Explaining and Implementing the Cooley-Tukey Fast Fourier Transform

Atikah Hussain

Third Year Project

BSc Mathematics

Queen Mary, University of London

May 2022

Abstract

This project aims to explain the derivation and usage of the Cooley-Tukey Fast Fourier Transform. We begin by motivating the need for such an algorithm through its application in efficient polynomial multiplication. Then we will analyse its algorithmic complexity and explain the tricks that make it fast, and even extend these ideas to work over finite fields. Finally, we will implement the algorithm using Python code to test its efficacy and speed by comparing it to more naive algorithms.

This report will also cover some supplementary material on complex numbers, linear algebra, computational complexity theory, and finite field theory.

Acknowledgements

I would like to give my sincere thanks to Professor Charles Leedham-Green for all the guidance, patience, and enthusiasm he has put towards this project in his role as supervisor. Without him, undertaking this project would not have been nearly as rewarding and insightful.

Contents

Chapter 1

Background for the FFT

1.1 Introducing Fast Fourier Transforms

Fast Fourier Transforms (FFTs) are a set of optimised algorithms with origins in Fourier Analysis, where they were used to compute the Discrete Fourier Transform of frequency samples of a signal. Now, FFTs have many different applications across mathematics (for fast multiplications, solving partial differential equations, etc.) and engineering (for signal processing, image and audio compression, etc.).

In essence, these algorithms are the fastest way of converting large polynomials from one form to another. The inverse of FFT algorithm will then use a very similar process to covert the input back to its original form. Since polynomials are so widely used, being able to manipulate them efficiently is crucial.

Aside from their importance, FFTs present a unique take on the classic divide-and-conquer method of speeding up an algorithm, using the periodicity and symmetry of complex numbers.

There exist many FFTs with different implementations and uses. Prime-factor FFTs make use of number theory rather than properties of complex numbers, while Brunn's FFT was designed to work best on real-valued data. However, the Cooley-Tukey FFT is the most well known and frequently implemented version of an FFT algorithm due to its accuracy and flexibility, and so it will be the focus of this report.

1.2 Time Complexity and Big O Notation

When describing the efficiency of an algorithm, we will consider how the number of basic operations needed to finish running the algorithm scales asymptotically as the size of the algorithm's input data, n, increases. This will be given in terms of big O notation.

In this report, we will regard algorithms with quasi-linear time complexity $O(n \log n)$ or lower as "efficient", and anything higher than this as "inefficient". Although classifying algorithms in this dichotomous way is not precise, it provides a rule of thumb for identifying whether an algorithm will run in good time on a computer for very large inputs.[1]

1.3 Polynomial Multiplication

To motivate the need for developing the Cooley-Tukey FFT algorithm, consider the problem of multiplying two polynomials, $A(x)$ and $B(x)$, both of degree $n-1$. The naive method for finding the product would be carry out a pairwise multiplication of each term of each polynomial and sum all of the products. While this is practical for polynomials of a relatively small degree, the computation becomes inefficient for larger degrees. This is because the naive multiplication method has time complexity $O(n^2)$, due to conducting at most n multiplications for each of the n terms in the polynomials. A faster polynomial multiplication method, like Karatsuba's algorithm, is still only $O(n^{\log_2 3})$, i.e. $O(n^{1.585\ldots})$ [1].

Our goal is to find an optimised way of multiplying polynomials. To start, we can consider the ways we represent polynomials, before trying to manipulate them. The most common representation takes the form

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

We can simplify this notation using a vector of just the coefficients, and refer to this form as the *coefficient representation* or *coefficient form* of a polynomial.

$$\vec{A} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$$

We can define an alternative representation using the following idea: a degree $n-1$ polynomial can be uniquely represented by a sequence of n distinct points. We will formalise this idea in a theorem and prove it as follows.

Theorem 1.3.1 (Polynomial Interpolation) *Let $\mathbb{F}$ be any field, $n > 0$ be an integer, and $a_0, \ldots, a_{n-1}$ be any n distinct elements of $\mathbb{F}$. Let $y_0, \ldots, y_{n-1}$ be any n elements of $\mathbb{F}$ that are not necessarily distinct.*

Then, there is a unique polynomial $f(x)$ over $\mathbb{F}$ of degree at most $n-1$ such that $f(a_i) = y_i$ for $i = 0, 1, \ldots, n-1$.

[1] Generally, polynomial time algorithms or lower are considered efficient, as outlined in Cobham's thesis [3], in which "efficiency" is equated to membership of the fundamental complexity class P. However, in the interest of optimising commonly used algorithms, as we aim to do in this report, polynomial time is not fast enough.

Proof:

First we must prove the existence of such a polynomials. Using the $a_0, \ldots, a_{n-1}$ values, we can construct the polynomial $f(x) = \sum_{i=0}^{n-1} d_i \prod_{j \neq i}(x - a_j)$. It is the sum of n different polynomials that we can produce using $n-1$ out of the n different values of a_i.

When we evaluate $f(a_i)$, all but one of the polynomial expressions summed together will cancel out, leaving just $d_i \ldots (a_i - a_{i-1})(a_i - a_{i+1}) \ldots$ which we require to equal y_i. All we have to do is divide y_i by the factored expression to solve for the unique value of d_i. We can do such a division since the polynomial is defined over an arbitrary field.

Now to prove that this polynomial function is unique. It suffices to prove that if $f(a_i) = 0$ for all i, then $f(x) = 0$.

So, suppose that $f(x)$ has this vanishing property, and that $f(x)$ has degree at most $n-1$. If the degree is 0, then $f(x)$ is constant, and this constant must be zero, as required. Now suppose that the degree is not 0, and let $f(x) = (x - a_{n-1})g(x) + c$ where $g(x)$ has degree less than $n-1$. Then, $c = 0$ and $g(a_i) = 0$ for $0 \leq i \leq n-1$. By induction on n, it follows that $g(x) = 0$, and so $f(x) = 0$ as required.

□

Now, we can refer to such a sequence of points as the *value representation* or *value form* of a polynomial.

Multiplication of two polynomials in their value form works when both sequences of $A(x_k)$ and $B(x_k)$ values are computed from the same sequence of x_k values. We can simply carry out an element-wise multiplication of the two sequences, and the product will be a new sequence of equal length as the initial sequences. In other words, each term in the new sequence is $C(x_k) = A(x_k) \cdot B(x_k)$.

We know the polynomial product, $C(x)$, in its coefficient form will have $2n-1$ terms when both initial polynomials, $A(x)$ and $B(x)$, had n terms. This means $A(x)$ and $B(x)$ would need to be evaluated at $2n$ different values of x_k before being multiplied in order for the product sequence to considered be a value representation of $C(x)$, according to Theorem 1.3.1. For instance, the product of two quadratic functions is a quartic function. A quartic function is made up of five terms, so the two quadratic functions would each have to be evaluated at five points in order for their product to represent the unique quartic function.

Element-wise multiplication of two sequences of equal length has a time complexity of just $O(n)$, so it is much more efficient than multiplication in coefficient form. We have reached our first goal of finding an efficient algorithm for polynomial multiplication.

However, this result is not entirely useful on its own, as in most contexts we will find ourselves using the coefficient representation of polynomials much more often than value representation. The reason being, certain operations such as division cannot

be done on polynomials in value form. We also would not be able to define a function that gives us other values using the value form.

So now we have another problem: can we efficiently convert between these two representations?

1.4 Polynomial Evaluation and Interpolation

Converting from coefficient form to value form is known as polynomial evaluation. The inverse, converting from value form to coefficient form, is known as polynomial interpolation.

Using Horner's Rule[2] or just naively evaluating and summing each of the n terms of a polynomial in coefficient form at n distinct values of x_k to get its value form has time complexity $O(n^2)$. Separate algorithms exists for polynomial interpolation such as Neville's algorithm and Lagrange's Interpolation algorithm, both with $O(n^2)$ as well. [2]

Suppose there did exist some algorithms for polynomial evaluation and interpolation which we consider efficient. Then, we could use them to convert two polynomials in coefficient form to their value form, and apply the $O(n)$ value form multiplication. Finally, we would just convert the product to coefficient form. Efficient algorithms used in succession on the same input can, in itself, be considered an efficient algorithm.

So far we have not come across any efficient algorithms. However, even if we did have such a polynomial evaluation algorithm, there is an issue with using two entirely unrelated algorithms for evaluation and interpolation: there are no standard choices for which values of x_k to use when deriving the value representation of a polynomial. This runs the risk of mistakenly multiplying two polynomials in their value representations which are not derived form the same sequence of x_k values, leading to an incorrect result. We also can't guarantee that both algorithms run calculations to the same degrees of accuracy, meaning accuracy could be lost if we keep converting back and forth between coefficient and value forms.

Ideally, we would have evaluation and interpolation algorithms which are exact inverses of each other, such that the input and output for multiplication is always valid and similarly accurate. This also will have an added benefit in terms of implementing the algorithm in code, as the two algorithms would share commonalities that make the implementations similar, and thus, straight-forward.

Our goal now is quite ambitious. We want to find an algorithm that is efficient, invertible, and uses a consistent set of x_k values for our value representations.

[2]Horner's Rule for Evaluation: $A(x_k) = a_0 + x_k(a_1 + x_k(a_2 + ...x_k(a_{n-1})$

1.5 Divide and Conquer with Symmetry

If we begin by considering the most basic polynomials with only one term, in the form $P(x) = ax^k$, it is easy to see that they all exhibit some symmetry. Specifically, a single-term polynomial is either an odd or even function when its degree is odd or even respectively.[3]

Definition 1.5.1 (Even Function) *Let f be a real-valued function of a real variable. Then f is an even function if the following holds for all x such that x and $-x$ are in the domain of f:*

$$f(-x) = f(x)$$

Definition 1.5.2 (Odd Function) *Let f be a real-valued function of a real variable. Then f is an even function if the following holds for all x such that x and $-x$ are in the domain of f:*

$$f(-x) = -f(x)$$

Any polynomial is made up of a series of terms which are odd and even functions. Using the basic properties of odd and even functions, we know that the sum of two even functions is an even function, and the sum of two odd functions is an odd function. So, we can separate all the even and odd degree terms of the polynomial knowing that these half-sized polynomials can each be evaluated in nearly half the time expected, using the definition of odd and even functions. This is because half of the evaluations we compute will use only a constant number of basic operations, and will therefore be done in $O(1)$ time rather than $O(n)$ time.

Example 1.5.1 (Splitting Polynomials) *Consider the following polynomial:*

$$A(x) = x^3 + x^2 - x - 1$$

We collect even and odd degree terms separately, then factor out x from the odd expression to get two polynomials in terms of x^2.

$$A(x) = (x^2 - 1) + (x^3 - x)$$
$$A(x) = (x^2 - 1) + x(x^2 - 1)$$

Now we use these two expressions to define smaller polynomials, with which we can rewrite $A(x)$:

[3]These definitions can be expanded for complex-valued functions [5].

$$A_{even}(x^2) = x - 1,\ A_{odd}(x^2) = x - 1$$

These functions were re-parameterised to take x^2 instead of x, so that the polynomials have odd and even degree terms rather than only even degree terms. That way, they may be split recursively in the same way, until we get polynomials with only one term:

$$A_{even}(x^2) = (-1) + x(1)$$
$$A_{e,e}(x^2) = -1,\ A_{e,o}(x^2) = 1$$

The expressions for A_{odd}, $A_{o,e}$ and $A_{o,o}$ follow. We can now infer that for each x_k and $-x_k$ value, we only need to evaluate

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2)$$
$$A(-x) = A_{even}(x^2) - xA_{odd}(x^2)$$

Let us evaluate $A(x)$ at $x_1 = 2$. First we evaluate each of the one-term polynomials, $A_{e,e}$, $A_{e,o}$, and so on, at $x_1 = 2$. Then we can evaluate each of the two-term polynomials:

$$A_{even}(2^2) = -1 + (2^2) \cdot 1 = 3$$

In this instance, $A_{odd}(2^2) = 3$ as well. Now, we can evaluate $A(x)$ at $x_1 = 2$:

$$A(2) = 3 + (2) \cdot 3 = 9$$

We can also very quickly evaluate $A(x)$ at $-x_1 = -2$:

$$A(-2) = 3 + (-2) \cdot 3 = -3$$

If we try evaluating $A(2)$ and $A(-2)$ by simple substitution, we see that the result is the same as we got with this recursive splitting method.

Although halving the total number of computations from naive polynomial evaluation is somewhat of an improvement, we still only have an $O(n^2)$ algorithm. This is because we evaluate $\frac{n}{2}$ values of x_k, with each evaluation using $n - 2$ smaller polynomials, each of which requiring 2 basic operations to compute. This gives exactly

$\frac{n}{2} \cdot (n-2) + 2$ steps when n is a power of two[4], meaning the algorithm has $O(n^2)$.

If we want to make full use of our recursion, we should aim to halve the number of computations at *every* recursive step, not just the first step. This would reduce the total number of recursions to roughly $\log_2 n$, reducing this evaluation algorithm's time complexity to $O(n \log n)$. This would be possible if we started with a single base value of x_k which could be used to derive all other values, similar to how, currently, we are using $+x_0, +x_1, \ldots, +x_{n\frac{n-1}{2}}$ to derive $-x_0, -x_1, \ldots, + - x_{\frac{n-1}{2}}$

Our issue is that squaring x_k and $-x_k$ for the first recursive step both give x_k^2, so we longer have the positive-negative pairs that allow us to exploit the symmetry of the odd and even functions.

1.6 Divide and Conquer with Roots of Unity

In this section, we will explore how we can remedy the loss of positive-negative pairs in recursion by expanding the domain of numbers that we use for x_k from the real numbers to the complex numbers.

In particular, we will use the group of n-th roots of unity, which come in positive-negative pairs when n is a power of 2. When roots of unity for n as a power of 2 are squared (as they will be at each recursive step), we get a subgroup of half the order that also has positive-negative pairs within it, and so on. For example, when each of the 8th roots of unity are squared, we get the 4th roots of unity.

Since we have pairs like this at every step, half of the required computations at each step can be done in $O(1)$ time, resulting in $O(n \log_2 n)$ time overall.

Another benefit to using n-th roots of unity is it gives us an answer to question of how to consistently choose our values of x_k for all possible polynomials. Since n-th roots of unity are a cyclic group under multiplication, we can simply choose the generator element, $\omega = e^{\frac{2\pi i}{n}}$, as our first x_k value. The rest of the values will follow, through multiplication.

If we substitute $x_k = \omega^k$ into the formulae for calculating $A(x_k)$ from split polynomials, we get

$$A(\omega^k) = A_{even}(\omega^{2k}) + \omega^k A_{odd}(\omega^{2k})$$
$$A(-\omega^k) = A_{even}(\omega^{2k}) - \omega^k A_{odd}(\omega^{2k})$$

[4] A_{even} and A_{odd} will only split evenly at each recursive step if n is a power of two. Otherwise, the recursion will end sooner for some polynomials than others, meaning the exact expression for number of steps in the algorithm will vary slightly. However, this wouldn't affect the overall time complexity.

Using the diagonal symmetry of power-of-2-th roots of unity, $-\omega^k = \omega^{k+\frac{n}{2}}$, we can adjust the second formula further such that we only need to use one half of the roots of unity to calculate the other half.

$$A(\omega^k) = A_{even}(\omega^{2k}) + \omega^k A_{odd}(\omega^{2k})$$
$$A(\omega^{k+\frac{n}{2}}) = A_{even}(\omega^{2k}) - \omega^k A_{odd}(\omega^{2k})$$

These final two formulae are the key to the divide and consquer of the FFT.

Chapter 2

Explaining the FFT

2.1 The Cooley-Tukey FFT Algorithm

Now that we have seen the benefit of using n-th roots of unity, we can describe the Cooley-Tukey FFT in full.

We begin with a polynomial in coefficient form, of degree $n-1$ (or length n, if we include terms with 0 as the coefficient, which we will when implementing the algorithm later). We check that $n = 2^k$ for some $k = 0, 1, 2...$ Depending on how the FFT is implemented, we either *cannot* proceed if $n \neq 2^k$, or we *can* by padding the polynomial with higher degree terms of coefficient 0 until we reach the required length. I have included two algorithms outlined in pseudocode in the Appendix, one for each scenario.

Next we calculate the first of the n-th roots of unity, ω, and take its powers up to $n/2$ to get the rest of the values required. We split the polynomial by its odd and even degree terms, factor x out of the odd degree terms, and define two new polynomials using the re-parameterisation $x \to x^2$. We continue splitting in this manner until we have only single-degree polynomials.

Then, we can start evaluating these polynomials at ω^k and $-\omega^k$ values, with the latter done in constant time, and storing the output. We combine these results to evaluate the larger polynomial which they were originally derived from. This continues recursively until we have evaluated the original polynomial at all the roots of unity.

Once the algorithm has finished running, we should have, as its output, a sequence of n complex values which gives the original polynomial in its value form.

It should be noted that although polynomial evaluation and interpolation in general are defined over arbitrary fields, the FFT is not because it evaluates the polynomial at n-th roots of unity, so it only works on polynomials defined over the field of

complex numbers.

To see the pseudocode for use of the FFT and inverse FFT for polynomial multiplication, see the Appendix. It uses the FFT twice, one for each polynomial being multiplied. It then carries out element-wise multiplications on the FFT outputs. Finally, we run the inverse FFT. All together, this algorithm has $O(n \log n)$ since we have roughly $3n \cdot \log_2 n + n$ major steps. We consider this efficient, and it certainly beats the naive $O(n^2)$ algorithm.

2.2 Inverting the FFT

In this section I will discuss how the forward algorithm (evaluation) can be adjusted to get a backwards algorithm (interpolation) to convert the product of the two polynomials from value representation to coefficient representation.

As mentioned in the introduction, the FFT was originally used for computing the Discrete Fourier Transform, which comes in different forms. The DFT can be thought of as a square matrix transformation applied on a column vector of coefficients to return a vector of values, for instance.

We can therefore consider how FFT works as a matrix transformation on a polynomial, to conceive of how its inverse might work. [4] Using matrices will also allow us to make use of intuitions from linear algebra. Our goal now is to determine how the original transformation matrix and its inverse relate, and to adapt our findings for the inverse FFT algorithm.

The forward transformation matrix, V, is a Vandermonde matrix, defined as follows.

Definition 2.2.1 (Vandermonde Matrix) *A Vandermonde matrix, V, is a matrix of rows $j = 0, 1, \ldots, n-1$ and columns $k = 0, 1, \ldots, n-1$ with elements $V_{j,k} = x_j^k$, where each x_j is a scalar.*

In other words, it is matrix in which each row is a geometric progression of some value. The product of a Vandermonde matrix and coefficients vector evaluates the polynomial at different values. For naive evaluation of a polynomial, we would have a matrix equation as follows:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

After deciding that x_k should be ω^k where $\omega = e^{\frac{2\pi i}{n}}$, we get the following matrix.

$$\begin{bmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

This transformation matrix is aforementioned DFT. Immediately, we notice that the matrix is symmetric, which supports our observation that only need about half as many points at every recursion step of the FFT, compared to naive evaluation. Each element of the matrix follows a simple pattern, $V_{jk} = \omega^{jk}$, where j and k are the row and column indices respectively, and counting begins at 0.

To undo the transformation, we have to determine the inverse of the transformation matrix and apply it to the vector of coefficients.

Notably, our matrix consists of complex numbers, which we will claim allows us an elegant way of finding the inverse.

To understand the claim, we require the following definition of a complex conjugate of a matrix, which will relate to the Vandermonde matrix later.

Definition 2.2.2 (Complex Conjugate of a Complex-valued Matrix) *For a matrix V where $v_{j,k} \in \mathbb{C}$ for $j = 0, \dots, n-1$ and $k = 0, \dots, n-1$, the complex conjugate of V is denoted as $\overline{V}$, and it has elements $\overline{v}_{j,k} = \overline{v_{j,k}}$.*

For an element $\overline{v}_{j,k}$ in exponential form, this would mean $\overline{v_{j,k}} = re^{-i\Phi}$ when $v_{j,k} = re^{i\phi}$ for $r, \Phi \in \mathbb{R}$.

We now claim we can find the inverse of a Vandermonde matrix to use in polynomial interpolation as follows

Theorem 2.2.1 (Inverse of a Vandermonde Matrix) *For a Vandermonde matrix V, its inverse V^{-1}, and its complex conjugate $\overline{V}$, we have the property that*

$$V^{-1} = \frac{1}{n}\overline{V}$$

Since the complex conjugate of a matrix is very easy to calculate, this theorem would imply that the inverse of the Vandermonde matrix can also easily be found, and always exists.

Proof:

To prove the above claim, we can prove the analogous claim that

$$P = V \cdot \overline{V} = nI$$

since we know that for any matrix and its inverse, $V \cdot V^{-1} = I$ by definition of I.

We also know that in general for the product of two matrices, A, any of its entry, a_{jk}, is given by the dot product of the j-th row of the first matrix, and the k-th column of the second matrix. In our case, where P is the product matrix and our two multiplying matrices are both symmetric, we can say p_{jk} is the dot product of the j-th row (or column) in V and the k-th row (or column) in $\overline{V}$.

Before we compute the dot product, we should understand what exactly appears in each entry of $\overline{V}$. Geometrically, the complex conjugate is a reflection of a point in the complex plane over the real axis. So for complex numbers in exponential form, the complex conjugate will negate the original number's exponent. Specifically for the roots of unity, our complex conjugate matrix will have entries $\overline{V}_{jk} = \omega^{-jk}$.

Our dot product can therefore be written as

$$p_{jk} = \sum_{m=0}^{n-1} \omega^{mj} \cdot \omega^{-mk} = \sum_{m=0}^{n-1} \omega^{m(j-k)}$$

A simple observation is that when $j = k$, the sum is

$$\sum_{m=0}^{n-1} \omega^{m \cdot 0} = \sum_{m=0}^{n-1} 1 = n$$

So this means all the diagonal entries are simply n. For the rest of the entries, we can evaluate the finite geometric series using the formula $S_n = \frac{r^n - 1}{r - 1}$.

$$p_{jk} = \frac{(e^{i2\pi(j-k)/n})^n - 1}{e^{i2\pi(j-k)/n} - 1} = \frac{e^{i2\pi(j-k)} - 1}{\dots} = 0$$

This result is because the numerator has $e^{i2\pi(j-k)} = 1^{(j-k)} = 1$. Now combining all entries together, we get the matrix

$$\begin{bmatrix} n & 0 & 0 & \dots & 0 \\ 0 & n & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & n \end{bmatrix}$$

which is just nI, as required. $\square$

Using the statement we just proved, it follows that the matrix multiplication for the inverse FFT, which uses the inverse DFT matrix, is simply

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix}$$

The only differences between the DFT matrix and the inverse DFT matrix is the normalising factor $\frac{1}{n}$, and the negative exponents. In practice this would mean the inverse FFT algorithm is the same as the forwards FFT algorithm, except we start with a slightly different choice values $x_k = \omega^{-k}$, and we divide by n at the end. The pseudocode outlining the full algorithm is given in the appendix.

These additional steps in the inverse FFT are both done in $O(1)$ time, so the overall time complexity for the inverse FFT is the same as the forwards FFT, $O(n \log n)$.

2.3 FFT Over Finite Fields

So far we have considered evaluation and interpolation of polynomials over arbitrary fields, and the FFT over complex numbers. We can extend these ideas to finite fields, but before we explore how, we must motivate the need to do so in the first place.

The main source of inefficiency for the FFT comes from the solution we devised for making the algorithm work on polynomials whose length is not a power of 2. Padding up the polynomial with 0 coefficients is fine for input lengths that are not too large. But at much larger values, the padding ends up causing a lot of unnecessary extra recursion. For instance, a polynomial of length $n = 16389$ is 5 terms too large to be $n = 2^{14}$. So we end up having to pad the polynomial by $2^{15} - 16389 = 16380$ terms of coefficient 0. Immediately, we can see how this can become excessive.

Instead, more involved adjustments can be made to the algorithm which would allow n to be the product of small primes, rather than just powers of 2. This would allow us to pad up the polynomials much less, in general.

Suppose we have 2 polynomials $A(x)$ and $B(x)$ both of degree n defined over a finite field $GF(q)$, where q is the order, and we want $C(x) = A(x)B(x)$. Using the FFT, we can get $A(x)$ and $B(x)$ in value representations and multiply them. We know $C(x)$ has degree $2n$, so according to Theorem 1.3.1, we need to evaluate the two polynomials at at least $2n - 1$ points. If $q < 2n - 1$, then this would not be possible over $GF(q)$ since we don't have enough elements in the finite field to evaluate over.

In order to acquire more elements, we can consider using an extension field of $GF(q)$, which would be a higher-order field that contains $GF(q)$ as a subfield. Since finite fields can only have orders which are prime numbers or prime powers, we know that the extension field should be of the form $GF(q^e)$ for some $e \in \mathbb{Z}^+$, $e > 1$, and $q^e > q$.

Finite fields have the property such that all of their non-zero elements are $q^e - 1$ -th roots of unity. For each e, we will search for a primitive r-th root of unity such that $r > 2n - 1$, and r is the product of only small primes (e.g. 2, 3, 5) found in the prime factorisation of $q^e - 1$. Then, our value for ω to use in the FFT would be the r-th root of unity. This means that for some choice of e, we can define our two polynomials over an extension field with a primitive r-th root of unity with the required properties. It goes without saying, we should aim to use the smallest choice of e possible.

Primitive roots of unity are ones which are not an m-th root of unity for any m satisfying $1 < m < n$, where n is a product of small primes. In other words, primitive roots of unity don't appear in smaller groups of roots of unity.

Now, we can pad the polynomials A(x) and B(x) to have degree $r - 1$ and begin splitting the polynomials to evaluate recursively, knowing that we can acquire $2n-1$ evaluations. However, unlike before, we don't have to split the polynomials into halves at every step. This is because r has prime factors other than 2, namely 3, or even 5. We will perform as many splits as there are prime factors in r, and each split will be the size of the current prime factor we are considering. For example, if $r = 48$ then we have prime factors $3 \cdot 2^4$. So we can split our polynomials in half 4 times and into thirds 1 time.

The implementation for this adjusted algorithm would be more complicated than the standard Cooley-Tukey algorithm which we have devised, but without too much impact on the runtime. This is because the worst-case algorithmic complexity would still be $O(n \log n)$ (since the worst case is when we have $n = 2^k + 1$ in the original algorithm, and for the new algorithm it is $n = p^k + 1$ for all the small primes p). Similarly, the best-case complexity would also be unchanged (best case for the original algorithm is when $n = 2^k$, and for the new algorithm would be $n = p^k$ for any of the small primes p). However, we could not claim that the FFT is optimised for extremely large n without considering where improvements could be made, which we have now done.

Chapter 3

Implementing the FFT

3.1 Goals of the Implementation

For this project, the intentions behind implementing the FFT and related code in a program are to:

1. Provide a deeper understanding of the mechanism behind the FFT, its strengths, and its weakness.
2. Verify the efficacy of the FFT and IFFT algorithms on their own and for fast polynomial multiplication.
3. Compare the relative speeds of the FFT with other evaluation algorithms, and the IFFT with other interpolation algorithms, to verify the complexity analyses in this report.

3.2 Limitations of the Implementation

Implementation the FFT and related code was done using Python. Python has a wide range of in-built features and libraries for mathematical and scientific scripting (such as for handling complex numbers), as well as visualising data (using *matplotlib*). Being a high-level language, Python code is also very readable, so it will reflect the pseudocode found in the Appendix without as many implementation-based distractions.

However, this comes at the cost of much less control over memory allocation and data usage, making Python code typically slower than code in other languages. Python is not fit for manipulating extremely large data sets or values requiring very high floating-point precision for this reason, but this will not detract from the goals outlined previously.

3.3 Testing the Efficacy of the FFT and IFFT

Tests for efficacy should be done by checking for equivalence between what we expect the output to be, and what the code computes. Initially, we can decide what to expect by doing manual computations on small examples. Then, we compare the outputs of other algorithms such as the DFT, Horner's Rule, and naive multiplication against the output of the FFT. Similarly, we check the output of Lagrange's interpolation algorithm against the output of the IFFT.

Another aspect of the FFT and IFFT's efficacy is to test the composite application of the FFT followed by the IFFT, and the IFFT followed by the FFT. We expect the output to be identical to the input, since the functions should have the following relationship

$$FFT^{-1}(x) = IFFT(x)$$

$$FFT(IFFT(x)) = x, \quad IFFT(FFT(x) = x$$

3.4 Testing the Runtime of FFT Multiplication

We will consider multiple ways of testing the runtime of the FFT multiplication algorithm. First, we can compare the runtime of FFT multiplication with the runtime of naive multiplication, and verify that the FFT multiplication is faster in general. Second, we can compare the runtime of the FFT at different input sizes and check if the growth is as expected according to our algorithmic complexity analysis.

The dependent variable in these tests will be the runtime measured in seconds. The actual timings are not of importance as these are machine and implementation dependent. However, the general trend which the timings follow as they vary on input size should be universal.

Our independent variable for testing in both cases will be the input size, n, which for each iteration of the test will be a power of 2. These values were chosen since we claimed the algorithmic complexity of FFT multiplication is $O(n \log_2 n)$, which gives us a nice ratio to measure how the time t grows.

If the runtime at n is

$$t = k \cdot (n \log_2(n))$$

then at $2n$ we would have

$$t = k \cdot (2n \log_2 2n) = 2kn \cdot (\log_2 2 + \log_2 n) = 2kn \cdot (1 + \log_2 n) \tag{3.1}$$

The growth factor of the runtime as n double would be

$$\frac{2kn \cdot (1 + \log_2 n)}{kn \cdot \log_2 n} = \frac{2(1 + \log_2 n)}{\log_2 n} = 2\left(\frac{1}{\log_2 n} + 1\right) \tag{3.2}$$

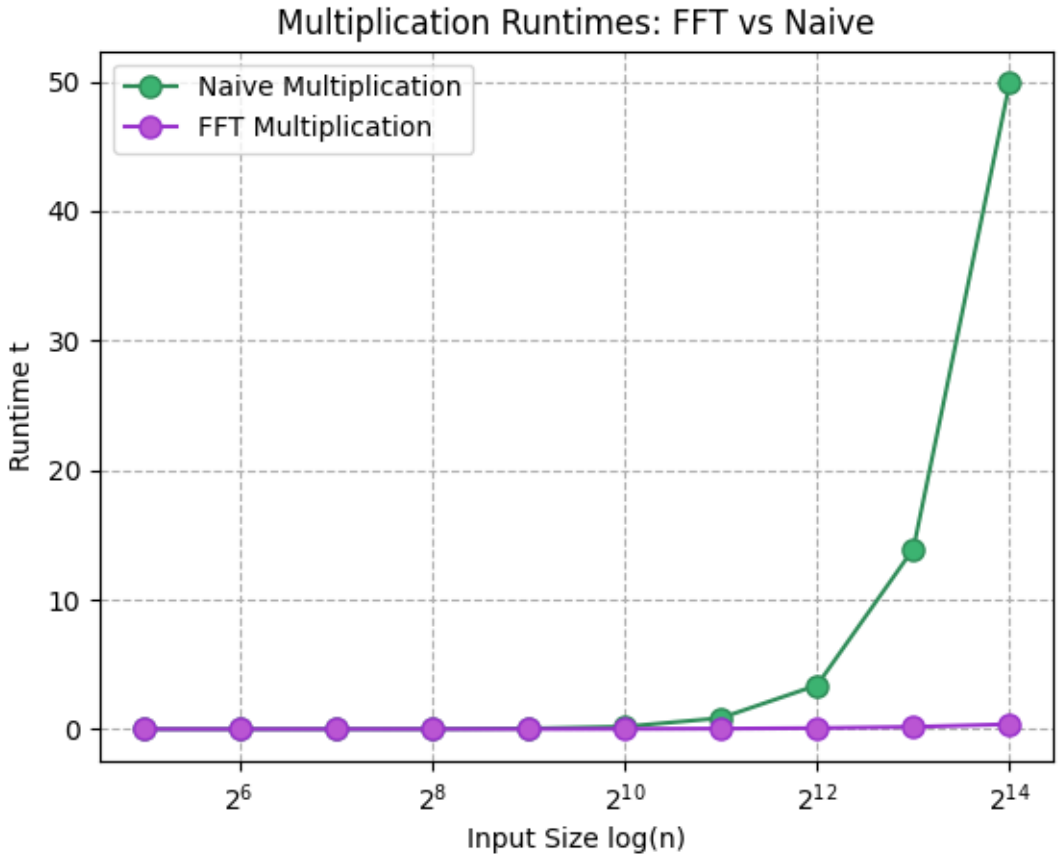

Figure 3.1: Plotting just the n values on a log scale shows us an intuitive comparison, in seconds, of just how much faster the FFT multiplication is compared to naive multiplication especially at larger values of n.

As $n \to \infty$, this expression tends to 2.

Using the same logic, we would expect the runtime of naive multiplication to grow by a factor of

$$\frac{k \cdot (2n)^2}{k \cdot n^2} = \frac{4n^2}{n^2} = 4 \tag{3.3}$$

So, approximately speaking, as the value of n doubles, the runtime of the naive multiplication method with be almost double that of the FFT multiplcaiton method.

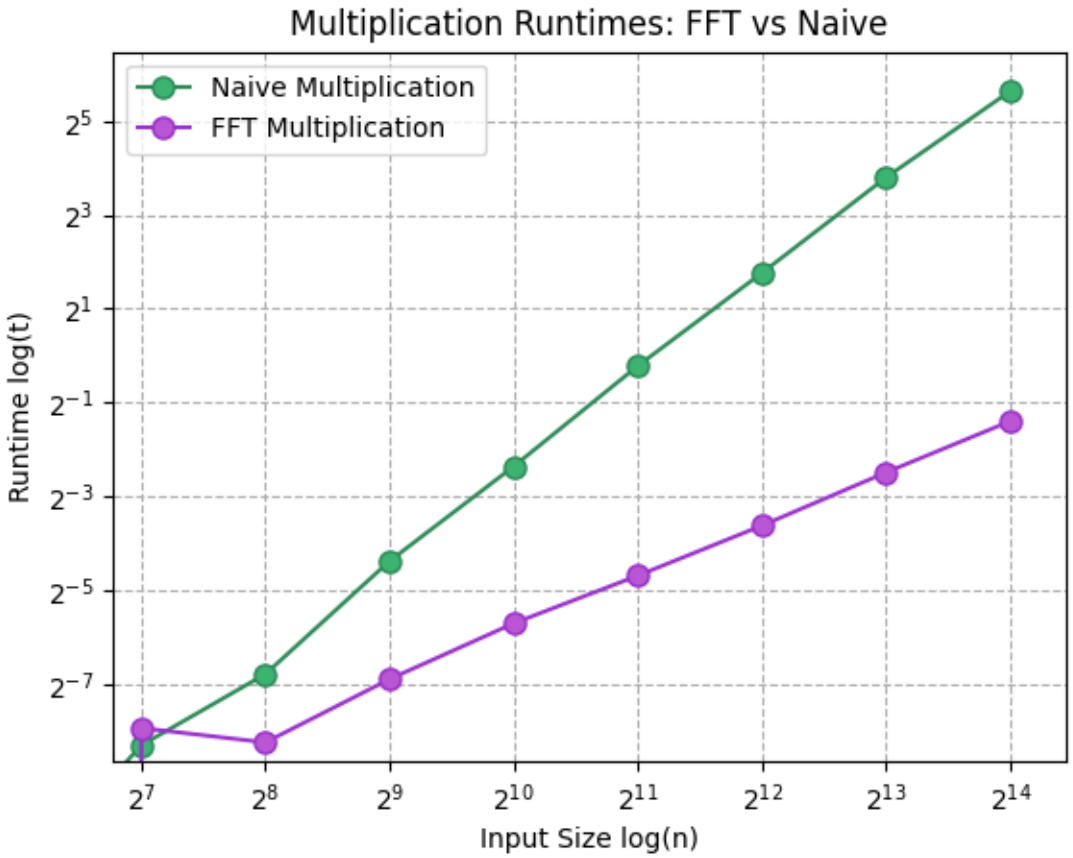

Figure 3.2: Plotting on a $\log_2(n)$-$\log_2(t)$ scale gives us linear plots of the runtime. For doubling values of n, we can see that the naive multiplication line almost twice the gradient of the FFT multiplication line. This makes sense since we expect the FFT line to have a gradient of just over 2, while the naive line has a gradient of 4, approximately. This supports our complexity analysis. Note that for smaller values of n, i.e. $n < 2^7$, it appears that the naive multiplication is actually faster. This is as expected since expression (3.2) (see above) is large for small values.

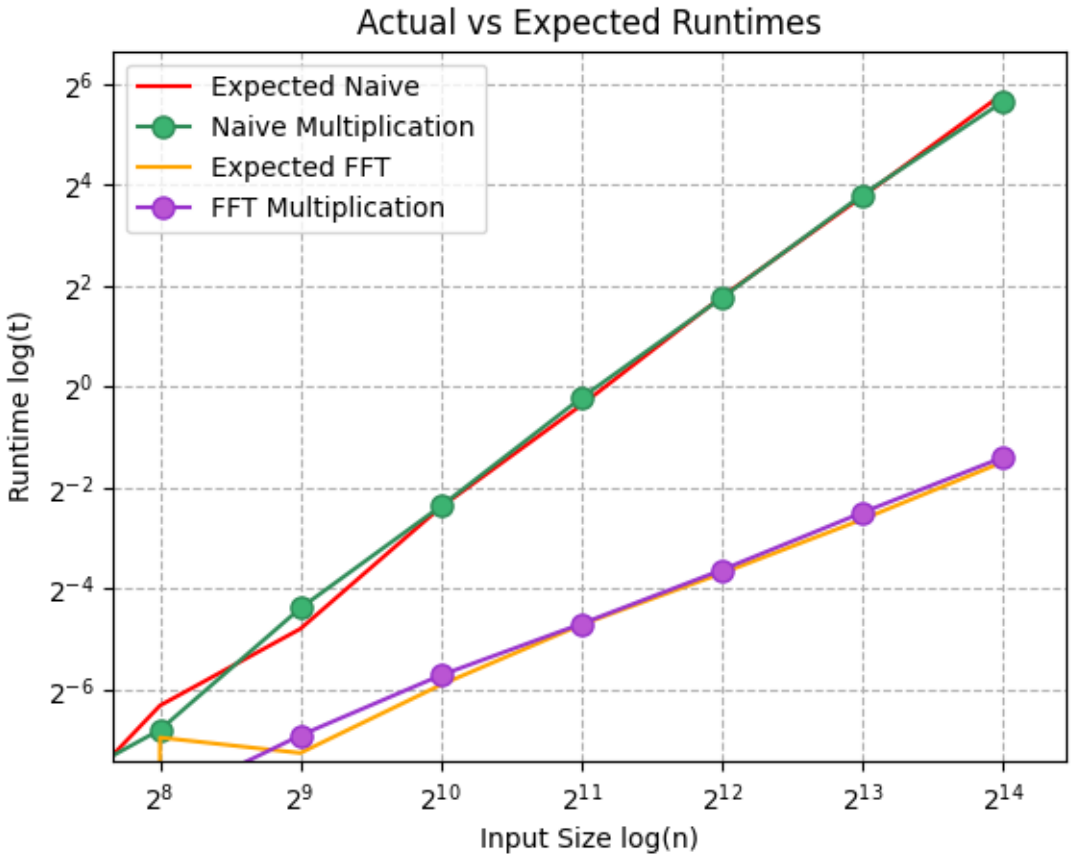

Figure 3.3: For this plot we calculated the expected growth factor multiplied by the current runtime to predict the runtime at the next value of n, and compared this to the actual runtimes for each algorithm. As we can cleary see, the empirical data closely aligns with our expectations. It would also be possible to use linear regression on $\log n$ and $\log t$ to check numerically if the gradients are as expected.

Appendix A

Pseudocode

Algorithm 1 Cooley-Tukey FFT for degree $n = 2^k - 1$ polynomials where $k \in \mathbb{Z}^+$

Require: P is a non-empty array of complex coefficients

```
procedure FFT(P)
    n ←len(P)
    if n = 1 then                                   ▷ Base case of recursion
        return P
    end if
    ω ← e^(2πi/n)
    P_even, P_odd ← P[:: 2], P[1 :: 2]              ▷ Split polynomial in half
    y_even, y_odd ←FFT(P_even), FFT(P_odd)          ▷ Recursive step
    y = [0] * n                                     ▷ Create an array to store output
    for j in range(n/2) do                          ▷ Calculate output using roots of unity
        y[j] ← y_even[j] + ω^j · y_odd[j]
        y[j + n/2] ← y_even[j] − ω^j · y_odd[j]
    end for
    return y
end procedure
```

Algorithm 2 Cooley-Tukey FFT for degree n polynomials where $n \in \mathbb{Z}^+$

Require: P is an array of complex coefficients

```
procedure FFT(P)                          ▷ Use an auxiliary procedure first to check n
    n ←len(P)
    if n < 1 then                         ▷ Handle degree 0 polynomials
        return
    end if
    if n not a power of 2 then
        pad P with 0s                     ▷ Pad with higher power terms of coefficient 0
    end if
    return Recursion(P)
end procedure

procedure RECURSION(P)
    n ←len(P)
    if n = 1 then                         ▷ Base case of recursion
        return P
    end if
    ω ← e^(2πi/n)
    P_even, P_odd ← P[:: 2], P[1 :: 2]    ▷ Split polynomial in half
    y_even, y_odd ←Recursion(P_even), Recursion(P_odd)   ▷ Recursive step
    y = [0] * n                           ▷ Create an array to store output
    for j in range(n/2) do                ▷ Calculate output using roots of unity
        y[j] ← y_even[j] + ω^j · y_odd[j]
        y[j + n/2] ← y_even[j] − ω^j · y_odd[j]
    end for
    return y
end procedure
```

Algorithm 3 Cooley-Tukey Inverse FFT

Require: P is a non-empty array of complex values

```
procedure IFFT(P)                                  ▷ Use an auxiliary procedure...
    n ←len(P)
    if n < 1 then
        return
    end if
    if n not a power of 2 then
        pad P with 0s
    end if
    output ←Recursion(P)
    return output / n                              ▷ ...to divide output by n at the end
end procedure

procedure RECURSION(P)
    n ←len(P)
    if n = 1 then
        return P
    end if
    ω ← e^(−2πi/n)                                 ▷ Negative exponent for roots of unity
    P_even, P_odd ← P[:: 2], P[1 :: 2]
    y_even, y_odd ← Recursion(P_even), Recursion(P_odd)
    y = [0] * n
    for j in range(n/2) do
        y[j] ← y_even[j] + ω^j · y_odd[j]
        y[j + n/2] ← y_even[j] − ω^j · y_odd[j]
    end for
    return y
end procedure
```

Algorithm 4 FFT for Polynomial Multiplication

Require: P and Q are both non-empty arrays of complex coefficients for polynomials of the same length, n

```
procedure FFT_MULTIPLICATION(P, Q)
    n ←len(P)
    P_values, Q_values ←FFT(P),FFT(Q)
    R_values ← [0] * n
    for i in range(n) do
        R_values[i] = P_values[i] * Q_values[i]
    end for
    return IFFT(R_values)
end procedure
```

Appendix B

Python Code

```python
# for fft.py, ifft.py, dft.py
from math import cos, sin, pi, exp, log, ceil

# for test_runtime.py
from time import time
from random import randint
from tabulate import tabulate

# for test_inbuilt_fft.py
import numpy as np
from scipy.fft import fft as scipy_fft
from scipy.fft import ifft as scipy_ifft

# for testing
from datetime import datetime as dt
import matplotlib.pyplot as plt

# for random_number_generating.py
from numpy.random import random_sample as np_rand
```

Listing B.1: "shared_import.py" - Imported libraries used in all multiple other files.

```python
from shared_imports import *
# Functions which are used across multiple files


def rect_form(theta, r=1, is_rounded=True, decimal_places=15):
    """theta: float, r: float, is_rounded: bool, decimal_places: int
    Convert a complex number in exponential form to its rectangular form, a+bi
    Auxiliary to the FFT and IFFT function."""
    if is_rounded == True:
        return round(r * cos(theta), decimal_places), round(r * sin(theta), decimal_places)
    else:
        r * cos(theta), r * sin(theta)

```

```
14
15 def is_not_power_of_2(N):
16     """N: int
17     Use bit-wise operator trick to check for powers of 2.
18     Auxiliary to the FFT function."""
19     return True if N <= 0 or N & (N - 1) != 0 else False
20
21
22 def padding(P):
23     """P: array of complex numbers
24     Add 0s to the front of an array of coefficients to reach an
       array length that is a power of 2.
25     Auxiliary to the FFT function."""
26     N = len(P)
27     next_pow_2 = pow(2, ceil(log(N) / log(2)))
28     padding = next_pow_2 - N
29     padded_polynomial = ([0] * padding) + P
30     return padded_polynomial
```

Listing B.2: "shared_functions.py" - This file contains functions for converting complex numbers to different forms

```
1  from shared_imports import *
2  from rounding import *
3  from shared_functions import is_not_power_of_2, padding, rect_form
4
5  def FFT_recursion(P):
6      """P: array of complex values
7      Get values from coefficients of a polynomial.
8      This algorithms has O(nlogn) time complexity."""
9
10     N = len(P)
11
12     # Base case: recursion stops when the polynomial is split into
       individual terms
13     if N == 1:
14         return P
15
16     # Define the first of the Nth roots of unity
17     a, b = rect_form(2 * pi / N) # Takes the angle as argument
18     w = complex(a, b)
19     W = [w**k for k in range(N // 2)]  # Store the first n/2 n-th
       roots of untiy
20
21     # Partition the polynomial by the parity of each term's position
22     P_even, P_odd = P[0::2], P[1::2]
23
24     # Recursive call to keep splitting the polynomial in half
25     V_even, V_odd = FFT_recursion(P_even), FFT_recursion(P_odd)
26
27     # Create a empty array to store the polynomial's values in
28     value_rep = [0] * N
29
30     # Calculate values in half the expected time by exploiting
```

```
    symmetries
    for k in range(N // 2):
        # print(f"n = {k}, w^n = {W[k]}")
        value_rep[k] = V_even[k] + W[k] * V_odd[k]
        value_rep[k + N // 2] = V_even[k] - W[k] * V_odd[k]

    return value_rep


def FFT(P):
    """
    Prepare the argument to pass to the FFT_recursion function.
    """
    # Create a copy of the argument to not alter it
    A_coeff = [coeff for coeff in P][::-1]
    N = len(A_coeff)
    # Use a bitwise AND operation to test if the array length is a
    power of 2
    if is_not_power_of_2(N):
        A_coeff = padding(A_coeff)
    A_val = FFT_recursion(A_coeff)

    return A_val


if __name__ == '__main__':
    A_coeff = [2, 3, 7, 8]
    A_val = FFT(A_coeff)
    print(A_coeff)
    print(A_val)  # should return [(20+0j), (-5-5j), (-2+0j), (-5+5j
    )]
```

Listing B.3: "fft.py" - Defining the FFT using two functions: one for padding the polynomial and one for recursion.

```
from shared_imports import *
from shared_functions import rect_form


def IFFT_recursion(P):
    """P: array of complex values
    Get coefficients from values of a polynomial.
    This algorithms has O(nlogn) time complexity."""

    N = len(P)

    # Check if the number of terms in the polynomial is a power of 2
    if N > 0 and N & (N - 1) != 0:
        return
        # need to pad the polynomial up

    # Base case: recursion stops when the polynomial is split into
    individual terms
    if N == 1:
```

```
        return P

    # Define the first of the Nth roots of unity
    a, b = rect_form(2 * pi / N)
    w = complex(a, -b)
    W = [w ** k for k in range(N // 2)]  # Store the first n/2 n-th
   roots of untiy

    # Partition the polynomial by the parity of each term's position
    P_even, P_odd = P[0::2], P[1::2]

    # Recursive call to keep splitting the polynomial in half
    V_even, V_odd = IFFT_recursion(P_even), IFFT_recursion(P_odd)

    # Create a empty array to store the polynomial's values in
    coeff_rep = [0] * N

    # Calculate values in half the expected time by exploiting
   symmetries
    for k in range(N // 2):
        coeff_rep[k] = V_even[k] + W[k] * V_odd[k]
        coeff_rep[k + N // 2] = V_even[k] - W[k] * V_odd[k]

    return coeff_rep

def IFFT(P):
    """
    Final steps after IFFT_recursion.
    """
    # Create a copy of the argument data
    A_val = [val for val in P]
    A_coeff = IFFT_recursion(A_val)
    # Divide each resulting coefficient value by N
    A_coeff = [c / len(A_coeff) for c in A_coeff]

    return A_coeff[::-1]

if __name__ == '__main__':
    A_val = [(20+0j), (-5-5j), (-2+0j), (-5+5j)]
    A_coeff = IFFT(A_val)
    print(A_coeff)  # should return [2, 3, 7, 8]
    print(A_val)
```

Listing B.4: "inverse_fft.py" - Defining the IFFT using two functions: one for recursion and one for dividing by n after recursion.

```
def multiply_values(A, B):
    """A and B are arrays of complex numbers.
    Carries out an element-wise multiplication, returns the product
   as an array"""
    C = [0 for _ in range(len(A))]
    for i in range(len(C)):
        C[i] = A[i]*B[i]
    return C
```

```
8
9  if __name__ == '__main__':
10     A = [(1+0j), 1j, (-1+0j), (-0-1j)]
11     B = [(1+0j), 1j, (-1+0j), (-0-1j)]
12     # A = [1, 0, 1]
13     # B = [3, 4] # to check padding..
14     C = multiply_values(A, B)
15     print(C)
```

Listing B.5: "multiply_values.py" - Defining a function for the element-wise multiplication of two arrays of numbers. Will be used as "value representation multiplication" for the FFT Multiplication code.

```
1  def naive_multiplication(A, B):
2      # Lengths & degrees of A(x) and B(x)
3      n_A = len(A); n_B = len(B)
4      d_A = n_A - 1; d_B = n_B - 1
5
6      # Degree & length of C(x), the product
7      d_C = d_A + d_B; n_C = d_C + 1
8
9      C = [0 for i in range(n_C)]
10     # Calculate new coefficients as the sum of pair-wise products of
       old coefficients
11     for i in range(n_A):
12         for j in range(n_B):
13             C[i + j] = C[i + j] + A[i] * B[j]
14
15     return C
16
17 if __name__ == '__main__':
18     A = [2, 3, 7]
19     B = [1, 0, 2]
20     # output as expected, from manual calculation: [2, 3, 11, 6, 14]
21     C = naive_multiplication(A, B)
22     print(C)
```

Listing B.6: "naive_polynomial_multiplication.py" - Defining a function to carry out the naive method of polynomial multiplication in quadratic time.

```
1  from multiply_values import multiply_values
2  from naive_polynomial_multiplication import naive_multiplication
3  from rounding import round_nums
4
5  def naive_polynomial_evaluation(P, X):
6      """
7      Evaluate the polynomial, P, given as an array of its
       coefficients at each of the values in array X
8      Return an array, Y, where Y[i] = P(X[i])
9      """
10     Y = [0] * len(X)
11     for i in range(len(X)):
12         # Evaluate the polynomial at each X[i]
13         for j in range(len(P)):
```

```
            # Evaluate each term of the polynomial at X[i] and add them
            Y[i] += P[j] * pow(X[i], len(P)-1-j)
    return Y

def horners_rule(A, x):
    """Use horner's rule to evaluate a polynomial A once, on a number x.
    Auxiliary function for horners_polynomial_evaluation function.
    """
    n = len(A)
    y = A[0]
    for i in range(n-1):
        y = A[i+1] + x*y
    return y

def horners_polynomial_evaluation(A, X):
    """Use horner's rule repeatedly to evaluate polynomial A, on an array of numbers X"""
    Y = []
    for x in X:
        Y.append(horners_rule(A, x))
    return Y

def lagrange_polynomial(X, i):
    """Calculate the i-th lagrange polynomial for a list of x values.
    Auxiliary function for lagrangian_polynomial_interpolation function."""
    X_ = X[:i] + X[i+1:] # all of X except X[i]

    denominator = 1
    for j in range(len(X_)):
        denominator *= X[i] - X_[j] # (xi - x1), (xi - x2), etc excluding (xi - xi)

    numerator = [1, -X_[0]]
    for j in range(1, len(X_)):
        factor = [1, -X_[j]] # (x - x1), (x - x2), etc excluding (x - xi)
        numerator = naive_multiplication(numerator, factor)

    return [numerator[j] / denominator for j in range(len(numerator))]

def lagrangian_polynomial_interpolation(X, Y):
    """
    Given length n arrays of X and Y values, return the coefficients of a unique degree n-1 polynomial
    """
    n = len(X)
    L = [lagrange_polynomial(X, i) for i in range(n)]

    YL = [0] * n # YL will be a list of the Y[i] * L[i] polynomials
```

```
    for i in range(len(L)): # For each polynomial in L
        YL[i] = [L[i][j]*Y[i] for j in range(len(L[i]))]  # Multiply each term in L_i(x) by y_i

    P = [0] * len(YL[0])
    for i in range(len(YL[0])): # For each term in a polynomial
        for j in range(len(YL)): # For the current polynomial
            P[i] += YL[j][i]

    return P


if __name__ == '__main__':
    P1 = [1, 0, 1]
    P2 = [3, 4]
    X = [0, 1, 2, 3]
    Y0 = horners_polynomial_evaluation(P2, X)
    Y1 = naive_polynomial_evaluation(P1, X)
    Y2 = naive_polynomial_evaluation(P2, X)
    Y3 = multiply_values(Y1, Y2)
    P3 = lagrangian_polynomial_interpolation(X, Y3)
    expected = naive_multiplication(P1, P2)

    print(f"P1 output = {Y1}")
    print(f"P2 output = {Y2}")
    print(f"Honer's Rule output: {Y0}") # [4, 7, 10, 13]
    print(f"Product polynomial's value form  = {Y3}") # [4, 14, 50, 130], using element-wise multiplication
    print(f"P3 = {round_nums(P3)}") # [3.0, 4.0, 3.0, 4.0] from lagrange interpolation
    print(f"Expected = {expected}") # [3, 4, 3, 4], from multiplication in value form
    print(f"P3 as expected? {round_nums(P3) == expected}")
```

Listing B.7: "slow_evaluation_and_interpolation.py" - Defining functions for Horner's Rule and Lagrange's Inteprolation algorithm to help test the efficacy of the FFT and IFFT by checking for equivalence of outputs given the same input.

```
from math import cos, sin, pi
from shared_functions import rect_form


def DFT_real_input(X):
    """
    For an array of real numbers, return its DFT as an array of complex numbers.
    Complex numbers are given as arrays of length 2,
    where the first element is the real component and second element is the imaginary component.
    This algorithm has O(N^2) time complexity.
    """
    N = len(X)
    Y = [[0, 0] for _ in range(N)]

```

```
    for k in range(N):  # k=0,1,2
        for n in range(N):  # n=0,1,2
            theta = -2 * pi * n * k / N
            a, b = rect_form(theta)
            a, b = round(a * X[n], 8), round(b * X[n], 8)
            # print(f"{a} + {b}i")
            Y[k] = [round(Y[k][0] + a, 5), round(Y[k][1] + b, 5)]

        print(Y[k])


def complex_mult(w, v):
    """
    For two iterables in the forms [a, b] and [c, d] where a, b, c, d are real numbers,
    return [ac - bd, bc + ad], which represents their product as a complex number.
    """
    return [w[0]*v[0] - w[1]*v[1], w[1]*v[0] + w[0]*v[1]]


def DFT_complex_input(X):
    """
    For an array of complex numbers, return its DFT as an array of complex numbers.
    Complex numbers are given as arrays of length 2,
    where the first element is the real component and second element is the imaginary component.
    This algorithm has O(N^2) time complexity.
    """
    N = len(X)
    Y = [[0, 0] for _ in range(N)]

    for k in range(N):  # k=0,1,2
        for n in range(N):  # n=0,1,2
            theta = -2 * pi * n * k / N
            a, b = rect_form(theta)
            a, b = complex_mult([a,b], X[n])
            a, b = round(a, 8), round(b, 8)
            print(f"{a} + {b}i")
            Y[k] = [round(Y[k][0] + a, 5), round(Y[k][1] + b, 5)]

        print(Y[k])

if __name__ == '__main__':
    A = [2, 3, 7, 8]
    B = [1, 0, 2]

    # Polynomials A and B evaluated at the 1st, 2nd, and 3rd roots of unity
    DFT_real_input(A)

    #C = [[2,1],[3,2],[0,5]]
    C = [[2,1],[3,1],[7,1]]
```

```
64      # DFT_complex_input(C)
```

Listing B.8: "dft.py" - Defining a function for computing the DFT of an array of coefficients. This method takes quadratic time since it does not employ the same tricks for efficiency as the FFT

```
1  from fft import FFT
2  from inverse_fft import IFFT
3  from multiply_values import multiply_values
4
5  def FFT_multiplication(A_coeff, B_coeff):
6      # Step 1 - Evaluting
7      A_val = FFT(A_coeff)
8      B_val = FFT(B_coeff)
9      # Step 2 - Multiplying
10     C_val = multiply_values(A_val, B_val)
11     # # Step 3 - Interpolation
12     C_coeff = IFFT(C_val)
13
14     return C_coeff
15
16 if __name__ == '__main__':
17     # A = [2, 3, 7]
18     # B = [1, 0, 2] # expected output from naive multiplication: [2,
       3, 11, 6, 14]
19     A = [1, 0, 1]
20     B = [0, 3, 4] # expected output from naive multiplication: [3,
       4, 3, 4]
21     C = FFT_multiplication(A, B)
22     print(C)
```

Listing B.9: "fft_polynomial_multiplication.py" - Defining a function for using the FFT and IFFT on two for fast polynomial multiplication.

```
1  from shared_imports import *
2  from fft import FFT
3  from inverse_fft import IFFT
4
5
6  def random_polynomial(n=1, min=-1000, max=1000, is_complex=False):
7      """n, min, max: int, is_complex: bool to define the polynomial
       over real or or complex values
8      Generate a random polynomial with coefficients within the
       specified bounds."""
9      diff = max - min
10     if is_complex == True:
11         return [complex(min + np_rand() * diff, min + np_rand() *
       diff) for _ in range(n)]
12     else:
13         return [min + np_rand() * diff for _ in range(n)]
14
15
16 if __name__ == "__main__":
17     rand_pol = random_polynomial(n=3)
```

```
    fft_rand_pol = FFT(rand_pol)
    ifft_rand_pol = IFFT(fft_rand_pol)
    print(rand_pol)
    print(fft_rand_pol)
    print(ifft_rand_pol)
```

Listing B.10: "random_number_generating.py" - Defining a function for generating polynomials pesudo-randomly to use for runtime testing.

```
def round_complex(number, decimal=5):
    """number: complex number, decimal: int"""
    return complex(round(number.real, decimal), round(number.imag, decimal))


def round_complex_nums(numbers, decimal=5):
    """numbers: array of complex numbers, decimal: int"""
    return [round_complex(number, decimal=decimal) for number in numbers]


def round_real_nums(numbers, decimal=5):
    """numbers: array of real numbers, decimal: int"""
    return [round(number, decimal) for number in numbers]


def round_nums(numbers, decimal=5):
    """
    numbers: array of numbers, decimal: int
    Only use if input is known to be either all complex or all real, but unsure which
    """
    if isinstance(numbers[0], complex):
        return round_complex_nums(numbers, decimal=decimal)
    else:
        return round_real_nums(numbers, decimal=decimal)

if __name__ == '__main__':
    A = [complex(1,2), 3, 4, complex(6,7)]
    round_nums(A)
```

Listing B.11: "rounding.py" - Functions used to round floating point values

```
from fft import FFT
from inverse_fft import IFFT
from numpy import mean


def test_forward_inversion(coeffs):
    """Test if IFFT(FFT(P)) = P"""
    error = [abs(coeff_1 - coeff_2) for coeff_1, coeff_2 in zip(coeffs, IFFT(FFT(coeffs)))]
    print(f"Forward Inversion Error = {error}")
    if mean(error) < 0.001:
        print("Inversion Successful")
```

```
    else:
        print("Inversion Failed")

def test_backward_inversion(values):
    """Test if FFT(IFFT(P)) = P"""
    error = [abs(value_1 - value_2) for value_1, value_2 in zip(
   values, FFT(IFFT(values)))]
    print(f"Backward Inversion Error = {error}")
    if mean(error) < 0.001:
        print("Inversion Successful")
    else:
        print("Inversion Failed")

if __name__ == '__main__':
    coeffs = [(2 + 0j), (3 + 0j), (7 + 0j), (8 + 0j)]
    values = [(20 + 0j), (-5 - 5j), (-2 + 0j), (-5 + 5j)]
    test_forward_inversion(coeffs)
    test_backward_inversion(values)

    ## Output:
    # Forward Inversion Error = [0.0, 0.0, 0.0, 0.0]
    # Inversion Successful
    # Backward Inversion Error = [0.0, 0.0, 0.0, 0.0]
    # Inversion Successful
```

Listing B.12: "test_efficacy.py" - Test whether composing the FFT and IFFT produce an output equal to the input.

```
from shared_imports import *
from naive_polynomial_multiplication import naive_multiplication
from fft_polynomial_multiplication import FFT_multiplication

def test_timing(f):
    """For an algorithm defined as a function, f,
    time its completion on different input sizes.
    Tabulate and return the time taken"""

    k = list(range(5, 15)) # range of input sizes to use input sizes
    n = [2**i for i in k] # all input sizes in an array
    number_of_tests = len(n)
    timings = [] # store the timings

    for i in range(number_of_tests):
        # Generate two polynomials of random coefficients
        A = [randint(0, 999) for _ in range(n[i])]
        B = [randint(0, 999) for _ in range(n[i])]

        t = time() # start timer
        f(A, B) # run algorithm
        t = time() - t # stop timer
        timings.append(t) # save the time
        print(f"{i+1}/{number_of_tests} tests done! n={n[i]} run
   complete! Time taken: {t}")

```

```
    # Tabulate and print
    table = {"n": n, "time (s)": timings}
    print(f"{f.__name__.replace('_', ' ')}:")
    print(tabulate(table, headers="keys"))

    return timings, n, k

def save_data_to_csv(function, label, data):
    """function: used to get the name of the test function as the title of the file,
    label: string to specify which algorithm the data is from,
    data: a 2D array of all the output from the testing
    This function will parse data and save it to a text file in CSV format. """

    unique_stamp = dt.now().strftime('%H%M%S%f') # so each run of the test has its own file
    file_name = f"{function.__name__}_{label}_{unique_stamp}"
    file_path = "Output/" + file_name + ".txt"
    with open(file_path, 'w+') as f:
        for i in range(len(data)): # each row
            for j in range(len(data[0])):  # each column
                f.write(f"{data[i][j]}") # record each piece of data
                if j != len(data[0]) - 1: # no comma on end of line
                    f.write(",")
            f.write("\n") # new line for each test iteration's data
        print(f"Successfully saved data to {file_path}")


if __name__ == '__main__':
    t_naive, n, k = test_timing(naive_multiplication)
    t_fft, n, k = test_timing(FFT_multiplication)
    save_data_to_csv(test_timing, "naive", [k, n, t_naive, t_fft])
```

Listing B.13: "test_runtime.py" - Measure the runtime in seconds of the FFT Multiplication and Naive Multiplication functions

```
from shared_imports import *

if __name__ == '__main__':

    k, n, t_naive, t_fft = np.transpose(np.loadtxt('Output/test_timing_naive_015624232811.txt',
                                    unpack=True,
                                    delimiter=','))
    # t_naive_expected = [t_naive[0] * 4] + [t_naive[i] * 4 for i in range(len(t_naive[:-1]))]
    # t_fft_expected = [t_fft[0] * 2] + [t_fft[i] * 2 for i in range(len(t_fft[:-1]))]
    # t_naive_expected = [(n[i] ** 2) / (10 ** 6.7) for i in range(len(n))]
    # t_fft_expected = [(n[i] ** 0.5) / (10 ** 1) for i in range(len(n))]

```

```
    # plt.plot(n, t_naive_expected, label='Expected Naive', color="red")
    plt.plot(n, t_naive, label=f'Naive Multiplication',
        color='seagreen', marker='o', markersize='8',
        markerfacecolor='mediumseagreen', markeredgecolor='seagreen')

    # plt.plot(n, t_fft_expected, label="Expected FFT", color="orange")
    plt.plot(n, t_fft, label=f'FFT Multiplication',
        color='darkorchid', marker='o', markersize='8',
        markerfacecolor='mediumorchid', markeredgecolor='darkorchid')


    ### Linear Regression Attempt
    # fit_naive = np.polyfit(np.log(n), np.log(t_naive), 1)
    # fit_fft = np.polyfit(np.log(n[2:]), np.log(t_fft[2:]), 1)
    # [1.92373537 - 14.8865877]
    # [1.01709443 - 10.95497214]
    # view the output of the model
    # print(fit_naive)
    # print(fit_fft)
    # plt.plot(n, [fit_naive[1] + fit_naive[0] * i for i in n])
    # plt.plot(n, [fit_fft[1] + fit_fft[0] * i for i in n])
    # plt.plot(n, [10 ** fit_naive[1] * i ** fit_naive[0] for i in n], label="naive regression")
    # plt.plot(n, [10 ** fit_fft[1] * i ** fit_fft[0] for i in n], label="fft regression")

    plt.grid(linestyle='--')
    plt.legend()
    plt.title("Multiplication Runtimes: FFT vs Naive")
    plt.xlabel('Input Size log(n)')
    plt.ylabel('Runtime log(t)')
    plt.xscale('log', base=2)
    plt.yscale('log', base=2)
    plt.show()
```

Listing B.14: "plotting.py" - Plotting the runtime testing output for FFT Multiplication and Naive Multiplicaiton graphically from a csv file of data.

Bibliography

[1] Victor Adamchik. Carnegie Mellon University, Algorithm Design and Analysis Lecture 2, Karatsuba Algorithm for Polynomial Multiplication. https://www.cs.cmu.edu/ 15451-s15/LectureNotes/lecture02.pdf, 2015.

[2] Birne Binegar. Oklahoma State University, Numerical Analysis Lecture 17, Algorithms for Polynomial Interpolation. https://math.okstate.edu/people/binegar/4513-F98/4513-l17.pdf, 1998.

[3] Alan Cobham. Cobham's thesis, the intrinsic computational difficulty of functions. https://www.cs.toronto.edu/ sacook/homepage/cobham_intrinsic.pdf, 1965.

[4] Erik Demaine. MIT OpenCourseWare, Design and Analysis of Algorithms Course, Divide & Conquer: FFT. https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/pages/lecture-notes/, 2015.

[5] Matthew Ondrus. A generalization of even and odd functions. *Involve*, 4(1), 2011.