Multiplication of Big Numbers

Ermin Hodžić

Jun 2023

In this text, we are going to be dealing with what we call *big numbers*. Such numbers far exceed the maximum value supported by fundamental single-variable data types in programming languages, such as the 64-bit integer, or the double type. Within a 64-bit integer variable, we are able to store numbers of up to 19 decimal digits. However, what do we do when that is no longer enough? If we, for whatever purpose, need to store numbers that have potentially hundreds or even thousands of digits, we can no longer rely on the basic built-in data types provided by programming languages, as there are no machine instructions that operate on such big numbers. Thus, we have to resort to writing our own code that implements such big numbers and the appropriate basic operations, such as addition and multiplication.

Before going into the technical details of implementing addition and multiplication on such numbers, we are going to talk about how dealing with big numbers actually reduces to dealing with polynomials. We do this because polynomials provide us with more flexibility, as they are not as restricted in their structure as numbers themselves are. Afterwards, we will go through a number of increasingly faster algorithms used to multiply polynomials.

Requirements. The final section of the manuscript is heavy on mathematics theory, containing a lot of linear algebra, complex numbers, trigonometry and group theory. All of it is necessary to understand how and why the *Fast Fourier Algorithm* works. If the mathematical proofs and analyses are too difficult, it should still be possible to understand the idea behind the algorithm, but it may not all "click" together without fully understanding the mathematical theory behind it that makes it work.

Contents

1 Numbers are Polynomials

A polynomial is an expression consisting of a linear combination (addition, subtraction and multiplication) of (non-negative) powers of variables. A general form of an n-order univariate (with only a single variable) polynomial P with the variable x would be:

$$P(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

Coefficients a_i can take any value, but it is assumed that a_n is not zero, as then the polynomial would not be of the order n, but lower. Polynomials are used for algebraic representations of some concepts in mathematics. For example, in geometry, an order-1 univariate polynomial would represent a straight line on a plane:

$$P(x) = a_1 x + a_0$$

Whereas an order-1 multi-variate polynomial would represent a (possibly higher-order) plane. In the three-dimensional space, that would be:

$$P(x, y) = ax + by + c$$

More relevant to our topic, we can also use polynomials to represent numbers, if we set the coefficients of the polynomial to the number's digits, and if the variable represents the base of the numeric system of the number. For example, the decimal number 52658 can be represented by a polynomial where the coefficients are its digits:

$$P(x) = 5x^4 + 2x^3 + 6x^2 + 5x + 8$$

If we evaluate $P(x)$ in the point $x = 10$, by substituting every occurrence of the variable x in the expression with the number 10, we will obtain the exact decimal numerical value of

$$52658 = 5 \cdot 10^4 + 2 \cdot 10^3 + 6 \cdot 10^2 + 5 \cdot 10 + 8$$

Now let us observe two numbers 123 and 456, and their respective polynomials $P(x)$ and $Q(x)$:

$$P(x) = 1 \cdot x^2 + 2 \cdot x^1 + 3 \cdot x^0$$
$$Q(x) = 4 \cdot x^2 + 5 \cdot x^1 + 6 \cdot x^0$$

Adding these two polynomials is straightforward – we simply perform pair-wise addition of the coefficients of the same order.

$$\begin{aligned} P(x) + Q(x) &= (1+4) \cdot x^2 + (2+5) \cdot x^1 + (3+6) \cdot x^0 \\ &= 5 \cdot x^2 + 7 \cdot x^1 + 9 \cdot x^0 \end{aligned}$$

We see that $P(x)+Q(x)$ corresponds to the number 579, which is indeed the sum of numbers 123 and 456. This example works without any complications because the pair-wise coefficient sums all result in numbers which are smaller than the base of the numeric system (in our case, the number 10). Let us examine how to handle the opposite case.

$$\begin{aligned} Q(x) + Q(x) &= (4+4) \cdot x^2 + (5+5) \cdot x^1 + (6+6) \cdot x^0 \\ &= 8 \cdot x^2 + 10 \cdot x^1 + 12 \cdot x^0 \end{aligned}$$

The result of adding 456 and 456 is 912, not 81012, so we cannot just directly read the coefficients of the resulting polynomial as the resulting number. This is because the coefficients are breaking the rule for numbers that no digit can be bigger than the base of the numeric system. Examining our resulting polynomial closer, which apparently has "12 ones", "10 tens" and "8 ten squares", we can see that the problem is one of ignoring the *carry over* within the coefficients. If we have "12 ones", what we really have is

"1 ten and 2 ones", and "1 ten" has no business being inside the coefficient that is reserved only for "ones", so the "1 ten" needs to carry over into the coefficient of the immediately higher order. Thus, our polynomial should look more like the following:

$$Q(x) + Q(x) = 8 \cdot x^2 + 11 \cdot x^1 + 2 \cdot x^0$$

Since the first-order coefficient 11 has the same problem, we need to carry over the higher-order part of its own value further up the chain. Doing so, we obtain:

$$Q(x) + Q(x) = 9 \cdot x^2 + 1 \cdot x^1 + 2 \cdot x^0$$

Finally, all the coefficients now represent proper decimal digits, and indeed they match the expected result of 912. From the above, we see how to handle polynomial addition, and how to get the resulting number from it. In the remainder of the text, to make both the code and the analysis simpler, we will make the assumption that the two polynomials are of the same order. This can be easily made true in practice by padding the lower-order polynomial with zeroes as its high-order coefficients. Additionally, we will assume that the conversion of the polynomial into a valid number is done as a post-processing step by a separate function, and we will not concern ourselves with that for the remainder of the text.

```
void polynomial_add(const int * A, const int * B, const int n, int * C) {
    for (int i = 0; i < n; i++) C[i] = A[i] + B[i];
}
```

```
void carryOver(int * A, int & n) {
    // We assume that the array A has already been allocated large enough to not cause
    overflow
    int carryOver = 0;
    for (int i = 0; i < n; i++) {
        A[i] += carryOver;
        carryOver = A[i] / 10;
        A[i] %= 10;
    }
    while (carryOver) {
        A[n] = carryOver;
        carryOver = A[n] / 10;
        A[n] %= 10;
        n++;
    }
}
```

There isn't really a way to make polynomial addition any faster than the linear algorithm we gave above. At the very least, we have to read the coefficients of the polynomials once, which represents a tight lower bound on the total addition time. However, multiplication is a different story, and provides a great example of how clever utilization of both the classic algorithmic techniques and mathematics provides significant running time improvements. We are going to go through these methods, from the slowest and most straightforward one to increasingly more complex (no pun intended! (later, it will be clear why I wrote this remark)) and fast algorithms.

2 The Naive Multiplication

In this approach, we simply multiply every term of one polynomial with every term of the other polynomial, and add up the terms that have the same order of the variables to obtain the final coefficients.

$$P(x) \cdot Q(x) = \sum_{i=0}^{n} p_i x^i \cdot \sum_{j=0}^{n} q_j x^j = \sum_{k=0}^{2n} x^k \Big(\sum_{i+j=k} p_i q_j \Big)$$

```
void polynomial_multiply(const int * A, const int * B, const int n, int * C) {
    // We assume that the array C has already been allocated large enough to not cause
    overflow
    memset( C, 0, (2*n - 2) * sizeof(C[0]) );
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i + j] += A[i] * B[j];
        }
    }
}
```

This algorithm is extremely simple and clean. Its running time, however, is $O(n^2)$, and it starts being very slow once the order of the two polynomials grows past a few thousands.

3 The Karatsuba Algorithm

We will now analyze a divide & conquer algorithm by Anatoly Karatsuba. The algorithm is based on the observation that multiplying two sums of two parts does not, in fact, require 4 multiplications. To illustrate, observe the following:

$$(a+b)\cdot(c+d) = ac + ad + bc + bd$$

Following this formula, we require 4 unique multiplications and 4 additions in order to compute the result. However, we can rewrite two of the terms the following way:

$$\begin{aligned} ad + bc &= ad + bc + (ac - ac) + (bd - bd) \\ &= (ac + ad + bc + bd) + (-ac - bd) \\ &= (a+b)(c+d) - ac - bd \end{aligned} \tag{1}$$

Thus, we have:

$$\begin{aligned} (a+b)\cdot(c+d) &= ac + ad + bc + bd \\ &= ac + bd + (ad + bc) \\ &= ac + bd + ((a+b)(c+d) - ac - bd) \end{aligned}$$

While this expression looks more complex, the important thing to note is that the number of required multiplications for computing the result is no longer 4. Now, we only need to perform 3 multiplications: ac, bd and $(a+b)(c+d)$. This is a good thing, because we have already seen that addition (and equivalently, subtraction) is done in $O(n)$ time, and that multiplication is the slow part. At this point, you might be wondering how the above expression is supposed to help us multiply polynomials in time faster than $O(n^2)$, when it's about the sum of numbers. Moreover, this expression may seem not only more complex but also useless, as the term $(a+b)\cdot(c+d)$ appears on both sides of the equation. To see why it is helpful, note that a polynomial is just a sum of terms, and that addition allows arbitrary grouping of terms. In other words, we can represent our polynomials as sums of their two halves. Assume that we split the polynomial in two parts with the split being made right before the m-th coefficient (for some $0 < m < n$):

$$P(x) = \sum_{i=0}^{n} p_i x^i = \sum_{i=0}^{m-1} p_i x^i + \sum_{i=m}^{n} p_i x^i = \sum_{i=0}^{m-1} p_i x^i + x^m \sum_{i=m}^{n} p_i x^{i-m}$$

$$\begin{aligned} P(x) &= a_0 x^0 + a_1 x^1 + ... + a_n x^n \\ &= (a_0 x^0 + ... + a_{m-1} x^{m-1}) + x^m (a_m x^0 + a_{m+1} x^1 + ... + a_n x^{n-m}) \end{aligned}$$

The two expressions in the parentheses also represent polynomials, but of a lower order. Let us denote these two polynomials that are composed of the lower-order and higher-order parts of $P(x)$ by $P_L(x)$ and $P_H(x)$, respectively. If we pick m to be equal to $\lfloor \frac{n}{2} \rfloor$, then both of these polynomials will have order roughly equal to half of that of the original polynomial.

$$P(x) = P_L(x) + x^m P_H(x)$$

Multiplication of two polynomials represented this way then looks like:

$$\begin{aligned} P(x) \cdot Q(x) &= (P_L(x) + x^m P_H(x))(Q_L(x) + x^m Q_H(x)) \\ &= P_L(x)Q_L(x) + x^m(\ P_L(x)Q_H(x) + P_H(x)Q_L(x)\ ) + x^{2m} P_H(x)Q_H(x) \end{aligned}$$

At last, we see the utility of Equation 1, which allows us to rewrite the middle term as:

$$P_L(x)Q_H(x) + P_H(x)Q_L(x) = (P_L(x) + P_H(x))(Q_L(x) + Q_H(x)) - P_L(x)Q_L(x) - P_H(x)Q_H(x)$$

We have thus divided the problem of calculating $P(x)Q(x)$ into 3 subproblems of the same form, to calculate $P_L(x)Q_L(x)$, $P_H(x)Q_H(x)$, and $(P_L(x) + P_H(x))(Q_L(x) + Q_H(x))$. Now we can use recursion to calculate the 3 products, and combine them into the final solution by raising the terms to the appropriate order (multiplications by x^m and x^{2m}) and adding them up together.

The running time of this algorithm will obviously depend on the value of m that we choose. By setting $m = \lfloor \frac{n}{2} \rfloor$, we obtain the optimal split and ensure that each subproblem has size at most half of the original one. The final act of adding the results from the subproblems together is done in linear time, and thus the total running time is then described by the recursive formula:

$$\begin{aligned} T(n) &= 3T(\frac{n}{2}) + O(n) \\ &= O(n^{\log_2 3}) \\ &\approx O(n^{1.59}) \end{aligned}$$

This represents a significant improvement over $O(n^2)$. Below, we give the code for the implementation of Karatsuba which uses a temporary storage array for intermediate calculations.

```
void Karatsuba(const int * A, const int * B, const int n, int * C) {
    // We assume that the array C has already been allocated as an array of size 2n
    // We place coefficients of A and B in the same array, placed one after the other
    for (int i = 0; i < n; i++) {
        C[i] = A[i];
        C[n + i] = B[i];
    }
    // The array temp is used to hold coefficients of (A_L + A_H) and (B_L + B_H) for their
    multiplication
    int * temp = new int [2 * n];
    KaratsubaRecursive(C, temp, n);
    delete temp;
}
```

```
void KaratsubaRecursive(int * A, int * temp, const int n) {
    int * B = A + n;
    if (n == 1) {
        A[0] = A[0] * B[0];
        A[1] = 0;
    }
    else {
        // Calculate (A_L + A_H) and store it in temp
        for (int i = 0; i < n/2; i++)
            temp[i] = A[i] + A[i + n/2];
        // Calculate (B_L + B_H) and store it in temp + n/2
        for (int i = 0; i < n/2; i++)
            temp[i + n/2] = B[i] + B[i + n/2];
        // Swap A_H and B_L so that A contains the coefficients of A_L and B_L, and B
    contains the coefficients of A_H and B_H
        for (int i = 0; i < n/2; i++)
            swap(A[n/2 + i], B[i]);
        // Calculate A_L * B_L, and leave the result in the first half of A
        KaratsubaRecursive(A, temp + n, n/2);
        // Calculate A_H * B_H, and leave the result in the second half of A (which is B),
    which corresponds to "raising the result by x^{n}"
        KaratsubaRecursive(B, temp + n, n/2);
        // Calculate (A_L + A_H) * (B_L + B_H)
```

```
        KaratsubaRecursive(temp, temp + n, n/2);

        // Calculate (A_L + A_H) * (B_L + B_H) - A_L * B_L - A_H * B_H
        for (int i = 0; i < n; i++)
            temp[i] -= A[i] + B[i];
        // "Raise the polynomial stored in temp by x^{n/2}" and add it to the final result
        for (int i = 0; i < n; i++)
            A[n/2 + i] += temp[i];
    }
}
```

4 Multiplication via the Fast Fourier Transform

To understand what a *Fast Fourier Transform* is, and then how it can be applied to enable the fastest known multiplication algorithm, we will have to dig deeper into math theory. Be warned, there lie linear algebra, group theory, number theory and complex numbers ahead!

To summarize what follows, in the following subsections we will first look into the relation between the coefficients of a polynomial and values it evaluates to, and explain what polynomial evaluation and interpolation are. Next, we will introduce the *Discrete Fourier Transform* and *roots of unity*, and show a few important properties of them, as well as their mutual relation. Finally, we will combine all the theory and look at the Fast Fourier Transform algorithm.

4.1 Polynomial Evaluation and Interpolation

The standard representation of a univariate polynomial of the n-th order is via the sum of its terms and their coefficients:

$$P(x) = \sum_{i=0}^{n} a_i x^i$$

Evaluating a polynomial is calculating its value in a given point x. For example, evaluation of $P(x)$ in the point $x = 4$ means calculating the value of

$$P(4) = \sum_{i=0}^{n} a_i 4^i$$

To get to where we are going with this more easily, let us look at what evaluation in some point $x = v$ looks like in matrix form.

$$P(v) = \sum_{i=0}^{n} a_i v^i = \begin{bmatrix} v^0 & v^1 & \cdots & v^n \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

If we want to evaluate $P(x)$ in both $x = v_0$ and $x = v_1$, we can write

$$\begin{bmatrix} P(v_0) \\ P(v_1) \end{bmatrix} = \begin{bmatrix} {v_0}^0 & {v_0}^1 & \cdots & {v_0}^n \\ {v_1}^0 & {v_1}^1 & \cdots & {v_1}^n \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

In fact, if the matrix in the middle, which is holding the powers of the values that the polynomial is being evaluated in, were a square matrix of the same order as the number of coefficients, it could have an inverse and we could get a formula to calculate the coefficients of the polynomial. So let's add more points $v_2, ..., v_n$ to the equation:

$$\begin{bmatrix} P(v_0) \\ P(v_1) \\ \vdots \\ P(v_n) \end{bmatrix} = \begin{bmatrix} {v_0}^0 & {v_0}^1 & \cdots & {v_0}^n \\ {v_1}^0 & {v_1}^1 & \cdots & {v_1}^n \\ \vdots & \vdots & \ddots & \vdots \\ {v_n}^0 & {v_n}^1 & \cdots & {v_n}^n \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

If the inverse of the middle matrix exists, then we can multiply both sides from the left with the inverse to obtain

$$\begin{bmatrix} {v_0}^0 & {v_0}^1 & \cdots & {v_0}^n \\ {v_1}^0 & {v_1}^1 & \cdots & {v_1}^n \\ \vdots & \vdots & \ddots & \vdots \\ {v_n}^0 & {v_n}^1 & \cdots & {v_n}^n \end{bmatrix}^{-1} \times \begin{bmatrix} P(v_0) \\ P(v_1) \\ \vdots \\ P(v_n) \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

In order for a matrix to have an inverse, it suffices for its determinant to not be zero. This process of calculating the coefficients of a polynomial, given a number of points that correspond to the order of the polynomial, as well as the values of the polynomial in those same points, is called *interpolation*.

The matrix whose inverse is required to exist for the above to be calculable has a very specific form, and it is called a Vandermonde matrix $V_n(v_0, ..., v_n)$.

$$V_n(v_0, ..., v_n) = \begin{bmatrix} {v_0}^0 & {v_0}^1 & \cdots & {v_0}^n \\ {v_1}^0 & {v_1}^1 & \cdots & {v_1}^n \\ \vdots & \vdots & \ddots & \vdots \\ {v_n}^0 & {v_n}^1 & \cdots & {v_n}^n \end{bmatrix}$$

The determinant of this matrix has a special form

$$det(V_n(v_0, ..., v_n)) = \prod_{0 \leq i < j \leq n} (v_j - v_i)$$

The requirement for a product to not be equal to zero is that none of its factors equal zero. Thus, the requirement for this determinant to be non-zero is that no pair of v_j and v_i is equal. Therefore, to guarantee that interpolation of an n-order polynomial $P(x)$ can be performed, we simply need to evaluate it in $n + 1$ different points.

Neat. However, why do we care about evaluation and interpolation in the context of multiplication of polynomials? Well, it turns out that once two polynomials have been evaluated in a given point, calculating the evaluation of the polynomial that is their product in that same point is trivial. Indeed, once we evaluate $P(x)$ in a point $x = v$, the result is just a number $P(v)$. Evaluating $Q(x)$ in that same point also gives us just a number $Q(v)$. Thus, the evaluation of $P(x)Q(x)$ in $x = v$ is also just a number $P(v)Q(v)$, which we obtain with just one multiplication of two numbers. So if $P(x)Q(x)$ has order m, and we evaluate $P(x)$ and $Q(x)$ in m different points $v_0, ..., v_m$, then we can calculate the coefficients of $P(x)Q(x)$ by calculating $P(v_0)Q(v_0)$, ..., $P(v_m)Q(v_m)$ in time $O(m)$, and then inverting the matrix $V_m(v_0, ..., v_m)$, and finally multiplying it with the vector of evaluated values $P(v_0)Q(v_0)$, ..., $P(v_m)Q(v_m)$.

The general algorithm for multiplication via evaluation and interpolation is given by the following. Let $P(x)$ and $Q(x)$ be two polynomials of order n. To obtain the coefficients of $P(x)Q(x)$:

1. **Evaluation:** Evaluate $P(x)$ and $Q(x)$ on $2n + 2$ different points
2. **Multiplication:** Pair-wise multiply the $2n + 2$ evaluated values together
3. **Interpolation:** Calculate the inverse of the Vandermonde matrix of the evaluation points, and multiply the inverted Vandermonde matrix with the vector of evaluated values to obtain the coefficients of $P(x)Q(x)$

Multiplication itself is the fastest part of this algorithm, and is done in $O(n)$ time. Evaluation requires $O(n^2)$ time in general case. Interpolation is by far the slowest part for the general case, requiring a little under $O(n^3)$ time for matrix inversion to execute. Even the naive polynomial multiplication algorithm is faster

than this. However, by carefully choosing the evaluation points $v_0, ..., v_{2n+1}$, evaluation and interpolation can be performed much faster, ultimately resulting in an algorithm that is faster than even the Karatsuba algorithm.

4.2 Complex Roots of Unity

Generally, a *complex number* z, consisting of a real part a and an imaginary part b, is written as $z = a + bi$, where i is the imaginary unit. z is not a real number because the imaginary unit is defined as $i^2 = -1$, and $\sqrt{-1}$ is not a real number. A *complex conjugate* of a complex number $z = a + bi$, denoted by $\overline{z}$, is defined as $\overline{z} = a - bi$. That is, a complex conjugate of a complex number is a complex number with the same real part, and a complex part of the same magnitude but opposite sign.

Now we introduce a group of special complex numbers that will be used for our purpose of evaluation and interpolation of polynomials. *N-th root of unity*, for a given positive integer n, is any number z such that $z^n = 1$. An n-th root of unity z is *primitive* if $z^k \neq 1$ for every $k \in \{1, ..., n-1\}$, i.e. the smallest positive integer for which z is a root of unity is n.

Complex n-th roots of unity are defined as

$$\omega_n^k = e^{\frac{2\pi ik}{n}} = \cos(\frac{2\pi k}{n}) + i\sin(\frac{2\pi k}{n}) \quad (k \in \mathbb{N})$$

From the definition, we then have that:

$$\overline{\omega_n^k} = \overline{e^{\frac{2\pi ik}{n}}} = \cos(\frac{2\pi k}{n}) - i\sin(\frac{2\pi k}{n}) = \cos(-\frac{2\pi k}{n}) + i\sin(-\frac{2\pi k}{n}) = e^{\frac{-2\pi ik}{n}} = \omega_n^{-k}$$

Claim 1. *ω_n^1 is a primitive n-th root of unity.*

Proof. First, we show that $(\omega_n^1)^n = 1$.

$$(\omega_n^1)^n = (e^{\frac{2\pi i}{n}})^n = e^{2\pi i} = cos(2\pi) + isin(2\pi) = 1 + i0 = 1$$

Next, we show that for a $k \in \{1, 2, ..., n\}$ such that $(\omega_n^1)^k = 1$, it must be that $k = n$.

$$1 = (\omega_n^1)^k = (e^{\frac{2\pi i}{n}})^k = e^{\frac{2\pi ik}{n}} = cos(\frac{2\pi k}{n}) + isin(\frac{2\pi k}{n})$$

Combining this with $(\omega_n^1)^n = 1$, we have:

$$cos(\frac{2\pi k}{n}) + isin(\frac{2\pi k}{n}) = cos(2\pi) + isin(2\pi)$$
$$(cos(\frac{2\pi k}{n}) - cos(2\pi)) + i(sin(\frac{2\pi k}{n}) - sin(2\pi)) = 0$$

A complex number is zero if both its real and its imaginary parts are zero, thus it must be that both:

$$cos(\frac{2\pi k}{n}) - cos(2\pi) = 0$$
$$sin(\frac{2\pi k}{n}) - sin(2\pi) = 0$$

For the first part:

$$cos(\frac{2\pi k}{n}) - cos(2\pi) = 0$$
$$cos(\frac{2\pi k}{n}) = cos(2\pi) = 1$$

We know that the cosine function periodically has value 1 only if its argument is 0 or 2π, thus it must be that:

$$\frac{2\pi k}{n} = 0 \vee \frac{2\pi k}{n} = 2\pi$$
$$k = 0 \vee k = n$$

For the second part:

$$sin(\frac{2\pi k}{n}) - sin(2\pi) = 0$$
$$sin(\frac{2\pi k}{n}) = sin(2\pi) = 0$$

We know that the sine function periodically has value 0 only if its argument is 0, π or 2π, thus it must be that:

$$\frac{2\pi k}{n} = 0 \vee \frac{2\pi k}{n} = \pi \vee \frac{2\pi k}{n} = 2\pi$$
$$k = 0 \vee k = \frac{n}{2} \vee k = n$$

Since both parts have to be zero, the only valid values of k are the ones that satisfy both the first and the second part, which is $k \in \{0, n\}$. Since for our proof $k = 0$ is not valid, the only remaining possibility is $k = n$, thus proving that no other positive integer value of k lower than n applies.

□

Claim 2. *For any $k < l \in \{1, 2, ..., n\}$, $\omega_n^k \neq \omega_n^l$.*

Proof. Assume that there exists a pair $k, l \in \{1, 2, ..., n\}$ such that $\omega_n^k = \omega_n^l$.

$$cos(\frac{2\pi k}{n}) + isin(\frac{2\pi k}{n}) = cos(\frac{2\pi l}{n}) + isin(\frac{2\pi l}{n})$$
$$\iff (cos(\frac{2\pi k}{n}) - cos(\frac{2\pi l}{n})) + i(sin(\frac{2\pi k}{n}) - sin(\frac{2\pi l}{n})) = 0$$
$$\iff cos(\frac{2\pi k}{n}) - cos(\frac{2\pi l}{n}) = 0 \quad \wedge \quad sin(\frac{2\pi k}{n}) - sin(\frac{2\pi l}{n}) = 0$$

a) The real part:

$$cos(\frac{2\pi k}{n}) - cos(\frac{2\pi l}{n}) = 0$$
$$\iff cos(\frac{2\pi k}{n}) = cos(\frac{2\pi l}{n})$$
$$\iff \frac{2\pi k}{n} = \frac{2\pi l}{n} \quad \vee \quad \frac{2\pi k}{n} = -\frac{2\pi l}{n}$$
$$\iff k = l \quad \vee \quad k = -l$$

b) The imaginary part:

$$sin(\frac{2\pi k}{n}) - sin(\frac{2\pi l}{n})$$
$$\iff sin(\frac{2\pi k}{n}) = sin(\frac{2\pi l}{n})$$
$$\iff \frac{2\pi k}{n} = \frac{2\pi l}{n} \quad \vee \quad \frac{2\pi k}{n} = \pi - \frac{2\pi l}{n} = \frac{2\pi \frac{n}{2}}{n} - \frac{2\pi l}{n} = \frac{2\pi(\frac{n}{2} - l)}{n}$$
$$\iff k = l \quad \vee \quad k = \frac{n}{2} - l$$

Only $k = l$ satisfies both a) and b), thus we conclude our proof.

□

So far, we have proved that (1) ω_n^1 is the primitive n-th complex root of unity, and (2) no two n-th complex roots of unity for $k \in \{1, 2, ..., n\}$ are the same. Remembering our matrix formulae for evaluation and interpolation of polynomials, that means that n-th complex roots of unity represent a set of n distinct points which would result in a non-zero determinant of the Vandermonde matrix. Additionally, note that $\{\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}\}$ forms a cyclic group of order n with respect to multiplication, generated by ω_n^1. Therefore, for any integer $k \in \{0, 1, ..., n-1\}$, we have that (3):

$$\omega_n^{n+k} = \omega_n^k$$
$$\omega_n^{-k} = \omega_n^{n-k}$$

4.3 Discrete Fourier Transform

The *Discrete Fourier Transform (DFT)* of a set of n complex numbers $\{v_0, v_1, ..., v_{n-1}\}$ transforms them into another set of numbers $\{f(v_0), f(v_1), ..., f(v_{n-1})\}$, defined by:

$$f(v_k) = \sum_{m=0}^{n-1} v_m e^{-\frac{2\pi i k m}{n}} = \sum_{m=0}^{n-1} v_m \omega_n^{-km}$$

We see that $f(v_k)$ is equal to the evaluation of the polynomial $P(x) = \sum_{m=0}^{n-1} v_m x^m$ in point $x = \omega_n^{-k}$. Therefore, evaluation of a polynomial $P(x) = \sum_{i=0}^{n-1} a_i x^i$ in the n-th complex roots of unity is the same as computing the DFT of $\{a_0, a_1, ..., a_{n-1}\}$, which we will denote as $DFT(P)$. Now, let us look at the resulting equation in the matrix form again.

$$\begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} (\omega_n^{-0})^0 & (\omega_n^{-0})^1 & \cdots & (\omega_n^{-0})^{n-1} \\ (\omega_n^{-1})^0 & (\omega_n^{-1})^1 & \cdots & (\omega_n^{-1})^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ (\omega_n^{-(n-1)})^0 & (\omega_n^{-(n-1)})^1 & \cdots & (\omega_n^{-(n-1)})^{n-1} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\iff \begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} \omega_n^{-0\cdot 0} & \omega_n^{-0\cdot 1} & \cdots & \omega_n^{-0\cdot(n-1)} \\ \omega_n^{-1\cdot 0} & \omega_n^{-1\cdot 1} & \cdots & \omega_n^{-1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^{-(n-1)\cdot 0} & \omega_n^{-(n-1)\cdot 1} & \cdots & \omega_n^{-(n-1)\cdot(n-1)} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

A great feature of the Vandermonde matrix in the n-th complex roots of unity is that it is very close to a unitary matrix. A *unitary matrix* M is a matrix whose inverse is equal to its transpose conjugate, i.e. $M^{-1} = \overline{M^T}$. A matrix is a unitary matrix iff its rows form an *orthonormal basis*, i.e. for every two of its rows, their inner product is equal to 0 if they are different rows, and equal to 1 if they are the same row. So let us see what the inner product of two rows i, j of the above matrix looks like.

$$\langle(\omega_n^{-i\cdot 0}, \omega_n^{-i\cdot 1}, \cdots, \omega_n^{-i\cdot(n-1)}), (\omega_n^{-j\cdot 0}, \omega_n^{-j\cdot 1}, \cdots, \omega_n^{-j\cdot(n-1)})\rangle = \sum_{k=0}^{n-1} \omega_n^{-ik}\overline{\omega_n^{-jk}} = \sum_{k=0}^{n-1} \omega_n^{-ik}\omega_n^{jk} = \sum_{k=0}^{n-1} \omega_n^{k(j-i)}$$

$$\iff \langle(\omega_n^{-i\cdot 0}, \omega_n^{-i\cdot 1}, \cdots, \omega_n^{-i\cdot(n-1)}), (\omega_n^{-j\cdot 0}, \omega_n^{-j\cdot 1}, \cdots, \omega_n^{-j\cdot(n-1)})\rangle = \sum_{k=0}^{n-1} (\omega_n^{(j-i)})^k$$

If $i \neq j$, then $i - j \neq 0$ and $\omega_n^{k(j-i)} \neq \omega_n^0 = 1$. Thus we can use the formula for the n-th sum of a geometric series:

$$\sum_{i=0}^{n-1} r^i = \frac{1 - r^n}{1 - r}$$

$$\implies \langle(\omega_n^{-i\cdot 0}, \omega_n^{-i\cdot 1}, \cdots, \omega_n^{-i\cdot(n-1)}), (\omega_n^{-j\cdot 0}, \omega_n^{-j\cdot 1}, \cdots, \omega_n^{-j\cdot(n-1)})\rangle = \frac{1-(\omega_n^{j-i})^n}{1-\omega_n^{j-i}} = \frac{1-(\omega_n^n)^{j-i}}{1-\omega_n^{j-i}} = \frac{1-1}{1-\omega_n^{j-i}}$$

$$\implies \langle(\omega_n^{-i\cdot 0}, \omega_n^{-i\cdot 1}, \cdots, \omega_n^{-i\cdot(n-1)}), (\omega_n^{-j\cdot 0}, \omega_n^{-j\cdot 1}, \cdots, \omega_n^{-j\cdot(n-1)})\rangle = 0$$

So we know that if the two rows are different, their inner product is zero.

Now, if $i = j$, then $\omega_n^{k(j-i)} = \omega_n^0 = 1$, and we have:

$$\langle(\omega_n^{-i\cdot 0}, \omega_n^{-i\cdot 1}, \cdots, \omega_n^{-i\cdot(n-1)}), (\omega_n^{-j\cdot 0}, \omega_n^{-j\cdot 1}, \cdots, \omega_n^{-j\cdot(n-1)})\rangle = \sum_{k=0}^{n-1} 1 = n$$

So the matrix V_n is not unitary in n-th complex roots of unity because the inner product of a row with itself does not equal 1. However, since we know it equals n, we can create a new matrix $M = \frac{1}{\sqrt{n}} V_{n-1}$, which should be unitary.

$$\langle\frac{1}{\sqrt{n}}(\omega_n^{-i\cdot 0}, \omega_n^{-i\cdot 1}, \cdots, \omega_n^{-i\cdot(n-1)}), \frac{1}{\sqrt{n}}(\omega_n^{-j\cdot 0}, \omega_n^{-j\cdot 1}, \cdots, \omega_n^{-j\cdot(n-1)})\rangle = \sum_{k=0}^{n-1} \frac{1}{n}(\omega_n^{(j-i)})^k = \frac{1}{n}\sum_{k=0}^{n-1}(\omega_n^{(j-i)})^k$$

Now when $i = j$, the result will be:

$$\frac{1}{n}\sum_{k=0}^{n-1}(\omega_n^0)^k = \frac{1}{n}\sum_{k=0}^{n-1} 1 = \frac{1}{n}n = 1$$

Then the inverse of the matrix $M = \frac{1}{\sqrt{n}} V_{n-1}$ is going to be:

$$M^{-1} = \overline{M^T} = \overline{(\frac{1}{\sqrt{n}} V_{n-1})^T} = \frac{1}{\sqrt{n}}\overline{V_{n-1}^T} = \frac{1}{\sqrt{n}}\overline{\begin{bmatrix} \omega_n^{-0\cdot 0} & \omega_n^{-0\cdot 1} & \cdots & \omega_n^{-0\cdot(n-1)} \\ \omega_n^{-1\cdot 0} & \omega_n^{-1\cdot 1} & \cdots & \omega_n^{-1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^{-(n-1)\cdot 0} & \omega_n^{-(n-1)\cdot 1} & \cdots & \omega_n^{-(n-1)\cdot(n-1)} \end{bmatrix}}$$

$$\iff M^{-1} = \frac{1}{\sqrt{n}}\begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \cdots & \omega_n^{0\cdot(n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \cdots & \omega_n^{1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \cdots & \omega_n^{(n-1)\cdot(n-1)} \end{bmatrix}$$

We see that the matrix on the right side is also a Vandermonde matrix, but instead of using powers of ω_n^{-1} as in the evaluation, it uses powers of ω_n^1 for interpolation. As the result, we have:

$$M = \frac{1}{\sqrt{n}} V_{n-1}(\omega_n^{-1}) \quad \wedge \quad M^{-1} = \frac{1}{\sqrt{n}} V_{n-1}(\omega_n^1)$$

$$\implies \begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = V_{n-1}(\omega_n^{-1}) \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\iff \begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \sqrt{n} M \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\iff M^{-1} \times \begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \sqrt{n} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\iff \frac{1}{\sqrt{n}} V_{n-1}(\omega_n^1) \times \begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \sqrt{n} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\iff \frac{1}{n} V_{n-1}(\omega_n^1) \times \begin{bmatrix} P(\omega_n^0) \\ P(\omega_n^1) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

The above is interesting – namely, if we evaluate the polynomial in the inverse n-th roots of unity ω_n^{-1} (i.e. if we compute its DFT), then we can interpolate it and calculate its coefficients by simply computing the DFT of its evaluation on the (non-inverse) n-th roots of unity ω_n^1, and dividing the result of that by n. The reason that this is great news is that the *Fast Fourier Transform (FFT)* algorithm enables us to compute the DFT of a set of n numbers in time $O(n \log n)$, as we will see in the following subsection.

Before going into FFT, let us update our general multiplication scheme via evaluation and interpolation with the findings that we have collected so far. Let $P(x)$ and $Q(x)$ be two polynomials of order $n-1$. To obtain the $2n$ coefficients of $P(x)Q(x)$:

1. **Evaluation:** Compute DFTs of $P(x)$ and $Q(x)$ with ω_{2n}^{-1} as the root
2. **Multiplication:** Pair-wise multiply the $2n$ evaluated values together
3. **Interpolation:** Calculate the DFT of the $2n$ multiplied evaluated values with ω_{2n}^1 as the root (i.e. compute the inverse DFT), and divide the numbers obtained as the result by $2n$, to obtain the coefficients of $P(x)Q(x)$

Now that steps 1 and 3 are reduced to the identical subproblem (computing the DFT in both cases, but with primitive roots of unity that are inverses of each other), the only thing that remains is to show how to efficiently compute the DFT.

4.4 The Fast Fourier Transform Algorithm

The *Fast Fourier Transform (FFT)* is an algorithm for efficient computation of the DFT of a set of numbers. Let us write down the problem that it solves formally. Let $a_0, a_1, ..., a_{n-1}$ be a set of n numbers, and let $P(x)$ represent the polynomial formed by the coefficients corresponding to those same numbers:

$$P(x) = \sum_{i=0}^{n-1} a_i x^i$$

The goal is to compute the *Discrete Fourier Transform* of $a_0, a_1, ..., a_{n-1}$ with the root ω_n^1, i.e. compute the values $P(\omega_n^0), P(\omega_n^1), ..., P(\omega_n^{n-1})$.

FFT is a divide & conquer algorithm, and it relies on splitting the polynomial into two parts of half the size, for which DFT can be recursively computed, and then combining that somehow into the overall solution and obtaining the DFT of the original polynomial. The divide step relies on splitting the terms of the polynomial into those that have even-order powers of x, and those that have odd-order powers of x. For our analysis, let us assume that n is an even number and it can be split into two equal halves.

$$\begin{aligned} P(x) &= \sum_{i=0}^{n-1} a_i x^i \\ &= a_0 x^0 + a_1 x^1 + ... + a_{n-1} x^{n-1} \\ &= (a_0 x^0 + a_2 x^2 + ... + a_{n-2} x^{n-2}) + (a_1 x^1 + a_3 x^3 + ... + a_{n-1} x^{n-1}) \\ &= (a_0 x^0 + a_2 x^2 + ... + a_{n-2} x^{n-2}) + x(a_1 x^0 + a_3 x^2 + ... + a_{n-1} x^{n-2}) \\ &= (a_0 (x^2)^0 + a_2 (x^2)^1 + ... + a_{n-2} (x^2)^{\frac{n}{2}-1}) + x(a_1 (x^2)^0 + a_3 (x^2)^1 + ... + a_{n-1} (x^2)^{\frac{n}{2}-1}) \\ &= P_E(x^2) + x P_O(x^2) \end{aligned} \tag{2}$$

If we denote the polynomial in x^2 that contains $P(x)$'s even-order coefficients by $P_E(x^2)$, and the one that contains $P(x)$'s odd-order coefficients by $P_O(x^2)$, we see that evaluating a polynomial in a point x can be reduced to evaluating two polynomials of half the size in x^2, and then computing the result through a single number multiplication and addition.

While this formula represents the basis of FFT, we need to analyze why it actually works and why it results in an efficient algorithm. For example, since the goal is to compute the DFT of the original polynomial, would each level of recursion have to evaluate all the subpolynomials in all the original n-th complex roots of unity, even as the problems keep shrinking to sizes $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...$? The running time of such an algorithm would be $\omega(n^2)$, as even just the leaf-level of the recursion tree would spend $\Theta(n^2)$ time. Therefore, we need to make it so that each subproblem evaluates the polynomial in at most half the points. This could be achieved if the values that the polynomial is being evaluated on represent $\frac{n}{2}$ pairs of values with the opposite sign, as then the following would be true:

$$P(\pm v) = P_E(v^2) \pm vP_O(v^2)$$

Thus, by evaluating $P_E(v^2)$ and $P_O(v^2)$, we would have the result ready for both $P(v)$ and $P(-v)$. However, we are squaring v for the next level of recursion, so we would lose the plus/minus pairs, right? With general real numbers, yes – but we are not working with general, or even real, numbers. We are evaluating the polynomial on the n-th complex roots of unity, and it turns out that (i) when you square n-th complex roots of unity, you obtain the $\frac{n}{2}$-th complex roots of unity; as well as that (ii) the n-th complex roots of unity form plus/minus pairs.

Claim 3. *Let n be an even number. Then the n-th complex roots of unity form plus/minus pairs $\pm 1, \pm\omega_n^1, ..., \pm\omega_n^{\frac{n}{2}-1}$*

Proof.

$$\omega_n^{k+\frac{n}{2}} = e^{\frac{2\pi i(k+\frac{n}{2})}{n}} = e^{\frac{2\pi ik}{n}} e^{\frac{2\pi i\frac{n}{2}}{n}} = \omega_n^k e^{\pi i} = \omega_n^k(\cos(\pi) + i\sin(\pi)) = \omega_n^k(-1 + i0) = -\omega_n^k$$

□

Claim 4. *When squared, the plus/minus pairs of n-th complex roots of unity form $\frac{n}{2}$-th complex roots of unity.*

Proof.

$$(\pm\omega_n^k)^2 = \omega_n^{2k} = e^{\frac{2\pi i2k}{n}} = e^{\frac{2\pi ik}{\frac{n}{2}}} = \omega_{\frac{n}{2}}^k$$

□

Thus, we now know for certain that at every step of recursion where the number of coefficients is n: (i) we are evaluating a polynomial of order n in the n-th roots of unity, (ii) the n-th roots of unity form plus/minus pairs, (iii) two subproblems need to be evaluated on roots of unity whose number is half the size of the current, whose primitive root we get by simply squaring the current primitive root, and that again form plus/minus pairs? Almost! The plus/minus pairs property of the n-th complex roots of unity relies on the assumption that n is an even number (Claim 3). The only way to guarantee that n is an even number at each step of recursion is for the original set of numbers whose DFT we are computing to have the number of elements which is a power of 2. Thus, we have to pad the high-order coefficients of the original polynomial with 0 until their number represents a power of 2. With this adjustment made, now the required math checks out at every level of recursion, and this leads to an algorithm whose running time is described by $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, which results in a total running time of $\Theta(n \log n)$. Below, we give a recursive implementation of the FFT algorithm for computing the DFT, given a primitive root (ω_n^1 or ω_n^{-1}).

```
1 void FFT(const complex<double> * P, const int n, complex<double> * DFT, const complex<double
    > primitiveRoot) {
2     // We assume that the array DFT has already been allocated as an array of size n, and
      that n is a power of 2
3     for (int i = 0; i < n; i++) {
4         DFT[i] = P[i];
5     }
6     // The array temp is used to separate the coefficients into even and odd
```

```
    complex<double> * temp = new complex<double> [n];
    FFTRecursive(DFT, temp, n, primitiveRoot);
    delete temp;
}
```

```
void FFTRecursive(complex<double> * DFT, complex<double> * temp, const int n, complex<double> root) {
    if (n > 1) {
        // Separate the coefficients
        for (int i = 0; i < n/2; i++) {
            temp[i] = DFT[2*i];
            temp[n/2 + i] = DFT[2*i + 1];
        }
        memcpy(DFT, temp, n * sizeof(DFT[0]));

        // Evaluate P_E(v^2)
        FFTRecursive(DFT, temp, n/2, root * root);
        // Evaluate P_O(v^2)
        FFTRecursive(DFT + n/2, temp, n/2, root * root);

        // P(±v) = P_E(v^2) ± vP_O(v^2)
        complex<double> omega_k(1, 0);
        for (int k = 0; k < n/2; k++, omega_k *= root) {
            complex<double> t = omega_k * DFT[n/2 + k];
            DFT[n/2 + k] = DFT[k] - t;
            DFT[k] = DFT[k] + t;
        }
    }
}
```

4.5 Division and Interpolation

Polynomial division of $P(x)$ with $Q(x)$ is the problem of finding polynomials $q(x)$ and $r(x)$, such that $P(x) = Q(x)q(x) + r(x)$, where $q(x)$ is the quotient polynomial and $r(x)$ is the remainder polynomial.

Having gone over our algorithm for fast polynomial multiplication, one might ask whether we can employ polynomial evaluation and interpolation to perform polynomial division, and consequently the division of big numbers. Based on the multiplication algorithm, the intuition here would be to evaluate $P(x)$ and $Q(x)$ by computing their DFTs, divide their values in the complex roots of unity to obtain evaluation of $P(x)/Q(x)$, and then finally interpolate the $P(x)/Q(x)$ polynomial by calculating the inverse DFT. However, this algorithm does not work, for multiple reasons.

The first reason is that there simply may not exist a polynomial that accurately represents $P(x)/Q(x)$, as is the case any time that the remainder $r(x)$ is not zero. Thus, we unfortunately cannot use interpolation to perform polynomial division in the general case. Even if we restrict our attention to cases where there would be no remainder, such as when we are dividing polynomials that represent big numbers whose division results in an integral value (something that we generally cannot guarantee), the approach still does not work, for **the second** reason, which is continuity. The process of interpolation of a polynomial's coefficients attempts to interpolate the function represented by the polynomial, given its values in a number of different points in a plane. In the case of multiplication of two polynomials, this works without any problems. $P(x)$ and $Q(x)$ represent functions that are continuous everywhere, and their product $P(x)Q(x)$ is also a polynomial that is continuous everywhere. However, in the case of division, the function $P(x)/Q(x)$ is generally not a continuous function. It has asymptotes at every point in which $Q(x)$ evaluates to 0. If such points intersect the interval of complex roots of unity that we have evaluated the two polynomials at, or even simply come close to it, then the behavior of the function $P(x)/Q(x)$ will not be accurately represented by interpolation. As an example, let us observe polynomials $P(x) = 9x$ (representing the number 90) and $Q(x) = 4x + 5$ (representing the number 45). $Q(x)$ has a root at $x = \frac{-5}{4}$, which is very close to a root of unity -1. Even though the polynomial that is the result of interpolating $P(x)/Q(x)$ should evaluate to 2 in $x = 10$, the inverse DFT of $P(x)/Q(x)$ is going to return weird coefficients, and the resulting polynomial will evaluate to 46 in $x = 10$.