Multiplying huge integers using Fourier transforms

Ando Emerencia (s1283936)

October 25, 2007
Outline

- The problem
- How does FFT Multiplication work?
- The implementation
- Results
- Questions
The problem

Multiplying huge integers occurs in many fields of Computational Science:

- Cryptography
- Number theory
- ...

820348901038490238478324
1739423249728934932894 ×

???
The problem

Traditional approaches to multiplication require $O(n^2)$ multiplications.

A simple example

\[
\begin{array}{c}
123 \\
456 \times \\
\hline
6 \cdot 3 + 6 \cdot 20 + 6 \cdot 100 + \\
50 \cdot 3 + 50 \cdot 20 + 50 \cdot 100 + \\
400 \cdot 3 + 400 \cdot 20 + 400 \cdot 100 = \\
\hline
56088
\end{array}
\]

Two integers of length 3, $3 \times 3 = 9$ operations, this is costly for very large integers.
Can this be done faster?

One way to do this faster is by using Fourier transforms, reducing the arithmetic complexity to $O(n \log n)$. 
How does FFT Multiplication work?

Multiplying polynomials

Coefficient form
A polynomial represented in coefficient form is described by a coefficient vector \( \mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \) as follows:

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

Degree
The degree of such a polynomial is the largest index of a nonzero coefficient \( a_i \).
How does FFT Multiplication work?

Vector convolution

Multiplying polynomials in coefficient form

Like multiplying integers, takes $\Theta(n^2)$ time:

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

Degree of polynomial product

The highest power of $x$ of polynomials $p$ and $q$ having $n$ coefficients is $x^{n-1}$, so their product has a highest power $x^{(n-1)+(n-1)} = x^{2n-2}$, but for symmetry reasons we define their product of size $2n$, with coefficient vector $c = [c_0, c_1, \ldots, c_{2n-1}]$, defining $c_{2n-1} = 0$. 
How does FFT Multiplication work?

The Interpolation Theorem for Polynomials

We have to find a different way to represent a polynomial other than the coefficient form, a representation that allows for faster multiplication than $\Theta(n^2)$.

The Interpolation Theorem for Polynomials

Given a set of $n$ points in the plane, $S = (x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})$, such that the $x_i$'s are all distinct, there is a unique degree-$(n - 1)$ polynomials $p(x)$ with $p(x_i) = y_i$, for $i = 0, 1, \ldots, n - 1$.

Figure: A set of $n$ distinct points in the plane uniquely defines an $(n - 1)$-degree polynomial.
How does FFT Multiplication work?
Calculating the polynomial product in a different form

Idea
We know that the product of polynomials $p$ and $q$ of degree $n - 1$ is a polynomial of degree $2n - 1$, so:

- We evaluate $p$ in $2n$ points;
- We evaluate $q$ in the same $2n$ points;
- So if we compute the products of $p$ and $q$ evaluated at these $2n$ points, we have uniquely defined the polynomial of their product (albeit not in coefficient form).

Not there yet
Evaluating $2n$ different inputs will take $O(n^2)$ time. The challenge is to find a set of inputs that has specific properties so that some of the outputs can be used to quickly evaluate other parts of the input.
How does FFT Multiplication work?

How to find $2^n$ distinct points that can be quickly evaluated?

**Primitive Roots of Unity**

A number $\omega$ is a **primitive $n$th root of unity**, for $n \geq 2$, if it satisfies the following properties:

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

**Special properties**

- **(Reflective property)** If $\omega$ is a primitive $n$th root of unity and $n \geq 2$ is even, then $\omega^{k+n/2} = -\omega^k$.
- **(Reduction property)** If $\omega$ is a primitive $(2n)$th root of unity, then $\omega^2$ is a primitive $n$th root of unity. So if $1, \omega, \omega^2, \ldots, \omega^{2n-1}$ are distinct, then $1, \omega^2, (\omega^2)^2, \ldots, (\omega^2)^{n-1}$ are also distinct (since the latter set is a subset of the former set).
How does FFT Multiplication work?

The Discrete Fourier Transform

DFT

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

Note

Done naively, this can still take \(O(n^2)\) time. That is why we have the Fast Fourier Transform.

Different methods

There are many ways in which an FFT can be implemented, in our implementation we used a recursive divide-and-conquer approach. This method is relatively simple but its use is widespread. In the following, we will explain this method in more detail.
How does FFT Multiplication work?

The Cooley-Tukey algorithm for FFT

A divide and conquer approach

If $n$ is even, we can divide a degree-$(n - 1)$ polynomial

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

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

$$p^{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$
$$p^{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

which we can combine into $p$ using the equation

$$p(x) = p^{\text{even}}(x^2) + xp^{\text{odd}}(x^2).$$
How does FFT Multiplication work?

Why is this faster?

Reuse of computations

By the reduction property, the values $(\omega^2)^0, \omega^2, (\omega^2)^2, (\omega^2)^3, \ldots, (\omega^2)^{n-1}$ are $(n/2)$th roots of unity. Thus, we can evaluate each of $p^{\text{even}}(x)$ and $p^{\text{odd}}(x)$ at these values, and we can reuse those same computations in evaluating $p(x)$.

Further reducing computation

If we use an even $n$, then by the reflective property, we know that $\omega^{k+n/2} = -\omega^k$. So we can reuse the computations of half the number of points to derive the values of the other half.
The implementation

Pseudo-code FFT algorithm

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
The implementation
How to multiply two polynomials?

Convolution
Now that we know how the FFT works, we can use it to multiply two polynomials given as coefficient vectors \( a \) and \( b \).

- Pad \( a \) and \( b \) with \( n \) zeros so that they will be of size \( 2n \):
  \[
a' = [a_0, a_1, \ldots, a_{n-1}, 0, 0, \ldots, 0].
\]
- Compute the FFT’s \( y = \text{FFT}(a) \) and \( z = \text{FFT}(b) \).
- Multiply the vectors \( y \) and \( z \) component-wise
  \[
m = y \times z = [y_0z_0, y_1z_1, \ldots, y_{2n-1}z_{2n-1}].
\]
- Compute the inverse FFT of \( m \).

Note
In practice, instead of \( 2n \), the next power of 2 is used instead. This is required by the divide and conquer algorithm.
The implementation

The Inverse Discrete Fourier Transform

Definition
We use the Inverse Discrete Fourier Transform to recover the coefficients from a vector $y$ of the values of a degree-$n-1$ polynomial $p$ at the $n$th roots of unity, $\omega^0, \omega^1, \ldots, \omega^{n-1}$.

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

Note
We will not derive this formula here, but it follows from the formulation of the DFT as a matrix multiplication of the column vector $a$ and a matrix of powers of $\omega$. The above formula can then be derived from multiplying $a$ by the inverse of this matrix.
The implementation

Multiplying big integers

The Base
An integer can be expressed as a polynomial of a certain base \( B \).

Example
The most simple implementation would be of base \( B = 10 \).

\[
\begin{align*}
a_5 & \quad a_4 & \quad a_3 & \quad a_2 & \quad a_1 & \quad a_0 \\
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6
\end{align*}
\]

But different bases can also be used, for example \( B = 100 \).

\[
\begin{align*}
b_2 & \quad b_1 & \quad b_0 \\
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6
\end{align*}
\]

Any positive integer can be used as base, but for the sake of simplicity, in our implementation we only used bases that are powers of 10.
The implementation

Correctness

Mathematica as verification
Mathematica supports multiplication of Big Integers.

Note
Note that when we multiply big integers, the coefficients $a_i$ that we get after the Inverse Fourier Transform are supposed to be integers. Because we divide by $n$, we incur a certain rounding error. In our results we will analyse this error.
The implementation
Complexity of multiplication with FFT

Arithmetic complexity
Because the algorithm FFT() calls itself two times with half
the input, and then performs the merge in a loop of
$O(n \log(n))$, so according to the Master Theorem, the
complexity is $O(n \log(n))$.

Actual complexity
The actual complexity of the algorithm in terms of the
number of operations is higher, because then we would have
to include ordinary multiplications as a factor in the
complexity. This also shows in practice, because the
overhead of these multiplications is significant, especially
when working with a large base number.
Results

Experimental settings

- We have written an implementation that compares the speed of a naive multiplication implementation versus an FFT one.
- Both the FFT and the naive approach first split the integers up in coefficients to a certain base $B$, as shown before.
- Where we let the program perform the same multiplication for a duration of 500ms and then count how many times the multiplication has been performed to get a good average of the time a single multiplication of the polynomials took.
- We performed tests on nine multiplications of different lengths (in decimal form): $5 \times 5$, $50 \times 50$, $100 \times 100$, $500 \times 500$, $1000 \times 1000$, $5000 \times 5000$, $10000 \times 10000$, $25000 \times 250000$, $50000 \times 50000$. 
Multiplying huge integers using Fourier transforms

Ando Emerencia
(s1283936)

Outline
The problem
How does FFT Multiplication work?
The implementation
Results
Experimental settings
Base 10
Base 100
Base 1000
Base 10000
Comparing FFT’s
Comparing Normal Multiplications
Comparing both multiplications
Maximum Square Error
Conclusions
Questions

Results

Base 10 FFT Multiplication vs. Base 10 Normal Multiplication

<table>
<thead>
<tr>
<th>size(a)+size(b)</th>
<th>Base 10 FFT</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.005</td>
<td>0.0007</td>
</tr>
<tr>
<td>100</td>
<td>0.055</td>
<td>0.031</td>
</tr>
<tr>
<td>200</td>
<td>0.122</td>
<td>0.115</td>
</tr>
<tr>
<td>1000</td>
<td>0.580</td>
<td>2.563</td>
</tr>
<tr>
<td>2000</td>
<td>1.281</td>
<td>10.25</td>
</tr>
<tr>
<td>10000</td>
<td>14.156</td>
<td>265.5</td>
</tr>
<tr>
<td>20000</td>
<td>35.125</td>
<td>1070.5</td>
</tr>
<tr>
<td>50000</td>
<td>85.75</td>
<td>6836.0</td>
</tr>
<tr>
<td>100000</td>
<td>258.0</td>
<td>28211.0</td>
</tr>
</tbody>
</table>
Multiplying huge integers using Fourier transforms

Ando Emerencia
(s1283936)

Outline
The problem
How does FFT Multiplication work?
The implementation
Results
Experimental settings
Base 10
Base 100
Base 1000
Base 10000
Comparing FFT's
Comparing Normal Multiplications
Comparing both multiplications
Maximum Square Error
Conclusions
Questions

Results
Base 100 FFT Multiplication vs. Base 100 Normal Multiplication

<table>
<thead>
<tr>
<th>size(a)+size(b)</th>
<th>FFT</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0021</td>
<td>0.0004</td>
</tr>
<tr>
<td>100</td>
<td>0.0258</td>
<td>0.0086</td>
</tr>
<tr>
<td>200</td>
<td>0.057</td>
<td>0.031</td>
</tr>
<tr>
<td>1000</td>
<td>0.259</td>
<td>0.701</td>
</tr>
<tr>
<td>2000</td>
<td>0.611</td>
<td>2.688</td>
</tr>
<tr>
<td>10000</td>
<td>6.344</td>
<td>66.5</td>
</tr>
<tr>
<td>20000</td>
<td>14.156</td>
<td>265.5</td>
</tr>
<tr>
<td>50000</td>
<td>37.0</td>
<td>1680.0</td>
</tr>
<tr>
<td>100000</td>
<td>86.0</td>
<td>6398.5</td>
</tr>
</tbody>
</table>
Multiplying huge integers using Fourier transforms

Ando Emerencia (s1283936)

Outline
The problem
How does FFT Multiplication work?
The implementation
Results
Experimental settings
Base 10
Base 100
Base 1000
Base 10000
Comparing FFT's
Comparing Normal Multiplications
Comparing both multiplications
Maximum Square Error
Conclusions
Questions

Results

Base 1000 FFT Multiplication vs. Base 1000 Normal Multiplication

![Graph showing comparison between FFT and Normal multiplication]

<table>
<thead>
<tr>
<th>size(a)+size(b)</th>
<th>10</th>
<th>100</th>
<th>200</th>
<th>1000</th>
<th>2000</th>
<th>10000</th>
<th>20000</th>
<th>50000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>0.0010</td>
<td>0.0248</td>
<td>0.055</td>
<td>0.259</td>
<td>0.580</td>
<td>2.930</td>
<td>6.094</td>
<td>37.0</td>
<td>86.0</td>
</tr>
<tr>
<td>Normal</td>
<td>0.0003</td>
<td>0.0048</td>
<td>0.015</td>
<td>0.320</td>
<td>1.219</td>
<td>28.313</td>
<td>117.25</td>
<td>742.5</td>
<td>2820.0</td>
</tr>
</tbody>
</table>
Results

Base 10000 FFT Multiplication vs. Base 10000 Normal Multiplication

<table>
<thead>
<tr>
<th>size(a)+size(b)</th>
<th>FFT</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0010</td>
<td>0.0003</td>
</tr>
<tr>
<td>100</td>
<td>0.0115</td>
<td>0.0031</td>
</tr>
<tr>
<td>200</td>
<td>0.026</td>
<td>0.009</td>
</tr>
<tr>
<td>1000</td>
<td>0.122</td>
<td>0.183</td>
</tr>
<tr>
<td>2000</td>
<td>0.269</td>
<td>0.708</td>
</tr>
<tr>
<td>10000</td>
<td>2.831</td>
<td>15.65</td>
</tr>
<tr>
<td>20000</td>
<td>6.25</td>
<td>65.6</td>
</tr>
<tr>
<td>50000</td>
<td>14.05</td>
<td>415.6</td>
</tr>
<tr>
<td>100000</td>
<td>35.9</td>
<td>1671.8</td>
</tr>
</tbody>
</table>
Results

Comparing FFT's of various bases

![Graph showing time per multiply (ms) versus size(a)+size(b) for different bases.]

- fft base 10
- fft base 100
- fft base 1000
- fft base 10000

Comparing FFT's of various bases
Results

Comparing Normal Multiplications at various bases
Results
Comparing Normal Multiplications with FFT Multiplications at various bases

![Graph showing comparison of time per multiplication vs size of input numbers for different bases. The graph compares normal multiplications (base 10, 100, 1000, 10000) and FFT multiplications (base 10000) at various sizes. The graph indicates that FFT multiplication is generally faster than normal multiplication, especially for larger sizes. The x-axis represents the sum of the sizes of the input numbers, and the y-axis represents the time per multiplication in milliseconds.]
Results
Maximum Square Error

Definition
In the inverse FFT procedure, when rounding down from doubles to integers, we keep track of the square of the highest deviation, that is:

\[ \text{SquareError} = ([x + 0.5] - x)^2 \]

Critical value
Note that if the Maximum Square Error comes close to 0.25, then the real error comes close to 0.5 and then there is a chance that we make rounding errors and end up with an incorrect answer.
Results

Maximum Square Error of the FFT multiplication at various bases

![Graph showing the maximum square error of FFT multiplication at various bases. The x-axis represents the sum of sizes of a and b, while the y-axis represents the time per multiply (ms). Different lines represent different bases: base 10, base 100, base 1000, base 10000, and a critical value line. The graph illustrates how the maximum square error increases with larger bases.](image-url)
Results

Conclusions

- Choosing a larger base will speed up the calculation of both the normal and the FFT multiplications.
- Choosing a larger base will increase the Maximum Square Error of the inverse FFT.
- Increasing the size of the input will increase the Maximum Square Error for the FFT.
- For number sizes 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.
- 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.
Questions?

References

  http://numbers.computation.free.fr/Constants/Algorithms/fft.html