Analysis of Large Integer Multiplication Algorithms
18-647: Blog Post

Aman Mohanty
Carnegie Mellon University

When was the last time you multiplied two large numbers using pen and paper? Let's give it a try shall we.

$$\begin{array}{cccccc} & & & 2 & 4 & 3 \\ & & \times & 1 & 7 & 9 \\ \hline & & 2 & 1 & 8 & 7 \\ & 1 & 7 & 0 & 1 & \\ & 2 & 4 & 3 & & \\ \hline & 4 & 3 & 4 & 9 & 7 \end{array}$$

Well, these numbers are not "large" but these work. This is a quadratic operation which was taught to us during our school days. It's quadratic because each digit of one number is multiplied with every digit of the other number. So, it's $\mathcal{O}(n^2)$. What this notation means is that if the system configurations, compiler etc. were kept constant, time taken by this algorithm will scale quadratically with the number of digits of the numbers (to make things easier its the longest number between the two). Figure 1 depicts this trend. Do our modern day computers and compilers use this method? If I were to multiply this pair of number using, say, Python, result would be on the screen no sooner than I ran the code. But would it be the same for large numbers, say numbers with about 1 million digits? This is what we will investigate in this blog. Let's begin shall we?

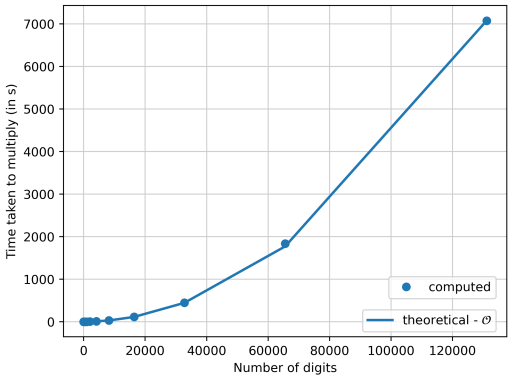


Figure 1: Runtime of quadratic algorithm.

Why is large number multiplication important?

Multiplication is the core of many mathematical, scientific and computational problems. Researchers have been busy throughout the later part of the 20^{th} century to discover the fastest multiplication algorithm. It was believed that the quadratic algorithm was the fastest until 1960 when Anatoly Karatsuba discovered an algorithm with complexity $\mathcal{O}(n^{1.58})$. Researchers believed that there was even a faster algorithm and in the last 50 years researchers discovered faster algorithms. In 2019, David Harvey and Joris van der Hoeven discovered the fastest algorithm with complexity $\mathcal{O}(n \log n)$.

Experiment setup

I ran all my experiments on Pittsburgh Supercomputing Center's Bridges-2 supercomputer. The regular memory (RM) nodes have 256GB RAM and Two AMD EPYC 7742 CPUS with 256MB L3 Cache. I have used Python3.8 to code the algorithms and have used the *random*, *time* and *csv* packages. These packages come with Python3.8 installation.

The naive algorithm - Quadratic

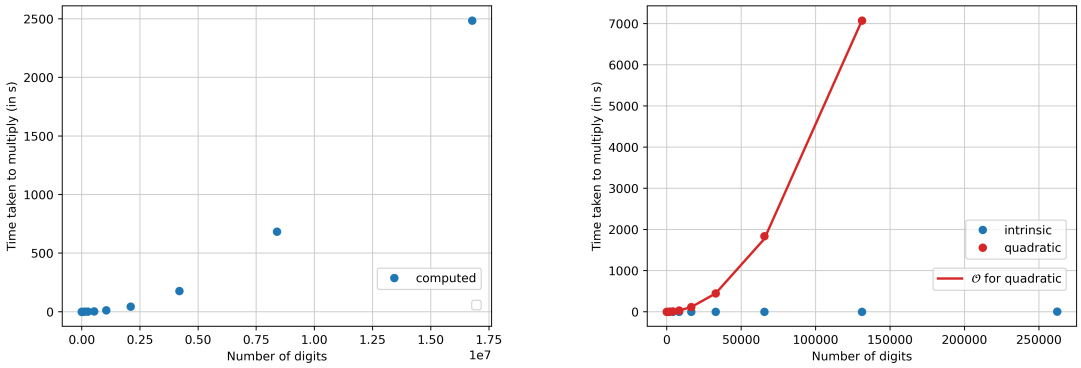
Let's start with the baseline naive quadratic algorithm. The below code snippet is a realization of the algorithm. The numbers are encoded in strings, the result is also encoded in a string. Figure 1 depicts the computed runtime (circles) of the quadratic algorithm along with the theoretical complexity (line). For numbers with more than 120,000 digits, the computation time is about 3 hours. I ran the code for 10 hours and these are the numbers that could be covered. Will the Python intrinsic multiplication operator take the same time?

```
def quadratic(num1, num2):
    """
    Quadratic algorithm
    """

    if len(num1) == 0 or len(num2) == 0:
        return "0"
    res = [0] * (len(num1) + len(num2))
    ind_n1 = 0

    for i in range(len(num1) - 1, -1, -1):
        c = 0
        n1 = ord(num1[i]) - 48
        ind_n2 = 0

        for j in range(len(num2) - 1, -1, -1):
            n2 = ord(num2[j]) - 48
            prod = n1*n2 + res[ind_n1+ind_n2] + c
            c = prod // 10
            res[ind_n1+ind_n2] = prod % 10
            ind_n2 += 1

        res[ind_n1+ind_n2] += c
        ind_n1 += 1

    i = len(res) - 1
    while (i >= 0 and res[i] == 0):
        i -= 1
    if (i == -1):
        return "0"

    s = ""

    while (i >= 0):
        s += chr(res[i] + 48)
        i -= 1

    return s
```

Intrinsic algorithm

We use Python3.8 multiplication operator $*$. The numbers are generated as strings, and then casted into integer before multiplying. Figure 2(a) depicts runtime of the intrinsic algorithm. Figure 2(b) compares runtime of the algorithm to that of the quadratic algorithm. Clearly, Python intrinsic multiplication algorithm is way faster than the quadratic algorithm. Now, the question is "Is this the fastest algorithm?".

```
def python(x,y):
    """
    Python internal multiplication
    """
    return int(x)*int(y)
```



(a) Runtime (b) Compared to quadratic algorithm

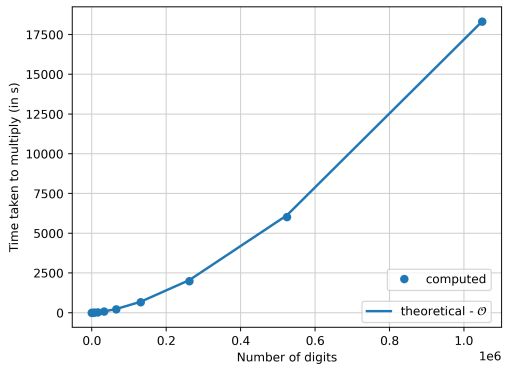
Figure 2: Runtime of intrinsic python multiplication operator.

Karatsuba algorithm

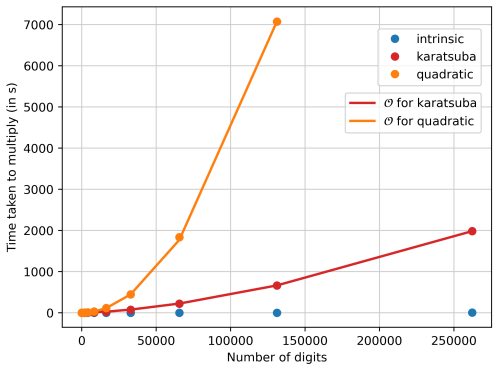
As stated before, in 1960 Karatsuba discovered an algorithm that was faster than the quadratic algorithm. Karatsuba's method involves breaking up the digits of a number and recombining them in a novel way that allows you to substitute a small number of additions and subtractions for a large number of multiplications. The method saves time because addition takes only $2n$ steps, as opposed to n^2 steps. Let's see if the Python intrinsic multiplication algorithm does better than the Karatsuba algorithm. Below code snippet demonstrates Karatsuba's algorithm. If either number is only one digit long, we perform the regular intrinsic multiplication otherwise we break the number using Karatsuba's method. Figure 3(a) depicts runtime of the karatsuba algorithm. Figure 3(b) compares runtime of the algorithm to that of quadratic and intrinsic algorithm. We can observe that karatsuba algorithm performs better than the quadratic algorithm but the python intrinsic multiplication operator beats the karatsuba algorithm by a fair margin.

```
def karatsuba(x,y):
    """
    Karatsuba algorithm
    """

    num1 = int(x)
    num2 = int(y)
    if len(x) == 1 or len(y) == 1:
```

```
        return num1*num2
    else:
        m = max(len(x),len(y))
        m2 = m // 2

        a = num1 // 10**(m2)
        b = num1 % 10**(m2)
        c = num2 // 10**(m2)
        d = num2 % 10**(m2)

        z0 = karatsuba(str(b),str(d))
        z1 = karatsuba(str(a+b),str(c+d))
        z2 = karatsuba(str(a),str(c))

        return str( (int(z2) * 10**(2*m2)) + ((int(z1) - int(z2) - int(z0))
    * 10**(m2)) + int(z0) )
```



(a) Runtime



(b) Compared to quadratic/intrinsic algorithm

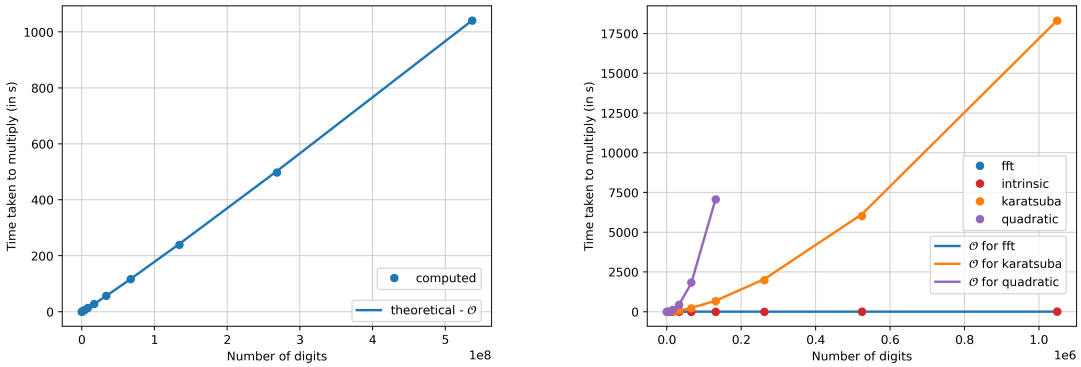
Figure 3: Runtime of karatsuba algorithm.

FFT algorithm

Finally, we arrive at David Harvey and Joris van der Hoeven's algorithm. Their method is a refinement of the major work that came before them. It splits up digits, uses an improved version of the fast Fourier transform, and takes advantage of other advances made over the past forty years. Below code snippet demonstrates an easy implementation of the FFT algorithm using numpy's fft module. Figure 4(a) depicts runtime of the FFT algorithm. It looks linear, doesn't it? But is it really linear? Well, its not. It's actually $\mathcal{O}(n \log n)$ and we will see the difference when it is plotted with lot more data points. Currently, the longest number is 536,870,912 and still it shows linear. Figure 4(b) compares runtime of the algorithm to that of the other algorithms. It can be observed that the algorithm's runtime matches with that of the intrinsic algorithm. So, Does Python3.8 uses FFT based multiplication? Actually, no. FFT is faster than python intrisic multiplication operator. Figure 5 depicts this trend. This figure is based on log scale to render better visualization. We can observe that intrinsic mulitplication operator is faster than the fft algorithm for numbers with about 100,000 digits. But beyond that the FFT algorithm crushes the intrinsic multiplication operator.

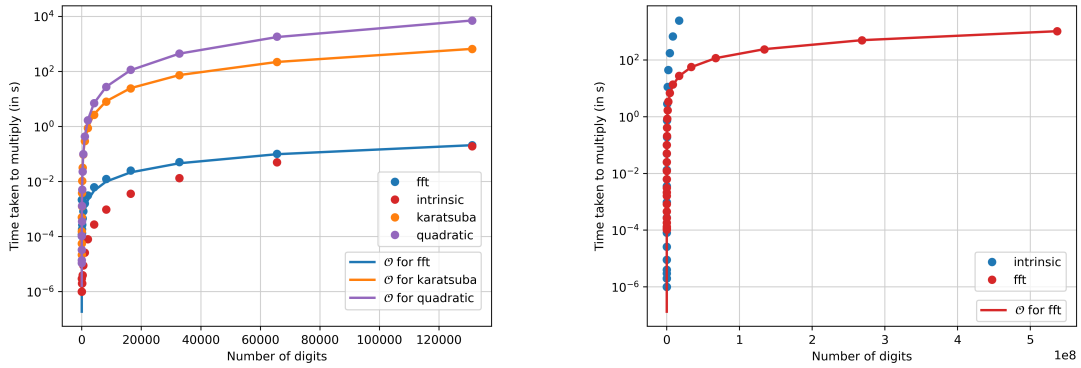
```
def fft(num1,num2):
    """
```

```
    FFT algorithm
    """
    N = len(num1)
    arr_a1=np.pad(num1,(0,N),'constant')
    arr_b1=np.pad(num2,(0,N),'constant')
    a_f=np.fft.fft(arr_a1)
    b_f=np.fft.fft(arr_b1)

    c_f=[0]*(2*N)

    for i in range( len(a_f) ):
        c_f[i]=a_f[i]*b_f[i]

    return np.fft.ifft(c_f)
```



(a) Runtime (b) Compared to other algorithms

Figure 4: Runtime of FFT algorithm.



(a) All algorithms (b) Intrinsic and FFT

Figure 5: Runtime comparison on log scale

Should we use FFT based algorithm for our operations?

How many times have you multiplied numbers with more than 100,000 digits? 10,000 digits? 100 digits? 50 digits? Seldom do we even multiply numbers with more than 10 digits. Then, does all these algorithms really matter? For us, with our day to day operations? No. Figure 6 depicts the runtime of these algorithms on smaller numbers (numbers with less than 50 digits). Intrinsic multiplication operator outperforms all other algorithms. FFT, supposedly the fastest algorithm, performs the worst for numbers with less than 5 digits, the range that we use the most. The intrinsic multiplication takes micro seconds to get the job done for numbers with less than 20 digits. So, unless you are dealing with numbers with more than 100,000 digits, the intrinsic multiplication operator, should get the job done for you.

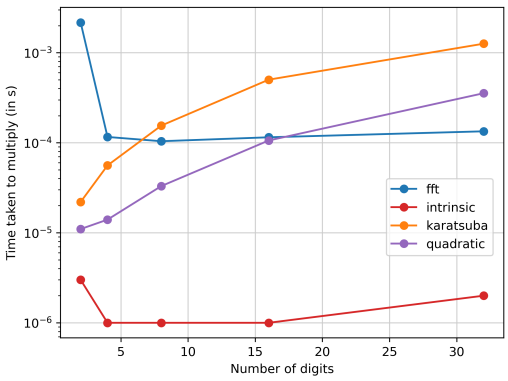


Figure 6: Runtime of different algorithms on log scale with maximum number of digits under 50.

Link to the code implementation: `https://github.com/amanmohanty/mult-analysis-blog`