# FAST MULTIPLICATION

YIPING DENG

ABSTRACT. The aim of this paper is to give a analysis of several Fast Multiplication algorithms and their asympototic behaviors.

## 1. Introduction

Multiplication of large integers were considered relatively expensive operation in the past. With the modern approach of algorithmic design, faster algorithms are developed so that multiplication of two large integers can be computed easily.

## 2. Basic Assumptions for analysis of Fast Multiplication

In this paper, we will assume that multiplication, addition, subtration and division of two single digit numbers, with carry, can be achieved in constant time.[1]

Based on the assumption, we can conclude:

**Lemma 2.1.** *(Time Complexity of Addition). Addition of two $n$ digits number can be achieved in $O(n)$, denoted $T(n) = O(n)$*

*Proof.* Consider $n = 1$, addition can be achieved by constant time, thus $T(1) = O(1)$. Inductively, assume $T(n) = O(n)$ holds for $n$, consider the addition of two $n+1$ digits numbers, $a$ and $b$ in base $B$. Define

$$S_1 = \left\lfloor \frac{a}{B^n} \right\rfloor + \left\lfloor \frac{b}{B^n} \right\rfloor$$

$$S_2 = a \mod B^n + b \mod B^n$$

The division and the modulo operation on such a number can be achieved in constant time. Calculation of $S_2$ involves in addition of two $n$ digit number. Using such expression, we can rewrite the addition

$$a + b = S_1 \cdot B^n + S_2$$

The additon of $S_1$ and $S_2$ [2]can also be achieved in constant time, resulting in the following expression

$$T(n+1) = T(n) + \mathit{constant}$$

Hence, by induction hypothesis,

$$T(n+1) = O(n) + \mathit{constant} = O(n+1)$$

□

Also consider multiplying one single digit integer with a $n$ digit integer in base $B$

*Key words and phrases.* Fast multiplication, FFT, Polynomial Multiplication.

[1]In modern computer, addition and multiplication of two 64 bits numbers can be calculated in one CPU cycle

[2]Take out the $n+1$ digits of $S_2$ and perform the carried single digit addition when calculating $S_1$

**Lemma 2.2.** *(Single Digit Multiply $n$ Digits) Multiplication of a single digit integer with a $n$ digit integer has a time complexity of $O(n)$.*

*Proof.* It is obvious for the case $n = 1$.

Inductively, given $a$ a single digit number in base $B$, $b$ a $k + 1$ digits number in base $B$ assume Lemma 2.2 holds for $n = k$. we compute

$$P_1 = a \cdot \left\lfloor \frac{b}{B^k} \right\rfloor$$

$$P_2 = a \cdot (b \mod B^k)$$

$$a \cdot b = P_1 \cdot B^k + P_2$$

Indeed,

$$T(k+1) = T(k) + \mathit{constant} = O(k) + \mathit{constant} = O(k+1)$$

□

## 3. Naive Approch

Grade-school approach is the most intuitive algorithm for integer multiplication. Such a approach works for integers in any basis. Without loss of generality, in algorithmic analysis, integers in the base of 2 is considered.

Using long multiplication, first we have $n$ single digit to n digits multiplication, each with time complexity of $O(n)$. Then we have $n - 1$ addition of $n$ at most $2n$ digits integer, with each addition of time complexity $O(2n)$. In total, it will give us a running time of

$$T(n) = n \cdot O(n) + (n-1) \cdot O(2n) = O(n^2)$$

## 4. Recursive Approach

Without loss of generality, given two binary integer $x$, $y$, with $n$ digits, $n = 2^k, k \in \mathbb{N}$, we decompose $x$, $y$ by their lower and higher halfs of the digits.

$$x = x_l * 2^{n/2} + x_r$$

$$y = y_l * 2^{n/2} + y_r$$

Thus, we rewrite $x * y$ as

$$\begin{aligned} x \cdot y &= (x_l \cdot 2^{n/2} + x_r) \cdot (y_l \cdot 2^{n/2} + y_r) \\ &= x_l y_l \cdot 2^n + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r \end{aligned}$$

Note that $x_l$, $x_r$, $y_l$, $y_r$ each contains $n/2$ digits. The addition in between takes $O(n)$ times in total, so the recursive approach will lead to a time complexity of

$$T(n) = 4T(n/2) + O(n)$$

solving the recurrence we have

$$T(n) = O(n^2)$$

Suprisingly, our recursive approach is no better than our grade-school approach. This is where the Gauss trick comes into play.

## 5. Gauss Trick

The mathematician Carl Friedrich Gauss once noticed that multiplication of two complex number, $z_1 = a + bi$, $z_2 = c + di$.

$$z_1 \cdot z_2 = ac - bd + (ad + bc) \cdot i$$

although seems to involve four read-number multiplication, it can in fact be done with just three: $ac$, $bd$, $(a + b)(c + d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd$$

Applying his trick to our multiplication problem leads to a asympotically faster recursive algorithm.

## 6. Recursive Approach Improved

Similar to Gauss' trick for complex number, we observe that

$$x_l y_r + x_r y_l = (x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r$$

This immediately changes the recurrence relation, since our multiplication can be achieved using 3 sub-multiplication. The resulting algorithm has a improved running time of [3]

$$T(n) = 3T(n/2) + O(n)$$

Again, solving the recurrence will give us

$$T(n) = O(n^{log_2 3})$$

where $log_2 3 \approx 1.59$.

Finally, the eternal question: *Can we do better?* It turns out that we can use fast Fourier transform to achieve a even faster algorithm.

## 7. The fast Fourier transform

We extend our discussion of multiplication from integer to polynomial. The product of two degree-$d$ polynomials is a polynomial of degree $2d$, for example:

$$(1 + 2x + 3x^2)(2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4$$

More generally, given two degree-$d$ polynomial $A(x) = a_0 + ... + a_d x^d$ and $B(x) = b_0 + ... + b_d x^d$, their product $C(x) = A(x) \cdot B(x) = c_0 + ... + c_{2d} x^{2d}$ has coefficients

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

Computing $c_k$ takes $O(k)$ steps. Finding all $2d + 1$ coefficients would require $\Theta(n^2)$ time. *Can we multiply polynomial faster than this?*

The resulting algorithm is called fast Fourier transform.

[3]Actually, the recurrence should read

$$T(n) \leq 3T(n/2 + 1) + O(n)$$

The one we are using has the same big-O running time.

**Algorithm 1:** fast Fourier Transform(polynomial formulation)[1]

```
Function FFT (A, ω):
    Input: Coefficient representation of polynomial $A(x)$ of degree $n-1$,
           where $n$ is a power of 2; $\omega$, an $n$th root of unity
    Output: Value representation of $A(\omega^0), A(\omega^1), ..., A(\omega^{n-1})$
    if $\omega = 1$ then
        return $A(1)$;
    end
    express $A(x)$ in the form of $A_e(x^2) + xA_o(x^2)$;
    call FFT ($A_e$, $\omega^2$) to evaluate $A_e$ at even power of $w$;
    call FFT ($A_o$, $\omega^2$) to evaluate $A_o$ at even power of $w$;
    for $j = 0$ to $n-1$ do
        compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$;
    end
    return $A(\omega^0), ..., A(\omega^{n-1})$;
```

FFT will transform a polynomial from coefficient representation to value representation where the points $\{x_i\}$ are $n$th root of unity.

$$\langle \textit{values} \rangle = FFT(\langle \textit{coefficients} \rangle, \omega)$$

The last remaining puzzle is the inverse operation. It turns out, amazingly, that

$$\langle \textit{coefficients} \rangle = \frac{1}{n} FFT(\langle \textit{values} \rangle, \omega^{-1})$$

Algorithm 1 has two recursive call, with each recursive call reducing the size $n$ by half. The remaining operation is bounded by $O(n)$, leading to a overall running time of $T(n) = 2T(n/2) + O(n) = n \log n$.

## 8. Integer multiplication using FFT

Indeed, we can consider two arbitray $n$-digit integers $x$, $y$ in base $b$ in the following manner. Contruct two polynomial

$$X(z) = \sum_{i=1}^{n} x_i z^{i-1}$$

$$Y(z) = \sum_{i=1}^{n} y_i z^{i-1}$$

where $x_i$ and $y_i$ represent, accordingly, the $i$th digit of $x$ and $y$ from the right. Note that $X(b) = x$, $Y(b) = y$. Using our fast Fourier transform, we can calculate $X(z) \cdot Y(z)$ in its coefficient representation relatively cheap. First convert both into its value representation, calculate its componentwise product of the values, and convert it back to coefficient representation.

Finally, once we obtained $P(z) = X(z) \cdot Y(z)$, evaluating $P(b)$ will give you exactly the multiplication of the integer.

## References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

P.O 182 College Ring 7, 28759 Bremen, Germany
*E-mail address*: y.deng@jacobs-university.de