MULTIPLYING HUGE INTEGERS USING FOURIER TRANSFORMS

ANDO EMERENCIA (S1283936)

Abstract. Multiplying huge integers of n digits can be done in time $O(n\log(n))$ using Fast Fourier Transforms (FFT), instead of the $O(n^2)$ time complexity normally required. In this paper we present this technique from the viewpoint of polynomial multiplication, explaining a recursive divide-and-conquer approach to FFT multiplication. We have compared both methods of multiplication quantitatively and present our results and conclusions in this paper, along with complexity analyses and error bounds.

1. Introduction

Multiplying huge integers is an operation that occurs in many fields of Computational Science: Cryptography, Number theory, just to name a few. The problem is that traditional approaches to multiplication require $O(n^2)$ multiplication operations, where n is the number of digits. To see why, assume for example that we want to multiply the numbers 123 and 456. The normal way to do this is shown below.

$$\begin{array}{rl}
123 & \\
\underline{456} & \times \\
6\cdot 3+6\cdot 20+6\cdot 100+ & \\
50\cdot 3+50\cdot 20+50\cdot 100+ & \\
\underline{400\cdot 3+400\cdot 20+400\cdot 100} & = \\
56088 &
\end{array}$$

We see that for two integers of length 3, this multiplication requires $3\times 3=9$ operations, hence its $O(n^2)$ complexity. Executing an $O(n^2)$ algorithm for huge n is very costly, so that is why it is preferred to use more efficient algorithms when multiplying huge integers. One way to do this more efficient (in $O(n\log(n))$), is by using FFT's.

The objective of this paper is to explain how this technique works, and analyze it to give answers to practical considerations such as: for what sizes of input does FFT multiplication become faster than normal multiplication, how big this speed-up is, does the base we choose to represent an integer in affect the running time of the multiplication algorithm and whether or not we should worry about rounding errors when using FFT multiplication.

The paper is structured as follows: section 2 explains how the Fast Fourier Transform works. Section 3 presents the specific FFT algorithm that we used in our implementation and shows how to use FFT's for integer multiplication. In

Date: November 7, 2007.

section 4 we present our results of comparing FFT multiplication with normal multiplication. Section 5 discusses the complexity of the algorithm and its correctness. Section 6 concludes the paper.

2. Basic concepts

In this paper we will explain the method of integer multiplication using FFT's in two steps: we will first show how FFT multiplication works for polynomials, and secondly, how to represent an integer as a polynomial. We start by defining some basic concepts about polynomials.

2.1. **Polynomials.**

The default notation for a polynomial is its *coefficient form*. A polynomial p represented in coefficient form is described by its coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ as follows:

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

We call x the *base* of the polynomial, and $p(x)$ is the evaluation of the polynomial, defined by its coefficient vector a, for base x. We conclude from (1) that the evaluation of p for a single input has complexity $O(n)$, since we would need to evaluate n multiplications.

The *degree* of such a polynomial is i, where $a_i \neq 0$ and $\forall j > i, a_j = 0$; ergo, it is the largest index of a nonzero coefficient a_i. By (1), we can also say that the degree of a polynomial is the exponent of the highest power of the base that occurs when we fully expand the polynomial. Note that the indexing of coefficient vectors starts at zero, so a polynomial defined by a coefficient vector of n nonzero coefficients has degree $n - 1$.

Multiplying two polynomials results in a third polynomial, and this process is called *vector convolution*. As with multiplying integers, *vector convolution* takes $O(n^2)$ time:

$$\begin{aligned} p(x)q(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1 + a_2b_0)x^2 + \\ \cdots + a_{n-1}b_{n-1}x^{2n-2}. \end{aligned} \tag{2}$$

This is because for each power of base x, we have to determine the combinations of indices for the two coefficient vectors that are multiplied. Each combination is a multiplication operation that needs to be calculated, and the number of combinations for a power of x is of $O(n)$, and since we have $O(n)$ powers of x in the polynomial product ($2n-2$, to be exact), the total complexity for polynomial multiplication adds up to $O(n^2)$.

We know that the highest power of x in the polynomials p and q, both having n coefficients, is x^{n-1}, so the polynomial of their product has highest power $x^{(n-1)+(n-1)} = x^{2n-2}$. This would imply a coefficient vector of length $2n - 1$. But by convention, we will pad this coefficient vector with a zero at the end, so that it is of length $2n$.

We think that this convention is because when we multiply coefficients of huge integers represented as polynomials (explained in detail later), some products will not fit in the base range that the integer is represented in, so then we have to rescale the coefficients (with a carry and such), causing us to use (at most) one extra power of x. So the length of the coefficient vector is always $2n$, but the degree might be $2n-1$ or $2n-2$, depending on the integers multiplied; for the sake of simplicity and to be safe, in this paper we will assume that it always has degree $2n-1$.

We have seen that multiplying two polynomials in coefficient form takes $O(n^2)$ time. To be able to do this in a lower time complexity, we need to use a different way to represent a polynomial. One such representation is formalized by the *Interpolation Theorem for Polynomials*.

2.2. **The Interpolation Theorem for Polynomials.** Given a set of n points in the plane, $S = (x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_n - 1, y_n - 1)$, such that the x_i's are all distinct, there is a unique $n-1$ polynomial $p(x)$ with $p(x_i) = y_i$, for $i = 0, 1, \dots, n-1$.

This theorem tells us that if we have an $n-1$ degree polynomial, and we evaluate it in n distinct points, then this collection of n input and output points is a unique representation for this polynomial. So like the *coefficient form*, this also is a way to uniquely specify a polynomial.

We know that the product of polynomials p and q of degree $n-1$ is a polynomial of degree $2n-1$, which we now know is uniquely defined by its evaluation in $2n$ distinct points. This gives us the basic idea for FFT multiplication:

- We evaluate p in $2n$ points;
- We evaluate q in the same $2n$ points;
- We compute the $2n$ products of p and q evaluated at these $2n$ points, giving us $2n$ evaluations of the product of p and q at $2n$ distinct points. By the *Interpolation Theorem for Polynomials*, this uniquely defines the polynomial of their product (albeit not in coefficient form).

But we are not done yet, because evaluating $2n$ different inputs will still take $O(n^2)$ time. So the challenge is to find a set of inputs that has specific properties so that we can reuse some of the outputs to evaluate other parts of the input more efficiently. The specific set of inputs used in FFT's are the *primitive roots of unity*.

2.3. **Primitive Roots of Unity.** A number ω is a *primitive nth root of unity*, for $n \geq 2$, if it satisfies the following properties:

- $\omega^n = 1$, that is, ω is an nth root of 1.
- The numbers $1, \omega, \omega^2, \dots, \omega^{n-1}$ are distinct.

This definition tells us that most of these roots of unity will be complex numbers, consisting of a real and an imaginary part. The set of primitive nth roots unity has a number of special properties, but in this paper we will only discuss those properties that we actually use in our FFT implementation to speed up calculations. These are the *reflective property* and the *reduction property*.

2.3.1. *The reflective property.* If ω is a primitive nth root of unity and $n \geq 2$ is even, then $\omega^{k+n/2} = -\omega^k$.

We will not prove or derive this property here. This property is used in implementations of FFT's to reduce the number of values that need to be computed to $n/2$, since the other $n/2$ values can be derived from the first half by a simple sign switch (we will explain in detail how this works for our specific FFT implementation later).

2.3.2. *The reduction property.* If ω is a primitive $(2n)$th root of unity, then ω^2 is a primitive nth root of unity.

We know that the $2n$ powers of ω $(1, \omega, \omega^2, \ldots, \omega^{2n-1})$ are distinct (by definition of ω as a $2n$th primitive root of unity), so the n powers of ω^2 $(1, \omega^2, (\omega^2)^2, \ldots, (\omega^2)^{n-1})$, are also distinct, since the latter set is a subset of the former set. This verifies the property.

The *reduction property* says that if we have the $2n$ roots of unity for a polynomial of degree $2n-1$, then half of these are also roots of unity for a polynomial of degree $n-1$. In our implementation, we actually use this property in reverse: we have calculated values for powers of ω^2 of a polynomial of degree $n-1$ that we use to calculate values for powers of ω for a polynomial of degree $2n-1$.

2.4. **The Discrete Fourier Transform.**

The *Discrete Fourier Transform* (DFT) of an $n-1$ degree polynomial $p(x)$, is its evaluation at the nth roots of unity, $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$.

This definition is a formalization of our basic idea to uniquely represent a polynomial of degree $n-1$ using its evaluations at n distinct points. We note that if this is done naively, it will still take $O(n^2)$ time. That is why we have the *Fast Fourier Transform*, which exploits the specific properties of the set of nth roots of unity in order to achieve lower complexity.

Here we also give the definition of the *Inverse Discrete Fourier Transform*, which is used to recover the coefficients of a polynomial given in its FFT representation (we will explain later when and how it is used).

2.5. **The Inverse Discrete Fourier Transform.**

Given a vector $\mathbf{y}$ of the values of an $n-1$ degree polynomial p at the nth roots of unity, $\omega^0, \omega^1, \ldots, \omega^{n-1}$, the inverse DFT computes the coefficients in the following way:

$$a_i = \sum_{j=0}^{n-1} y_j \omega^{-ij}/n.$$

We will not derive this formula here, but it follows from the formulation of the DFT as a matrix multiplication of the column vector $\mathbf{a}$ and a matrix of powers of ω. The above formula can then be derived by multiplying $\mathbf{a}$ by the inverse of this matrix. The implementation of the inverse FFT is very much like the implementation of the FFT, in fact in our implementation, we make use of the recursive procedure of the FFT, to calculate the inverse FFT.

Note that there is a division in the above formula (by convention, the division is done in the inverse DFT, but it could also be done in the FFT, or in both (dividing by $\sqrt{n}$)). When the polynomial represents an integer, we expect its coefficients to be integers as well, so we have to perform a rounding operation. And since we have a division here, this rounding operation might incur a certain rounding error. In our results we will further analyze the magnitude and severity of this error. Note that this error is due to the limited precision in which we can represent numbers on computers, it does not occur in exact arithmetic.

3. Applied method

There are many ways to implement the Fast Fourier Transform method. Some[SS] methods are based on ring theory, others[GO] on primality theory, and many more exist. Some are recursive and some are not. In this section we will explain the method we used in our implementation, a recursive divide-and-conquer approach to FFT multiplication that is based on the original FFT algorithm of *Cooley-Tukey*.

3.1. **The Cooley-Tukey algorithm for FFT.**

The elegance of the *Cooley-Tukey* algorithm lies its divide-and-conquer nature, so before presenting the pseudo-code for this algorithm, we first explain how a polynomial can be split up into two polynomials of half the original degree, and how we can combine their results to compute the results of the original polynomial.

If n is even, we can divide an $n-1$ degree polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two $(n/2-1)$ degree polynomials

$$\begin{aligned} p_{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p_{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned} \tag{3}$$

which we can combine into p using the equation

$$p(x) = p_{\text{even}}(x^2) + x p_{\text{odd}}(x^2). \tag{4}$$

We can easily verify that the above formula for combining the two polynomials is correct, by entering x^2 into the formulas of p_{even} and p_{odd}. In the following, let "all n powers of ω" denote the set of values ω^k, with k ranging from 0 to $n-1$; we will also assume that n is a power of 2.

Let our recursive FFT procedure be declared with two parameters, one specifying the $n-1$ degree polynomial p, and the other, which we call ω, specifying the nth primitive root of unity. So by definition of the DFT, what we have to do in this procedure is evaluate p at all n powers of ω. Ignoring the base case for now, we first split the polynomial up into two polynomials p_{even} and p_{odd}, according to equation (3).

Now we perform a recursive call to our FFT procedure for these two polynomials, passing ω^2 as second parameter, which is justified by the *reduction property*, which tells us that ω^2 is a primitive $n/2$th root of unity and since n is a power of two, p_{even} and p_{odd} are both of degree $n/2-1$ (since they are half the size of the original polynomial, by definition of even and odd).

So by these recursive calls, we now have the values for p_{even} and p_{odd}, evaluated at all the $n/2$ powers of ω^2. Since we know by the *reduction property* that all $n/2$ powers of ω^2 are a subset of the powers of the n powers of ω, and to be more specific, we know that they match exactly half the values of the n powers of ω; we can combine their results using equation (4), which will give us evaluations of p for half the values in the set of n powers of ω. The other half, we can derive from these values by using the *reflective property*. This idea is formalized in equation (5):

$$\begin{aligned} p(\omega^k) &= p_{\text{even}}(\omega^{2k}) + \omega^k p_{\text{odd}}(\omega^{2k}) \\ p(\omega^{k+n/2}) &= p_{\text{even}}(\omega^{2k}) - \omega^k p_{\text{odd}}(\omega^{2k}) \end{aligned} \tag{5}$$

The first line of equation (5) is just equation (3) filled in for ω^k. Note that this use of the formula is valid, since we have called p_{even} and p_{odd} with the value ω^2, their values at the kth index are actually their evaluations at the kth power of ω^2 ($(\omega^2)^k$). Furthermore, equation (4) specifies that p_{even} and p_{odd} be called with a square, which they are, since $(\omega^2)^k = (\omega^k)^2 = \omega^{2k}$.

Now since we have $n/2$ values in which p_{even} and p_{odd} are evaluated, we can loop equation (5) $n/2$ times, with k ranging from 0 to $n/2-1$. Recall that this will give us evaluations of p for exactly half the values (the lower half) in the set of n powers of ω in which we have to evaluate p. The other half is calculated in the second line of equation (5).

The second line of equation (5) is justified by the *reflective property*, which we recall telling us that $\omega^{k+n/2} = -\omega^k$. So by this property, we have to flip a sign wherever ω^k occurs. Note that the only actual difference between the first and second line of equation (5) is a single minus sign; this is because the other two occurrences of ω^k are squared, and we know that for every number a, real or complex, $a^2 = (-a)^2$, so we do not need to change these products.

The complexity of this algorithm is indeed $O(n\log(n))$, in section 5 we will show why this is true.

3.2. **Pseudo-code FFT algorithm.**

As an illustration, we present the algorithm explained above, in pseudo-code (based on the code in [SS]) below.

```
Algorithm FFT(a, omega)
  Input: An n-length coefficient vector a = [a_0,a_1,...,a_(n-1)]
         and a primitive nth root of unity omega (n = a power of 2)
  Output: A vector y of values of the polynomial for a
          at the nth roots of unity.
  if n=1 then
    return y = a.
  end
  // divide step
  a_even = [a_0,a_2,a_4,...,a_(n-2)]
  a_odd  = [a_1,a_3,a_5,...,a_(n-1)]
  // recursive calls with omega^2 as n/2th root of unity
  y_even = FFT(a_even, omega^2)
  y_odd  = FFT(a_odd,  omega^2)
  x = 1 // storing powers of omega
  // combine step, using x = omega^i
  for (i=0;i<n/2,i++)
    y[i]     = y_even[i]+x*y_odd[i]
    y[i+n/2] = y_even[i]-x*y_odd[i] // because of reflective prop.
    x = x*omega
  end
  return y
end
```

Most of the above pseudo-code we have explained already, except for the basis step. We see here that the basis step is in fact conceptually very simple: since n is a power of two, we keep splitting the polynomial up into two equal parts, until we eventually end up with polynomials that have a coefficient vector of length one.

We know that when a polynomial is specified by a coefficient vector of length one, it only has a coefficient for the 0th power of the base (which is 1), so this reduces the polynomial to a single constant (having the value of this coefficient). Represented in 2D, it would show up as a horizontal line, so no matter what roots of unity we have to evaluate this polynomial in, its value will always be this constant, and that is why we may return `y = a` in the above pseudo-code.

3.3. **Multiplying two polynomials using FFT.** Now that we know how the FFT works, the other steps in the algorithm to multiply two polynomials are easy.

Remember that the polynomial product of two $n-1$ degree polynomials p and q, has degree $2n-1$, so it requires evaluations in at least $2n$ distinct points to be uniquely identified. In order to get $2n$ evaluations, we use the $2n$th primitive roots of unity, and so for our algorithm we need a polynomial with a coefficient vector of at least length $2n$.

So what pad the coefficient vectors of p and q ($\mathbf{a}$ and $\mathbf{b}$, respectively), to at least length $2n$, using zeros:

$$\mathbf{a}' = [a_0, a_1, \ldots, a_{n-1}, 0, 0, \ldots, 0].$$

To be more precise, for our divide-and-conquer algorithm it is required that n is a power of 2, so instead of padding the coefficient vectors to length $2n$, we pad them to length 2^k, where k is the lowest integer such that $2^k \geq 2n$. In the following we will assume without loss of generality that $2^k = 2n$.

The next step is computing the FFT's $\mathbf{y}$=FFT($\mathbf{a}$) and $\mathbf{z}$=FFT($\mathbf{b}$), with our FFT algorithm.

Now we have the evaluations of the polynomials p and q at the same $2n$ distinct inputs (the $2n$th roots of unity), so if we multiply the $2n$ evaluations of p with the respective $2n$ evaluations of q, we calculate $2n$ products in total, that together uniquely represent the polynomial product of p and q:

$$\mathbf{m} = \mathbf{y} \times \mathbf{z} = [y_0 z_0, y_1 z_1, \ldots, y_{2n-1} z_{2n-1}].$$

The remaining step is to compute the inverse FFT of $\mathbf{m}$, to transform the vector of polynomial evaluations (FFT form), to the vector of its coefficients (coefficient form).

This completes the explanation on how to multiply two polynomials using FFT's. In the following subsection, we explain how an integer can be represented as a polynomial, so that we can apply our algorithm to multiply integers.

3.4. **Representing an integer as a polynomial.**

When we represent an integer as a polynomial, we have a choice in what base B to use. Any positive integer can be used as a base, but for the sake of simplicity we restrict ourselves to choosing a base that is a power of 10.

With a base that is a power of 10, converting an integer given in decimal form to its coefficient vector is trivial. Consider the integer 123456, whose polynomial form using $B = 10$ is $\mathbf{a} = [6, 5, 4, 3, 2, 1]$:

$$\overset{a_5}{1}\;\overset{a_4}{2}\;\overset{a_3}{3}\;\overset{a_2}{4}\;\overset{a_1}{5}\;\overset{a_0}{6}$$

A coefficient vector for the same number using a base $B = 100$, would be $\mathbf{b} = [56, 34, 12]$:

$$\overset{b_2}{12}\quad\overset{b_1}{34}\quad\overset{b_0}{56}$$

So this is all very intuitive. There is one note of caution here: assume a base $B = 10^k$, then if k is not a divisor of the length of the input integer, then there are multiple ways to split up the integer, depending on where we start. By our own convention, we choose only to leave room at the coefficient with highest index (i.e. start splitting the number up from the right), this is useful since we have to rescale the coefficients. Consider for example the number 12345, then (for $B = 100$), we choose to split it up as [45,23,1], and not [5,34,12].

In our implementation we used a standard big integer library that lets us represent integers as polynomials, and the setting of the base B is a parameter to this library.

Once we have represented our integers as polynomials, we can simply use the multiplication algorithm for polynomials explained in the previous section, with one additional step (in $O(n)$ time): we have to rescale the coefficients so that they fit in the base that the integer is represented in (recall this is why we defined the polynomial product to have an extra coefficient), this is implemented simply by a `for`-loop and keeping track of a *carry*.

4. Results

4.1. Experimental setup.

- We have written an implementation that compares the speeds of normal multiplication and FFT multiplication.
- Both the FFT and the normal implementation first represent the integer as a polynomial by splitting it up in coefficients to a certain base B, as shown in the previous section.
- For both multiplication methods, we then let the program perform the same multiplication for a duration of 500ms, and count how many times the multiplication has been performed in this duration, in order to get a good average of the time a single multiplication took. Some of the longer multiplications would take more than 500ms for a single evaluation, so we also set a limit that the multiplication has to be executed at least 5 times.
- We performed tests using nine different multiplications, each with two randomly generated integers of the same length (here we mean by length the number of digits in decimal form), for different lengths: 5, 50, 100, 500, 1000, 5000, 10000, 25000, 50000.
- We performed the tests for four different bases, where by base we mean the base we use to split the integer up into coefficients: 10, 100, 1000, 10000.
- In the following, let "an input size of n" denote that the algorithms have to perform a multiplication of two integers that both consist of $n/2$ digits, when represented in decimal form (so their product consists of n digits).

In the following subsections we list the results thus obtained.

4.2. **Base 10 FFT multiplication vs. base 10 normal multiplication.** Figure 1 shows a comparison of these two methods at this base. The individual values of the measurements taken are also listed in the table below.

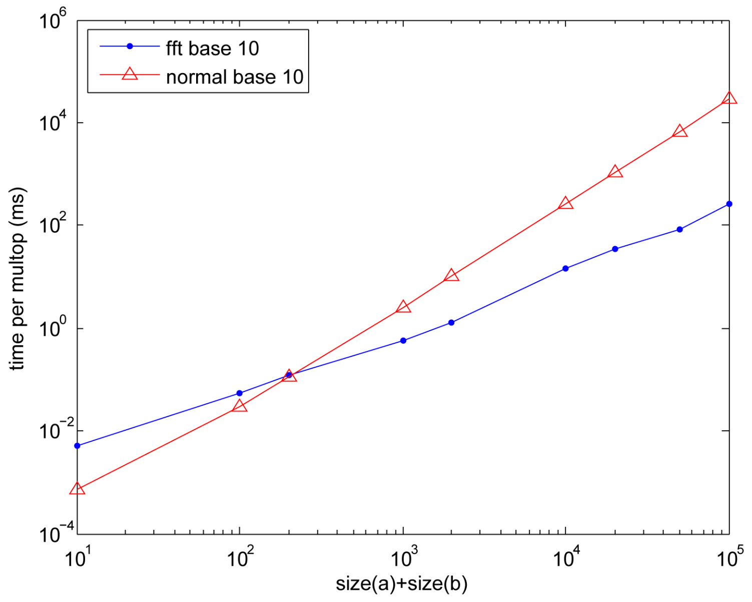

FIGURE 1. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are drawn on a logarithmic scale (base 10).

time(ms)	**10**	**100**	**200**	**1000**	**2000**	**10000**	**20000**	**50000**	**100000**
FFT	0.005	0.055	0.122	0.580	1.281	14.156	35.125	85.75	258.0
Normal	0.0007	0.031	0.115	2.563	10.25	265.5	1070.5	6836.0	28211.0

We see here that for an input size of more than 200, the FFT method becomes increasingly faster than normal multiplication. For an input size of 10^5, the FFT multiplication algorithm takes 258ms, while normal multiplication requires more than 28 seconds, so the time required differs by a ratio of more than a hundred.

4.3. **Base 100 FFT multiplication vs. base 100 normal multiplication.** Figure 2 shows a comparison of these two methods at this base. The individual values of the measurements taken are also listed in the table below.

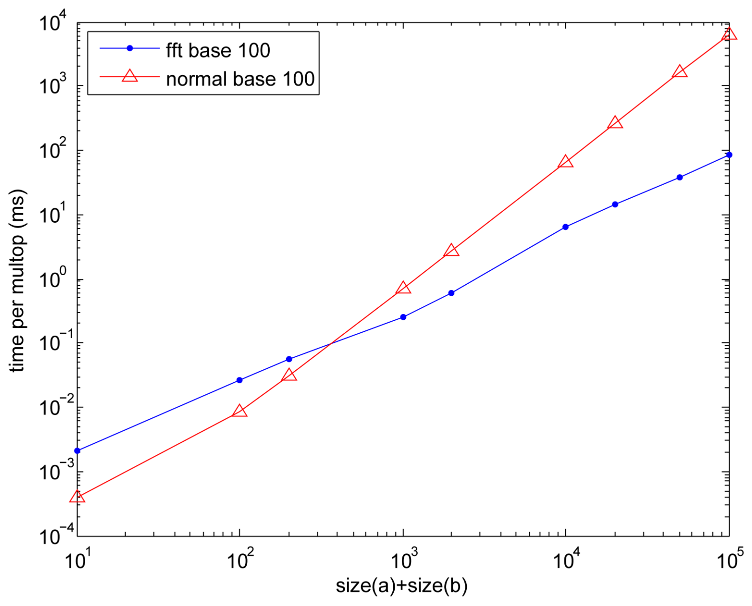

Figure 2. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are drawn on a logarithmic scale (base 10).

time(ms)	**10**	**100**	**200**	**1000**	**2000**	**10000**	**20000**	**50000**	**100000**
FFT	0.0021	0.0258	0.057	0.259	0.611	6.344	14.156	37.0	86.0
Normal	0.0004	0.0086	0.031	0.701	2.688	66.5	265.5	1680.0	6398.5

We see here that both the FFT multiplication and the normal multiplication speed up when we use base $B = 100$ instead of base $B = 10$, for example for an input size of 10^5, the time for a normal multiplication goes down from 28 seconds to 6.4 seconds, and for the FFT multiplication it goes down from 258ms to 86ms. The shape of the graph however roughly stays the same, so we know that the ratio between both methods has not changed by much (around 50-70% speed increase for FFT and around 60-80% speed increase for normal multiplication).

Another thing that we note here is that the point where FFT multiplication becomes faster that normal multiplication has shifted a bit towards the right: FFT multiplication now becomes faster for an input size of around 400 (this is 200 when $B = 10$).

4.4. **Base 1000 FFT multiplication vs. base 1000 normal multiplication.** Figure 3 shows a comparison of these two methods at this base. The individual values of the measurements taken are also listed in the table below.

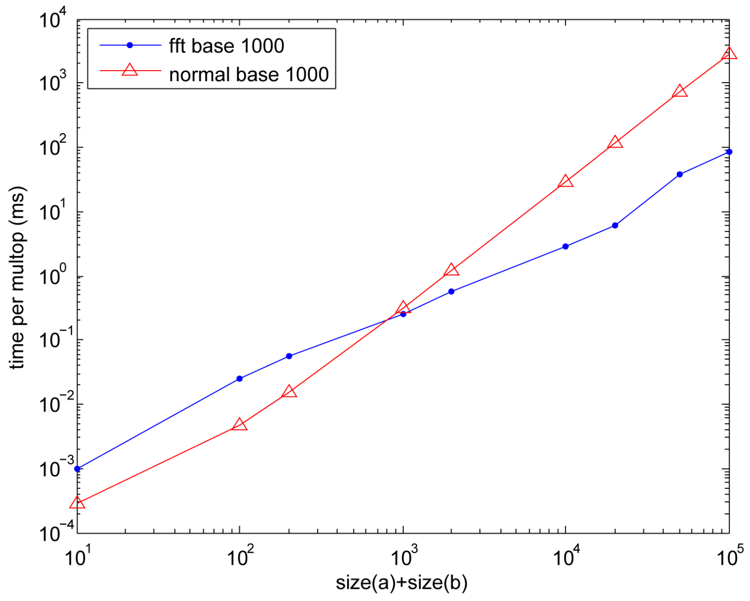

FIGURE 3. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are drawn on a logarithmic scale (base 10).

time(ms)	**10**	**100**	**200**	**1000**	**2000**	**10000**	**20000**	**50000**	**100000**
FFT	0.0010	0.0248	0.055	0.259	0.580	2.930	6.094	37.0	86.0
Normal	0.0003	0.0048	0.015	0.320	1.219	28.313	117.25	742.5	2820.0

With a base $B = 1000$ instead of 100, we see pretty much the same changes as when we changed to $B = 100$ from $B = 10$. Overall, the times per multiplication operation decrease by a factor, the ratio in times between the two multiplication methods stays roughly the same, and the intersection point for the two lines moves slightly to the right.

4.5. **Base 10000 FFT multiplication vs. base 10000 normal multiplication.** Figure 4 shows a comparison of these two methods at this base. The individual values of the measurements taken are also listed in the table below.

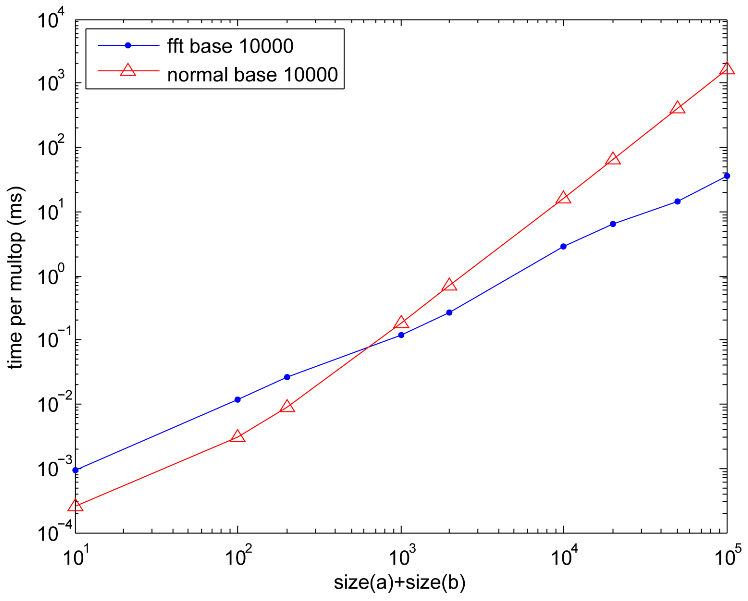

FIGURE 4. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are drawn on a logarithmic scale (base 10).

time(ms)	**10**	**100**	**200**	**1000**	**2000**	**10000**	**20000**	**50000**	**100000**
FFT	0.0010	0.0115	0.026	0.122	0.269	2.831	6.25	14.05	35.9
Normal	0.0003	0.0031	0.009	0.183	0.708	15.65	65.6	415.6	1671.8

Here we see the same type of changes as before. We think that because the intersection shifts a tiny bit to the right each time we increase the base, the normal:FFT time ratio measured for the points on the right side of the intersection becomes a bit smaller as well. However the scale by which this difference increases, roughly stays the same.

4.6. **Comparing FFT's of various bases.** Figure 5 shows a comparison of the times spent per multiplication operation for the FFT multiplication method only, for the four bases that we discussed earlier.

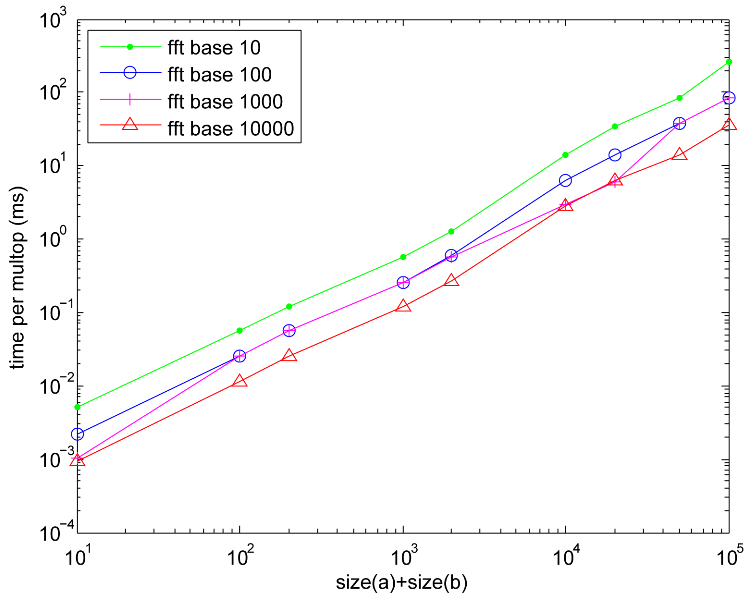

Figure 5. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are dawn on a logarithmic scale (base 10).

We see here that FFT multiplications indeed speed up when we use a larger base B. We can also conclude that the relative speed increase is not dependent on the input size.

4.7. **Comparing normal multiplications of various bases.** Figure 6 shows a comparison of the times spent per multiplication operation for the normal multiplication method only, for the four bases that we discussed earlier.

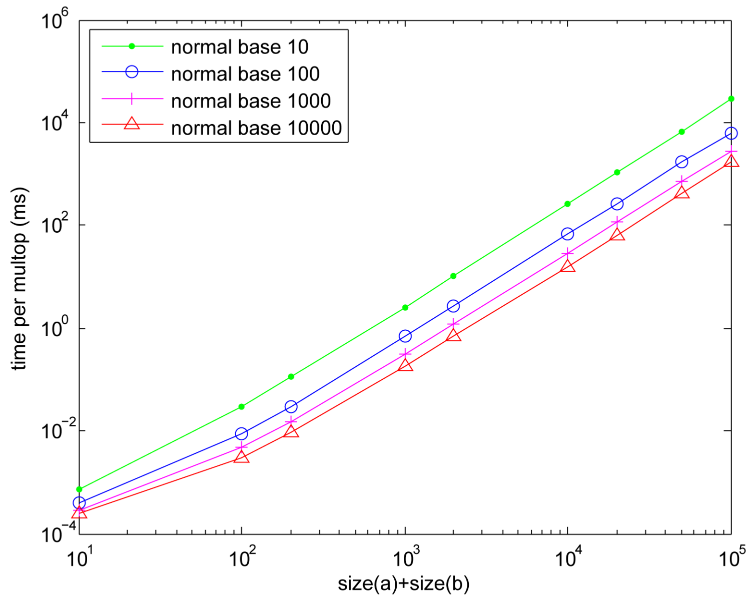

FIGURE 6. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are dawn on a logarithmic scale (base 10).

This image confirms our findings that normal multiplications speed up when we use a larger base B. And the relative speed increase here is also not dependent on the input size.

4.8. **Comparing FFT multiplications with normal multiplications at various bases.** Figure 7 shows a comparison of these two methods at the various bases we discussed earlier.

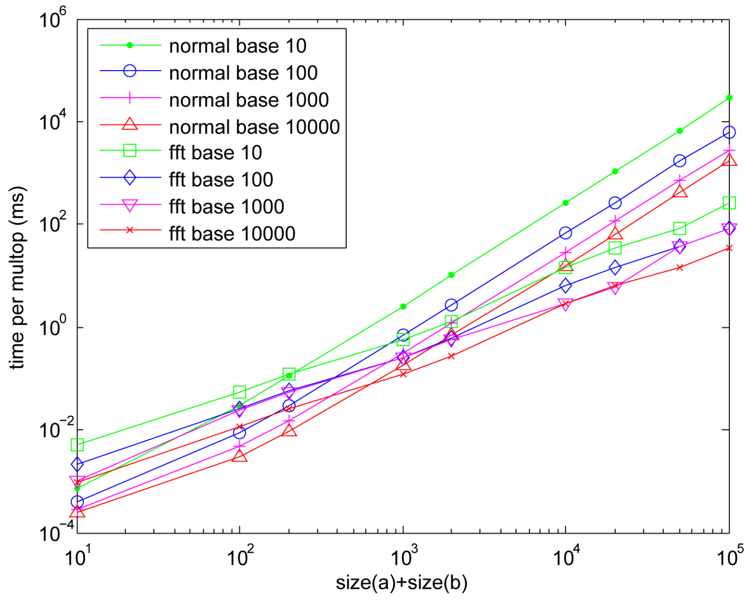

Figure 7. A plot showing the average time spent per multiplication operation as a function of the sum of the lengths of the integers to be multiplied. Both axes are dawn on a logarithmic scale (base 10).

We see here that even the FFT multiplication at the lowest base eventually becomes faster than the normal multiplication at the highest base, and that this difference increases with the input size (as it should, since it represents a difference of $O(n^2 - n\log(n))$).

4.9. **Maximum Square Error.** Remember that in the inverse FFT procedure, we incurred a rounding error because there was a division and then we had to round to integers. In our implementation, we keep track of the square of the highest deviation caused by the rounding function, that is:

$$\texttt{SquareError} = (\lfloor x + 0.5 \rfloor - x)^2$$

We compute this `SquareError` for every coefficient of the polynomial and keep track of its maximum. Note that the rounding error can never be greater than 0.5, so by definition, the `SquareError` can never be greater than 0.25.

However, this measurement is a bit misleading. Let us have an x of 0.4 for example, representing a coefficient that is supposed to be 0. We then calculate: $\lfloor 0.4 + 0.5 \rfloor = \lfloor 0.9 \rfloor = 0$; $(0 - 0.4)^2 = 0.16$. So this gives a `SquareError` of 0.16, but we did classify the value as 0, so our answer is still correct. For comparison, let us now have an x of 0.6, representing a coefficient that is supposed to be 0. We calculate: $\lfloor 0.6 + 0.5 \rfloor = \lfloor 1.1 \rfloor = 1$; $(1 - 0.6)^2 = (0.4)^2 = 0.16$.

So while one of these two examples was classified wrong, their `SquareError` value is the same, since an absolute rounding error of 0.6 gives exactly the same `SquareError` as an absolute rounding error of 0.4. We cannot do much about this, since the program does not know that 0.6 should have been 0; by rounding it assumes that it should have been 1. This causes the answer to be incorrect, without the program having measured a *Maximum Square Error* that is (almost) 0.25. We conclude that if our *Maximum Square Error* gets close to 0.25, there already is a possibility that some coefficients have been rounded to the wrong number, causing the answer to be incorrect.

In the following subsection we plot values of this *Maximum Square Error* (which of course only occurs for FFT multiplication) at the various bases.

4.10. **Maximum Square Error of the FFT multiplication at various bases.** Figure 8 shows a comparison of the *Maximum Square Error* measured in FFT multiplication, for the four bases we discussed earlier.

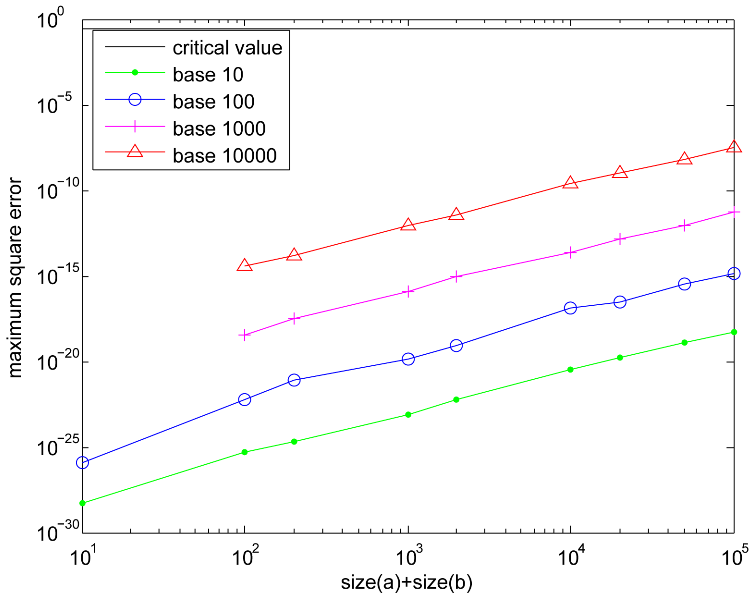

Figure 8. A plot showing the *Maximum Square Error* as a function of the sum of the lengths of the integers to be multiplied. Both axes are dawn on a logarithmic scale (base 10). The horizontal line labeled *critical value* is drawn at the theoretical maximum of the *Maximum Square Error* (0.25).

Note that the first two values for the base 1000 and base 10000 lines are missing here; since they are 0, their log is undefined.

We see here that the *Maximum Square Error* increases with the input size and with the base. We also note that for the multiplications we performed, the *Maximum Square Error* is still in a very safe range. Note that when we have an input size of 10^7 or higher, we may have to choose a smaller base in order to avoid rounding errors from causing our answer to be incorrect.

5. Discussion

5.1. **Correctness.** To verify that the outcomes of our program were correct, we simply copy/pasted our input to Mathematica, which supports multiplication of Big Integers. We then compared the output of our program with the output of Mathematica and found that our program indeed gives correct answers for all results we presented.

We have also put our implementation online, so the readers of this paper can test our implementation (in C++) for themselves:
`http://ando.home.fmf.nl/fft_multiplication_implementation.zip`.
See the included `Readme.txt` for instructions.

5.2. **Complexity.** Looking back at the pseudo-code for our FFT algorithm, we see that the algorithm calls itself two times with half the input (a polynomial of half the original size), and then performs the merge in a loop of $O(n)$, so according to the *Master Theorem*, the complexity of the FFT procedure is $O(n\log(n))$.

As for the other steps in FFT multiplication: padding a vector can be done in $O(n)$ time, multiplying two vectors component-wise also takes $O(n)$ time (since we have to perform $O(n)$ multiplications), the inverse FFT has the same complexity as the FFT, and rescaling the coefficients takes $O(n)$. So the overall complexity of FFT multiplication, dictated by the complexity of the FFT procedure, is $O(n\log(n))$.

5.3. **Future work.** A way to improve our implementation would be to have the program determine the optimal base (using the input size for example), to optimize speed and correctness. However, for very large numbers, we then would have to use bases smaller than 10, so bases that are not powers of 10; in which case the conversion of integers to polynomial form becomes non-trivial.

But then again, if we want to design an FFT multiplication function that should always be correct, we should use one of the more complex and optimized FFT algorithms, that is less prone to rounding errors.

5.4. **Integer division.** Another question posed is whether we could use FFT's for integer division. The problem here is in representing a quotient as a polynomial; a quotient can not be written as a sum of (individually evaluated) quotients like a multiplication can be written as a sum of multiplications.

We could for example split $\frac{28}{7}$ up into $\frac{8}{7}+\frac{20}{7}$, but then we would need the $\frac{1}{7}$ part from the first quotient to evaluate the second quotient. Rounding down each quotient to an integer is not an option since that would cause an incorrect answer. So it might be possible to use FFT's for integer division if there is an algorithm that distributes the fractional parts of quotients to match other quotients, but this is non-trivial. Even if such an algorithm exists, it would have to have a low complexity to make it useful.

6. Conclusion

- Choosing a larger base will speed up the calculation of both the normal and the FFT multiplications. The relative size of this speed increase does not depend on input size.
- Choosing a larger base will increase the *Maximum Square Error* in FFT multiplications.
- Increasing the size of the input will increase the *Maximum Square Error* in FFT multiplications.
- For number lengths of around 10000 and higher (depending on the base used), FFT multiplication can easily be more than 100 times as fast as a normal multiplication implementation.
- The input size where FFT multiplication becomes faster than normal multiplication increases with the base size. But no matter what base we choose, FFT multiplication will become faster than normal multiplication at some point.
- When working with numbers of 10^7 or higher, we may have to choose a smaller base in order to avoid rounding errors from causing an incorrect answer.

References

[GO] I.J. Good: 1958-1960, 'The interaction algorithm and practical Fourier analysis', in *J. R. Statist. Soc. B 20 (2)*, pp. 361-372.

[GT] M.T. Goodrich, R. Tamassia: 2002, 'Algorithm Design Foundations, Analysis, and Internet Examples', pp. 488-495.

[GS] X. Gourdon, P. Sebah: 2001, 'FFT based multiplication of large numbers', `http://numbers.computation.free.fr/Constants/Algorithms/fft.html`

[SS] A. Schönhage, V. Strassen: 1971, 'Schnelle Multiplikation großer Zahlen', in *Computing 7*, pp. 281-292.