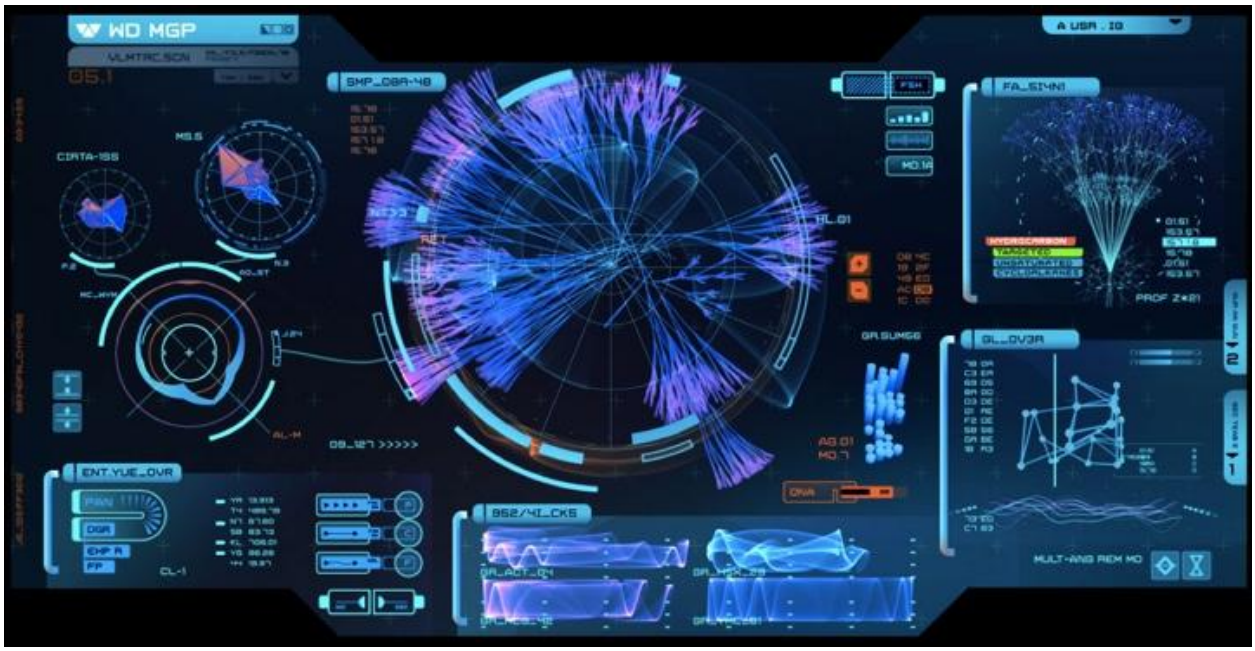
XIE JAROD 28710097

MODEL PROJECT REPORT



Introduction

This project focuses on implementing efficient algorithms for complex number arithmetic, Fast Fourier Transforms (FFT), and polynomial multiplication. The goals include developing a system for complex numbers using double-precision floating-point representation, creating FFT and inverse FFT algorithms for vectors of size 2k (with adaptability for non-power-of-2 sizes), and implementing both naive and FFT-based algorithms for multiplying polynomials with integer coefficients. The project culminates in a comparative analysis of the efficiency of these algorithms, considering factors such as computational complexity and execution time.

Key Project Objectives:

1. **Complex Number Arithmetic:**
    - Implement an arithmetic system for complex numbers using double-precision floating-point numbers, facilitating operations involving both real and imaginary parts.
2. **Fast Fourier Transforms (FFT) and Inverse FFT:**
    - Develop FFT and inverse FFT algorithms tailored for vectors of size 2k, with a crucial enhancement to handle vectors of sizes that are not powers of 2.
3. **Polynomial Multiplication Algorithms:**
    - Implement a naive algorithm for multiplying polynomials with integer coefficients.
    - Implement an FFT-based algorithm for polynomial multiplication, as detailed in the course material.
4. **Efficiency Comparison(BenchMark):**
    - Conduct a comparative analysis of the efficiency of the implemented algorithms, considering factors such as computational complexity, execution time, and resource utilization.

Complex Number Arithmetic:

To implement an arithmetic system for complex numbers using double-precision floating-point numbers, I create a structure to represent complex numbers and define methods for the basic arithmetic operations: addition, subtraction, multiplication, and division and others.

Here's a brief explanation of the arithmetic operations for complex numbers:

- Addition: (a + bi) + (c + di) = (a + c) + (b + d)i
- Subtraction: (a + bi) - (c + di) = (a - c) + (b - d)i
- Multiplication: (a + bi) * (c + di) = (ac - bd) + (ad + bc)i
- Division: (a + bi) / (c + di) = [(a + bi) * (c - di)] / (c^2 + d^2)

.

Fast Fourier Transforms (FFT) and Inverse FFT:

Fast Fourier Transform (FFT) is a crucial algorithm in signal processing and various scientific applications for efficiently computing the discrete Fourier transform of a sequence..In the context of this project, we aim to develop FFT and Inverse FFT algorithms specifically designed for vectors of size 2k, optimizing performance for sequences with a power-of-2 length. Moreover, our implementation also includes vectors of sizes that are not powers of 2, ensuring versatility and applicability to real-world datasets of varying lengths. The inverse FFT is equally important, allowing us to efficiently transform frequency domain representations back to the original time domain. This project explores the intricacies of these algorithms, emphasizing efficiency, versatility, and adaptability to diverse data sizes.

Here is the pseudo-code for the FFT algorithm:

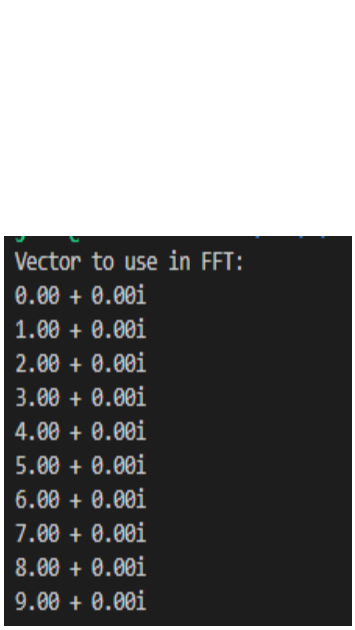
```
1.  Function FFT(a: Array of Complex, n: Integer, invert: Integer)
2.      if n <= 1 then
3.          return
4.      end if
5.
6.      even = AllocateMemory(n / 2 * sizeof(Complex))
7.      odd = AllocateMemory(n / 2 * sizeof(Complex))
8.
9.      for i from 0 to n / 2 - 1
10.         even[i] = a[i * 2]
11.         odd[i] = a[i * 2 + 1]
12.     end for
13.
14.     FFT(even, n / 2, invert)
15.     FFT(odd, n / 2, invert)
16.
17.     ang = (2.0 * PI / n) * (invert ? -1 : 1)
18.     w = {cos(ang), sin(ang)}
19.     wn = {1.0, 0.0}
20.     for i from 0 to n / 2 - 1
21.         t = multiply_c(wn, odd[i])
```

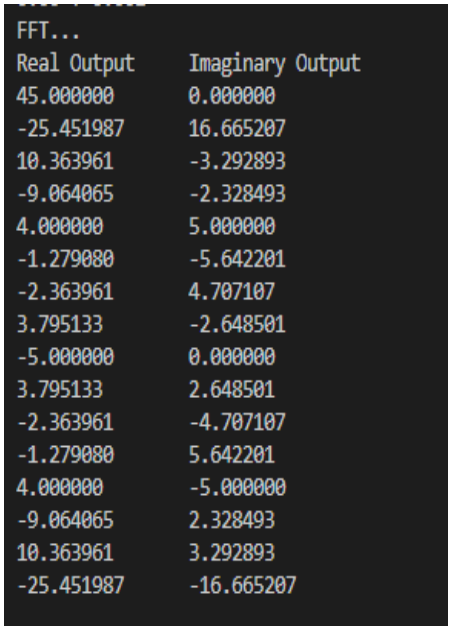
```
22.         a[i] = add_c(even[i], t)
23.         a[i + n / 2] = subtract_c(even[i], t)
24.         wn = multiply_c(wn, w)
25.     end for
26.     FreeMemory(even)
27.     FreeMemory(odd)
28. End Function
```

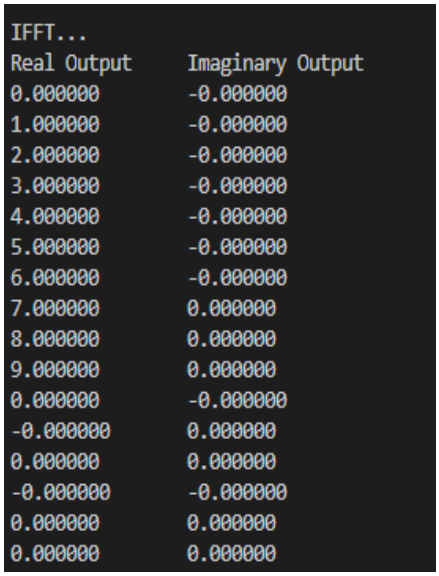
Here is the pseudocode for IFFT :

```
1.  Procedure IFFT(a: Array of Complex, n: Integer)
2.      FFT(a, n, 1)
3.      for i from 0 to n - 1
4.          a[i].real /= n
5.          a[i].imag /= n
6.      end for
7.  End Procedure
```

I will now present some results from the FFT/IFFT functions that I have implemented for the project.







At the beginning, we have the vector (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) with all its coordinates being real numbers. After applying the FFT, we obtain a vector of size 2^k (16 in this case). When we subsequently apply the IFFT to the result of the FFT operation on the initial vector, we retrieve the original vector, effectively padded with zeros to achieve a size of 2^k.

Polynomial Multiplication Algorithms:

Polynomial multiplication lies at the core of various mathematical and computational tasks, and this project encompasses the implementation of two distinct algorithms for this purpose. Firstly, a naive algorithm is introduced, providing a straightforward approach to multiply polynomials with integer coefficients. While conceptually simple, the naive algorithm serves as a benchmark for comparison against more advanced methods. Secondly, an FFT-based algorithm is implemented, drawing inspiration from the course material. This advanced approach leverages the efficiency of Fast Fourier Transforms to enhance the speed and computational efficiency of polynomial multiplication. The project's objective is to explore and compare the performance of these two algorithms, shedding light on their strengths, weaknesses, and respective applications in real-world scenarios.

Here is the pseudocode for naive algorithm :

```
1. Function multiplyPolynomials(poly1: Array of Term, size1: Integer, poly2: Array of Term, size2: Integer, resultSize: Reference to Integer) -> Array of Term
2.     resultSize = size1 * size2
3.     result = AllocateMemory(resultSize * sizeof(Term))
4.     for i from 0 to resultSize - 1
5.         result[i].coefficient = 0
6.         result[i].exponent = 0
7.     end for
8.     for i from 0 to size1 - 1
9.         for j from 0 to size2 - 1
10.            coeff = poly1[i].coefficient * poly2[j].coefficient
11.            exp = poly1[i].exponent + poly2[j].exponent
12.            result[i * size2 + j].coefficient += coeff
13.            result[i * size2 + j].exponent += exp
14.        end for
15.    end for
16.    return result
17. End Function
```

Here is the pseudocode for FFT_based algorithm :

```
1. Procedure FFT_mult(poly1: Array of Integer, size1: Integer, poly2: Array of Integer, size2: Integer, result: Reference to Array of Integer, resultSize: Reference to Integer)
2.         fftSize = 1
3.         while fftSize < (size1 + size2 - 1)
4.                 fftSize *= 2
5.         end while
6.
7.         // Allocate memory for FFT arrays
8.         fftPoly1 = AllocateMemory(fftSize * sizeof(Complex))
9.         fftPoly2 = AllocateMemory(fftSize * sizeof(Complex))
10.        for i from 0 to fftSize - 1
11.                fftPoly1[i] = (i < size1) ? Complex(poly1[i], 0.0) : Complex(0.0, 0.0)
12.                fftPoly2[i] = (i < size2) ? Complex(poly2[i], 0.0) : Complex(0.0, 0.0)
13.        end for
14.
15.        FFT(fftPoly1, fftSize, 0)
16.        FFT(fftPoly2, fftSize, 0)
17.
18.        for i from 0 to fftSize - 1
19.                fftPoly1[i] = multiply_c(fftPoly1[i], fftPoly2[i])
20.        end for
21.        IFFT(fftPoly1, fftSize)
22.        resultSize = size1 + size2 - 1
23.        for i from 0 to resultSize - 1
24.                result[i] = round(fftPoly1[i].real)
25.        end for
26.
27.        FreeMemory(fftPoly1)
28.        FreeMemory(fftPoly2)
29. End Procedure
```

I will now present some results from the naive algorithm that I have implemented for the project. (Don't forget to remove the quote to use this part of the code)

```
Example of 2 polynomial to multiply
2x^2 + 3x^1 + 4x^0
1x^1 + 5x^0
Multiplication of the 2 polynomial
Result before assembling like terms: 2x^3 + 10x^2 + 3x^2 + 15x^1 + 4x^1 + 20x^0
Result after assembling like terms: 20x^0 + 19x^1 + 13x^2 + 2x^3
```

As evident from the outcome, the result aligns with our anticipated expectations.

I will now present some simple results from the FFT-based algorithm that I have implemented for the project.

```
FFT_mult of the 2 polynomial
Polynomial 1: 4x^0 + 3x^1 + 2x^2
Polynomial 2: 5x^0 + 1x^1
Result: 20x^0 + 19x^1 + 13x^2 + 2x^3
```

As observed, utilizing identical polynomials yields consistent results.

BenchMarks :

This project conducts a benchmark comparison between the FFT-based algorithm and the naive algorithm for polynomial multiplication. Polynomial multiplication, a fundamental operation in diverse applications, is approached by the straightforward naive method and the more advanced FFT-based technique. The objective is to assess the performance of these algorithms across different polynomial sizes, revealing their respective strengths and weaknesses. Through concise benchmarking, the project aims to guide the selection of the most suitable algorithm for specific computational tasks.

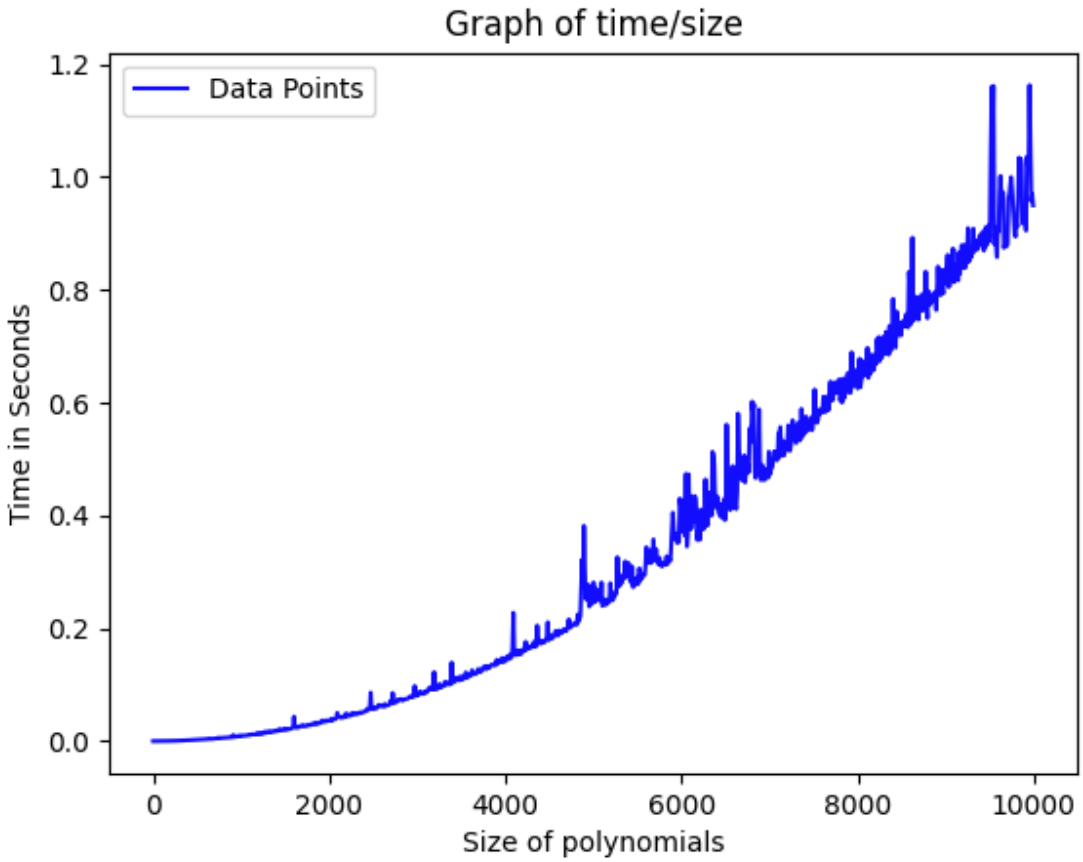
Let's briefly discuss the theoretical complexities of the naive algorithm for polynomial multiplication and the FFT-based algorithm.

The naive algorithm for polynomial multiplication involves multiplying each term in one polynomial with every term in the other polynomial.If the polynomials have degrees m and n, the resulting polynomial will have a degree of m+n.The time complexity of the naive algorithm is $O(m \cdot n)$, as it requires $m \cdot n$ multiplications.

The FFT-based algorithm employs Fast Fourier Transforms to reduce the time complexity of polynomial multiplication. If the polynomials have degrees m and n, the time complexity of the FFT-based algorithm is $O((m+n) \cdot \log(m+n))$.The reduced time complexity is particularly advantageous when dealing with large polynomials.

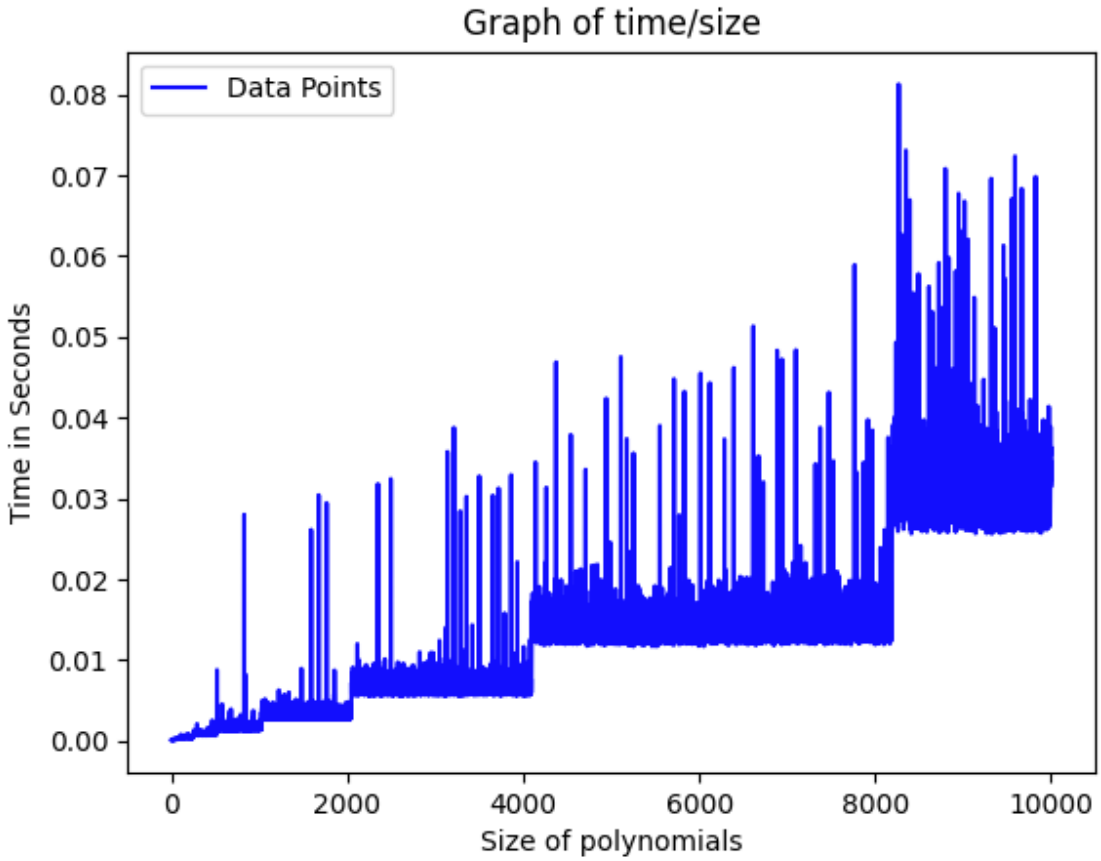
Let's observe how these algorithms perform in practice for both cases:

1. For the naive algorithm we have this graph.



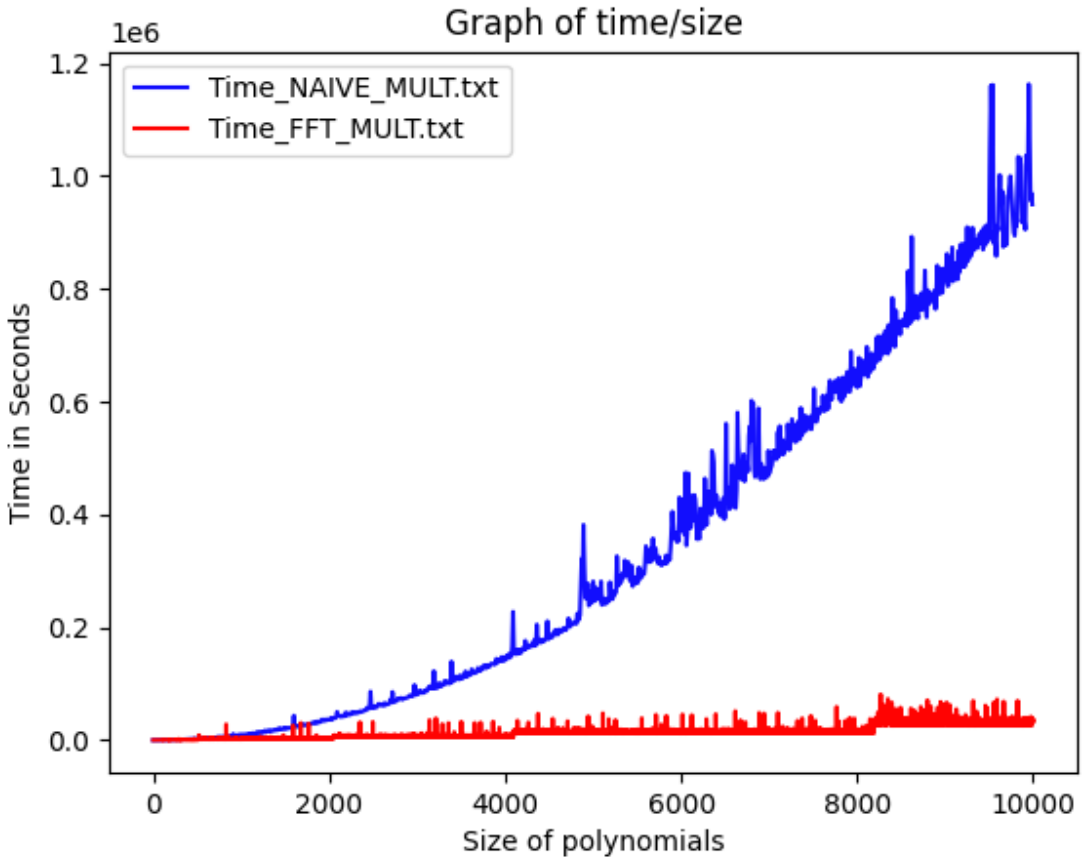
The naive multiplication algorithm is based on the classical pairwise matching of the coefficients of polynomials. The complexity of this algorithm is quadratic, meaning that the computation time increases proportionally to the square of the input size. The theoretical results, graphically represented, demonstrate an exponential growth in the required time for large polynomial sizes, thus emphasizing the inefficiency of this approach for extensive calculations.

2. Let’s look at the FFT-based algorithm now.

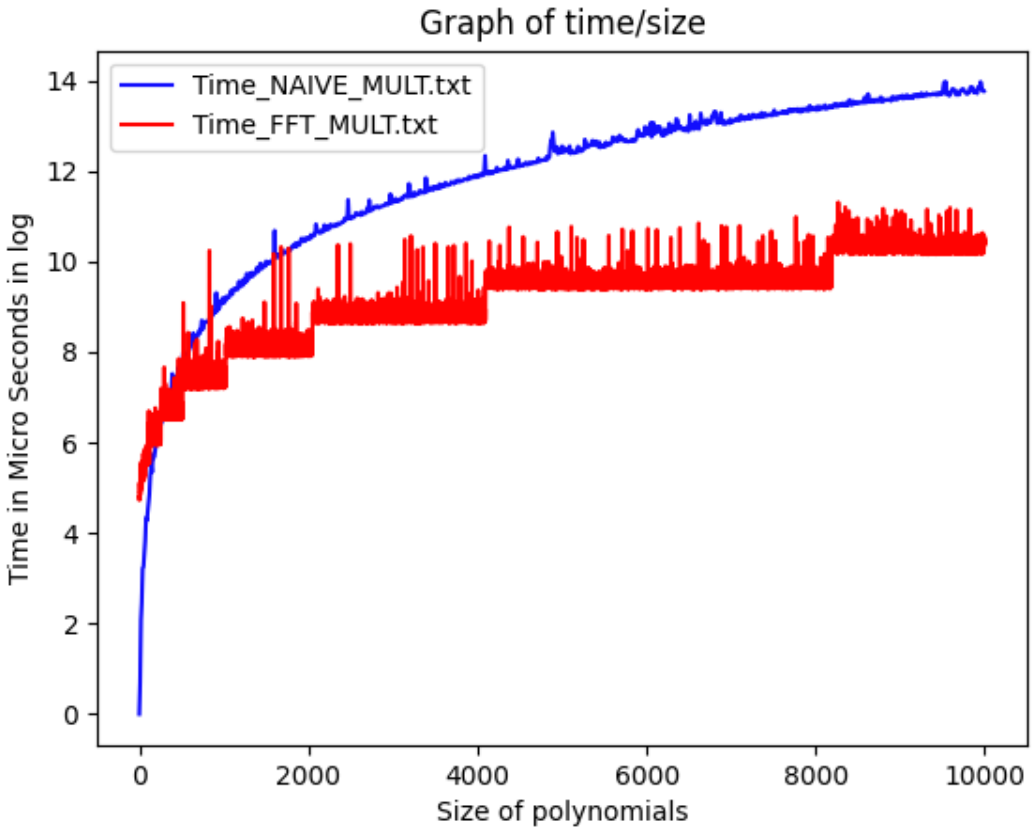


In contrast, the multiplication algorithm utilizing the Fast Fourier Transform (FFT) is significantly more efficient for large inputs. The FFT reduces the complexity of polynomial multiplication from O(n^2) to O(nlogn) through a "divide and conquer" approach. The corresponding graph depicts a much slower increase in computation time with the input size, making the FFT a preferred method for applications requiring fast and efficient polynomial multiplications.

3. Results and analysis



The initial graph (Linear Scale) depicts the progression of computation time relative to the polynomial degree. The curve associated with the FFT implementation seems to exhibit a linear-logarithmic trend, in line with the anticipated behavior for the FFT. The execution time shows a comparatively gradual increase with the input size. Additionally, the stepped pattern comes from our FFT algorithm extending the size of vectors that are not powers of 2 to the next power of 2. Conversely, the naive approach demonstrates a considerably swifter escalation in computation time as the polynomial degree increases, consistent with its quadratic algorithmic complexity O(n^2).



The second graph employs a logarithmic scale on both axes, which is an effective method for representing data that spans several orders of magnitude. This approach is particularly adept at elucidating the variations in growth rates of different algorithms. When observed on this scale, the Fast Fourier Transform (FFT) curve closely resembles a straight line, which serves as a graphical affirmation of its computational complexity, O(n log n). The logarithmic representation has the distinct advantage of converting polynomial growth patterns into linear forms, thereby simplifying the interpretation of complex relationships.

Another noteworthy observation is that the naive algorithm demonstrates faster performance for smaller polynomial sizes, specifically, everything below approximately 400 exhibits quicker execution with the naive algorithm. We can deduce that the naive algorithm is better for smaller polynomials.

4. Pros and Cons of each algorithm

	Naive algorithm	FFT-Based Algorithm
Pros	• No advanced knowledge required • Low setup cost • No initial overhead	• Reduced Complexity • Efficiency for Large Inputs • Versatility
Cons	• Quadratic Complexity • Inefficiency for Large Inputs	• Complex Implementation • Initial Overhead

5. Difficulties encountered

Managing memory resources posed a significant challenge for me as I encountered frequent memory loss and memory leaks, the origins of which were initially unclear. However, through the use of tools like Valgrind and strategically placed printf statements, I successfully identified and overcame this challenge.

Another challenging aspect was implementing the FFT algorithm with my custom complex number library that I constructed from the ground up. I encountered type errors at times, which proved to be intricate to resolve. Fortunately, with the aid of online documentation, I successfully navigated and resolved these challenges.

6. Possible improvement in my code

Improving the code could involve several strategies to enhance performance, readability, and maintainability. One avenue for improvement is optimizing the FFT algorithm implementation for better efficiency. This might include exploring more advanced FFT variants, optimizing memory usage, or parallelizing certain computations.

Exploring advanced compiler optimizations and profiling tools could uncover opportunities for performance enhancements. Continuous refinement and optimization can contribute to a more robust and efficient codebase.

Conclusion

In conclusion, this project has been a rewarding exploration of polynomial multiplication algorithms, highlighting significant differences between the naive algorithm and the Fast Fourier Transform (FFT)-based algorithm. The simplicity of the naive algorithm offers advantages for small polynomial sizes, while the reduced complexity of the FFT algorithm stands out notably for larger inputs. Challenges faced, such as memory resource management and FFT implementation, were successfully overcome, providing a profound understanding of algorithmic intricacies.