Bachelor Project: Polynomial Multiplication

Author: Rasmus Ladefoged
Supervisor: Srikanth Srinivasan

Department of Computer Science
University of Copenhagen
Denmark
November 27, 2024

Contents

1 Abstract

This thesis investigates the optimization of polynomial multiplication algorithms. This is fundamental in computational mathematics with widespread applications in areas such as digital signal processing and cryptography. Beginning with the conventional naive approach, which operates with quadratic time complexity, the study advances to more advanced methods including Karatsuba's algorithm and Fast Fourier Transform (FFT) techniques which improve on this runtime. The algorithms discussed in this thesis are:

- Naive Polynomial Multiplication
- Karatsuba's algorithm tailored to Polynomial multiplication
- Discrete Fourier Transform
- Recursive Fast Fourier Transform
- Iterative Fast Fourier Transform

Through a detailed examination encompassing pseudo-code, implementation in C, and empirical testing, the study quantitatively analyzes each algorithm's time and space complexities. The findings demonstrate that while Karatsuba's algorithm significantly improves on the naive method, FFT methods further enhance performance. This thesis highlights the importance of Amortized runtime analysis and its shortcomings, when theory becomes practical. Further then this thesis also contains a detailed walkthrough of converting pseudo-code to actual implementation and the challenges that lie therein.

The sourcecode for this project can found on my Github: https://github.com/RasmusLC1/Polynomial-Multiplication

2 Introduction

In this paper we will be exploring the optimisation of polynomial multiplication algorithms. Algorithmic optimisations, such as polynomial multiplication is crucial in many fields. For polynomial multiplication, this include cryptography, signal processing and many more. We will start with the naive approach which is the intuitive approach that is taught in school. Then we will look at an optimisation of this in the form of Karatsuba's algorithm. Afterwards we will look at alternative ways to calculate polynomials using the Discrete Fourier Transform algorithm. We will then optimise that using first a recursive and then an iterative approach to the Fast Fourier Transform algorithm.

We will be testing and discussing these implementations, their advantages and disadvantages and see if the practical aligns with the theoretical. The goal of this paper is to explore polynomial multiplication as this is a fundamental and important part of many large projects and if you can optimise the foundations of those projects. Then the effects will scale to the rest of the project as well. While this paper focuses on polynomial multiplication, the principles are applicable to a broad range of computational problems.

3 Choice Of Programming Language

Before we even begin we need to discuss how to implement the algorithms, we need to determine what programming language we will implement them in. The easiest and most straight forward approach would be to use Python, which is great for mathematical applications and it is very easy to handle polynomials with it due to its straightforward array setup.

The problem with Python is that is it very slow since it is an interpreted language with dynamic typing, which are both great for writing the code quickly and cleanly. But if we want to observe the runtime accurately, then we need something much faster.

For this we will be using C, which is compiled, meaning we do not have to interpret the code line by line as we would with Python. This will greatly increase the runtime of the code. Secondly it is statically typed. We will specify each type for the compiler to interpret, thereby again improving the runtime.

We also get direct access to memory, which is a double edged sword. C is notoriously dangerous to write in due to this, but being able to allocate pointers to memory will greatly help in increasing the efficiency of the code. We will also be able to allocate and free the exact amount of memory we need and specifically when we need it.

There is also little to no overhead except for the overhead we create ourselves. This means that when running the code we will get a reliable and predictable result every time. It also means that the overhead that another programming language might handle in the background will be visible to us.

Finally C is also very low level and close to assembly code, meaning things like multiplication, loops and if statements will run very fast since they will be translated straight to assembly code by C.

All these things combined means that C would be the ideal language for this project. I will be compiling the code with the -O2 flag for optimisation to showcase a realistic scenario. This flag is commonly used to instruct the compiler to calculate constants at compilation time instead of runtime, improve loops, copy propagation and many other things. What this does in effect is make the naive approach much faster, which will make the task of outperforming it more challenging. But it will also showcase the importance of algorithms better if we succeed in outperforming the naive approach.

4 Naive Polynomial Multiplication

The most straightforward way to multiply polynomials, is the naive approach. Here you multiply each value from the first polynomial with every value from the second polynomial and add them together at the end. This approach is very simple, but it is also very time consuming as it runs in $O(n^2)$ time. The goal of this paper is to improve this and explore better polynomial multiplication methods.

To implement naive polynomial multiplication is very straight forward, since it is a very simple algorithm. The code can be seen below:

```
void Naive_Polynomial_Multiplication(int *input1, int *input2, int n, int *out){
    for (int i = 0; i < n; i++) { // For each element of input1
        for (int j = 0; j < n; j++) { // For each element of input2
            out[i + j] += input1[i] * input2[j];
        }
    }
}
```

Here we have two for loops that iterate over each number in the polynomials, which is what creates the runtime of $O(n^2)$.
Next we will be looking at the first real successful attempt at improving this algorithm's runtime, Karatsuba's algorithm.

5 Karatsuba's algorithm

The idea with Karatsuba's algorithm is to use a divide-and-conquer approach to optimize the multiplication of large numbers[12], for this paper we will be modifying it to work with polynomials. The optimisation in Karatsuba's algorithm is achieved by splitting each polynomial into halves: a high part and a low part, based on their position relative to the midpoint of the polynomial's decimal representation. For two polynomials to be multiplied, we perform recursive multiplications on their high and low parts. This can be seen in the pseudo-code below. We were unable to find a polynomial version of Karatsuba's algorithm, but the overall idea stays the same, so we will start by explaining the version that handles numbers in the pseudo-code and then discuss the polynomial version in the actual implementation.

5.1 Pseudo-Code

```
function karatsuba(num1, num2)
    input: integer num1 and integer num2
    output: num1 * num2
    // Base case for recursion
    if (num1 < 10 or num2 < 10)
        return num1 * num2

    m = max(size_base10(num1), size_base10(num2))
    m2 = floor(m / 2) // Half the size, adjusted for odd digits

    high1, low1 = split_at(num1, m2)
    high2, low2 = split_at(num2, m2)

    z0 = karatsuba(low1, low2)  // Product of the low parts
    z2 = karatsuba(high1, high2)  // Product of the high parts
    z1 = karatsuba(low1 + high1, low2 + high2)  // Product of sums

    // Combine the results with appropriate positional shifts
    return (z2 * 10 ^ (m2 * 2)) + ((z1 - z2 - z0) * 10 ^ m2) + z0
```

Here we see that we start by checking if either num1 or num2 er less than 10, if that's the case then the naive approach is the best.

If they are greater than 10, then we find the larger of the two numbers and set that to m. Then we half m and store it as m2, which is important for the divide step. Then we floor it to account for odd numbers.

Then we get to the triple recursive step, were we take the lowest half of each number and call karatsuba with them, then the same for the highest parts. in the number 1204, then low would be 04 and high would be 12. Then finally we call a third time where we take the sum of the low and high together so high+low.

Finally we do the conquer step, where we combine the results together and return the result.

5.2 Implementation of Karatsuba

We walked through the pseudo-code for Karatsuba's algorithm for numbers, but we will now look at our implementation for polynomials:

```
// Karatsuba multiplication for polynomials
void Karatsuba_Polynomial(int *input1, int *input2,int length_input1,
                          int length_input2, int *result) {

    // Check if either number is 2 digits, if yes then multiply.
    // C is really fast for small number multiplication,
    // so it doesn't need to check for 1 digit
    if (length_input1 <= 1 || length_input2 <= 1) { // Base case for the smallest size
        Array_Multiplication(input1, input2, length_input1, length_input2, result);
        return;
    }
    // Calculate half length
    int half_length1 = (length_input1 >> 1) + (length_input1 % 2);
    int half_length2 = (length_input2 >> 1) + (length_input2 % 2);

    // Find the longest number and set it to half length
    int half_length;
    if (half_length1 >= half_length2) {
        half_length = half_length1;
    } else {
        half_length = half_length2;
    }

    // Allocate memory
    int *low1 = calloc(half_length, sizeof(int));
    int *low2 = calloc(half_length, sizeof(int));
    int *high1 = calloc(half_length, sizeof(int));
    int *high2 = calloc(half_length, sizeof(int));
    int *result_low = calloc(length_input1 +
                        length_input2 - 1, sizeof(int));
    int *result_high = calloc(length_input1 +
                        length_input2 - 1, sizeof(int));
    int *result_middle = calloc(length_input1 +
                        length_input2 - 1, sizeof(int));

    // Seperate the polynomials into highs and lows
    memcpy(low1, input1, half_length * sizeof(int));
    memcpy(low2, input2, half_length * sizeof(int));
    memcpy(high1, input1 + half_length, (length_input1 -
                        half_length1) * sizeof(int));
    memcpy(high2, input2 + half_length, (length_input2 -
                        half_length2) * sizeof(int));

    // First 2 recursive calls
    Karatsuba_Polynomial(low1, low2, half_length, half_length, result_low);
    Karatsuba_Polynomial(high1, high2, half_length, half_length, result_high);

    // Third recursive call
    // long long product_middle  = karatsuba(low1 + high1, low2 + high2);
    int *temp1 = calloc(half_length, sizeof(int));
    int *temp2 = calloc(half_length, sizeof(int));
    Array_Addition(low1, high1, half_length, temp1);
```

```
    Array_Addition(low2, high2, half_length, temp2);
    Karatsuba_Polynomial(temp1, temp2, half_length, half_length, result_middle);

    // Calculate middle coefficients (result_middle = result_middle -
                                    result_low - result_high)
    Array_Subtraction(result_middle, result_low, length_input1 +
                                    length_input2 - 1, result_middle);
    Array_Subtraction(result_middle, result_high, length_input1 +
                                    length_input2 - 1, result_middle);

    // Assemble final result
    // result = result_low + (result_middle << half_length1) +
                                (result_high << (half_length1 * 2))
    int max_length = length_input1 + length_input2 - 1;

    for (int i = 0; i < max_length; i++) {
        result[i] = result_low[i];  // Set result to low part

        // Add middle part if within its valid range
        if (i >= half_length) {
            result[i] += result_middle[i - half_length];
        }

        // Add high part if within its valid range
        if (i >= 2 * half_length) {
            result[i] += result_high[i - 2 * half_length];
        }
    }
    free(low1); free(low2); free(high1); free(high2);
    free(result_low); free(result_middle); free(result_high);
    free(temp1); free(temp2);
}
```

As can be seen, then the simplicity of the pseudo-code has been lost and the code is now almost 4 times longer. But the overall idea still stays the same, there is just a lot of overhead and memory management now.

If we look at the start, then it is the same. Here we start by checking if either of the polynomials have a length less or equal to 1, if yes then we just calculate them naively. This is the recursive exit condition and can be thought of the same as the exit condition in a while loop.

The way we calculate the length is different. Here instead of calculating it in the recursive call itself we pass it along to the Karatsuba function itself. This is a minor optimisation, since we will be keeping track of the length of each array and then halving that length in each iteration. This means that we don't need to compute the length inside the function itself.

We then add the modulus 2 value to it to account for odd numbers, this does the same as floor in the pseudo-code. Doing all this gives us an accurate length for the polynomials in each iteration that we can use. We then find the longest of the two polynomials and set that to the new half length.

Next we allocate memory for the arrays which will be used. This is quite expensive to do each recursive call, since it adds overhead. A faster way to do this would be to use memory pointer instead. But we were unable to implement this successfully, but it should not affect the results enough to make a significant difference.

Next we use memcpy to copy the appropriate halfs of the input arrays onto the high and low arrays we just created. This effectively splits each array in two. This is the crux of the Karatsuba algorithm and will be used for the recursive calls and the final calculation.

Finally we calculate the middle result and subtract the low and high value from it. This is setup for the

for loop, where we will calculate the result. Then we find the maximum possible length of our new output array. We then enter a for loop from 0 to the max length. Inside the loop we start by setting the low value, then we check if the variable i is beyond the half length. If yes then we can add the result of the middle section to the low result. This is because we are now in the middle part of the polynomial. Finally we check if the result is twice as big as the half length. If yes then we can add the high part of the polynomial.

This final section is where the problem with implementing Karatsuba's algorithm as a polynomial comes in. It is much simpler to implement it with numbers, which is why the pseudo-codee is much easier to understand than the actual implementation. But doing Karatsuba with Polynomials facilitates better comparisons with the FFT algorithm and allows different use cases than the version that computes numbers.

5.3 Master Theorem

To calculate the runtime we will use the Master Theorem[4][P.94], which is the following:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \tag{5.1}$$

- T(n) is the algorithm's runtime
- $a \geq 1$ is the number of subproblems at each recursion level
- $b > 1$ is the factor by which the subproblem's size is reduced in each call
- f(n) is the cost of the work that is done outside the recursive calls, like going through a for loop

This theorem provides a way to solve recursion problems, which are more difficult to calculate than a series of nested for loops. As an example with the naive approach we could easily see that the runtime would be quadratic, since there were 2 nested for loops that went from 0 to n. But in recursion we need to use the master theorem.

To analyse the runtime, we must then see which of the following 3 cases apply:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } f(n) = O(n^c) \text{ where } c < \log_b a, \\ O(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{\log_b a}), \\ O(f(n)) & \text{if } f(n) = \Omega(n^c) \text{ where } c > \log_b a \text{ and regularity condition holds.} \end{cases}$$

If we now look at the runtime in our Karatsuba implementation, we see that there are 3 recursive calls. Each of these then half the size of our n value. This means that we get the following runtime:

$$T(n) = 3T(n/2) + O(n) \tag{5.2}$$

We can then check the cases and see that our c value is 1, because

$$f(n) = O(n)$$

Which means c = 1, since c is the exponent of n here. If we now apply the first case in the master theorem we see that:

$$1 < log_2 3 \tag{5.3}$$

This means that we apply the first case and use it to get the following runtime:

$$O(n^{log_2 3}) = O(n^{1.59}) \tag{5.4}$$

This is a significant improvement from the Naive approach. But we can build further upon this idea and try to optimise it further. This can be done using the Faster Fourier Transform algorithm, but before we can look at that we first need to understand the algorithm it is based on. The Discrete Fourier Transform or DFT.

6 Proof of Correctness for Karatsuba

To prove the correctness of Karatsuba's algorithm, we need to show that the final result of the algorithm matches the mathematical product of x and y using the naive approach.

6.1 Splitting Step

By definition, the split of x and y is:

$$x = x_{high} \cdot B^m + x_{low}$$

$$y = y_{high} \cdot B^m + y_{low}$$

Here, B is the base, 10 is the standard and m is the number of digits in the lower half (i.e., $m = \lceil \frac{n}{2} \rceil$). The integers x_{high} and x_{low} are the high and low parts of x, respectively, and similarly for y.

for the number x = 1024, then x_{low} would be 24 and x_{high} would be 10 and m would be $\frac{4}{2} = 2$ since there are 4 digits in total.

6.2 Positional Notation

In positional notation, a number is represented as a sum of its digits, each multiplied by a power of the base. For a base B, an n-digit number x can be expressed as:

$$x = d_{n-1} \cdot B^{n-1} + d_{n-2} \cdot B^{n-2} + \cdots + d_1 \cdot B + d_0 \tag{6.1}$$

where d_i are the digits of x. This also applies to polynomials, but in this proof we will be working with numbers instead of polynomials for simplicity. But it is more or less the same end result.

6.3 Intermediate Products

We then compute three intermediate products using Karatsuba:

$$low = x_{low} \cdot y_{low} \tag{6.2}$$

$$z_{high} = x_{high} \cdot y_{high} \tag{6.3}$$

$$z_{mid} = (x_{high} + x_{low}) \cdot (y_{high} + y_{low}) - z_{high} - z_{low} \tag{6.4}$$

6.4 Combining the Results

We have now divided the problem into sub-problems, it is now time to combine and conquer the problems. We therefore need to show that the result will be the same as using the naive approach for $x \cdot y$. The naive approach can be described as:

$$xy = (x_{high} \cdot B^m + x_{low}) \cdot (y_{high} \cdot B^m + y_{low}) \tag{6.5}$$

This can then be expanded to:

$$xy = x_{high} \cdot y_{high} \cdot B^{2m} + (x_{high} \cdot y_{low} + x_{low} \cdot y_{high}) \cdot B^m + x_{low} \cdot y_{low} \tag{6.6}$$

Now, let's use this for our intermediate z products:

$$z_{high} = x_{high} \cdot y_{high} \tag{6.7}$$

$$z_{low} = x_{low} \cdot y_{low} \tag{6.8}$$

Next we look at z_{mid}:

$$z_{mid} = (x_{high} + x_{low}) \cdot (y_{high} + y_{low}) - z_{high} - z_{low} \tag{6.9}$$

We then write out the polynomials and reduce it based on our earlier definitions of z_{high} and z_{low}:

$$z_{mid} = \cancel{x_{high} \cdot y_{high}} + x_{high} \cdot y_{low} + x_{low} \cdot y_{high} + \cancel{x_{low} \cdot y_{low}} - \cancel{z_{high}} - \cancel{z_{low}} \tag{6.10}$$

$$z_{mid} = x_{high} \cdot y_{low} + x_{low} \cdot y_{high} \tag{6.11}$$

We can now use our z values to compute Karatsuba's algorithm:

$$xy = z_{high} \cdot B^{2m} + z_{mid} \cdot B^m + z_{low} \tag{6.12}$$

If we now replace the z values with their x and y representations, then we can see that the algorithm matches the naive approach:

$$xy = x_{high} \cdot y_{high} \cdot B^{2m} + (x_{high} \cdot y_{low} + x_{low} \cdot y_{high}) \cdot B^m + x_{low} \cdot y_{low} \tag{6.13}$$

From this we can see that combining z_{high}, z_{mid}, and z_{low} following the steps in the Karatsuba algorithm returns the correct product xy.

Conclusion

Karatsuba's algorithm correctly computes the product of two integers by breaking the problem into smaller parts, solving the sub-problems recursively, and then combining the results. The proof shows that each step maintains the correctness of the final product. This then demonstrates the overall correctness of the algorithm.

7 DFT

Before discussing the Fast Fourier Transform algorithm, we will start by discussing the algorithm it is based on. Discrete Fourier Transform is the process of taking a function of time and converting it into a function of frequency. It takes that signal and decomposes it into the frequencies that it consists of. This is useful for many different applications and foundational for most modern physics, engineering and applied math that work with signal processing, image analysis and more. Signal processing is beyond the scope of this paper though and we will instead be focusing on its ability to calculate polynomials.

Discrete refers to working with sequences or sets of values at distinct time points, as opposed to continuous functions which are defined over a continuous range of values. It works by transforming a finite sequence of equally spaced samples of a function into a list of coefficients of complex sinusoidal basis functions.

Complex sinusoidal basis functions are functions that are used to decompose a signal into its most fundamental frequencies. These functions are complex exponential, meaning they consists of imaginary and real numbers. When looking at formula for the DFT we see this:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{nk}{N}} \tag{7.1}$$

Here we see that we are working with a complex exponential which is derived from Euler's formula, which we will discuss next.

7.1 Euler's Formula

Euler's formula can be described like this:

$$e^{i\theta} = \cos(\theta) + i\sin(\theta) \tag{7.2}$$

as can be seen then $e^{i\theta}$ looks similar to the formula for DFT:

$$e^{-i2\pi \frac{nk}{N}} \tag{7.3}$$

The best way to understand the formula is to visualise what it does. Which can be seen in figure 1. Here we see that the real numbers lie on the x axis while the imaginary numbers lie on the y axis. What then happens is that instead of moving in one direction like we normally do on graphs, we instead move in a circle.

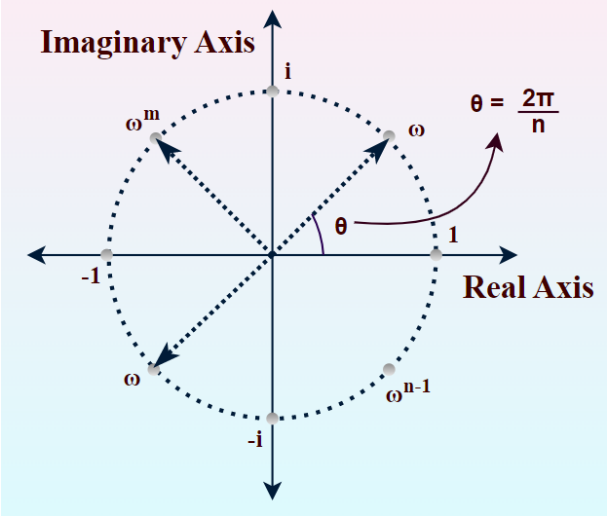


Figure 1: Source: https://www.geeksforgeeks.org/nth-roots/

There are stops along the way, these depend on how many roots of unity we have. If we have 1 root of unity, then it is simply 1. If we have two, then the roots of unity become 1 and -1. And if we have 4 roots of unity, then they are 1, -1. i and -1. The relationship between them can be described the following way, where there are 4 roots of unity:

$$\text{For } n = 1, \text{ the root of unity is } e^{\frac{2\pi i \cdot 0}{1}} = e^0 = 1. \tag{7.4}$$

$$\text{For } n = 2, \text{ the roots of unity are:} \tag{7.5}$$

$$e^{\frac{2\pi i \cdot 0}{2}} = e^0 = 1 \text{ (for } k = 0) \tag{7.6}$$

$$e^{\frac{2\pi i \cdot 1}{2}} = e^{\pi i} = -1 \text{ (for } k = 1). \tag{7.7}$$

$$\text{For } n = 4, \text{ the roots of unity are:} \tag{7.8}$$

$$e^{\frac{2\pi i \cdot 0}{4}} = e^0 = 1 \text{ (for } k = 0) \tag{7.9}$$

$$e^{\frac{2\pi i \cdot 1}{4}} = e^{\frac{\pi i}{2}} = i \text{ (for } k = 1) \tag{7.10}$$

$$e^{\frac{2\pi i \cdot 2}{4}} = e^{\pi i} = -1 \text{ (for } k = 2) \tag{7.11}$$

$$e^{\frac{2\pi i \cdot 3}{4}} = e^{\frac{3\pi i}{2}} = -i \text{ (for } k = 3). \tag{7.12}$$

As can be sen as we increase the number of roots of unity, then new numbers inserts themselves in between the existing numbers, so at n = 2 we have the roots 1 and -1, but at n=4, we have the roots 1, i, -1, -i. This is because we need to think of the function as forming a circle. When you increase the number of roots, we do not extend the circle, instead we extend the number of stops we make whilst travelling around. If we scale it to n=8, we would be inserting our new points between the existing ones

we already have in n=4, making it look like this:

$$\text{For } n = 8, \text{ the roots of unity are:} \tag{7.13}$$
$$e^{\frac{2\pi i \cdot 0}{8}} = e^0 = 1, \tag{7.14}$$
$$e^{\frac{2\pi i \cdot 1}{8}} = e^{\frac{\pi i}{4}} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, \tag{7.15}$$
$$e^{\frac{2\pi i \cdot 2}{8}} = e^{\frac{\pi i}{2}} = i, \tag{7.16}$$
$$e^{\frac{2\pi i \cdot 3}{8}} = e^{\frac{3\pi i}{4}} = -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, \tag{7.17}$$
$$e^{\frac{2\pi i \cdot 4}{8}} = e^{\pi i} = -1, \tag{7.18}$$
$$e^{\frac{2\pi i \cdot 5}{8}} = e^{\frac{5\pi i}{4}} = -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, \tag{7.19}$$
$$e^{\frac{2\pi i \cdot 6}{8}} = e^{\frac{3\pi i}{2}} = -i, \tag{7.20}$$
$$e^{\frac{2\pi i \cdot 7}{8}} = e^{\frac{7\pi i}{4}} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i. \tag{7.21}$$

As can be seen, then we now have 4 new points, which have been inserted between the existing points of unity. What this means is that instead of having 4 stops, when going around the circle, we now have 8 instead.

If we go back and look at n = 4 again, we can discover another interesting property of Euler's formula. We have already calculated k=0 to k=3, but what happens of we try k=4?

$$e^{\frac{2\pi i \cdot 4}{4}} = e^{2\pi i} = e^0 = 1 \text{ (for } k = 4 \text{ and } n = 4). \tag{7.22}$$
$$e^{\frac{2\pi i \cdot 5}{4}} = e^{\frac{5\pi i}{2}} = i \text{ (for } k = 5 \text{ and } n = 4). \tag{7.23}$$
$$\tag{7.24}$$

As can be seen, when k goes beyond the n value, it loops back on itself, showing that Euler's formula forms a circle with the roots of unity. This will be true for any $n = 2^m$ values that we use, so that when the k value increases beyond the n value, it effectively loops back since circles do not have an endpoint. The reason why it loops back on itself at $k = 5$, despite $e^{\frac{\pi i}{2}} \neq e^{\frac{5\pi i}{2}}$, is due to the nature of complex exponential functions, which have a period of 2π. If we now consider adding 2π to our angle, we can see why $k = 5$ is equivalent to $k = 1$:

$$e^{\frac{5\pi i}{2}} = e^{\frac{\pi i}{2} + 2\pi i \cdot k} \quad \text{for } k = 1 \tag{7.25}$$
$$= e^{\frac{\pi i}{2} + 2\pi i} \tag{7.26}$$
$$= e^{\frac{\pi i}{2}} \tag{7.27}$$

7.2 Inverse DFT

Inverse DFT or IDFT is the process of converting the complex numbers in the frequency domain we have just created back to it's original sequence in the time domain. What this means is that IDFT simply inverses everything we just did, so we reconstruct the original time domain.

- Time Domain is how the signal changes over time and is usually plotted with time on the horisontal axis.
- Frequency Domain shows how much of a signal lies within each given frequency band. The frequency is then plotted on the horizontal axis and the magnitude of the signal on the vertical axis

An example of conversion from time to frequency can be seen in figure 2. Here in the first row we see that we have a high frequency sine wave. This is then shown in the frequency domain as a single spike at a specific frequency, which is the frequency of the sine wave in the time domain. What we can read from this is that we have a single high frequency in the time domain.

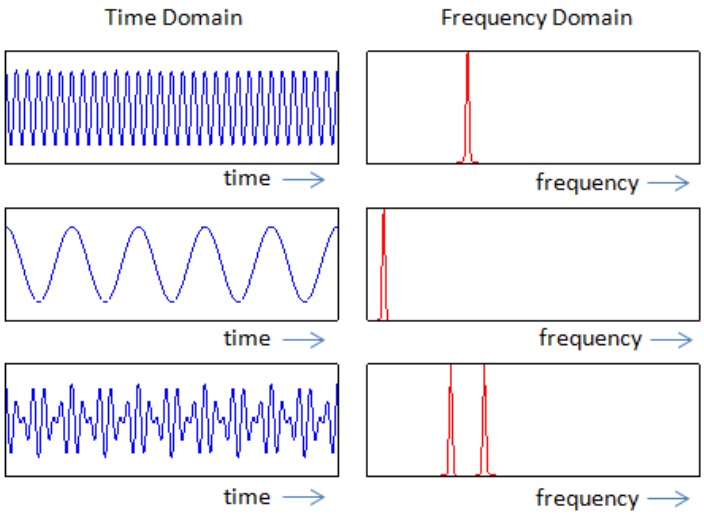


Figure 2: Time domain to frequency domain[14]

In the second row, we see a lower frequency since wave and therefore the spike in the frequency domain is further to the left. In the third row, we have a more complex signal, which constitute multiple since waves. Therefore in the frequency domain, we see two spikes, with different frequency.This shows how often each sine wave occurs in this signal.

As can seen from these examples, then frequency domain works as a great way to break down more complex signals, like in figure 2 row 3, where from the frequency domain we can read that there are multiple sine waves mixed together.

Therefore the process of converting time domain to frequency domain is crucial as it allows for the manipulation and analysis of signals. We use DFT to convert from time to frequency domain and then to reconstruct the time domain we use IDFT. In this paper we will use it for polynonmial multiplication. But we still need to convert it to time domain and then back to frequency domain for this to work. Here we use DFT to calculate the new values in the frequency domain and IDFT to convert them back to time domain so we can analyse them.

The mathematical formula for IDFT is very simple, as just need to invert the DFT, which we can do using a sign change on the exponential value:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{i2\pi \frac{nk}{N}} \cdot \frac{1}{N} \tag{7.28}$$

7.3 Implementation

To implement DFT and IDFT in C is very straight forward and resembles the naive approach, as we need to loop over every element in a given array and apply DFT to it and save it in an output array:

```
DEFINE TAU (2 M_PI)
void DFT(complex double *in, int n, complex double *out) {
    // 2 nested for loops is what causes the runtime n^2
    for (int i = 0; i < n; i++) { // For each output element
        // initialise each element to 0
        out[i] = 0;
        for (int j = 0; j < n; j++) { // For each input element
            // Compute DFT function
            out[i] += in[j] * cexp(-I * TAU * j * i / n);
        }
    }
}
```

As can be seen we have two for loops, which both run from 0 to n, which means that our runtime is $O(n^2)$. This means that our runtime is quadratic, which is not great when we start working with larger numbers. But in reality the code is much slower than the naive approach because of all the overhead. At this point all we have done is convert our time domain to frequency domain. Next we need to combine the polynomials together with point-wise multiplication. Then we can finally convert them back to time domain and normalise:

```
void IDFT(complex double *in, int n, complex double *out) {
    // 2 nested for loops is what causes the runtime n^2
    for (int i = 0; i < n; i++) { // For each output element
        // initialise each element to 0
        out[i] = 0;
        for (int j = 0; j < n; j++) { // For each input element
            // Compute inverse DFT function by changing the sign
            // This is the only change from DFT
            out[i] += in[j] * cexp(I * TAU * j * i / n);
        }
        out[i] /= n; // Scale by 1/n, ensuring proper normalization
    }
}
```

```
    // Apply DFT to both polynomials
    complex double fa[n], fb[n];
    DFT(padded_a, n, fa);
    DFT(padded_b, n, fb);
    // Point-wise multiply the DFTs
    for (int i = 0; i < n; i++) {
        fa[i] *= fb[i];
    }
    // Apply IDFT to get the product polynomial
    IDFT(fa, n, dft_result);
}
```

In the naive approach we only needed to through go the two for loops once. But in the DFT we need to do it three times. It however gets much worse when you look at the calculation that is done in each loop iteration:

$$out[i] += in[j] * cexp(-I * TAU * j * i/n); \tag{7.29}$$

As can be seen then there are 4 multiplications, 1 division in an exponential function to compute the DFT vs the single multiplication in the naive approach. But it gets worse still, because we also need to calculate the DFT for each polynomial and then do the IDFT, which includes an extra normalisation step. So in total we can expect the DFT to run rougly 18 times slower than the naive approach based on there being 6 calculations in each DFT and IDFT. One of these being exponential means it will in probably be even slower than that. Besides this there is an extra hidden cost. The DFT algorithm uses a complex double array, where the naive approach uses a standard integer array. The size of a complex double type is 16 bytes compared to just 4 from integer type. This means that we need to spend more time on allocating memory and potentially also computing. There are also extra holder arrays in the code also need to be allocated. So in conclusion using DFT for polynomial multiplication is not a good idea and very inefficient compared to more specialised solutions like Karatsuba's algorithm, which would scale much better as previously discussed.

We will therefore next look at ways to improve the DFT calculation, using the Fast Fourier Transform (FFT) to improve the runtime.

8 FFT Background

8.1 History

The fast Fourier Transform (FFT) algorithm was developed as a more efficient algorithm for calculating the discrete Fourier Transform (DFT) of a sequence of N numbers. The DFT is widely used in many fields

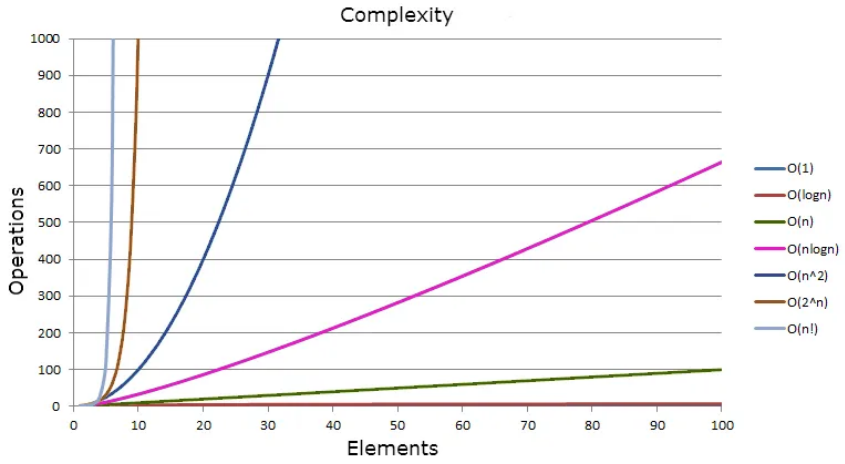


Figure 3: Time Complexity [13]

to get the spectrum of frequency content of a signal and to allow computation of discrete convolution and correlation [7].

The problem with the DFT algorithm is that it is slow, running in $O(n^2)$ time, as we have previously discussed. Therefore a faster method was needed to compute the DFT. The FFT algorithm was the solution to this and was first documented by Cooley and Tukey in 1965[3].

This discovery allowed the DFT to be calculated with $O(nlogn)$ operations rather than $O(N^2)$[3]. This has been critical since the DFT is such a fundamental algorithm in so many fields. The difference between $O(N^2)$ and $O(nlogn)$ can be seen in figure 3.

This improvement was critical back in the 60's, when compute power was much slower than it is today. For comparison, my phone has 12 GB ram where Appollo 11 computer that did the first moon landing only had 32 KB [9]. That is more than 3 million times more memory in my phone, than one of the most advanced computers of that time.

One of the most important achievements of the FFT algorithm was slowing down the nuclear arms race between the USSR and the USA [16]. In 1963 a treaty was signed between the major nuclear powers, the United States, the United Kingdom, and the Soviet Union to ban nuclear tests in atmosphere, space and underwater. This was the due to radioactive deposits being found in food produce and to prevent other countries from developing their own nuclear weapons[11].

The issue was that the only viable option left was to conduct nuclear tests underground, since this would not spread radiation. From a strategic perspective this would be dangerous, since both sides would be in the dark about the others developments. Therefore they needed a way to measure the tests the other side conducted. This is where the Fast Fourier Transform comes in.

The only way to measure the size of the explosions was by analysing seismological time-series data using the DFT algorithm. The problem was that the DFT is to slow. So in 1965 Cooley and Tukey managed to develop a faster version of the DFT, the Fast Fourier Transform. This allowed the US scientists to compute the seismological time-series data much more efficiently[16].

8.2 FFT explanation

The basic idea of the Fast Fourier Transform (FFT) algorithm is to use the symmetries and periodic properties of the DFT algorithm to reduce the required amount of computations needed to compute the DFT. This then reduces the compute cost and lowers the runtime down to $\mathrm{O}(n \cdot log_2 n)$.

To understand this we need to look back at Euler's formula $e^{i\theta} = \cos(\theta) + i\sin(\theta)$ in figure 1. This shows that if we know a point, then we will be able to get its inverse value as well. This only works on n > 1 where $n = 2^m$. You can get it to work with other values as well, but that is beyond the scope of this paper. The reason why this works is that if we look at figure 1 then we can simply inverse our first

result to get the other point in the circle since its center is at 0,0. This means that each point has an exact inverse diagonally across on the circle.

The way the FFT algorithm utilises this is by halving a polynomial into even and uneven elements that are determined by the indices. So the even indices values go into their own new polynomial and are ordered from $0, 1, 2, ..., \frac{n}{2}$. This can then be split again and create a new smaller even and uneven polynomial until they reach a length of n = 1. What this does is that it halves the complexity in each recursive iteration. When splitting the polynomial in half, we should think of it as splitting the circle in half, leaving us with an odd and even side. We can then recursively repeat until a length of n=2 has been reached. This process and eventually end up with just the two pairs of opposites that we can then add together.

To do this the FFT algorithm uses the divide-and-conquer strategy similar to Karatsuba's algorithm. This lowers the amount of computations and reduces the complexity as well, making it run significantly faster than the DFT algorithm.

When the polynomials reach n = 1 length the DFT can then be trivially solved with a single computation, where we just assign the output as the input. This is then returned and then the previous recursive call takes that values and uses it to compute the next part of the polynomial in the for loop. This DFT is then computed and it returns its output for the next part to compute. At the end we have two arrays of $\frac{n}{2}$ size that we compute. This means that if we compute an polynomial of size 8, we would need to calculate the following polynomial sizes:

$$n = 1, n = 1, n = 2, n = 1, n = 1, n = 2, n = 4, \tag{8.1}$$

At the end we then need to combine each of the polynomials together. We know we have the odd and even indices separated. These can then be combined using something called the Twiddle Factor $e^{-2\pi i \frac{k}{N}}$[8]. The Twiddle Factor is derived from Euler's formula and therefore has the same cyclical property as we described earlier. This accounts for the phase shift introduced by the odd indices of the sequence. We therefore need to do the following calculation to combine the odd and even indices back together:

$$X[k] = \text{EVEN}[k] + e^{-i2\pi \frac{k}{N}} \cdot \text{ODD}[k] \tag{8.2}$$

$$X[k + \frac{N}{2}] = \text{EVEN}[k] - e^{-i2\pi \frac{k}{N}} \cdot \text{ODD}[k] \tag{8.3}$$

Here we see that the twiddle factor has been multiplied onto the ODD indices, which accounts for the phase shift. They are then combined together, similar to Euler's formula: $e^{i\theta} = \cos(\theta) + i\sin(\theta)$. We then subtract the two to get the inverse and inserted into the other half of the polynomial. This is then repeated $\frac{n}{2}$ times until we have completed the DFT.

9 Recursive FFT

9.1 Pseudo-Code

Before we start with the implementation we will look at the pseudo-code for the recursive FFT[4] first:

```
RECURSIVE-FFT(a)
    input: Array a of n complex values where n is a power of 2.
    output: Array y the DFT of a.

    n = a.length  // n is a power of 2
    if n == 1:
        return a
    w_n = e^(2 * pi * i/n)
    w = 1

    a_even = (a0, a2, ..., an-2)
    a_odd = (a1, a3, ..., an-1)
    y_even = RECURSIVE-FFT(a_even)
    y_odd = RECURSIVE-FFT(a_odd)

```

```
    for k = 0 to n/2 - 1:
        y[k] = y_even[k] + w * y_odd[k]
        y[k+n/2] = y_even[k] - w * y_odd[k]
        w = w * w_n
    return y // y is assumed to be a column vector
```

Here see that we take an input A of length n. We then check if the length of a is 1, if yes then we simply return A as there is nothing to do. This is our recursive exit condition when we have a section of length n=1 as discussed previously. Next we set the twiddle factor $\omega_n = e^{2 \cdot pi \cdot \frac{i}{n}}$ to the principal root unity from euler's formula and the unity root factor ω to 1.

Next we split a into even a_even and odd a_odd. Then we do a double recursive FFT call with the even and odd arrays and store the output in y_even and y_odd respectively.

Then we iterate over half the array length and compute the FFT using the twiddle factor and store the result in y, which we then return. As can be seen then we calcualte twice, but the only difference between the first and second result is that we inverse the twiddle factor that is multiplied onto y_odd. This is the main optimisation of the FFT algorithm as we use the properties of Euler's formula to inverse our first result to get the second.

9.2 Implementation

The way we have implemented the algorithm is a bit more complicated than the pseudo code, but this is because we have used memory pointers to save overhead. We will therefore start by explaining the memory allocation first:

```
// FFT function to allocate memory and call actual FFT functionn
void Recursive_FFT(complex double *input, int n, complex double *out) {
    // Assign memory outside the recursive loop to save overhead
    // We need 4 arrays, in_even, in_odd, out_even and out_odd. Therefore we need 4 * n
    int n_bitshifted = n << 2; // * 4
    complex double *allocated_memory =
                    (complex double *)malloc(n_bitshifted * sizeof(complex double));

    // Call the actual function
    Recursive_FFT_ext(input, n, out, allocated_memory, n);
    free(allocated_memory);

}
```

Here we see that we start by increasing our n size by 4 to make room for the 4 arrays:

- a_even
- a_odd
- y_even
- a_odd

hen we allocate memory for it and call the actual recursive FFT function. If we now look at that then we can better see the advantage of handling memory this way:

```
void Recursive_FFT_ext(complex double *input, int n, complex double *out,
            complex double *allocated_memory, int allocated_memory_size) {
    // Check if n == 1 and return f(1) (f[0])
    if (n == 1) {
        out[0] = input[0];
        return;
    }

    // Save n/2 in a variable to save computations
    int n_half = n >> 1;
```

```
11
12      // Use the passed allocated_memory instead of allocating memory
13      // set pointer to start of allocated_memory
14      complex double *even_values = allocated_memory;
15      // set pointer to the second quarter of allocated memory
16      complex double *odd_values = allocated_memory + n_half;
17      // set pointer to the third quarter of allocated memory
18      complex double *out_even_values = odd_values + n_half;
19      // set pointer to the fourth quarter of allocated memory
20      complex double *out_odd_values = out_even_values + n_half;
21
22      // Seperate into odd and even numbers
23      int i_double = 0;
24      for (int i = 0; i < n_half; i++) {
25          i_double = i << 1;
26          even_values[i] = input[i_double];
27          odd_values[i] = input[i_double + 1];
28      }
29
30      // Double recursive call, half the allocated memory since we are splitting the data
31      // Here there are 2 recursive calls, which each split the array in half, therefore:
32          // T(n) = 2T(n/2) + O(n)
33      // Applying the master theorem, here log_2(2) = 1 and C = 1 because f(n) = O(n)
34      // Therefore we get the second option and our runtime becomes:
35          // T(N) = (n log n)
36      Recursive_FFT_ext(even_values, n_half, out_even_values,
37                          out_odd_values + n_half, allocated_memory_size);
38      Recursive_FFT_ext(odd_values, n_half, out_odd_values,
39                          out_odd_values + n_half, allocated_memory_size);
40
41      // Compute the FFT output
42      complex double tmp = 0;
43      complex double w = 1;
44      complex double w_n = cexp(-I * TAU / n);
45      for (int k = 0; k < n_half; k++) {
46          // Use defined TAU to save multiplication and calculate e^{i*TAU*n/k}
47          // directly in tmp to save calculations
48          tmp = w * out_odd_values[k];
49          out[k] = out_even_values[k] + tmp;
50          out[k + n_half] = out_even_values[k] - tmp;
51          w *= w_n;
52      }
53  }
```

We start by checking for n == 1, since that means we have reached the end of the polynomial and we can start computing the DFT. Next we half the size of n, so far so good, but now we get to the memory handling. The allocated memory is split into the four sections we previously discussed. This will be discussed next and is from line 11 to 21 in the pseudo-code.

Since we halved n we now have a memory size of 2n instead of 4n. We start by setting the memory pointer for even values at the location of the allocated memory. Then we set the memory pointer for the odd values to the allocated memory + the new n half value. We have now split our data into two different parts of the memory. We then add the out even values pointer next to the even values by saying odd values + n half. This has the side effect of placing the input and output arrays next to each other in memory, which should lead to improved performance as there is less space to iterate over when reading and writing. Finally we set the pointer for the odd values the same way.

Next we need to separate the polynomial into odd and even, this is on line 24 to 29. This is done by running it through a for loop from i to n half. In each iteration we use the variable i_double to double the i value. This is done to account for the separation of the array into two halves. The even getting the

i_double position whilst the odd array gets the i_double + 1 value. Doing this we split the polynomial in two.

Next we do our two recursive FFT calls on line 36 to 38. Which we have already discussed. Finally we get to the DFT computation, which is similar to the FFT pseudo-code, but slightly changed for optimisations. We start by setting $\omega = 1$ and $\omega_n = cexp(-I * TAU/n)$, this is similar to the pseudocode, we are just using TAU instead of $2 \cdot PI$. Next we will save $\omega * out\_odd\_values[k]$ in a tmp variable that we can simply invert when computing the FFT. We then add this tmp value along with the out even values to the output array on k position. Then we inverse tmp when calculating the out even values on position k + n half. These 3 lines (48 to 51) essentially phase shift each value into the correct position with the correct weight, thereby aligning the output polynomial. Then finally we assign the new ω value to be $\omega* = \omega_n$

This is then repeated for the other polynomial fb and then they are multiplied together point wise, before sending the combined result to the IFFT, which then computes the frequency domain by inversing the FFT. This is done the exact same way the DFT did it:

```
1     // Apply FFT to both polynomials
2     complex double fa[n], fb[n];
3     Recursive_FFT(padded_a, n, fa);
4     Recursive_FFT(padded_b, n, fb);
5
6     // Point-wise multiply the FFTs
7     for (int i = 0; i < n; i++) {
8         fa[i] *= fb[i];
9     }
10
11    // // Apply IFFT to get the product polynomial
12    Recursive_IFFT(fa, n, fft_result);
```

9.3 Computation time

We will again be using the Master Theorem since we are working with a recursive function.

- a is the number of sub-problems that are solved at each recursion level, which is 2 since we have two recursive calls in each iteration
- b is the reduction of n, which is halved for each recursive problem, which means b is 2
- f(n) is the work done outside the recursive calls, here we go through two separate for loops that each go to $\frac{n}{2}$. Meaning it is O(n)

This then gives us the runtime:

$$T(n) = 2T(\frac{n}{2}) + O(n^1) \tag{9.1}$$

Here we see that our runtime correlates with case two from the Master Theorem:

$$O(n^1) = \theta(n^{log_2 2}) \tag{9.2}$$

$$O(n^1) = \theta(n^1) \tag{9.3}$$

We can therefore say that the runtime of the FFT algorithm is:

$$O(n^{log_2 2} \cdot logn) = O(n \cdot logn) \tag{9.4}$$

10 Proof of Correctness for the FFT

In this section we will focus on the mathematical proof that the FFT algorithm will do the complete DFT calculation every time and cover every element of the polynomials. We will prove this through induction.

10.1 Base Case

When $n = 1$, the FFT is applied to a constant polynomial (degree zero). The output is therefore simply the input itself, as there are no computations to perform. This shows that the FFT is correct for $n = 1$.

10.2 Inductive Hypothesis

We assume that the FFT correctly computes the Discrete Fourier Transform (DFT) for any polynomial of degree less than n at their respective n-th roots of unity.

10.3 Inductive Step

For $n > 1$ where $n = 2^m$ for some integer m, consider a polynomial of degree $n-1$ will be split into two smaller polynomials:

- The even-indexed polynomial $f_e(x) = a_0 + a_2x + \ldots + a_{n-2}x^{\frac{n}{2}-1}$.
- The odd-indexed polynomial $f_o(x) = a_1 + a_3x + \ldots + a_{n-1}x^{\frac{n}{2}-1}$.

Each recursive step halves the size of the problem and repeats the same process. Below we see the next stage, with the polynomial f_e, the same is done for from f_o:

- The even-indexed polynomial $f_{ee}(x)$ in the next recursion becomes $a_0 + a_4x + \ldots + a_{n-2}x^{\frac{n}{2^2}-4}$.
- The odd-indexed polynomial $f_{eo}(x)$ in the next recursion becomes $a_2 + a_6x + \ldots + a_{n-1}x^{\frac{n}{2^2}-2}$.

The recursion continues until the polynomials can no longer be split and we have a length of n = 1. Assuming correctness of the recursive FFT computations for $f_e(\omega_{\frac{n}{2}}^k)$ and $f_o(\omega_{\frac{n}{2}}^k)$ for $k = 0, 1, \ldots, \frac{n}{2} - 1$, we have:

$$\omega_n^k = e^{2\pi i \frac{k}{n}} \tag{10.1}$$

Next we introduce $\omega_{\frac{n}{2}}^k$, which refers to the k-th power of a primitive $\frac{n}{2}$-th root of unity. The relationship can then be described as:

$$\omega_{\frac{n}{2}}^1 = e^{2\pi i \frac{1}{\frac{n}{2}}} = e^{2\pi i \frac{2}{n}} \tag{10.2}$$

As can be seen, this is the twiddle factor from earlier. We can then rewrite the function $\omega_{\frac{n}{2}}^k$ to:

$$\omega_{\frac{n}{2}}^k = e^{2\pi i \frac{k}{\frac{n}{2}}} = e^{2\pi i \frac{2k}{n}} \tag{10.3}$$

This shows that $\omega_{\frac{n}{2}}^k$ consists of every other root of unity from ω_n^k, which confirm that:

$$\omega_{\frac{n}{2}}^k = \omega_n^{2k} \tag{10.4}$$

When the division step is complete we need to recombine using the twidddle factor we just found:

$$f(\omega_n^k) = f_e(\omega_{\frac{n}{2}}^k) + \omega_n^k \cdot f_o(\omega_{\frac{n}{2}}^k) \tag{10.5}$$

This function consists of both the odd and even polynomials that are squared $\frac{n}{2}$ roots of unity. This is shown in the previous step $\omega_{\frac{n}{2}}^k = \omega_n^{2k}$. This can also be derived with the simple equation:

$$f(x) = f_e(x^2) + xf_o(x^2) \tag{10.6}$$

ω_n^k is the k-th n-th root of unity and is responsible for correctly aligning the function.

Doing this combines the result of the two smaller DFT calculations into a single DFT calculations. This interleaves the results from the odd and even indexed polynomials and recombines them using the roots of unity.

This shows that by correctly combining $f_e(x)$ and $f_o(x)$ according to the identity, the FFT accurately computes f at ω_n^k.

10.4 Completeness

For $k = 0, 1, \ldots, n-1$, the values of $f(\omega_n^k)$ computed by the FFT are correct based on the inductive hypothesis and the algorithm's symmetry which we discussed in the previous section. Here symmetry pertains to the this section specifically:

```
out[k] = out_even_values[k] + tmp;
out[k + n_half] = out_even_values[k] - tmp;
```

Here we use the symmetry of Euler's formula to save computations by simply inverting the tmp value and inserting it in the second half of the output. By exploiting this symmetry, the FFT needs to calculate only half of the values explicitly to determine the values for the entire set, thereby increasing efficiency.

11 Iterative FFT

We have established the foundation for the FFT algorithm and proved its correctness mathematically. Now the next step is improving it, the way we will do this is by getting rid of recursion. The reasoning for this is that the recursion adds a lot of overhead through the extra memory manipulation. If we can avoid this, then we should be able to output the result faster. We are also using the -O2 flag, which means that loops will be much faster, so we want to take advantage of that. The way we convert the Recursive FFT algortihm to an iterative version is by using the butterfly diagram, which is a phenonomen where data structures itself like a butterfly[4][P. 916].

11.1 Pseudo-code

Before we work on implementing the iterative approach we will look at the pseudo-code for the iterative FFT[4]

```
Iterative-FFT(a)
    input: Array a of n complex values where n is a power of 2.
    output: Array A the DFT of a.

    bit-reverse-copy(a, A)
    n <- a.length
    for s = 1 to log(n) do
        m <- 2s
        wm <- exp(2i/m)
        for k = 0 to n-1 by m do
            w <- 1
            for j = 0 to m/2 - 1 do
                t <- w A[k + j + m/2]
                u <- A[k + j]
                A[k + j] <- u + t
                A[k + j + m/2] <- u - t
                w <- w wm
    return A
```

Here we start by bit reversing and copying a onto A. Then we find the length of a and enter the for loops, which goes from s to $log_2 n$. Here set the segment length m to by $2s$. This is because each stage of the FFT operates on segments that have a length of $2s$. Next we calculate the principal root of unity ω_m for that sector. This is the foundation for calculating all the the other roots of unity for the section.

Next we get to the middle for loop. Here we iterate over each segment of the polynomial. Here we initialise the unity root factor ω to 1 and calculate the half length of the segment m.

Finally we get to the inner for loop, here we finally get to compute the FFT. In the previous two for loops, we have gradually been segmenting and reducing the size of the polynomial and now we are able to iterate over a half segment $m/2$ of this. We start by computing the twiddle factor t. We then compute the FFT the same way as we did in the recursive version. Where we use the symmetry of Euler's formula to more efficiently compute the DFT.

Finally we compute the new unity root factor by multiplying it with the current segments root of unity factor. This then returns the FFT. Now that we know what the Iterative FFT does line we can look at how we implemented it in C

11.2 Bit Reversal

To start with we look at the bit reversal and explain it a bit further. This step reorders the data so the FFT algorithm can access the correct bits in sequential order, this improves efficiency and makes it easier to compute. For instance if we look at index 3: 011 in binary, then reversing the bits returns 110, which is 6. This means that instead of working with index 3 we are now working with index 6. Below is the code that reverses the bits:

```
unsigned int Bit_Reverse(unsigned int x, int log2n) {
    int n = 0;
    for (int i = 0; i < log2n; i++) {
        n <<= 1; // Shift n one left (n *= 2)
        n |= (x & 1); // Finds least significant bit
        x >>= 1; // shift x one right (x /= 2)
    }
    return n;
}
```

Here we see that it takes an integer x and $log_2 n$ value and goes through 0 to $log_2 n$. It then doubles the n value and finds the least significant bit using an or statement. It then halves the size of x using bit shifting. This process is then repeated until all bits have been reversed.

We then use this reversed bit number in the iterative FFT algorithm to set the output array to the bit reversed input array:

```
void Iterative_FFT(complex double* input, int n, complex double* output) {
    int reverse_bit;
    int log2n = log2(n);

    for (unsigned int i = 0; i < n; ++i) {
        reverse_bit = Bit_Reverse(i, log2n);
        output[i] = input[reverse_bit];
    }
```

11.3 Core FFT processing

We now have the bit reversal setup complete. So now we continue with the Iterative FFT approach. We will be avoiding duplicate calculations by saving repeat calculations in variables to reduce the number of required computations, thereby optimising the pseudo-code. If we now look at the second part of the Iterative FFT method then we see that it is very similar to the pseudo-code:

```
    int fft_segment_length, fft_half_segment_length;
    complex double unity_root_factor, segment_root_of_unity, twiddle_factor, tmp;
     for (int s = 1; s <= log2n; s++) {
        fft_segment_length = 1 << s; // pow(2, s)
         // Principal root of unity for the current segment
        segment_root_of_unity = cexp(-I * TAU / fft_segment_length);

        for (int k = 0; k < n; k += fft_segment_length) {
            // Initialize unity root factor (w) to 1, use 0*I to create complex number
            unity_root_factor = 1 + 0 * I;
            fft_half_segment_length = fft_segment_length >> 1; // /2
            for (int j = 0; j < fft_half_segment_length; j++) {
                // Twiddle factor application: https://en.wikipedia.org/wiki/Twiddle_factor
                twiddle_factor = unity_root_factor *
                                 output[k + j + fft_half_segment_length];
                tmp = output[k + j];

```

```
                // Applying FFT butterfly updates
                output[k + j] = tmp + twiddle_factor;
                output[k + j + fft_half_segment_length] = tmp - twiddle_factor;

                // Update the unity root factor
                unity_root_factor *= segment_root_of_unity;
            }
        }
    }
}
```

We start by organising the polynomial by bit reversing it which later in the code will allow us to more efficiently compute the FFT. Then we go through a series of nested for loops that seperates the FFT into smaller and smaller segments, similar to how the recursive FFT broke each iteration down to $\frac{n}{2}$ problem size. We then finally apply the twiddle factor onto the two relevant segments of the polynomial and continue to do this until we have gone through each element. Because there are no recursive calls, there is less overhead management that we found in the recursive approach. This leads to improved runtime which we will look at next.

12 Programming Methodology

12.1 Implementation

We have already looked at each algorithm in depth and their implementation in C, so we understand how each of these algorithms operate. The algorithms that we have implemented are:

- Naive polynomial multiplication
- Karatsuba's algorithm
- Discrete Fourier Transform
- Recursive Fast Fourier Transform
- Iterative Fast Fourier Transform

To generate polynomials in C, we will be using arrays. These arrays will use a random number generator to generate a number that we can split into an array. This works fine for polynomials less than n = 32 in length. But if were to scale our implementation to test polynomials of larger sizes then we would run into integer overflow issues since C does not support numbers of this size. There is also the option of simply assigning each array element a number between 0 and 9, thereby generating a polynomial as well. This could work fine, but ideally we would like to have more flexibility with the function. Also when getting into very large polynomials using a dedicated library is gonna be faster and safer than generating 65.000+ random numbers in a for loop.

Therefore to generate arrays from a number we will implement GNU Multiple Precision Arithmetic Library (GMP)[5]. This is a library that can handle multiplication of extremely large numbers and it is able to generate random numbers of basically any size. We will therefore be using it for this project. What we do now is that we generate two random GMP numbers of n size and convert them into polynomials by assigning each digit to a space in an array. These arrays can then be sent to our algorithms and handled in there. This is done the following way:

```
int mpz_to_int_array(mpz_t input_int, int *output_array) {
    // Convert mpz_t to a string in base 10
    char* int_str = mpz_get_str(NULL, 10, input_int);
    int len = strlen(int_str);

    // Store digits in reverse order
    for (int i = 0; i < len; i++) {
        // Convert character to integer (digit)
        int digit = int_str[len - 1 - i] - '0'; // Reverse the index to store in reverse order
        output_array[i] = digit;
```

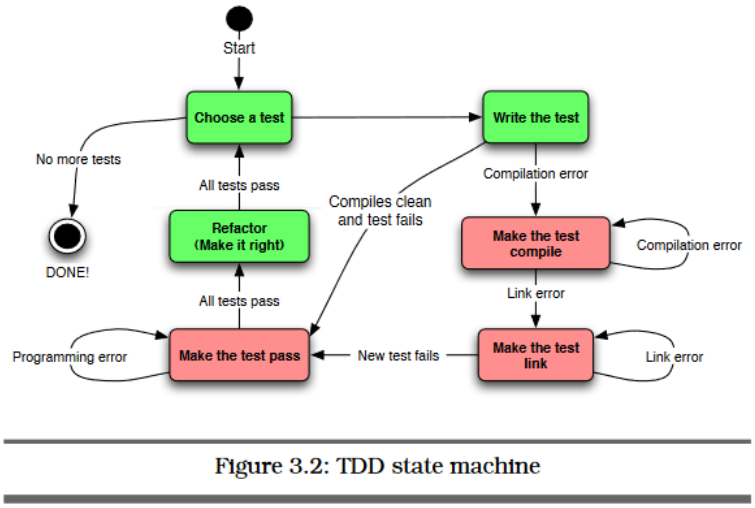


Figure 3.2: TDD state machine

Figure 4: Test Driven Development Graph[6][P. 71]

```
    }

    // Free the allocated string
    free(int_str);
    return len;
}
```

The mpz_t data is how GMP stores numbers. We then convert the mpz_t variable into a a string and get the length with strlen. Next we enter a for loop, where we convert each character of the string to an integer in reverse order. Then we assign it to the output array and repeat that process for each character.

We then send this array to the algorithms, which then returns the solution in a array form. But we then want to check if the result is correct. Earlier we did this by converting the numbers back to GMP form, the legacy code for this can be found in the helper file. But the current way we are doing it is by going through each element of both arrays and comparing each element to ensure that they are the same. This is a trivial check and will therefore not be showcased.

Besides correctness then we want to ensure that all the algorithms use the same standardised setup, so that we can easily compare them. This is done with a polynomial_multiply function that each algorithm has. These functions take two GMP numbers a and b, the n size and a result which is an integer array of size n. This will act as our output and each functions return a time, which is the amount of time it has taken to compute their tasked algorithm.

We start by doing the data handling for each function. This involves converting the GMP numbers over to arrays. When the data handling is done, we start the timer and run the relevant algorithm and the result is saved in the result array. We then stop the timer, return the result and the time and compare the result in the callee function where it is compared against the other results to ensure uniformity and correctness.

12.2 Test Driven Development

Now that the data handling has been configured, we need to measure the algorithms and determine if the algorithms runtime was as expected. If they do not, then there could be three reasons:

1. Memory overhead and extra computational costs
2. inefficient implementation of the algorithms
3. Errors in the code

All three of these have been encountered several times and the code base has been rewritten several times. In fact there has been a total of 57 contributions to the Repository where this project is stored [10]. Using the command:

```
git log --since="150 days ago" -w -p | grep "^[-\+] " | wc
```

We are able to get all the details of this project and see how much has been modified. Running it then returns us the following output:

9129 53523 420570

What these numbers mean are:

- 9129: This is the number of lines that match the filter, indicating the total number of lines added or removed.
- 53523: This is the total number of words in the changed lines.
- 420570: This is the total number of bytes in the changed lines.

If we now want to count the total number of lines we can enter the following Linux command:

wc -l *

This counts all the lines of code in the code directory and returns 1611 lines of code. What this shows is that roughly every line has been edited roughly 6 times on average. This is obviously not fully accurate, but it shows the usage of test driven development[6]. The idea with this is to write the tests first and then write code until the tests pass and then refactor the code to improve the structure.

This has been the development cycle of this project. We started out by first writing the data handling and setting up the tests. Then we wrote the Naive algorithm and DFT algorithm as these are trivial and used them to test the other algorithms. We then kept iterating over and over until all the tests we could think of passed without error or warnings. The idea can be seen in figure 4. Here we see that the tests are what drives the development of the project.

Our work process has been to start by writing the basic mathematics tests to see if the algorithms worked, with simple polynomials of n=2 size. Those tests then failed quite a lot, since we needed to familiarise ourselves with the check libary[2] and setup the data handling. When this was working with the naive and DFT algorithms, we were able to run the tests on the other algorithms until they one by one passed the tests.

Then we got to the large polynomial test, which is where things started falling apart as our code was overflowing due to not accounting for C's built in limitations with integer sizes. We therefore needed to rewrite much of the code and datahandling to implement GMP instead. Then we ran the basic math tests with GMP instead, which failed and we needed to rewrite the basic math tests until they worked as intended. Then we ran the different algorithms on the new test setup until all the bugs had been fixed. Then we moved back to the large numbers and got them running.

This process has then been iterated over again and again until we now have a fully functional and complete code base that runs reliably and at the expected runtime. Next we will look at the actual tests that we wrote.

12.3 Testing

There are two important types of tests that needs to be done for these algorithms, the first and most important is correctness. We have already mathematically proven that the algorithms compute the correct number, but we still need to prove that our implementation of them is correct. Secondly we need to calculate their actual speed and if it matches their asymptotic notation.

12.3.1 Correctness

To start with we will look at correctness. Here we have written a suite of unit test that tests all the function equally. This has been done using a library called check [2]. This allows us to test different parameters and edge cases to see if the code fails. We ended up with the following tests to tests our implementation of the algorithms:

- Basic math, where we do each number from 0 to 10 and see if they are correct
- Zero test, which tests if it can do two zero polynomials together
- Large polynomial test, here we test two polynomials each of size n=1024 and see if they compute correctly
- Negative test, to see if they correctly compute negative polynomials
- even test, to see if the functions correctly compute two polynomials of even length
- odd test, to see if the functions correctly compute two polynomials of odd length

When running these tests we get the following result:

```
Running suite(s): BasicMathSuite
100%: Checks: 500, Failures: 0, Errors: 0
Running suite(s): ZeroSuite
100%: Checks: 5, Failures: 0, Errors: 0
Running suite(s): Large polynomials Suite
100%: Checks: 50, Failures: 0, Errors: 0
Running suite(s): NegativeSuite
100%: Checks: 5, Failures: 0, Errors: 0
Running suite(s): evenSuite
100%: Checks: 500, Failures: 0, Errors: 0
Running suite(s): oddSuite
100%: Checks: 500, Failures: 0, Errors: 0
```

This shows that the implementation is correct. We can therefore move onto the runtime, which is the more interesting tests.

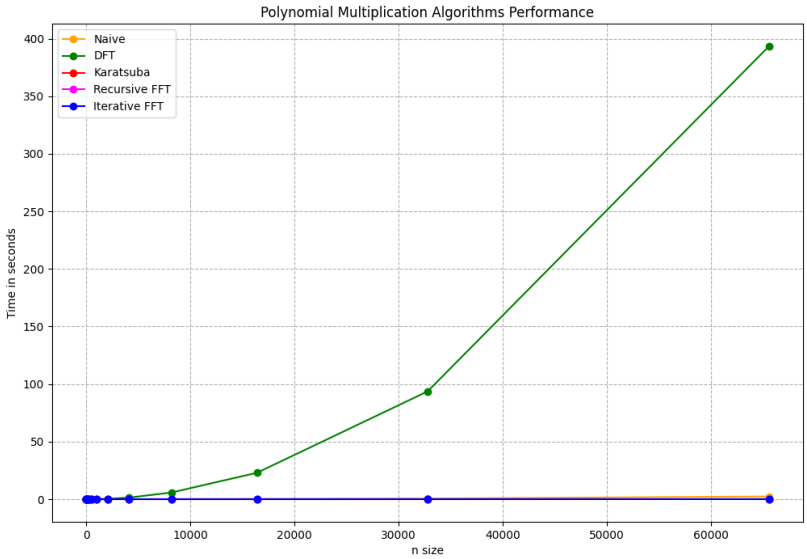
12.3.2 Runtime

To test the runtime we started by using clock. But this proved to be inaccurate for smaller n size which are calculated in nanosecond. Instead we will be using clock_gettime() with CLOCK_MONOTONIC which is a built in Linux function. This measures the absolute elapsed wall-clock time since an unspecified fixed point in the past [1]. This will allow us to get the time with great accuracy. This will then be placed around the function calls inside the Polynomial_Multiplication() functions so we only measure the time each algorithm takes and not their seperate data handling.

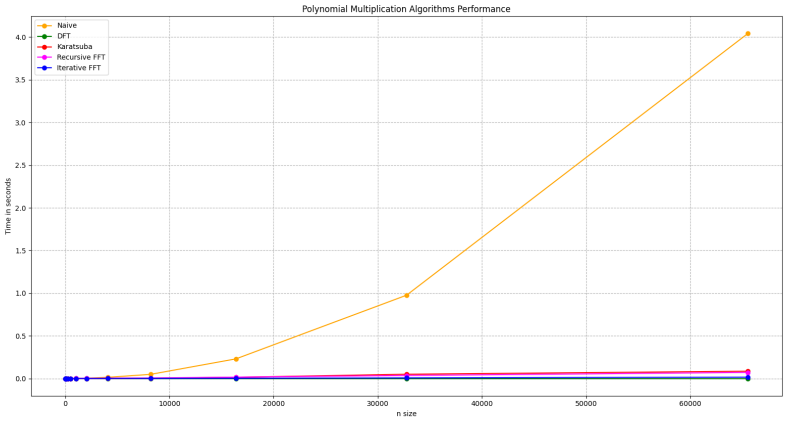
To start with we will discuss the DFT algorithm, which grows worse than expected. Theoretically it should grow similar to the Naive approach. But due to all the data handling and overhead then it performs significantly worse as can be seen in figure a. We will therefore be removing it from the graph so we can focus on the other algorithms and see their performance more clearly.

When looking at the performance of the Naive approach in figure b we see that it scales in computation time much faster than the other algorithms. But it is nowhere near as bad as the DFT algorithm and it does not fully look like the $O(n^2)$ graph we looked at in figure 3. The reason for this is likely that C is extremely optimised towards multiplication, especially with the O2 flag enabled. If we try to disable the O2 flag, then we see in figure c that it is now much slower at computing polynomials. The computation time for an n size of 2^{16} has now increased by 3.5 times from 4 to 14 seconds. This shows that the O2 flag does what it is intended to do and optimises the compiled program to give a better performance.

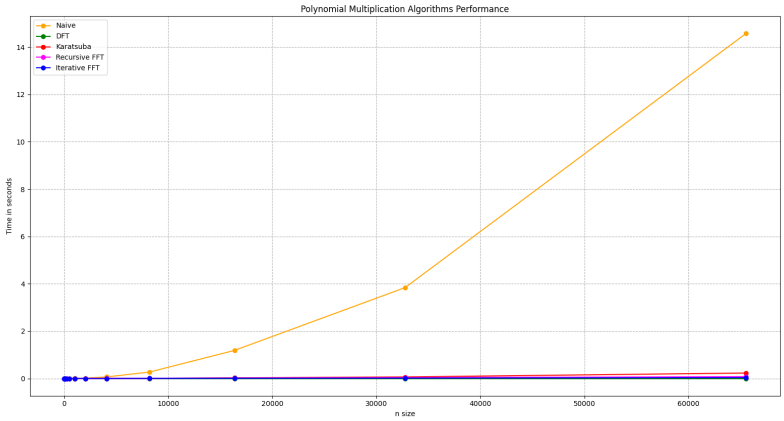
If we now exclude the Naive approach, then we see in figure d that our results align with the expected theory and Karatsuba's algortihm scales significantly worse than the FFT algorithms. This aligns with the theory from the amortized runtime analysis which calculated that Karatsuba would run at $O(n^{1.59})$. We can optimise Karatsuba by looking at our previous findings where we found that the Naive approach performs much better when applying the O2 flag to the compiler. This means that instead of breaking the recursive loop at single length polynomials, maybe we can try to increase that size limit?



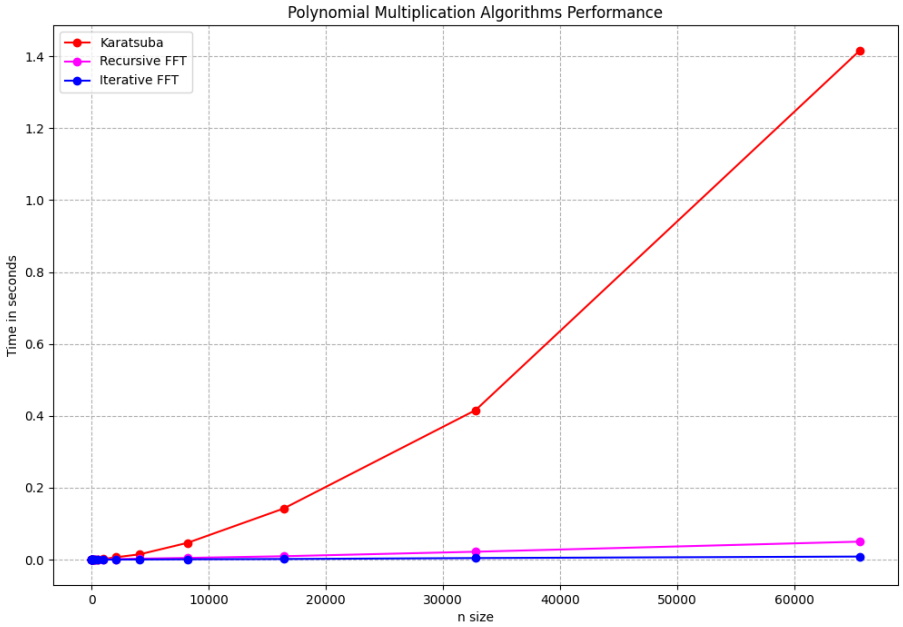
(a) Runtime including DFT



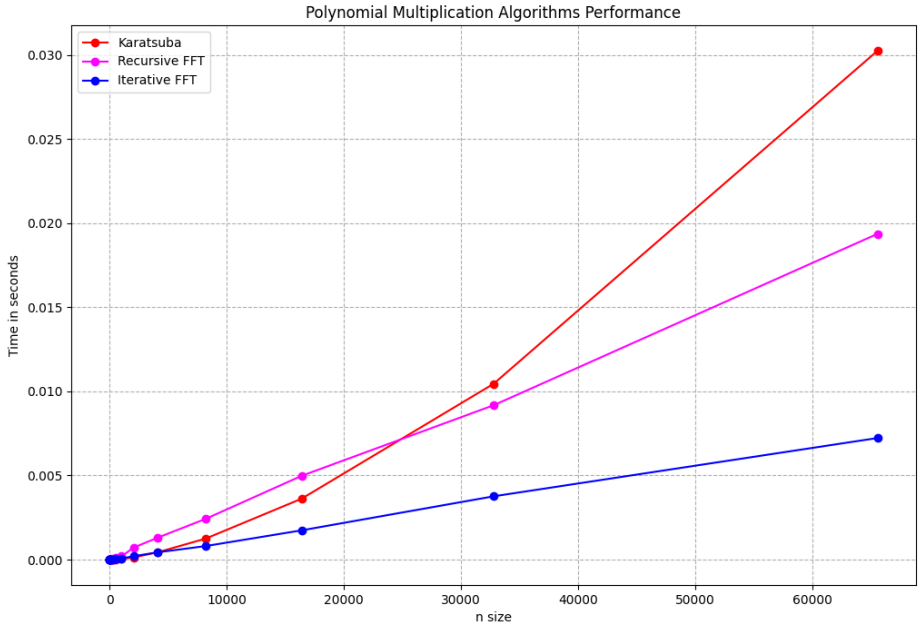
(b) Runtime Excluding DFT



(c) Runtime excluding the 02 flag and DFT



(d) Runtime excluding DFT and naive



(e) Runtime when optimising Karatsuba

12.3.3 Optimising Karatsuba's algorithm

The line we will be looking at is specifically this one here from Karatsuba:

```
if (length_input1 <= 1 || length_input2 <= 1) {
        Array_Multiplication(input1, input2, length_input1, length_input2, result);
        return;
    }
```

It sits at the top of Karatsuba's algorithm and is used to break the recursive loop and do naive polynomial multiplication when the size of the polynomials are small enough that it would be more efficient to simply use the Naive approach to calculate instead. In the current implementation we have a length of 2, but from our tests we know that the Naive approach performs significantly better than expected due to optimisations in C.

Therefore we wrote a small binary search test to find the optimal breaking point where we would switch over to the naive approach. We ran this test 1000 times and came up with the number 250. If we now try to insert that into our code it will look like this instead:

```
if (length_input1 <= 250 || length_input2 <= 250) {
        Array_Multiplication(input1, input2, length_input1, length_input2, result);
        return;
    }
```

If we now try to run the test and compare. Then we see in figure e that Karatsuba's algorithm now runs faster than the recursive FFT algorithm when $n < 30.000$, though after that point the Recursive FFT algorithm begins to scale significantly better. This shows that optimising algorithms is not only about mathematical optimisations, but also knowing and taking advantage of the strengths of the programming language the algorithm is being implemented in. Yes Karatsuba scales worse than the FFT algorithm, but by delaying this scaling we can greatly improve the utility of the algorithm.

The Iterative FFT algorithm still scales significantly better though and closely resembles the $O(n \cdot logn)$ scaling as seen in figure 3, which indicates a succesfull implementation. Though the Recursive FFT algorithm starts to display the same properties when the complexity increases, sadly we are not able to scale the tests further to the limitations of C. The Recursive FFT algorithm has to much overhead and memory management, which is causing it to only start running in $O(n \cdot log \cdot n)$ time after the memory allocation costs has become negliable. Therefore the closest of the two FFT algorithms to run in $O(n \cdot log \cdot n)$ time would be the Iterative version. We would like to test with larger numbers, but going beyond n^{16} results in the elements of the integer arrays to overflows. But based on the tests we can predict that the Iterative FFT algorithm would only continue to outperform the other algorithms, closely followed by the Recursive FFT.

The results that these graphs have been calculated with are attached in appendix in figure 9, figure 10 and figure 11

12.4 Choosing the right pseudo-code

When testing the time, we also found that there were drastic difference in performance between the pseudo-codes. As could be seen when we optimised Karatsuba's algorithm, then small changes can make a big difference. For instance the Recursive FFT algorithm listed on Wikipedia[15] runs significantly worse than the one from Introduction to Algorithms [4] that we used. This difference in performance can be seen in figure 8. Here we see that Wikpedia's version scales significantly worse than the the one given in Introduction to Algorithms. If we examine it a bit closer at Wikipedia's approach:

```
    X0,...,N-1 <- ditfft2(x, N, s):              DFT of (x0, xs, x2s, ..., x(N-1)s):
    if N = 1 then
        X0 <- x0                                       trivial size-1 DFT base case
    else
        X0,...,N/2-1 <- ditfft2(x, N/2, 2s)              DFT of (x0, x2s, x4s, ..., x(N-2)s)
        XN/2,...,N-1 <- ditfft2(x+s, N/2, 2s)            DFT of (xs, xs+2s, xs+4s, ..., x(N-1)s)
        for k = 0 to N/2-1 do                          combine DFTs of two halves into full DFT:
            p <- Xk
```

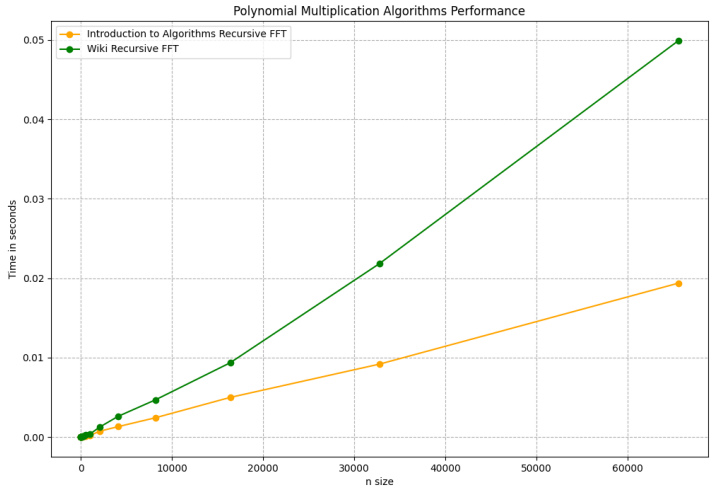


Figure 7: Wikipedia's Recursive FFT vs Introduction to Algorithms

```
            q <- exp(-2i/N k) Xk+N/2
            Xk <- p + q
            Xk+N/2 <- p - q
        end for
    end if
```

Most of the code is the same as the one we implemented, but there is one key difference. The twiddle factor computation has been moved into the for loop at the end. Which means that we now need to compute it $\frac{n}{2}$ times instead of just once. This is a significant cost increase that is not reflected when running an amortized runtime analysis on the algorithms. The only way to really know which one to pick is by closely examining the pseudo-code line by line and trying to use common sense to see which one is faster. Or if it is not obvious implement both, test them and pick the better one. The results from our tests can be found in figure 12.

12.5 Known issues

The code is reliable and has passed all our tests successfully. But there are still limitations, the main one is down to C and how it handles numbers. Since we are storing our result in an integer array, then we can store very large polynomials. But there is a size limit before it segfaults, this limit is an n size of $n = 2^{16}$. If we go beyond this point then each element of the array exceed the 64 bit limit that C can handle. There is no way I know to get around this for polynomials. With GMP you can exceed it with numbers, but for C integer arrays it is not possible.

Another issue lies in the way we generate polynomials from a whole number, this does not allow us to test polynomials containing negative numbers. This should not pose an issue though and it could relatively easily be implemented by doing the datahandling and setup different. But it would not add anything significant to this paper. Therefore we will not be implementing it.

Aside from this there are no known issues with the code. The main thing that could have been optimised would maybe be the runtime. We could implement caching, more efficient memory allocation and other methods that could improve the efficiency of the code. To prevent ballooning of the project though, then this is not necessary, though a more efficient implementation would obviously be desirable.

13 Discussion

Throughout this paper we have been discussing a variety of algorithms and their implementation. We will now try to gather the key findings and discuss them.

13.1 Efficiency

Our findings show that the computational efficiency matches the expected result. The main interesting finding has been the improved performance of Karatsuba's algorithm, which far exceeded its expected runtime and even outperformed the Recursive version the FFT algorithm for most of the tests, though not the iterative.

This difference as previously described is because of the computational overhead, that comes from memory management. This is an important thing to consider when doing amortized runtime analysis, where we simply look at the number of computation, but not their actual cost.

This can also be seen in the difference between the naive approach and the the DFT algorithm. The DFT algorithm being more than 100 times slower despite having the exact same amortized runtime analysis cost at $n = 2^{16}$. This difference only increases with complexity size as can be seen in figure a, showing that it scales much slower. We therefore believe that while amortized runtime analysis is a great way to estimate the computational cost of an algorithm, then it is not a bible. There are issues and things, which are not accounted for, such as more expensive computations and extra calculations within a loop.

The analysis is generally accurate though, as can be seen if we exclude the DFT algorithm which can be seen in figure b. Therefore we would argue that doing an amortized runtime analysis of an algorithm is an important step when implementing algorithms and will help pick the optimal algorithm to use. Then when the optimal algorithm has been picked, we can then implement it and see if the implemented runtime matches the expected runtime. Doing it this way saves a lot of implementation time, since the amortized runtime analysis can be done much faster than a full implementation of an algorithm. It also acts as a great guideline to see if ones code run as expected.

Another point on efficiency, is that we know from the test that the size of the polynomials make a great difference, as could be seen when we tried optimising Karatsuba's algorithm and found that below an n size of $n = 250$ the naive approach works better. This is due to the cost of allocating memory, which takes a lot of time, but that time does not really scale compared a lot and therefore the Karatsuba and FFT algorithms will generally perform better the larger the polynomials they are computing.

13.2 implementation discussion and challenges

As discussed previously there is a lot of memory management when working with polynomials in C. This has also been the main issue in implementation. Namely ensuring that the program is memory safe and won't overflow. We have not really concerned ourselves to much with safety and trying to fix security flaws in the code itself as this is not part of the scope of the project. Instead our priority has been to get the algorithms implemented, tested and optimised.

The main challenge in this project has been implementing Karatsuba and the recursive FFT algorithm. This is due to the memory management and the recursive calls. This means that we always need to know the current size of the array and be able to read and write from the correct arrays. This has proven quite challenging and as can be seen, then the recursive FFT algorithm has been implemented more successfully since we were able to use memory pointers, which are more efficient than creating and allocating new arrays. This is what we needed to do in Karatsuba since we were unable to implement memory pointers successfully in Karatsuba.

This limitation did not prove particularly significant though, since we were still able to optimise Karatsuba to be able to outperform the Recursive FFT algorithm in most applications by more effectively utilising the Naive approach. There are many other small optimisations in the implementation as well. Such as saving the twiddle factor $e^{-i*TAU*n/k}$ as a variable in the FFT algorithms so that we do not need to calculate it twice. Another example is using TAU instead of PI to prevent the extra calculations. These optimisations might seem small but they add up as the computations scale. Especially saving the twiddle factor in a variable is valuable as can be seen in the DFT where the cost of this computation is partly responsible for the poor computational performance or the difference between the two Recursive FFT pseudo-codes. Therefore being able to optimise this in the FFT is critical.

Another challenge was to ensure that the program was accurate. We ran into many issues with off by 1 errors in our calculations, especially in the Karatsuba algorithm. This was resolved through testing

and reading through the code and trying to run the code by hand. This led to finding errors such as a rounding error in our implementation of Karatsuba's algorithm. Another way was to use the basic math tests in the white box test to compute the output with small polynomials to see what it would produce and compare it to the one computed by either hand or the naive approach. This gradually led to the implementations becoming more and more stable.

Another major issue was the testing setup and timing of the algorithms. Since they handle the data so differently we needed to find a fair way to measure them. What we ended up doing was to measure only the algorithm itself and not the data handling. The reasoning for this is while the data handling is an important part of the implementation of the algorithms, then it is not the algorithms themselves that require it. It is just our way of implementing them that requires it. It also adds constant time to the algorithms that make them harder to measure. If you include all the datahandling in the polynomial_multiply() functions, then the difference between the FFT and the Naive approach is still 2 seconds, but now the FFT algorithm takes 9 seconds where the naive would take 11. This therefore does not clearly reflect the actual computation times of the algorithms. The algorithms are different though, since the DFT and FFT algorithms need to run twice, then we have placed the timer start before the first DFT/FFT call and the timer stop after the IDFT/IFFT algorithm has been completed. This provides a fair and equal way to measure the different algorithm, as the conversion to frequency and then time domain plays an important part in computational cost of the DFT based algorithms.

13.3 Importance of efficient algorithmic implementation

Throughout the project we have gradually optimised the implementations more and more through an iterative process using Test Driven Development. This has led to great improvement, such as the optimisations to Karatsuba's algorithm, which can be seen in figure e or the optimisations to the Recurisive FFT algorithm as can be seen in figure 8. Iterating over your code and testing is key to developing fast and efficient code. If you had to use polynomial multiplication in the industry for cryptography, then improving the runtime down to $O(n \cdot logn)$ from $O(n^2)$ would drastically improve the performance of the overall project. Improving the fundamental algorithms is critical in improving basically any project as if you can improve these, then the rest of the project will benefit as well.

Therefore efficient algorithmic implementations, should be key in most projects as it can greatly improve the effectiveness of the rest of the code. This project here is merely a test, that showcases the importance in polynomial multiplication. But there are many other critical algorithms, such as shortest path algorithms and search/sorting algorithms[4] that are widely used many placed in the industry. Sometimes, such as with the FFT algorithm, it can also be used to prevent war as we previously discussed[11].

If we had more time then there are many more things which we would have liked to look into, such as the Toom Cook algorithm. But our priority has been on focusing and maintaining a narrow vision as we would rather do our best with a small selection of algorithms, than spread out to much and increase the scope beyond our capabilities.

14 Conclusion

In this paper we have examined the various ways to do polynomial multiplication and the challenges that implementing them can bring. We have found that amortized runtime analysis is a great tool to get a rough estimate of the computational cost of an algorithm. But the only real way to measure its efficiency is by implementing it yourself.

We found that for polynomial multiplication, the choice of algorithm matters a lot. For small to medium sized polynomials, especially when working with C, then the Naive algorithm is the best choice. But the more we scale the project, then the optimisations in Karatsuba and FFT algorithm start to shine through.

We also found that the amortized runtime analysis does not consider the overhead, such as memory handling and complex calculations, which can be costly in algorithmic implementation. As could be seen when running the DFT algorithm. We also found that by utilising the strength of the Naive approach and the strength of Karatsuba, we can compute the results much more efficiently than the amortized runtime analysis would suggest and even outperform the recursive FFT algorithm for most tests. In the end though, the Iterative FFT algorithm seems to be optimal for medium to large polynomials.

References

[1] Ismail Ajagbe. Understanding timekeeping and clocks in linux. *https://www.baeldung.com/linux/timekeeping-clocks*, 2024.

[2] check. What is check? *https://libcheck.github.io/check/*, 2020.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math Comput*, 1965.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[5] GMP. The gnu multiple precision arithmetic library. *https://gmplib.org/*, 2024.

[6] J.W. Grenning. *Test Driven Development for Embedded C*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2011.

[7] T Johnson Don H Burrus C. Sidney Heideman, Michael. Gauss and the history of the fast fourier transform. *Archive for History of Exact Sciences*, 1985.

[8] Umair Hussaini. Twiddle factors in dsp for calculating dft, fft and idft. *https://technobyte.org/twiddle-factors-dsp-dft-fft-idft/*, 2020.

[9] Graham Kendall. The first moon landing was achieved with less computing power than a cell phone or a calculator. *https://psmag.com/social-justice/ground-control-to-major-tim-cook*, 2019.

[10] Rasmus Ladefoged. Polynomial multiplication. *https://github.com/RasmusLC1/Polynomial-Multiplication*, https://github.com/RasmusLC1/Polynomial-Multiplication.

[11] JFK library. Nuclear test ban treaty. *https://www.jfklibrary.org/learn/about-jfk/jfk-in-history/nuclear-test-ban-treaty*, 2024.

[12] Wolfram Mathworld. Karatsuba multiplication. *https://mathworld.wolfram.com/KaratsubaMultiplication.html*, 2024.

[13] Brian Ryu. Big o notation. *https://medium.com/@s.brianryu/big-o-notation-e52ce96d9486*, 2019.

[14] sharetechnote. Time domain vs frequency domain. *https://www.sharetechnote.com/html/RF_Handbook_TimeDomain_F*

[15] Wikipedia. Cooley–tukey fft algorithm. *https://en.wikipedia.org/wiki/Cooley*

[16] Stefan Wörner. Fast fourier transform, numerical analysis seminar. Swiss Federal Institute of Technology Zurich, *N/A*.

n_size	Naive	DFT	Karatsuba	Recursive FFT	Iterative FFT
2	0.000002	0.000944	0.000001	0.000002	0.000011
4	0.000001	0.000003	0.000002	0.000002	0.000002
8	0.000001	0.000010	0.000003	0.000003	0.000002
16	0.000002	0.000037	0.000007	0.000006	0.000003
32	0.000003	0.000155	0.000016	0.000010	0.000005
64	0.000005	0.000654	0.000038	0.000016	0.000007
128	0.000015	0.002135	0.000102	0.000040	0.000013
256	0.000054	0.007229	0.000191	0.000053	0.000016
512	0.000159	0.026553	0.000851	0.000136	0.000037
1024	0.000606	0.091299	0.001724	0.000208	0.000075
2048	0.002131	0.341296	0.004956	0.000608	0.000156
4096	0.008140	1.353876	0.014601	0.001095	0.000333
8192	0.035709	6.258063	0.045403	0.002188	0.000714
16384	0.136618	23.262430	0.145702	0.005270	0.002027
32768	0.557519	91.135094	0.418277	0.009357	0.003615
65536	2.140545	369.237152	1.466449	0.024163	0.012641

Figure 8: Computation Times for Various Algorithms

n_size	Naive	Karatsuba	Recursive FFT	Iterative FFT
2	0.000044	0.000001	0.000002	0.000011
4	0.000046	0.000002	0.000002	0.000002
8	0.000049	0.000003	0.000003	0.000002
16	0.000054	0.000007	0.000006	0.000003
32	0.000066	0.000016	0.000010	0.000005
64	0.000097	0.000038	0.000016	0.000007
128	0.000209	0.000102	0.000040	0.000013
256	0.000525	0.000191	0.000053	0.000016
512	0.001698	0.000851	0.000136	0.000037
1024	0.005899	0.001724	0.000208	0.000075
2048	0.070709	0.004956	0.000608	0.000156
4096	0.141008	0.014601	0.001095	0.000333
8192	0.442001	0.045403	0.002188	0.000714
16384	1.633481	0.145702	0.005270	0.002027
32768	6.705897	0.418277	0.009357	0.003615
65536	32.635822	1.466449	0.024163	0.012641

Figure 9: Computation Times Excluding -O2 Flag

n_size	Naive	Karatsuba	Recursive FFT	Iterative FFT
2	0.000002	0.000001	0.000044	0.000006
4	0.000001	0.000001	0.000002	0.000002
8	0.000001	0.000001	0.000003	0.000003
16	0.000002	0.000001	0.000005	0.000003
32	0.000003	0.000001	0.000009	0.000005
64	0.000005	0.000002	0.000015	0.000007
128	0.000015	0.000002	0.000028	0.000013
256	0.000054	0.000006	0.000072	0.000026
512	0.000208	0.000021	0.000155	0.000054
1024	0.000824	0.000065	0.000303	0.000115
2048	0.003303	0.000202	0.000879	0.000248
4096	0.008348	0.000381	0.001105	0.000341
8192	0.039508	0.001155	0.002229	0.000888
16384	0.150598	0.003552	0.004644	0.001694
32768	0.552095	0.010445	0.009608	0.004010
65536	2.418637	0.032706	0.020583	0.009065

Figure 10: Computation Times Without Karatsuba Optimization

n_size	**Introduction to Algorithms Recursive FFT multiplication**	**Wiki Recursive FFT multiplicatio**
2	0.000028	0.000025
4	0.000001	0.000002
8	0.000002	0.000003
16	0.000003	0.000006
32	0.000006	0.000011
64	0.000030	0.000023
128	0.000017	0.000050
256	0.000049	0.000125
512	0.000097	0.000270
1024	0.000194	0.000389
2048	0.000714	0.001229
4096	0.001302	0.002600
8192	0.002420	0.004678
16384	0.004979	0.009345
32768	0.009175	0.021841
65536	0.019366	0.049900

Figure 11: Performance comparison of Recursive FFT implementations