$$

$$X(0) = x(0) \cdot W_1^0 + W_2^0 \cdot x(8) \cdot W_1^0 = x(0) + x(8)$$

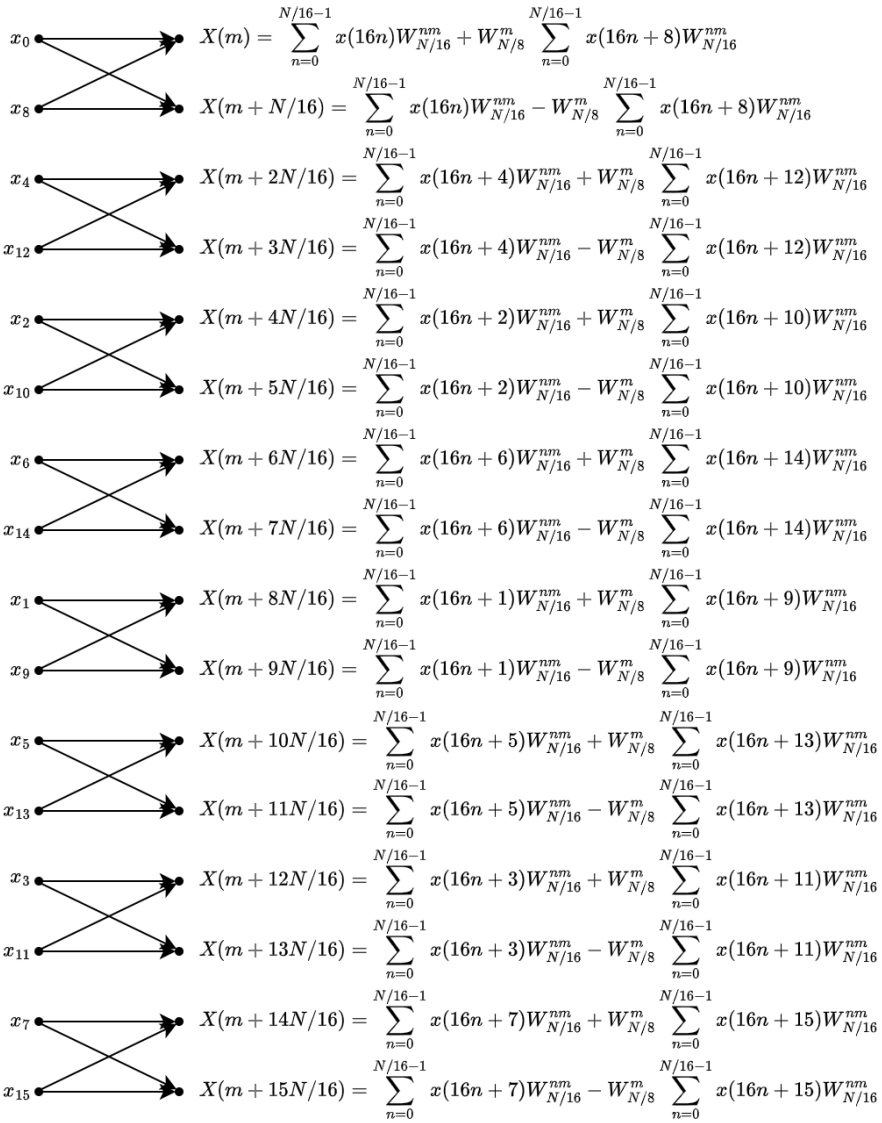

FIGURE 15. 16-point Decimation in Time FFT Stage 1

Similarly, we can derive the second point $X(1)$:

$$X(m) = \sum_{n=0}^{N/16-1} x(16n)W_{N/16}^{nm} - W_{N/8}^{m} \sum_{n=0}^{N/16-1} x(16n+8)W_{N/16}^{nm}$$

$$X(1) = x(0) \cdot W_1^0 - W_2^1 \cdot x(8) \cdot W_1^0 = x(0) - e^{-j2\pi/2} \cdot x(8) = x(0) + x(8)$$

We derive $X(2)$ and $X(3)$ for further demonstration:

$$X(m) = \sum_{n=0}^{N/16-1} x(16n+4)W_{N/16}^{nm} + W_{N/8}^{m} \sum_{n=0}^{N/16-1} x(16n+12)W_{N/16}^{nm}$$

$$X(2) = x(4) \cdot W_1^0 + W_2^2 \cdot x(8) \cdot W_1^0 = x(12) + e^{-j2\pi 2/2} \cdot x(12) = x(4) + x(12)$$

$$X(m) = \sum_{n=0}^{N/16-1} x(16n+4)W_{N/16}^{nm} - W_{N/8}^{m} \sum_{n=0}^{N/16-1} x(16n+12)W_{N/16}^{nm}$$

$$X(3) = x(4) \cdot W_1^0 - W_2^1 \cdot x(12) \cdot W_1^0 = x(4) - e^{-j2\pi 3/2} \cdot x(12) = x(4) + x(12)$$

and so forth.

6.3. **Algorithm** The recursive FFT algorithm from [7] is reproduced in Algorithm 6. Note that typically n is known ahead of time and we can precompute the maximum W_N roots of unity and use that set to derive all others (e.g. $W_4^2 = W_8^4$).

Algorithm 6 Fast Fourier Transform Decimation-in-Time

Input: a, an n-length coefficient vector where n is a power of 2, and w, a primitive root of unity

Output: A vector, y, of values of the polynomial for a at the nth roots of unity.

```
if n = 1 then
    return y = a
end if
x ← w_0
a_even ← [a_0, a_2, a_4, ..., a_{n-2}]
a_odd ← [a_1, a_3, a_5, ..., a_{n-1}]
y_even ← FFT(a_even, w^2)
y_odd ← FFT(a_odd, w^2)
for i ← 0 to n/2 − 1 do
    y_i ← y_{i,even} + x · y_{i,odd}
    y_{i+n/2} ← y_{i,even} − x · y_{i,odd}
    x ← x · w
end for
return y
```

6.4. **Time Complexity Analysis** As we have given a large number of equations, examples, and diagrams, it should be clear that the FFT recursively divides summation calculations into half. Thus, the FFT decreases the stages of the DFT to $\mathcal{O}(\log n)$, with $\mathcal{O}(n)$ operations in each stage. In total this gives a run time complexity of $\mathcal{O}(n \log n)$. In practice the FFT is highly efficient as the number of points is known ahead of time and all twiddle factors may be precomputed and stored.

6.5. **The Inverse DFT and FFT.** The DFT has an inverse:

$$x(n) = \frac{1}{N} \sum_{m=0}^{N-1} X(m) e^{j2pim/N}$$

and the Inverse FFT derivation is much the same as the FFT. We leave this as an exercise for the reader. Note the $1/N$ term at the front. Fortunately, because we have restricted N to be a binary power, we may simply shift the result of the summation, which is the same as dividing by a binary power.

7. Multiplication via the FFT

Now that we have the tools of the FFT, multiplication via the FFT is rather straightforward[5]. Given our digits, we pad both digits to some binary power D, then zero-pad to $2D$. This is to account for the typical $2n$ bits necessary from an n-bit multiply. Given that we are using the radix-2 FFT, we extend n to D and the $2n$ to $2D$. We then perform the FFT on our input coefficients, multiply each of the coefficients together, then perform the IFFT. We then adjust our carries across bases and are finished.

7.1. **Algorithm** Algorithm 7, modified slightly from from [6], provides the basic algorithm for multiplying two positive integers using the FFT and IFFT.

Algorithm 7 Basic FFT Multiplication

Output: Given two polynomial coefficient vectors x, y with length less than or equal to a binary power D.
Zero-pad x, y to length $2D$.
$X = FFT(x)$
$Y = FFT(y)$
$Z = X \cdot Y$ (element-wise multiplication)
$z = IFFT(z)$
$z = round(z)$ (See section 7.2 regarding this step.)
$z = carry\_ones(z)$
$z = remove\_leading\_zeros(z)$
return z

For our example we can revisit the same multiplication from section 5, 698 310 488 572 646 777 019 184 × 144 585 992 498 882 884 065 634 = 100 965 915 062 655 948 833 325 499 910 140 535 809 533 122 656. First we set

$$x = \{4, 8, 1, ..., 9, 6\},$$
$$y = \{4, 3, 6, ..., 4, 1\}.$$

A quick count shows $max(x, y) = 24$, so we set $D = 2^{\lceil log_2 24 \rceil} = 32$ and $2D = 64$, zero padding the coefficients for x and y. We can now take the FFT of x and y,

[5]Note that we are giving the simplest form of this FFT multiply. Section 7.2 discusses additional considerations for practical use of the algorithm.

$X = FFT(x)$ and $Y = FFT(y)$, respectively. The first and last few terms of each are:

$$X = \{121, 37.667 - 86.807i, -24.665 - 27.021i, ..., 37.666 + 86.807i\},$$
$$Y = \{130, 43.664 - 98.099i, -39.298 - 38.374i, ..., 43.664 + 98.099i\}.$$

We then multiply the elements of X and Y together to obtain Z,

$Z = X \cdot Y$,

$Z = \{15730.0, -6871.016 - 7485.332i, -67.622 + 2008.389i, ..., -6871.016 + 7485.332i\}$.

We then take the IFFT of Z to obtain z and round the results:

$$z = IFFT(Z),$$
$$z = round(z),$$
$$z = \{16, 44, 52, ..., 68, 33, 6, 0, ..., 0\}.$$

At this point we carry the ones from each base power to the next in our resultant coefficient polynomial:

$$z = \{6, 5, 6, ..., 9, 0, 0, 1, ..., 0\}.$$

We can then remove all the leading zeroes, or remove all consecutive zero entries descending through the polynomial coefficient degrees until we reach the first non-zero entry. At that point, we have our answer:
698 310 488 572 646 777 019 184 × 144 585 992 498 882 884 065 634 =
100 965 915 062 655 948 833 325 499 910 140 535 809 533 122 656.

7.2. **Floating-Point Complications and Time Complexity Analysis** There are multiple issues using the FFT to perform large integer multiplication. First, as the interim results from the prior section showed, the FFT requires complex, floating-point values. As we also saw, even though we are using single-digit coefficients, the FFT produces values which are very large for a single coefficient. There are two issues: enough numerical precision for our primitive roots of unity, and preventing overflow within each of the coefficients.

The FFT is remarkably well behaved in terms of stability[9], but we must have enough bits to ensure that we carry enough precision through the FFT calculations[12]. At a minimum, we need to be able to differentiate twiddle factors, or points on the unit circle. The methods to determine when to split these calculations further to provide smaller blocks of numbers or more precision are quite involved. Interested parties are referred to [2] which provides a helpful amount of detail. These details involving the FFT computations for arbitrarily-sized numbers require another $\log \log n$ term in our big O notation, yielding a final time complexity of $\mathcal{O}(n \log n \log \log n)$[2, 9, 12].

8. Modulo Arithmetic and Prime Numbers

Fourier was primarily concerned with decomposing time-domain signals into their frequency domain content via sines and cosines to understand heat flow [1], and the Discrete Fourier Transform is the discrete version of the Fourier transform and indeed gives cyclic frequency domain content via Euler's $e^{-i2\pi n/N}$. From a mathematical point of view, the DFT, and by extension the FFT, can continue to carry out transforms so long as we use a primitive root of unity in the transform and that root has an inverse. In this paper so far we have used the complex

exponential $e^{-i2\pi n/N}$ as our primitive root of unity in the DFT/FFT, but there are other primitive roots of unity using primes and modulo arithmetic.

We may more abstractly define the DFT of a signal x of length D, where D^{-1} exists, and g, a primitive D-th root of D exists, then the DFT of x, $X = DFT(x)$ is defined as:

$$X(m) = \sum_{n=0}^{D-1} x(n) g^{-jm}$$

and if using modulo arithmetic for some prime p, then

$$X(m) = \sum_{n=0}^{D-1} x(n) g^{-jm} \mod p.$$

If we use modulo arithmetic instead of the complex exponential as our primitive root of unity g, then we call the transform a Number-Theoretic Transform (NTT). There are a large number of NTTs used in the multiplication of large integers, typically dealing with prime numbers [1, 5, 6, 8, 13, 14]. The most widely-used NTT for large integer multiplication is Schönhage-Strassen. Here the NTT is sometimes called a Fermat Number Transform (FNT)[6].

For background regarding the multitude of necessary prime number properties requisite to compute the FNT in Schönage-Strassen, we highly recommend reviewing [13].

9. Schönhage-Strassen

The Schönhage-Strassen algorithm typically uses a prime number in the format $2^n - 1$ generating a finite field/ring $\mathbb{Z}_{2^n+1}$. The primitive root of unity must be a generator for the chosen prime. Schönhage-Strassen is typically used for numbers with hundreds of thousands of digits. Because the complex FFT is so well behaved, the complex FFT is used even though Schönhage-Strassen does not have the same issues with floating-point precision.

The 2^n-1 prime, when coupled with other restrictions on the generator, grouping of digits, and length of the padded input vectors, yield several nice properties for the algorithm that reduce all operations aside from the multiplication of transformed numbers to shifts and adds, which helps to compensate for some of the complexity of the algorithm [6].

For our example, we will pick a small prime and generator to illustrate the algorithm with our previous example of $1234 \times 6789 = 8\,377\,626$. We will use the prime $p = 337$ and primitive root of unity/generator $w = 85$. We group single decimal digits for ease, and take the NTT of our input vectors:

$$X = NTT(x) = \{10, 329, 298, 126, 2, 271, 43, 301\}$$
$$Y = NTT(y) = \{30, 32, 298, 34, 2, 36, 43, 271\}.$$

We can then point-wise multiply our individual transforms modulo p to obtain the polynomial coefficients of our result in $\mathbb{Z}_p$:

$$Z = X \cdot Y \mod p$$
$$Z = \{300, 81, 173, 240, 4, 320, 164, 17\}.$$

[6]In honor of Fermat's contributions rather than having anything to do with a Fermat number.

We then take the inverse Number Theoretic Transform or INTT:

$$z = INTT(Z)$$
$$z = \{36, 59, 70, 70, 40, 19, 6, 0\}.$$

From here we then simply perform our typical carry operation to yield the final answer:

$$z = carry\_ones(z)$$
$$z = \{6, 2, 6, 7, 7, 3, 8, 0\}.$$

We can then strip leading zeroes and write our final answer, $1234 \times 6789 = 8\,377\,626$.

9.1. **Algorithm** The algorithm here is taken from [6] and adapted slightly to be consistent with the notation we have been using throughout the paper.

Algorithm 8 Schönhage-Strassen FNT

Input: Given $0 \leq x, y < 2^n + 1$, this algorithm returns $xy \mod (2^n + 1)$.
Choose NTT size $D = 2^k | n$.
With $n = DM$, set a recursion length $n' \geq SM + k$ such that $D|n'$ (e.g. $n' = DM'$).
Split x and y into D parts of M bits each, and store these parts, considered residues modulo $(2^{n'} + 1)$ into two respective arrays $X_0, X_1, ..., X_{D-1}$ and $Y_0, Y_1, ..., Y_{D-1}$. Note that each element could have $n' + 1$ bits later on.
for $0 \leq j < D$ **do**
  $X_j = (2^{jM'} X_j) \mod (s^{n'+1})$
  $Y_j = (2^{jM'} Y_j) \mod (s^{n'+1})$
end for
$X = NTT(X)$ (Use $2^{2M'}$ as D-th root $\mod (2^{n'1} + 1)$)
$Y = NTT(Y)$
for $0 \leq j < D$ **do**
  $Z_j = X_j Y_j \mod (2^{n'} + 1)$
end for
$z = NTT(Z)$
for $0 \leq j < D$ **do**
  $c_j = Z_{D-j}/2^{k+jM'} \mod (2^{n'} + 1)$
  if $c_j > (j+1)2^{2M}$ **then**
    $c_j = c_j - (2^{n'} + 1)$ (c_j may be negative)
  end if
end for
Perform carry operations as done previously for c_j's.
return $z \mod (2^n + 1)$

Note that above we have used NTT to denote the Number Theoretic Transform that is the finite-field FFT.

9.2. **Time Complexity Analysis** Similar to the complex FFT, the Schönhage-Strassen algorithm yields a complexity of $\mathcal{O}(n \log n \log\log n)$. The $n \log n$ term is rather obvious as it is the same as the complex FFT. The $\log\log n$ term comes from having to reduce the size of the interim products with the NTT. This extra complexity of $\log\log n$ yields a final time-complexity for the Schönhage-Strassen algorithm of $\mathcal{O}(n \log n \log\log n)$.

10. Conclusion

We have provided a survey of large integer multiplication algorithms, tracing from grade school multiplication to the first advancement with Karatsuba, then Toom-Cook (specifically Toom-3), and culminating with advanced approaches that built on convolution and transforms using primitive roots of unity and the complex FFT, culminating in finite-field arithmetic with Schönhage-Strassen. These algorithms took us from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n \log\log n)$. In 2019 David Harvey and Joris van der Hoeven found an $\mathcal{O}(n \log n)$ method [8]. The method is highly specialized and due to a multitude of complicated constants, does not eclipse the methods we have already covered until galactic-scale numbers come into play (millions of digits and beyond). Accordingly, the algorithms presented in this paper are still relevant and used in big number code libraries.

We hope this paper will prove a useful introduction and reference to large integer multiplication methods for those interested in the subject.

Acknowledgments

I would like to extend my heartfelt thanks to Dr. Huang for allowing me to pursue an open-ended survey topic and providing guidance and motivation along the way. I am aware that an undergraduate survey of existing algorithms has few tangible benefits for a professor, but this has been a tremendously rewarding experience for me. Thank you for patience during the weeks where I was making no visible progress. Thank you for your mentorship and gentle steering during this survey. Most of all, thank you for sharing your passion for algorithms and mathematics. That passion is infectious and has made working with you not just a rewarding, but joyful experience.

References

[1] Jean Baptiste Joseph Baron Fourier et al. *The analytical theory of heat*. Courier Corporation, 2003.

[2] Richard P Brent and Paul Zimmermann. *Modern computer arithmetic*. Vol. 18. Cambridge University Press, 2010.

[3] David M Burton. "The history of mathematics: An introduction". In: *Group* 3.3 (1985), p. 35.

[4] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[5] Richard E Crandall. "Integer convolution via split-radix fast Galois transform". In: *Center for Advanced Computation Reed College* (1999).

[6] Richard E Crandall and Carl Pomerance. *Prime numbers: a computational perspective*. Vol. 2. Springer, 2005.

[7] Michael T Goodrich and Roberto Tamassia. *Algorithm design and applications*. Vol. 363. Wiley Hoboken, 2015.

[8] David Harvey and Joris Van Der Hoeven. "Integer multiplication in time O(nlog\,n)". In: *Annals of Mathematics* 193.2 (2021), pp. 563–617.

[9] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[10] Anatolii Alexeevich Karatsuba. "The complexity of computations". In: *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211 (1995), pp. 169–183.

[11] Richard G Lyons. *Understanding digital signal processing, 3/E*. Pearson Education India, 1997.

[12] John G Proakis and Dimitris G Manolakis. *Digital signal processing: principles, algorithms, and edition*. 1995.

[13] Joseph H Silverman. *A friendly introduction to number theory*. Pearson, 2014.

[14] Charles Van Loan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[15] Wikipedia. *Karatsuba algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-November-2023]. 2023. URL: `http://en.wikipedia.org/w/index.php?title=Karatsuba%5C%20algorithm%5C&oldid=1171220798`.

University of Illinois at Springfield
Email address: `jt.miller@txbinary.com`