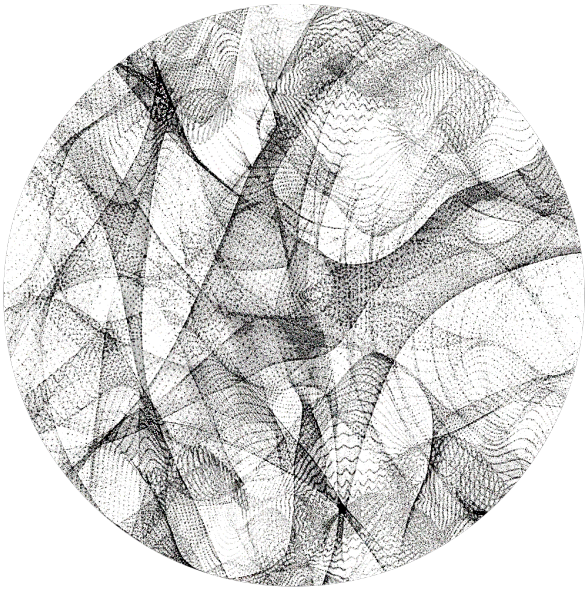
Group 19 Project

Implementation and comparative analysis of Naive and Fast Fourier Transform (FFT) based polynomial multiplication algorithms

Muzhou Wu
muzhou.wu@etu.sorbonne-universite.fr

Md Shadnan Azwad Khan
md_shadnan_azwad.khan@etu.sorbonne-universite.fr

Erekle Shatirishvili
erekle.shatirishvili@etu.sorbonne-universite.fr



Numerical and Symbolic Algorithms Modeling
(MODEL, MU4IN901)

Sorbonne Université

October 19, 2024

Contents

1 Introduction **3**

2 Implementation **5**
- 2.1 Complex Arithmetic . . . 5
- 2.2 FFT and Inverse FFT Implementation . . . 7
- 2.3 FFT-Based Polynomial Multiplication . . . 10
- 2.4 Naive Polynomial Multiplication . . . 11
- 2.5 Test Functions . . . 11
- 2.6 Execution . . . 12

3 Results and Analysis **18**
- 3.1 Verification . . . 18
  - 3.1.1 Complex Arithmetic . . . 18
  - 3.1.2 FFT and IFFT . . . 18
  - 3.1.3 Polynomial Multiplication . . . 19
- 3.2 Benchmarking . . . 20
- 3.3 Naive Polynomial Multiplication . . . 20
- 3.4 FFT-based Polynomial Multiplication . . . 20
- 3.5 Comparative Analysis . . . 21

4 Discussion and Conclusion **24**

References **25**

Introduction

Polynomials may be defined as expressions of the form:

$$a_0 + a_1x + a_2x^2 + ...a_{n-1}x^{n-1} + a_nx^n = \sum_{k=0}^{n} a_kx^k$$

where x is an indeterminate, and $a_0, .., a_n$ are constants or coefficients of the polynomial. The degree of a polynomial is the highest power of x with a non-zero coefficient. A polynomial can have real, integer, or complex coefficients, and can be univariate (one indeterminate) or multivariate (more than one indeterminate). In this report, we focus on univariate polynomials with a focus on their multiplication.

Polynomial multiplication is a fundamental operation in algebra with applications in (but not limited to) error correction codes, cryptography, and digital signal processing. The product of two polynomials $P(x), Q(x)$, is another polynomial $R(x)$ of the form

$$P(x) = \sum_{k=0}^{n} a_kx^k \quad Q(x) = \sum_{k=0}^{m} b_kx^k$$

$$R(x) = P(x)Q(x) = \sum_{k=0}^{n+m} c_kx^k \quad c_k = \sum_{i=0}^{k} p_iq_{k-i}$$

where n and m are the degrees of $P(x)$ and $Q(x)$ respectively, c_k are the coefficients in $R(x)$, and p_j an q_j are coefficients of x^j for some j-th term in $P(x)$ and $Q(x)$ respectively. This sum of terms representation of $R(x)$ arises from repeatedly applying the distributive, associative, and commutative laws.

The naive polynomial multiplication algorithm (computing each coefficient of a polynomial multiplication in a brute force fashion) has a time complexity of $O(N^2)$, making it impractical (or inefficient) for large polynomials. In comparison, the Fast Fourier Transform (FFT), leveraging the convolution property of the transform to achieve a significantly lower time complexity of $O(N \log N)$. Figure 1.1 illustrates this difference in time complexity for the two algorithms (N being the maximum of the degrees of the two polynomials being multiplied).

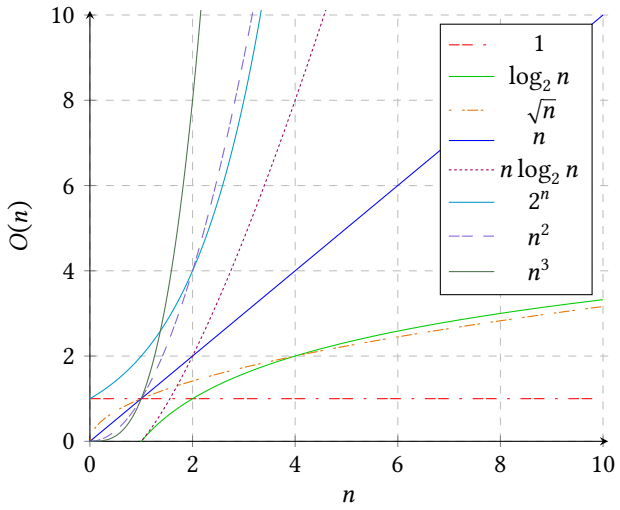


Figure 1.1: A comparison of common functions (number of operations $f(N)$ with respect to input size N) used for asymptotic analysis of algorithms

The FFT is an algorithm that computes the Discrete Fourier Transform (DFT). The DFT in turn is a sampled or discretized form of the Fourier Transform which converts a function into its frequency domain representation. Since complex numbers are often used to describe oscillatory behaviour [1] and to represent magnitude and phase, it is convenient to use complex valued functions to represent the frequency components of the original function.

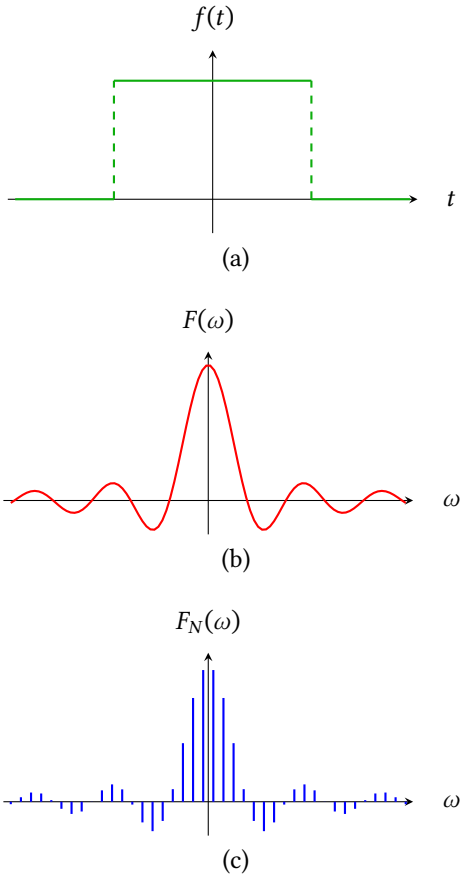


Figure 1.2: A basic representation of DFT: (a) a rectangular function (b) Fourier transform of the rectangular function (c) DFT of the rectangular function

The objective of this project is to develop a polynomial multiplication algorithm based on the Fast Fourier Transform (FFT) and evaluate its efficiency in comparison to the naive approach. Functions are also implemented for the arithmetic of complex numbers, to facilitate the FFT implementation. The report is divided into several sections. The implementation section explains the organization of our code, the main functions, and discusses the challenges encountered during the development phase. The results and analysis section outlines the experimental methodology and discusses the results obtained. The final section concludes our report with suggestions on possible directions for further work.

Implementation

The source code directory encompasses `functions.c` and `functions.h` files, containing the primary algorithmic code. Additionally, `testing.c` and `testing.h` files present functions to evaluate the efficiency of polynomial multiplication algorithms. Finally, `main.c` oversees algorithm execution, offering users options and interactivity through a basic command-line interface (CLI).

In the following sub-sections we start by examining the complex arithmetic functions, the FFT and inverse FFT functions, and the polynomial multiplication functions for FFT-based and naive approaches. Then we provide explanations for the test functions and the main execution of the algorithms.

2.1 Complex Arithmetic

Subsequent functions require complex number operations for their procedures. We can define a complex number as

$$a = a_{real} + i \cdot a_{imag}$$

where i or imaginary unit satisfies $i^2 = -1$ and a_{real} and a_{imag} are real numbers with $Re(a) = a_{real}$ being the real component and $Im(a) = a_{imag}$ being the imaginary component of a. Directly following from this, the `Complex` structure, which encapsulates real and imaginary components, is defined:

```c
typedef struct {
    double real;
    double imag;
} Complex;
```

The main operations or arithmetic for complex numbers $a = a_{real}+i\cdot a_{imag}$, $b = b_{real}+i\cdot b_{imag}$ and $c = c_{real}+i\cdot c_{imag}$ are

Addition of two complex numbers (`c_add`):

$$c = a + b = (a_{real} + i \cdot a_{imag}) + (b_{real} + i \cdot b_{imag}) = (a_{real} + b_{real}) + i \cdot (a_{imag} + b_{imag})$$

$$\therefore Re(c) = Re(a) + Re(b)$$

$$\therefore Im(c) = Im(a) + Im(b)$$

We can produce the code directly from this simplification:

```c
Complex c_add(Complex a, Complex b){
    return (Complex){a.real + b.real, a.imag + b.imag};
}
```

Subtraction of one complex number from another (`c_sub`):

$$c = a - b = (a_{real} + i \cdot a_{imag}) - (b_{real} + i \cdot b_{imag}) = (a_{real} - b_{real}) + i \cdot (a_{imag} - b_{imag})$$

$$\therefore Re(c) = Re(a) - Re(b)$$

$$\therefore Im(c) = Im(a) - Im(b)$$

We can produce the code directly from this simplification:

```c
Complex c_sub(Complex a, Complex b){
    return (Complex){a.real - b.real, a.imag - b.imag};
}
```

Multiplication of two complex numbers (`c_mul`):

$$\begin{aligned} c = a \cdot b &= (a_{real} + i \cdot a_{imag})(b_{real} + i \cdot b_{imag}) \\ &= a_{real} \cdot b_{real} + i \cdot a_{real} \cdot b_{imag} + i \cdot a_{imag} \cdot b_{real} + i^2 \cdot a_{imag} \cdot b_{imag} \\ &= a_{real} \cdot b_{real} + i \cdot a_{real} \cdot b_{imag} + i \cdot a_{imag} \cdot b_{real} - a_{imag} \cdot b_{imag} \\ &= (a_{real} \cdot b_{real} - a_{imag} \cdot b_{imag}) + i \cdot (a_{real} \cdot b_{imag} + a_{imag} \cdot b_{real}) \\ \therefore Re(c) &= Re(a) \cdot Re(b) - Im(a) \cdot Im(b) \\ \therefore Im(c) &= Re(a) \cdot Im(b) + Im(a) \cdot Re(b) \end{aligned}$$

We can produce the code directly from this simplification:

```
Complex c_mul(Complex a, Complex b){
    return (Complex){
    (a.real * b.real) - (a.imag * b.imag),
    (a.real * b.imag) + (a.imag * b.real)};
}
```

Division of one complex number by another (`c_div`):

$$\begin{aligned} c = \frac{a}{b} &= \frac{a_{real} + i \cdot a_{imag}}{b_{real} + i \cdot b_{imag}} = \frac{(a_{real} + i \cdot a_{imag})(b_{real} - i \cdot b_{imag})}{(b_{real} + i \cdot b_{imag})(b_{real} - i \cdot b_{imag})} \\ &= \frac{(a_{real} \cdot b_{real} + a_{imag} \cdot b_{imag}) + i \cdot (a_{real} \cdot b_{imag} - a_{imag} \cdot b_{real})}{b_{real}^2 + b_{imag}^2} \\ \therefore Re(c) &= \frac{Re(a) \cdot Re(b) + Im(a) \cdot Im(b)}{Re(b)^2 + Im(b)^2} \\ \therefore Im(c) &= \frac{Im(a) \cdot Re(b) - Re(a) \cdot Im(b)}{Re(b)^2 + Im(b)^2} \end{aligned}$$

It is easy to see that the denominator in this case has to satisfy the condition that $Re(b)^2 + Im(b)^2 \neq 0$ to produced a defined output. We can produce the code directly from the simplified form (and in keeping with the condition):

```
Complex c_div(Complex a, Complex b){
    double denominator = (b.real * b.real) + (b.imag * b.imag);
    if (denominator == 0){
        printf("Error: Division by zero\n");
        return (Complex){0, 0};
    }
    return (Complex){
    ((a.real * b.real) + (a.imag * b.imag)) / denominator,
    ((a.imag * b.real) - (a.real * b.imag)) / denominator};
}
```

Exponential of a complex number (`c_exp`):

$$c = e^a = e^{a_{real} + i \cdot a_{imag}} = e^a_{real} \cdot e^{i \cdot a_{imag}} = e^{a_{real}} \cdot \cos(a_{imag}) + i \cdot e^{a_{real}} \cdot \cos(a_{imag})$$

$$\therefore Re(c) = \exp(Re(a)) \cdot \cos(Im(a))$$

$$\therefore Im(c) = \exp(Re(a)) \cdot \sin(Im(a))$$

We can produce the code directly from this simplification:

```
Complex c_exp(Complex c){
    double real_part = exp(c.real);
    return (Complex){
    real_part * cos(c.imag),
    real_part * sin(c.imag)};
}
```

Conjugation of a complex number (c_conj):

$$c = a^* = (a_{real} + i \cdot a_{imag})^* = a_{real} - i \cdot a_{imag} == a_{real} + i \cdot (-a_{imag})$$

$$\therefore Re(c) = Re(a)$$

$$\therefore Im(c) = -Im(a)$$

We can produce the code directly from this simplification:

```
Complex c_conj(Complex c){
    return (Complex){
    c.real,
    -c.imag};
}
```

2.2 FFT and Inverse FFT Implementation

The Fast Fourier Transform (FFT) is an algorithm used to compute the Discrete Fourier Transform (DFT) and its inverse. The DFT maps a sequence of N complex numbers $x_0, x_1, ..., x_{N-1}$ to a sequence of N complex numbers $X_0, X_1, ..., X_{N-1}$ according to the formula:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{\frac{-2i\pi kn}{N}}, \quad 0 \leq k < N$$

We can express this as a matrix multiplication [2]:

$$\begin{bmatrix} X[0] \\ X[1] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} W_N^0 & W_N^0 & \cdots & W_N^0 \\ W_N^0 & W_N^1 & \cdots & W_N^{N-1} \\ \vdots & \vdots & & \vdots \\ W_N^0 & W_N^{N-1} & \cdots & W_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[N-1] \end{bmatrix}$$

where $W_N = e^{\frac{-2i\pi}{N}}$ is the primitive N-th root of unity. The FFT algorithm computes this matrix multiplication more efficiently by exploiting the symmetry and periodicity properties of the W_N factors. The central insight of the FFT is that to compute the DFT of a sequence of N points (where N is assumed to be a power of 2), one can recursively compute the DFT of two sequences of $N/2$ points, which reduces the computational time drastically.

The most well-known FFT algorithm is the Cooley-Tukey algorithm specifically the Radix-2 decimation-in-time (DIT) FFT variation [3]. This algorithm recursively breaks down a DFT of any composite size $N = N_1 N_2$ into many smaller DFTs of sizes N_1 and N_2, along with multiplications by complex roots of unity traditionally known as twiddle factors [4, 5].

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{\frac{-2i\pi k(2m)}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{\frac{-2i\pi k(2m+1)}{N}} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{\frac{-2i\pi k(m)}{N/2}} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{\frac{-2i\pi k(m)}{N/2}} \cdot e^{\frac{-2i\pi k}{N}} \\ \therefore X_k &= X_k^{even} + e^{\frac{-2i\pi k}{N}} \cdot X_k^{odd} \end{aligned}$$

This approach involves recursively splitting an array into even and odd parts. By applying the FFT to each part and repeating the process until reaching the base case of a single element, we can efficiently compute the FFT for the entire original array. This recursive strategy forms the basis of the algorithm.

Algorithm 1 Radix-2 decimation-in-time (DIT) FFT

Input: An array $x = x_0, x_1, x_2, \dots, x_{N-1}$ such that N is a power of 2
Output: $X = X_0, X_1, X_2 \dots, X_{N-1}$ is the FFT of $x = x_0, x_1, x_2, \dots, x_{N-1}$

if $N = 1$ **then**
  $X_0 \leftarrow x_0$
else
  Split X into X^{even} and X^{odd}
  Call function $FFT(X^{even})$ and $FFT(X^{odd})$
  for $k = 0$ to $\frac{N}{2} - 1$ **do**
    $t \leftarrow \exp\left(\frac{-2 \cdot \pi \cdot k}{n}\right) \cdot X_{odd}[k]$
    $X_k \leftarrow X_k^{even} + t$
    $X_{k+\frac{N}{2}} \leftarrow X_k^{even} - t$
  end for
end if

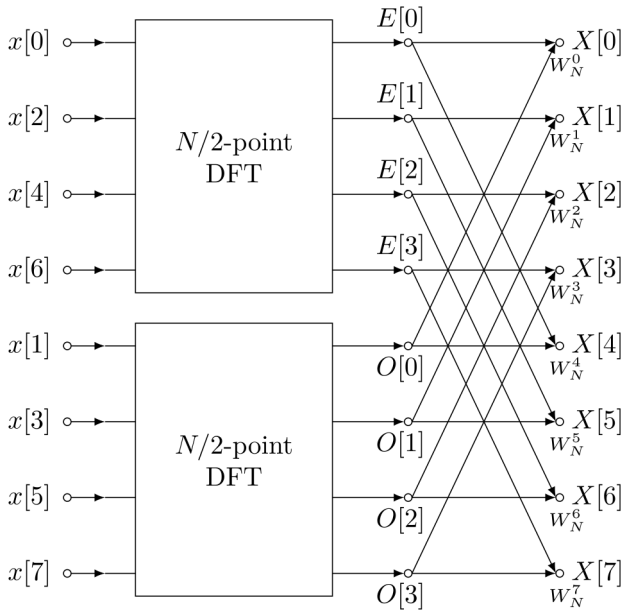


Figure 2.1: Butterfly Operations: Dividing and Conquering with the FFT[6]

It follows directly that the code implementation be of the form:

```
void fft(Complex *x, int n){
  if (n < 2) return;
  int half_n = n >> 1;
  Complex even[half_n], odd[half_n];
  for (int i = 0; i < half_n; ++i){
    even[i] = x[i * 2];
    odd[i] = x[i * 2 + 1];
  }
  fft(even, half_n);
  fft(odd, half_n);
  for (int k = 0; k < half_n; ++k){
    Complex t = c_mul(c_exp((Complex){0, -2 * PI * k / n}),  odd[k]);
    x[k] = c_add(even[k], t);
    x[k + half_n] = c_sub(even[k], t);
  }
}
```

It is known that utilizing the properties of the DFT, we can express the inverse DFT (IDFT) in terms of the DFT itself.

The IDFT is given by:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{2\pi i k n / N}, \quad 0 \leq n < N$$

where N as usual is assumed to be a power of 2.

It can be said that the IDFT is the DFT with conjugation applied once before and once after a DFT followed by a division by N [7]. That is to say

$$F^{-1}(x) = \frac{F(x^*)^*}{N}$$

The inverse FFT (IFFT) can be computed using this same principle. The algorithm can be expressed as such:

Algorithm 2 Inverse FFT (IFFT) using FFT results

Input: An array $X = X_0, X_1, X_2, \ldots, X_{N-1}$ such that N is a power of 2
Output: $x = x_0, x_1, x_2 \ldots, x_{N-1}$ is the IFFT of $X = X_0, X_1, X_2, \ldots, X_{N-1}$

1: **for** $k = 0$ to $N - 1$ **do**
2: $\quad X_k \leftarrow$ Conjugation of X_k
3: **end for**
4: Call function $FFT(X)$
5: **for** $k = 0$ to $N - 1$ **do**
6: $\quad X_k \leftarrow$ Conjugation of X_k term
7: $\quad X_k \leftarrow \frac{X_k}{N}$
8: **end for**

The code is thus:

```
void ifft(Complex *x, int n){
  for (int i = 0; i < n; ++i){
    x[i] = c_conj(x[i]);
  }
  fft(x, n);
  for (int i = 0; i < n; ++i){
    x[i] = c_conj(x[i]);
    x[i] = c_div(x[i], (Complex){n, 0});
  }
}
```

For both the FFT and IFFT functions we need N to be a power of 2, to satisfy this assumption arrays are zero-padded to the next power of 2 from the given user input for size of array. This zero-padding is done during the call to FFT or IFFT as we will see during the main execution function.

The next power of 2 is computed using the following function:

```
unsigned int get_next_pow2(int n){
  if (n && !(n & (n - 1)))
    return n;
  unsigned int count = 0;
  while (n != 0){
    n >>= 1;
    count++;
  }
  return 1 << count;
}
```

2.3 FFT-Based Polynomial Multiplication

An interesting feature of the Fourier Transform following from the convolution theorem may be used for polynomial multiplication [8, 9]. Let P and Q be two functions with convolution $P * Q$. Let F be the Fourier Transform operator. According to the convolution theorem

$$F(P * Q) = F(P) \cdot F(Q)$$

$$F(P \cdot Q) = F(P) * F(Q)$$

By applying the Inverse Fourier Transform F^{-1}, we can alternatively write

$$P * Q = F^{-1}F(P) \cdot F(Q))$$

$$P \cdot Q = F^{-1}(F(P) * F(Q)$$

So a convolution in one domain corresponds with the point-wise multiplication in the other domain. This property is maintained in the case of polynomials $P(x)$ and $Q(x)$ of some variable x.

Subsequently we may write the FFT-based polynomial multiplication algorithm as

Algorithm 3 FFT-based polynomial multiplication

Input: Arrays $P = P_0, P_1, P_2, \ldots, P_{N-1}$ and $Q = Q_0, Q_1, Q_2, \ldots, Q_{N-1}$ such that N is a power of 2

Output: $R = R_0, R_1, R_2, \ldots, R_{N-1}$ is the product of polynomials P and Q

1: Call function $FFT(P)$ and $FFT(Q)$
2: **for** $k = 0$ to $N - 1$ **do**
3: $\quad R_k \leftarrow P_k \cdot Q_k$
4: **end for**
5: Call function $IFFT(R)$

When we perform FFT-based polynomial multiplication we start by taking the next power of 2 of the maximum of the two degrees of the two input polynomials (defined in the function as size). The remaining coefficients of the polynomial are zero-padded, and the standard FFT-based polynomial multiplication is performed.

It is important to note that the FFT employs double floating-point values for complex numbers, while our input polynomials consist of integer coefficients as real components. After the FFT and IFFT operations to produce the polynomial product, the output has to have integer coefficient since the product of two polynomials with integer coefficients is another polynomial with integer coefficient.

However, due to the implementation of FFT and IFFT using double floating-point values, minor errors may be introduced. Since casting a double into an integer in C truncates the decimal part, potential issues arise. To mitigate this, we round to the nearest integer. This rounding maintains the integrity of the overall operation, as floating-point errors are negligible, and the result is guaranteed to be integers.

We can write the code of this algorithm as such:

```
int *fft_poly_mul(int *poly1, int m, int *poly2, int n, int size){

  int fft_size = get_next_pow2(2 * size);
  int *result = malloc((m + n - 1) * sizeof(int));
  Complex fft_poly1[fft_size];
  Complex fft_poly2[fft_size];
  Complex fft_result[fft_size];

  for (int i = 0; i < fft_size; ++i){
    fft_poly1[i] = (i < m) ? (Complex){poly1[i], 0} : (Complex){0, 0};
    fft_poly2[i] = (i < n) ? (Complex){poly2[i], 0} : (Complex){0, 0};
  }

  fft(fft_poly1, fft_size);
```

```
  fft(fft_poly2, fft_size);

  for (int i = 0; i < fft_size; ++i){
    fft_result[i] = c_mul(fft_poly1[i], fft_poly2[i]);
  }

  ifft(fft_result, fft_size);

  for (int i = 0; i < (m+n)-1; ++i)
    result[i] = (int)round(fft_result[i].real);
  return result;
}
```

2.4 Naive Polynomial Multiplication

We have previously expressed the product R(x) of two polynomials P(x) and Q(x), which can be written as

$$R(x) = P(x)Q(x) = \sum_{k=0}^{n+m} \left(\sum_{i=0}^{k} p_i q_{k-i} \right) x^k$$

The code can be a direct implementation of this formula:

```
int *naive_poly_mul(int poly1[], int m, int poly2[], int n){

  int *result = calloc((m + n - 1), sizeof(int));

  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        result[i + j] += poly1[i] * poly2[j];
  return result;
}
```

2.5 Test Functions

The following function is used to test polynomial multiplication:

```
void test_poly_mul(int size, bool printornot){

    // Define the size of the polynomials and initialize them
  int p1[size], p2[size];

  initialize_polynomial(p1, size);
  initialize_polynomial(p2, size);

  // Declare clock variables before measurements
  clock_t start_naive, end_naive, start_fft, end_fft;

  // Measure the time taken by the naive multiplication
  start_naive = clock();
  int *result_naive = naive_poly_mul(p1, size, p2, size);
  end_naive = clock();
  double time_naive = ((double)(end_naive - start_naive)) / CLOCKS_PER_SEC;

  // Measure the time taken by the FFT-based multiplication
  start_fft = clock();
  int *result_fft = fft_poly_mul(p1, size, p2, size, size);
  end_fft = clock();
  double time_fft = ((double)(end_fft - start_fft)) / CLOCKS_PER_SEC;

  if (printornot){
```

```
        printf("Polynomial 1: ");
        poly_print(p1, size);

        printf("Polynomial 2: ");
        poly_print(p2, size);

        // Naive Polynomial Multiplication
        printf("\n%s\n", "Result of Naive Polynomial Multiplication: ");

        poly_print(result_naive, (2 * size) - 1);

        // FFT-based Polynomial Multiplication
        printf("\n%s\n", "Result of FFT-based Polynomial Multiplication: ");
        poly_print(result_fft, (2 * size) - 1);
    }

  printf("\nTime taken by naive multiplication: %.17g seconds\n", time_naive);
  printf("Time taken by FFT-based multiplication: %.17g seconds\n", time_fft);
}
```

The two polynomials are initialized to a *size* corresponding to the degree of the polynomials being tested. The naive and FFT-based polynomial functions are used to find the product of the two polynomials The clock variable is set and measured after each algorithm is run to find the time taken by each respective algorithm. If the Boolean to print is set to true we print the results of the multiplication. In any case, we print the time taken by each algorithm.

The following function initializes polynomials for testing:

```
void initialize_polynomial(int *polynomial, int size){
    for (int i = 0; i < size; i++){
        polynomial[i] = i + 1;
    }
}
```

2.6 Execution

User input is used to run the functions we have implemented. The main execution function runs through each section (Complex Arithmetic, FFT and IFFT, Polynomial Multiplication, Testing, and Benchmarking):

```
int main(){
    char choice;

    print_section_header("Complex Arithmetic:");

    printf("\n%s", "Do you wish to perform Complex Arithmetic? (y/n): ");
    scanf(" %c", &choice);

    while (choice == 'y' || choice == 'Y'){
        perform_complex_arithmetic();
        printf("\n%s", "Do you wish to continue doing Complex Arithmetic? (y/n):
            ↪  ");
        scanf(" %c", &choice);
    }

    print_section_header("FFT and IFFT:");

    printf("\n%s", "Do you wish to perform FFT and IFFT? (y/n): ");
    scanf(" %c", &choice);

    while (choice == 'y' || choice == 'Y'){
        perform_fft_and_ifft();
        printf("\n%s", "Do you wish to continue doing FFT and IFFT? (y/n): ");
```

```
        scanf(" %c", &choice);
    }

    print_section_header("Polynomial Multiplication:");

    printf("\n%s", "Do you wish to perform Polynomial Multiplication? (y/n): ");
    scanf(" %c", &choice);

    while (choice == 'y' || choice == 'Y'){
        int degree1;
        int degree2;

        printf("Enter the degree of Polynomial 1 to be generated: ");
        scanf("%d", &degree1);

        printf("Enter the degree of Polynomial 2 to be generated: ");
        scanf("%d", &degree2);

        printf("%s\n", "Using random integers as polynomial coefficients...");

        int *poly1 = (int *)malloc((degree1 + 1) * sizeof(int));
        int *poly2 = (int *)malloc((degree2 + 1) * sizeof(int));

        for (int i = 0; i < degree1 + 1; ++i){
            poly1[i] = rand() % 201 - 100;
        }

        for (int i = 0; i < degree2 + 1; ++i){
            poly2[i] = rand() % 201 - 100;
        }

        perform_poly_multiplication("Naive", poly1, degree1 + 1, poly2, degree2
            ↪ + 1);
        perform_poly_multiplication("FFT", poly1, degree1 + 1, poly2, degree2 +
            ↪ 1);

        free(poly1);
        free(poly2);

        printf("\n%s", "Do you wish to continue doing Polynomial Multiplication?
            ↪  (y/n): ");
        scanf(" %c", &choice);
    }

    print_section_header("Testing Polynomial Multiplication:");

    printf("\n%s", "Do you wish to compare Naive and FFT Polynomial
        ↪ Multiplication? (y/n): ");
    scanf(" %c", &choice);

    int degree;
    char toprintornot;

    while (choice == 'y' || choice == 'Y'){
        printf("Enter the degree of polynomials to be generated: ");
        scanf("%d", &degree);
        printf("Do you wish to print the Polynomials and Result? (y/n): ");
        scanf(" %c", &toprintornot);
        bool printornot = (toprintornot == 'y' || toprintornot == 'Y');

        perform_testing(degree, printornot);

        printf("\n%s", "Do you wish to continue Testing? (y/n): ");
```

```
        scanf(" %c", &choice);
    }


    print_section_header("Benchmarking Polynomial Multiplication:");

    printf("\n%s", "Do you wish to benchmark Naive and FFT Polynomial
        ↪ Multiplication? (y/n): ");
    scanf(" %c", &choice);

    int n;

    while (choice == 'y' || choice == 'Y'){
        printf("\n");
        printf("Enter n for the upper bound 2^n size of polynomials to be
            ↪ benchmarked: ");
        scanf("%d", &n);

        bool printornot = false;

        for (int i = 1; i <= n; i++){
            if(i==1){
                printf("\n");
                printf("Best case s=2^n: ");
                perform_testing(pow(2, i), printornot);
                printf("\n");
                printf("Average case s= 1.5*2^n = Worst case s=1+2^n:: ");
                perform_testing(1.5*(pow(2, i)), printornot);
            } else if (i==n) {
                printf("\n");
                printf("Best case s=2^n: ");
                perform_testing(pow(2, i), printornot);
            } else {
                printf("\n");
                printf("Best case s=2^n: ");
                perform_testing(pow(2, i), printornot);
                printf("\n");
                printf("Worst case s=1+2^n: ");
                perform_testing(1+pow(2, i), printornot);
                printf("\n");
                printf("Average case s= 1.5*2^n = Worst case s=1+2^n:: ");
                perform_testing(1.5*(pow(2, i)), printornot);
            }
        }

        printf("\n%s", "Do you wish to continue Benchmarking? (y/n): ");
        scanf(" %c", &choice);
    }

    printf("\n
        ↪ ------------------------------------------------------------------\n
        ↪ ");
    return 0;
}
```

A simple function to print section headers:

```
void print_section_header(const char *section_name){
    printf("\n
        ↪ ------------------------------------------------------------------\n
        ↪ ");
    printf("\n%s\n", section_name);
}
```

Complex number operations or arithmetic are verified using the following function:

```
void perform_complex_arithmetic(){
    Complex num1;
    Complex num2;

    char choicecomplex;

    printf("Do you want to use random numbers? (y/n): ");
    scanf(" %c", &choicecomplex);

    printf("\n");

    if (choicecomplex == 'y' || choicecomplex == 'Y'){
        // Generate random complex numbers
        // Seed the random number generator with the current time
        srand(time(NULL));

        // Generate random values for the real and imaginary parts
        num1.real = ((double)rand() / RAND_MAX) * 200 - 100; // Random real part
        num1.imag = ((double)rand() / RAND_MAX) * 200 - 100; // Random imaginary
            ↪  part
        num2.real = ((double)rand() / RAND_MAX) * 200 - 100; // Random real part
        num2.imag = ((double)rand() / RAND_MAX) * 200 - 100; // Random imaginary
            ↪  part
    }else{
        // Taking user input for real part
        printf("Enter the real part for Complex Number 1: ");
        scanf("%lf", &num1.real);

        // Taking user input for imaginary part
        printf("Enter the imaginary part for Complex Number 1: ");
        scanf("%lf", &num1.imag);

        // Taking user input for real part
        printf("Enter the real part for Complex Number 2: ");
        scanf("%lf", &num2.real);

        // Taking user input for imaginary part
        printf("Enter the imaginary part for Complex Number 2: ");
        scanf("%lf", &num2.imag);

        printf("\n");
    }

    Complex sum = c_add(num1, num2);
    Complex difference = c_sub(num1, num2);
    Complex product = c_mul(num1, num2);
    Complex quotient = c_div(num1, num2);
    Complex exp_num1 = c_exp(num1);
    Complex exp_num2 = c_exp(num2);
    Complex conj_num1 = c_conj(num1);
    Complex conj_num2 = c_conj(num2);

    printf("Complex Number 1: ");
    c_print(num1);

    printf("Complex Number 2: ");
    c_print(num2);

    printf("Sum: ");
    c_print(sum);

    printf("Difference: ");
```

```
    c_print(difference);

    printf("Product: ");
    c_print(product);

    printf("Quotient: ");
    c_print(quotient);

    printf("Exponential of Complex Number 1: ");
    c_print(exp_num1);

    printf("Exponential of Complex Number 2: ");
    c_print(exp_num2);

    printf("Conjugate of Complex Number 1: ");
    c_print(conj_num1);

    printf("Conjugate of Complex Number 2: ");
    c_print(conj_num2);
}
```

For FFT and IFFT we first take user input for size of the arrays. The arrays are then initialized to the next power of 2 of the given size. FFT and IFFT results are verified using the following function:

```
void perform_fft_and_ifft(){
    int size;
    printf("Enter the size of the array to be generated for FFT and IFFT: ");
    scanf("%d", &size);
    printf("%s\n", "Using random double precision floting point numbers for the
        ↪ array elements...");

    int fftSize = get_next_pow2(size);
    Complex input[fftSize];
    for (int i = 0; i < size; ++i)
      input[i] = (Complex){((double)rand() / RAND_MAX) * 200 - 100, ((double)
          ↪ rand() / RAND_MAX) * 200 - 100};

    printf("\nInput Vector:\n");
    c_vec_print(input, size);
    fft(input, fftSize);
    printf("\nFFT:\n");
    c_vec_print(input, fftSize);
    ifft(input, fftSize);
    printf("\nIFFT:\n");
    c_vec_print(input, fftSize);
}
```

Polynomial multiplication for naive or FFT-based approaches are verified using the following function:

```
void perform_poly_multiplication(const char *method_name, int *poly1, int m, int
    ↪  *poly2, int n){
    int result_size = m + n - 1;
    int input_size_max = get_next_pow2((m >= n) ? m : n);

    printf("\n%s Polynomial Multiplication:\n", method_name);

    printf("Polynomial 1: ");
    poly_print(poly1, m);

    printf("Polynomial 2: ");
    poly_print(poly2, n);

```

```
    printf("Result of multiplication: ");

    if (strcmp(method_name, "Naive") == 0){
      int *naive_result = naive_poly_mul(poly1, m, poly2, n);
      poly_print(naive_result, result_size);
      free(naive_result);
    } else if (strcmp(method_name, "FFT") == 0){
      int *fft_result = fft_poly_mul(poly1, m, poly2, n, input_size_max);
      poly_print(fft_result, result_size);
      free(fft_result);
    }
}
```

The following function is used to perform testing:

```
void perform_testing(int size, bool printPolynomials){
    printf("\nSize of polynomial: %d\n", size);
    test_poly_mul(size, printPolynomials);
}
```

The following function prints complex numbers:

```
void c_print(Complex c){
  // Check the sign of the imaginary part
  if (c.imag >= 0)
    printf("%.15lf + %.15lfi\n", c.real, c.imag); // For positive imaginary part
        ↪ , print a+bi
  else
    printf("%.15lf - %.15lfi\n", c.real, -c.imag); // For negative imaginary
        ↪ part, print a-bi
}
```

The following function prints polynomials:

```
// Function to print a polynomial with lower powers to higher powers
void poly_print(int poly[], int n){
  for (int i = 0; i < n; i++){
    if(poly[i]==0) printf("0");
    else{
      if (i == 0){
        printf("%s", poly[i] < 0 ? " - " : " ");
        printf("%d", abs(poly[i]));
      }
      else
        printf("%dx^%d", abs(poly[i]), i);
      if (i < n - 1)
        printf(" %c ",(poly[i+1] < 0) ? '-' : '+');
    }
  }
  printf("\n");
}
```

The following function prints complex vectors:

```
void c_vec_print(Complex *vector, int size){
  for (int i = 0; i < size; ++i)
    c_print(vector[i]);
}
```

Results and Analysis

We verify that our code produces the desired outputs for each function and then benchmark.

3.1 Verification

The outputs of the functions given below have been reproduced many times and with different parameters, we present some examples of outputs as a demonstration.

3.1.1 Complex Arithmetic

The outputs produced for the complex number operations match expectations.

```
Complex Arithmetic:

Do you wish to perform Complex Arithmetic? (y/n): y
Do you want to use random numbers? (y/n): y

Complex Number 1: -81.087297844275497 + 66.616205483030626i
Complex Number 2: 32.090909468052388 + 14.348675550124000i
Sum: -48.996388376223109 + 80.964881033154626i
Difference: -113.178207312327885 + 52.267529932906626i
Product: -3558.019452986042779 + 974.279291257332488i
Quotient: -1.332278233226852 + 2.671555123093191i
Exponential of Complex Number 1: -0.000000000000000
- 0.000000000000000i
Exponential of Complex Number 2: -18154739265539.867187500000000
+ 84550721973289.000000000000000i
Conjugate of Complex Number 1: -81.087297844275497 - 66.616205483030626i
Conjugate of Complex Number 2: 32.090909468052388 - 14.348675550124000i
```

3.1.2 FFT and IFFT

The FFT indeed performs the proper operation, as does the IFFT. The only caveat is the rounding errors due to using double-precision floating-point numbers.

```
FFT and IFFT:

Do you wish to perform FFT and IFFT? (y/n): y
Enter the size of the array to be generated for FFT and IFFT: 10
Using random double precision floting point numbers for the array elements...

Input Vector:
-2.305473528013323 + 15.247022321097091i
13.352223631624227 - 87.278297165072658i
-88.713631680567573 - 39.545028349172803i
-2.695680550623536 - 69.800929431710827i
-72.928822866142184 - 70.604771175703391i
44.547746118413158 - 22.522326429617749i
-21.240197644215158 + 67.013938989031089i
-3.851739179273949 - 51.282069902532768i
-54.937162043031847 - 57.280825431123759i
-3.501947924262822 - 39.579353173998811i
```

```
3.633123684503659 - 52.730405588043112i
-62.910683622076498 - 90.3443706176915951

FFT:
-251.552245603665881 - 498.707415954539329i
-26.124810373472215 + 105.739191573964021i
-275.225031458733383 + 163.814922342737248i
132.985773907886426 + 107.641579617140863i
38.196640386337670 - 211.233204515293778i
-108.541031067803061 + 169.695772554147936i
5.089119424141529 + 29.730875228932398i
-48.584962318124354 - 240.3365285338532541
-221.432082551267001 + 222.907277486709518i
3.037747452149318 + 269.685277047015290i
-11.981339803566556 + 21.007634492313684i
327.465964478604178 + 128.317761679499711i
-85.898145980154254 + 36.479045840203327i
59.735763346386499 + 36.706441298324876i
344.862001018546550 - 100.2695598012764441
81.079062694521411 + 2.773286781527318i

IFFT:
-2.305473528013323 + 15.2470223210970881
13.352223631624222 - 87.2782971650726581
-88.713631680567573 - 39.5450283491728031
-2.695680550623543 - 69.8009294317108271
-72.928822866142184 - 70.6047711757034051
44.547746118413158 - 22.5223264296177561
-21.240197644215154 + 67.0139389890310891
-3.851739179273957 - 51.2820699025327761
-54.937162043031847 - 57.2808254311237591
-3.501947924262824 - 39.5793531739988111
3.633123684503659 - 52.730405588043119i
-62.910683622076498 - 90.3443706176915811
-0.000000000000007 - 0.000000000000007i
0.000000000000004 - 0.000000000000007i
-0.000000000000007 + 0.000000000000000i
0.000000000000001 - 0.000000000000007i
```

3.1.3 Polynomial Multiplication

Naive and FFT produce valid outputs for the polynomial multiplication.

```
Polynomial Multiplication:

Do you wish to perform Polynomial Multiplication? (y/n): y
Enter the degree of Polynomial 1 to be generated: 5
Enter the degree of Polynomial 2 to be generated: 8
Using random integers as polynomial coefficients...

Naive Polynomial Multiplication:
Polynomial 1:  - 56 - 45x^1 - 88x^2 + 95x^3 + 86x^4 - 65x^5
Polynomial 2:  - 28 + 35x^1 + 70x^2 - 69x^3 + 86x^4 + 9x^5 - 58x^6 + 14x^7 + 26x^8
Result of multiplication:  1568 - 700x^1 - 3031x^2 - 5026x^3 - 6954x^4 + 13178x^5
- 7535x^6 - 1280x^7 + 15754x^8 - 12728x^9 - 6531x^10 + 7444x^11 + 1326x^12
- 1690x^13

FFT Polynomial Multiplication:
Polynomial 1:  - 56 - 45x^1 - 88x^2 + 95x^3 + 86x^4 - 65x^5
Polynomial 2:  - 28 + 35x^1 + 70x^2 - 69x^3 + 86x^4 + 9x^5 - 58x^6 + 14x^7 + 26x^8
Result of multiplication:  1568 - 700x^1 - 3031x^2 - 5026x^3 - 6954x^4 + 13178x^5
- 7535x^6 - 1280x^7 + 15754x^8 - 12728x^9 - 6531x^10 + 7444x^11 + 1326x^12
- 1690x^13
```

3.2 Benchmarking

To properly conduct the experiment, we run the algorithms 5 times and average the values on 2 computers that have different configurations (10 times overall). We then analyse and compare the time complexity graph characteristics. We utilize computers with different builds to inspect whether that is a factor on the graph characteristics.

The hardware and software specifications of the systems used for the testing are as follows:

Machine	**Computer 1**	**Computer 2**
Processor	AMD Ryzen 7 6800HS, 3.2 Ghz, 8 cores CPU, 16 logical processors	Apple M1 chip, 3.2 GHz, 8 cores CPU (4 performance cores, 4 efficiency cores)
Memory	16GB	16GB
Operating System	Debian GNU/Linux 12.2.0-14 (bookworm)	ARM64 Apple Darwin 23.2.0
Compiler	GCC version 12.2.0	Apple clang version 15.0.0

Table 3.1: Technical specifications of the test systems

Testing of the polynomial multiplication is done through time complexity analysis. Since we are comparing the results of the FFT-based approach with the naive approach, we are interested in checking how fast polynomial multiplication is performed with regard to the input size (corresponding to the degrees of the polynomials being multiplied).

For FFT-based polynomial multiplication we are especially interested how it scales with input sizes of powers of 2. In fact, the best case scenario for the FFT-based approach is when input size s can be expressed as 2^n for some $n \in \mathbb{Z}^+$ as no zero-padding needs to be done.

Since we take the next power of 2, the worst case is therefore when the input size s is only larger by 1 than 2^n i.e. $s = 2^n + 1$ as it will be zero-padded to the next power of 2 i.e. 2^{n+1}. In the worst case for large input sizes, almost half the array will be zero padded. The average case is halfway between the best case and the worst case, therefore $s = (2^n + 2^{n+1})/2 = 2^{n-1} + 2^n = 2^n(1 + 2^{-1}) = 1.5 \cdot 2^n$.

We limit our study to an upper bound of $2^{15} = 32768$ in all three cases. For the best case, the upper bound would therefore be for $n = 15$. For the worst case it would be $n = \log_2(32768) - 1 = 15 - 1 = 14$. For the average case it would similarly be $n = 14$. For $n = 14$, in the worst case and average case, the array of size s will be zero padded to size 2^{15}. For $n = 1$, the worst case and the average case is the same $1.5 \cdot 2^1 = 2^1 + 1 = 3$, so we only consider one instance.

Our testing is simple, we run both the naive and FFT-based polynomial multiplication algorithms for input sizes given in Table 3.2 and record the time taken until completion.

3.3 Naive Polynomial Multiplication

For the naive approach the computers produced the results as shown in Figure 3.1 (a).

The two computers produce a graph with similar outputs, although different in scale they are both comparable to the graph of $1.5 \cdot 10^{-9} \cdot s^2$. Therefore we can say that they have a time complexity that can be characterized as $O(s^2)$.

3.4 FFT-based Polynomial Multiplication

For the FFT-based approach the computers produced the graph in Figure 3.1 (b) . As predicted, the zero padding means that each average case and worst case has a time comparable to the next power of 2, or the best case for the next n value for size $s = 2^n$.

n	Best case ($s = 2^n$)	Worst case ($s = 2^n + 1$)	Average case ($s = 1.5 \cdot 2^n$)
1	2	3	-
2	4	5	6
3	8	9	12
4	16	17	24
5	32	33	48
6	64	65	96
7	128	129	192
8	256	257	384
9	512	513	768
10	1024	1025	1536
11	2048	2049	3072
12	4096	4097	6144
13	8192	8193	12288
14	16384	16385	24576
15	32768	-	-

Table 3.2: Best, Worst, and Average case input sizes for FFT-based polynomial multiplication

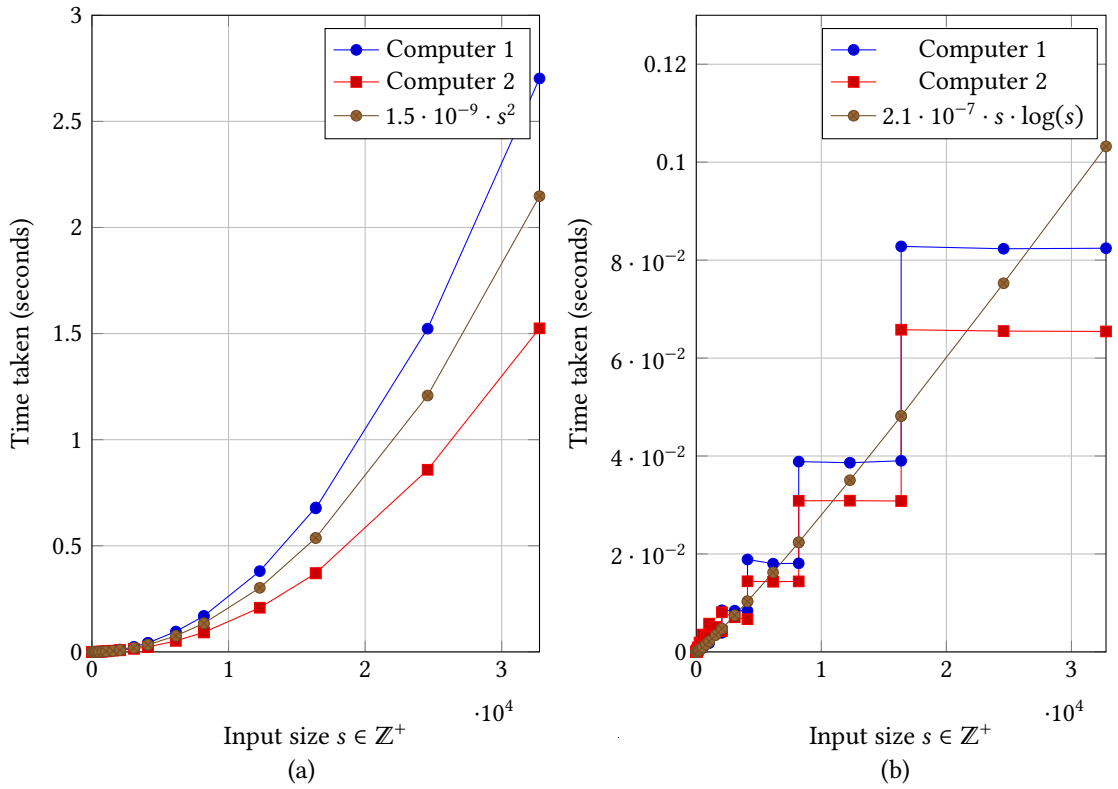


Figure 3.1: Polynomial multiplication for (a) Naive, and (b) FFT-based algorithms (input size 0 to 2^{15})

However, it is not easy to see what the graphs asymptotic bound might be. The given reference curve of $2.1 \cdot 10^{-7} \cdot s \cdot log(s)$, while similar enough to how the average case increases with size, is not unique as a best fit for this range. Similar functions (even a straight line) can also be used, but we know that FFT should have a time complexity of $O(s \cdot \log s)$ so we stick by that idea.

3.5 Comparative Analysis

The average values for the computers are provided in Table 3.3. We can see in Figure 3.2 that the naive polynomial multiplication was faster than the FFT-based algorithm until size $s = 1024$. It becomes faster again for size $s = 1025$, after that from size $s = 1536$ onward, FFT is faster until the end as can be seen in Table 3.3 and Figure 3.3.

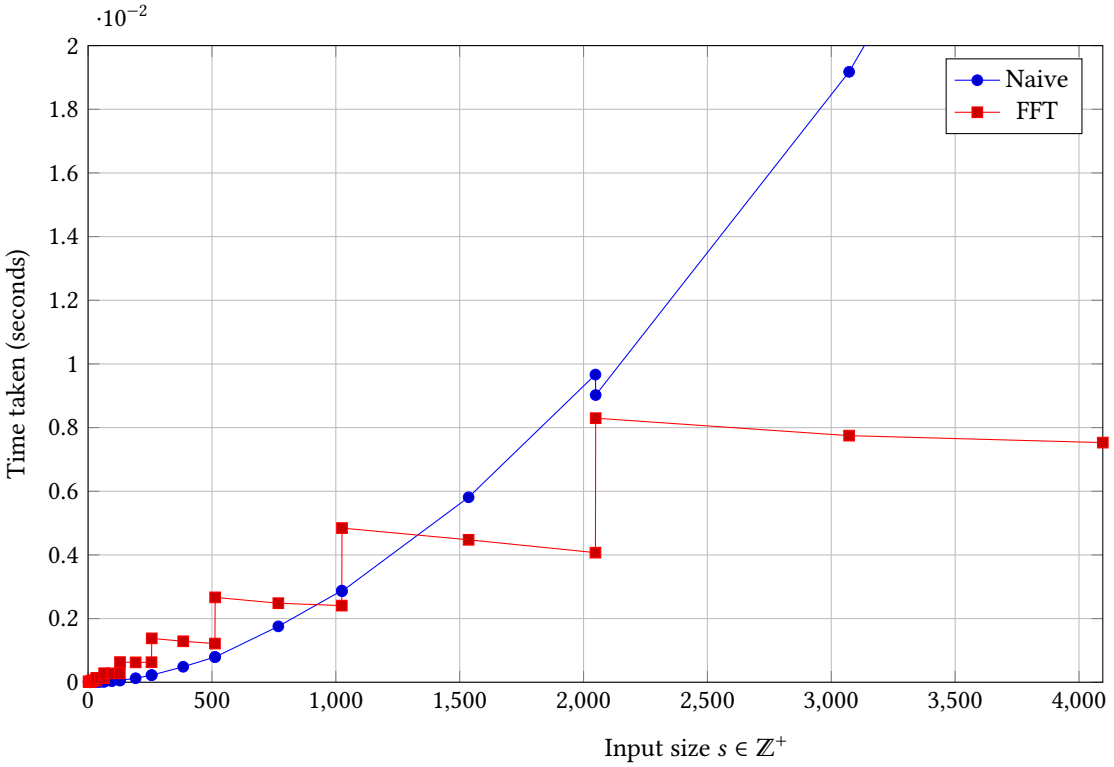


Figure 3.2: Naive vs FFT-based polynomial multiplication (input size 0 to 2^{12})

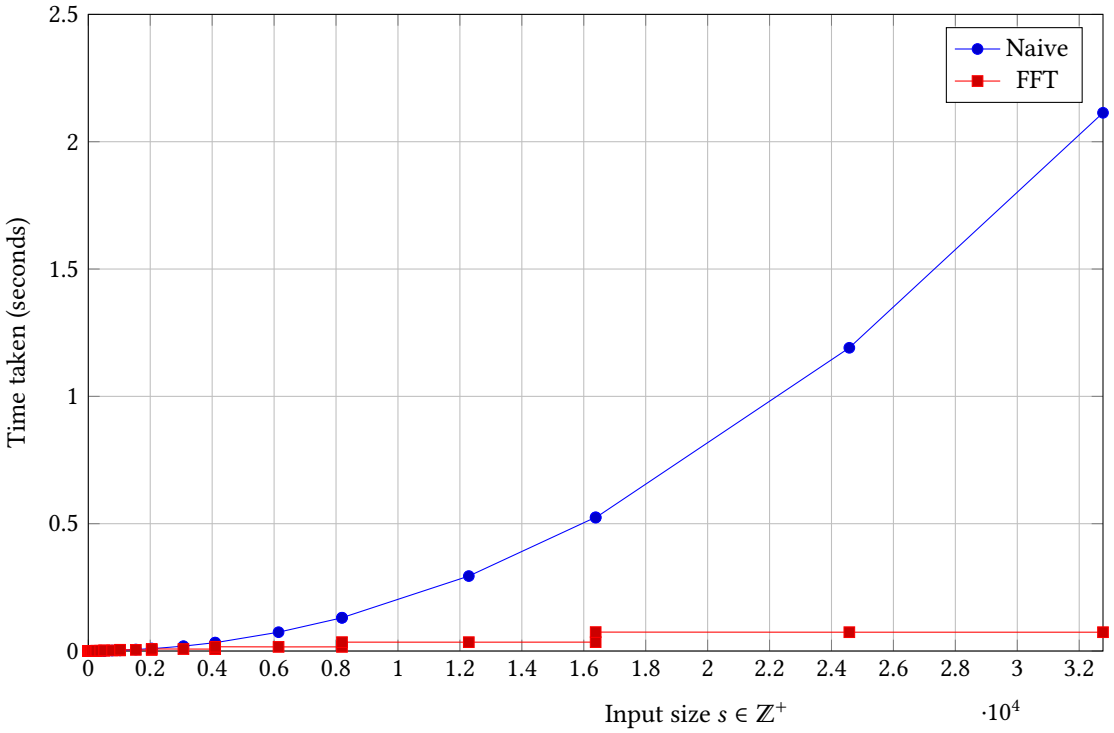


Figure 3.3: Naive vs FFT-based polynomial multiplication (input size 0 to 2^{15})

It is possible that the reason it takes a while for FFT to catch up and then cross naive in speed is because of the overhead that comes from zero padding the input arrays. However, the naive algorithm is considerably slower at large sizes and the main motivation behind using FFT is because at larger input sizes, it outperforms the naive approach by a considerable amount.

Input size	Naive (seconds)	FFT (seconds)
2	0.0000062	0.0000276
3	0.0000012	0.0000057
4	0.0000008	0.0000048
5	0.0000008	0.0000118
6	0.0000012	0.0000112
8	0.0000014	0.0000109
9	0.0000011	0.0000265
12	0.0000016	0.0000258
16	0.0000022	0.0000261
17	0.0000069	0.0000628
24	0.0000048	0.0000596
32	0.0000065	0.0000601
33	0.0000072	0.0001389
48	0.0000099	0.000132
64	0.0000162	0.0001267
65	0.0000167	0.0002841
96	0.0000327	0.0002803
128	0.0000575	0.0002813
129	0.0000584	0.0006323
192	0.000127	0.0006252
256	0.0002231	0.000629
257	0.0002245	0.0013789
384	0.0004869	0.0012874
512	0.0007931	0.0012163
513	0.0007923	0.0026666
768	0.0017564	0.0024837
1024	0.0028796	0.002408
1025	0.0028576	0.0048439
1536	0.0058121	0.0044751
2048	0.0096659	0.0040721
2049	0.0090229	0.0082959
3072	0.0191761	0.0077474
4096	0.0328554	0.0075286
4097	0.0326467	0.0166457
6144	0.0739256	0.0161733
8192	0.1304272	0.0162405
8193	0.1307594	0.0348568
12288	0.2943422	0.0347521
16384	0.5252685	0.0349198
16385	0.5236159	0.0742997
24576	1.1907204	0.0739146
32768	2.1135408	0.0739322

Table 3.3: Input size (degree of the polynomials) and average time taken (in seconds) for polynomial multiplication with naive and FFT-based approaches

Discussion and Conclusion

We have demonstrated our code works and produces the desired output for all the functions. The rounding errors introduced by the FFT due to using double-precision floating-point numbers, unless a highly precise output is desired, can be considered negligible. At least for our purposes while doing integer coefficient polynomial multiplications, the FFT-based output always agreed with the brute force naive approach. The performance analysis revealed that as expected, the FFT-based polynomial multiplication is considerably faster than the naive approach.

However, there could be some improvements in the code. Those running our executable will also note that the CLI could be further optimized. To improve our implementation we could code an in-place algorithm with data reordering (such as bit reversal). The indexes of each element that performs butterfly operation is calculated recursively in the current algorithm. However, by using the nature of bit-reversal permutation, the indexes can be re-arranged in only one for-loop. This will improve memory locality, hence the overall performance. Furthermore, as the algorithm does not contain recursive function calls but only for-loops, techniques such as multi-threading (i.e. OpenMP) and vectorization can be applied to make the computation even faster. We could also compare and inspect results with other polynomial multiplication algorithms such as the Karatsuba Algorithm [10] which has a time complexity of $O(n^{\log_2 3})$ for multiplications [11].

References

[1] Alexander A Antonov. "Oscillation processes as a tool of physics cognition". In: *American Journal of Scientific and Industrial Research* 1.2 (2010), pp. 342–349.

[2] Duraisamy Sundararajan. *The discrete Fourier transform: theory, algorithms and applications*. World Scientific, 2001.

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

[4] Henri J Nussbaumer and Henri J Nussbaumer. *The fast Fourier transform*. Springer, 1982.

[5] James W Cooley, Peter AW Lewis, and Peter D Welch. "Historical notes on the fast Fourier transform". In: *Proceedings of the IEEE* 55.10 (1967), pp. 1675–1677.

[6] Yangwenbo99. *DIT-FFT-butterfly*. CC BY-SA 4.0, via Wikimedia Commons. [Online; accessed 6-January-2024]. 2024. url: https://commons.wikimedia.org/wiki/File:DIT-FFT-butterfly.svg.

[7] Pierre Duhamel, B Piron, and Jacqueline M Etcheto. "On computing the inverse DFT". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.2 (1988), pp. 285–286.

[8] Robert T Moenck. "Practical fast polynomial multiplication". In: *Proceedings of the third ACM symposium on Symbolic and algebraic computation*. 1976, pp. 136–148.

[9] Paul Heckbert. "Fourier transforms and the fast Fourier transform (FFT) algorithm". In: *Computer Graphics* 2.1995 (1995), pp. 15–463.

[10] Anatolii Alekseevich Karatsuba and Yu P Ofman. "Multiplication of many-digital numbers by automatic computers". In: *Doklady Akademii Nauk*. Vol. 145. 2. Russian Academy of Sciences. 1962, pp. 293–294.

[11] André Weimerskirch and Christof Paar. "Generalizations of the Karatsuba algorithm for efficient implementations". In: *Cryptology ePrint Archive* (2006).