关于乘法算法的认识

作者：许兆晖　　　　　　　学号：1900011330　　　　　　　辅导老师:陈斌
（北京大学物理学院 2019 级本科 6 班）

【摘要】乘法算法是当今离散数学各类算法的基础，在计算领域具有无可替代的基石地位，然而真正意义上的现代乘法算法却直到上世纪 60 年代才有所发展,并在近几年达到其巅峰。本文回顾了乘法算法的历史发展，并根据实验寻找适用于家用计算机的合理运算方式。
【关键词】乘法算法，快速傅里叶变换

【正文】

1．历史回顾[1]

作为四则运算之一的乘法有着悠久的历史：古巴比伦人于公元前 4000 年发明了乘法，随后各个古代文明也相继独立发展出了自己的乘法计算，其中较为典型的有竖式乘法。然而尽管乘法算法在长达 5000 年的时间里有所演进，通用乘法算法却始终无法突破$O(n^2)$的效率极限，以至于 1960 年数学家 Andrey　Kolmogoro 在一次研讨会上声称不会存在比这更高效的通用乘法算法。

然而，仅仅在一周后，与会者之一的年轻数学家 Karatsuba 便发现了效率更高的乘法算法，其时间复杂度约为$O(n^{1.58})$，在随后的 10 年里，新的算法不断将这个指数推向 1。1971 年 Schönhage 与 Strassen 基于快速傅里叶变换(FFT)提出了时间复杂度为$O(n\log n\log\log n)$的算法，他们猜想最快的乘法算法将具有$O(n\log n)$的时间复杂度。最终在 2019 年，Harvey 等人触及了这个乘法算法的“圣杯”。但是，尽管$O(n\log n)$已经被广泛认为是乘法算法的时间复杂度极限，其证明仍然是未解决的问题。

2．基础乘法算法与 Toom-Cook 类乘法算法

基础乘法算法，无论是竖式乘法还是其变种，均可以等价于包含进位的多项式卷积，也即，设$A=\sum_{k=0}^{n}a_kr^k$、$B=\sum_{k=0}^{n}b_kr^k$，则有$C=A\times B=\sum_{k=0}^{n}\left(\sum_{p=0}^{k}a_pb_{k-p}\right)r^k$，其中假设$a_k$与$b_k$均小于 r，并且 r 以内的乘法是已知的（称为基本乘法），因而也被视作时间复杂度的衡量单位。以上内容可以表达为伪代码：

```
input a[n], b[n]
# convolution
for k in range(n)
    c[k] = 0
    for p in range(k)
        c[k] += a[p] * b[k - p]
# carry-on
for k in range(n)
    c[k + 1] += c[k] / r
    c[k] = c[k] mod r
output c[n]
```

Karatsuba 于 1960 年发现并于 1962 年发表了基于二分法思想的 Karatsuba 乘法算法。其基本想法来源于两位数乘法，也即$A = a_0 + a_1 r$、$B = b_0 + b_1 r$，我们有$C = A \times B = a_0 b_0 + (a_1 b_0 + a_0 b_1) r + a_1 b_1 r^2$，从表达式形式来看自然需要进行 4 次一位数的乘法。一个很自然的想法是希望利用已有的乘法结果减少乘法所需次数，注意到交叉项$a_1 b_0 + a_0 b_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$，因此只需进行 3 次一位数乘法和减法，后者相对时间可以忽略不计，便可以完成计算[2]。对于一般情况，只需使用二分递归的方法即可以完成优化，可以表示为伪代码：

```
function Karatsuba(a[m], b[m])
    if m < THERSHOLD
        # bottom out
        return Basecase(a[m], b[m])
    # bisecting
    a0[m/2], a1[m/2] = a[0: m/2], a[m/2 : m]
    b0[m/2], b1[m/2] = b[0: m/2], b[m/2 : m]
    c0[m/2] = Karatsuba(a0[m/2], b0[m/2])
    c1[m/2] = Karatsuba(a0[m/2] + a1[m/2], b0[m/2] + b1[m/2])
    c2[m/2] = Karatsuba(a1[m/2], b1[m/2])
    return c0[m/2] + (c1[m/2] - c0[m/2] - c2[m/2]) * r + c2[m/2] * r^2
input a[n], b[n]
output Karatsuba(a[n], b[n])
```

由于原来每 4 次乘法计算现在被 3 次乘法计算所取代，因此 Karatsuba 乘法的时间复杂度$M(n)$满足的方程是：$M(n) = 3M\left(\frac{n}{2}\right)$。其解为$O(n^{\log_2 3}) = O(n^{1.58})$，相比于基础乘法在渐进意义上有了很大进步。值得一提的是，Python 中的整数乘法使用的正是 Karatsuba 乘法。

Toom-Cook 类算法最早由 Toom 于 1963 年提出、Cook 于 1966 年改进，其基本思想与 Karatsuba 算法一致，但是更为复杂。具体来说，以最早的 Toom-3 为例[3]：

$$A = a_0 + a_1 r + a_2 r^2$$
$$B = b_0 + b_1 r + b_2 r^2$$

$$w_0 = a_0 b_0$$
$$w_1 = (a_0 + a_1 + a_2)(b_0 + b_1 + b_2)$$
$$w_2 = (a_0 - a_1 + a_2)(b_0 - b_1 + b_2)$$
$$w_3 = (a_0 + 2a_1 + 4a_2)(b_0 + 2b_1 + 4b_2)$$
$$w_4 = a_2 b_2$$

$$C = w_0 - \frac{3w_0 - 6w_1 + 2w_2 + w_3 - 12w_4}{6k} r - \frac{2w_0 - w_1 - w_2 + 2w_4}{2} r^2 + \frac{3w_0 - 3w_1 - w_2 + w_3 - 12w_4}{6} r^3 + w_4 r^4$$

其中 9 次乘法运算被 5 次乘法运算代替，同理求得时间复杂度为$O(n^{\log_3 5}) = O(n^{1.46})$。

注意到以上表达式还可以以一种完全不同的方式来理解：将 A 和 B 视作 r 的多项式，则$w_0=A(0)B(0)$，$w_1=A(1)B(1)$，$w_2=A(-1)B(-1)$，$w_3=A(2)B(2)$，$w_4=\lim_{t\to\infty}\frac{A(t)B(t)}{t^4}$，然后由$C(r)=A(r)B(r)$反解出系数。以上分析显然暗示着更多将多项式卷积转化为逐点积的算法的存在，也为乘法算法进一步的发展指明了前进的方向。

3．快速傅里叶变换与快速数论变换

将多项式卷积转化为逐点积的意义是显然的：前者具有$O(n^2)$的时间复杂度，而后者仅为$O(n)$，Toom-Cook 类算法（Karatsuba 算法显然为其特例）均按照这样的思路进行。一般来说，若取点为q_k，为了在点式与多项式系数之间相互转换，需要解 n 元线性方程：

$$\begin{pmatrix}1 & q_0 & \cdots & q_0^{n-1}\\ 1 & q_1 & \cdots & q_1^{n-1}\\ \vdots & \vdots & \ddots & \vdots\\ 1 & q_{n-1} & \cdots & q_{n-1}^{n-1}\end{pmatrix}\begin{pmatrix}f_0\\ f_1\\ \vdots\\ f_{n-1}\end{pmatrix}=\begin{pmatrix}F(q_0)\\ F(q_1)\\ \vdots\\ F(q_{n-1})\end{pmatrix}$$

一般来说完成这样的乘法仍然需要$O(n^2)$次基本乘法，相比于基础乘法算法没有优化。然而，借助快速傅里叶变换算法，我们可以将点式与多项式系数之间转换的时间复杂度减小为$O(n\log n)$。具体来说，我们取$q_k=e^{\frac{2ik\pi}{n}}$，则有：

$$\begin{pmatrix}F_0\\ F_1\\ \vdots\\ F_{n-1}\end{pmatrix}=\begin{pmatrix}1 & 1 & \cdots & 1\\ 1 & e^{\frac{2i\pi}{n}} & \cdots & e^{\frac{2i(n-1)\pi}{n}}\\ \vdots & \vdots & \ddots & \vdots\\ 1 & e^{\frac{2i(n-1)\pi}{n}} & \cdots & e^{\frac{2i(n-1)(n-1)\pi}{n}}\end{pmatrix}\begin{pmatrix}f_0\\ f_1\\ \vdots\\ f_{n-1}\end{pmatrix}$$

与：

$$\begin{pmatrix}f_0\\ f_1\\ \vdots\\ f_{n-1}\end{pmatrix}=\frac{1}{n}\begin{pmatrix}1 & 1 & \cdots & 1\\ 1 & e^{-\frac{2i\pi}{n}} & \cdots & e^{-\frac{2i(n-1)\pi}{n}}\\ \vdots & \vdots & \ddots & \vdots\\ 1 & e^{-\frac{2i(n-1)\pi}{n}} & \cdots & e^{-\frac{2i(n-1)(n-1)\pi}{n}}\end{pmatrix}\begin{pmatrix}F_0\\ F_1\\ \vdots\\ F_{n-1}\end{pmatrix}$$

即离散傅里叶变换(DFT)的形式。再取$n=2^m$，此时我们发现每一行乘法均可以被以二分法优化，具体来说：

$$F_k=\sum_{u=0}^{2^m-1}f_ue^{\frac{iuk\pi}{2^{m-1}}}=\sum_{u=0}^{2^{m-1}-1}f_{2u}e^{\frac{iuk\pi}{2^{m-2}}}+e^{\frac{ik\pi}{2^{m-1}}}\sum_{u=0}^{2^{m-1}-1}f_{2u+1}e^{\frac{iuk\pi}{2^{m-2}}}=G_k+e^{\frac{ik\pi}{2^{m-1}}}H_k$$

$$F_{k+2^{m-1}}=\sum_{u=0}^{2^m-1}(-1)^uf_ue^{\frac{iuk\pi}{2^{m-1}}}=\sum_{u=0}^{2^{m-1}-1}f_{2u}e^{\frac{iuk\pi}{2^{m-2}}}-e^{\frac{ik\pi}{2^{m-1}}}\sum_{u=0}^{2^{m-1}-1}f_{2u+1}e^{\frac{iuk\pi}{2^{m-2}}}=G_k-e^{\frac{ik\pi}{2^{m-1}}}H_k$$

也即计算两组$n=2^{m-1}$的离散傅里叶变换即可得到所求。重复以上操作直至$m=0$，可以得到时间复杂度$M(n)$满足方程：

$$M(n)=2M\left(\frac{n}{2}\right)+O(n)$$

解得$M(n)=O(n\log n)$，此即快速傅里叶变换(FFT)。具体实现伪代码如下：

```
function FFT(f[n])
    if n == 1
```

```
        return f[0]
    for k in range(n/2)
        g[k] = f[2k]
        h[k] = f[2k + 1]
    G[n] = FFT(g[n/2])
    H[n] = FFT(h[n/2])
    for k in range(n/2)
       F[k] = G[k] + exp(i * 2 * k * Pi / n) * H[k]
       F[k + n/2] = G[k] - exp(i * 2 * k * Pi / n) * H[k]
    return F[n]
# IFFT is similar and therefore neglected
input a[n], b[n]
A[n], B[n] = FFT(a[n]), FFT(b[n])
for k in range(n)
    C[k] = A[k] * B[k]
c[n] = IFFT(C[n])
# carry-on
for k in range(n)
    c[k + 1] += c[k] / r
    c[k] = c[k] mod r
output c
```

（其中递归可进一步用蝴蝶变换优化，但是这不涉及算法阶的改变）

然而注意到，快速傅里叶变换涉及了实数运算，因此这样的乘法算法有效输入范围必然受到精度限制。具体来说，对于 d 位二进制定点精度，计算过程中产生的最小定点数为$e^{\frac{i\pi}{2^{m-1}}} \sim 1 + \frac{i\pi}{2^{m-1}}$，为了确保计算结果的正确需要$-\log\frac{\pi}{2^{m-1}} \sim m = \log\left(\frac{n}{\log r}\right) < d$。对一般的 64 位电脑，取$d = 52$（双精度浮点）、$r$=$2^{64}$得到$n \sim 2^{58}$，已经超过计算机的内存存储量，因此是合理的。但如果作为通用算法，我们必须考虑浮点误差带来的影响。假如定点精度根据需要可调整，然后递归使用算法，则可得到算法时间复杂度满足方程：

$$M(n) = O(n\log n)M[O(\log n)]$$

解得算法时间复杂度：

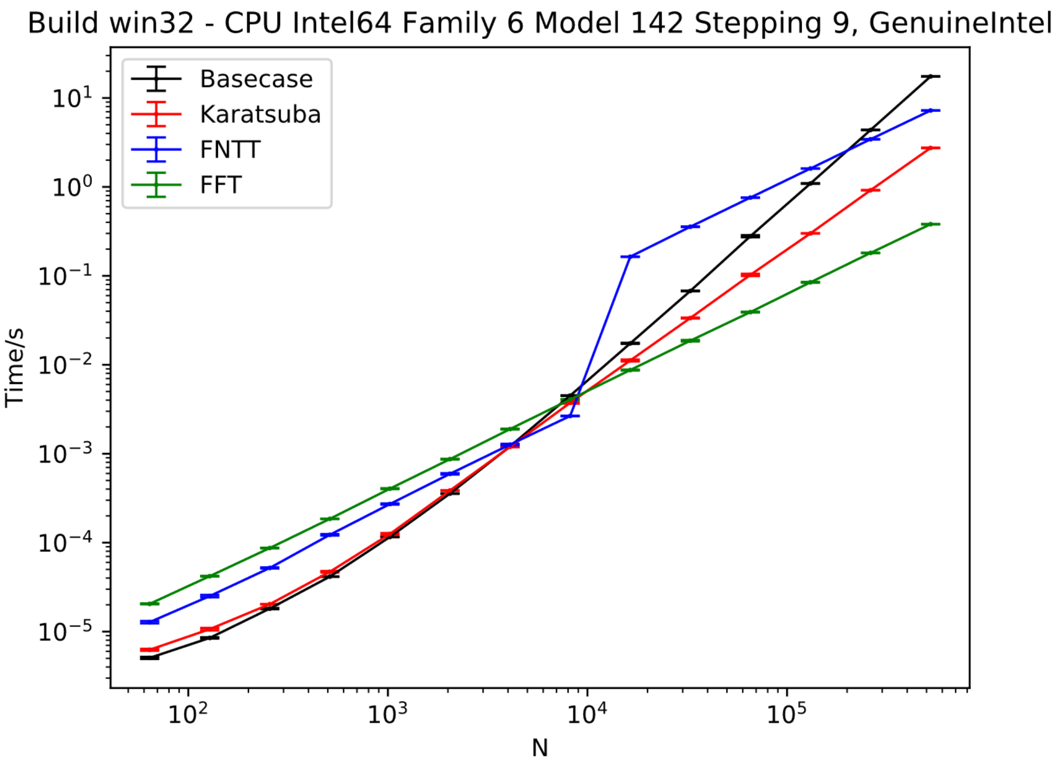
$$O\left(n\log^2 n\log^{(2)2} n \ldots\right)$$

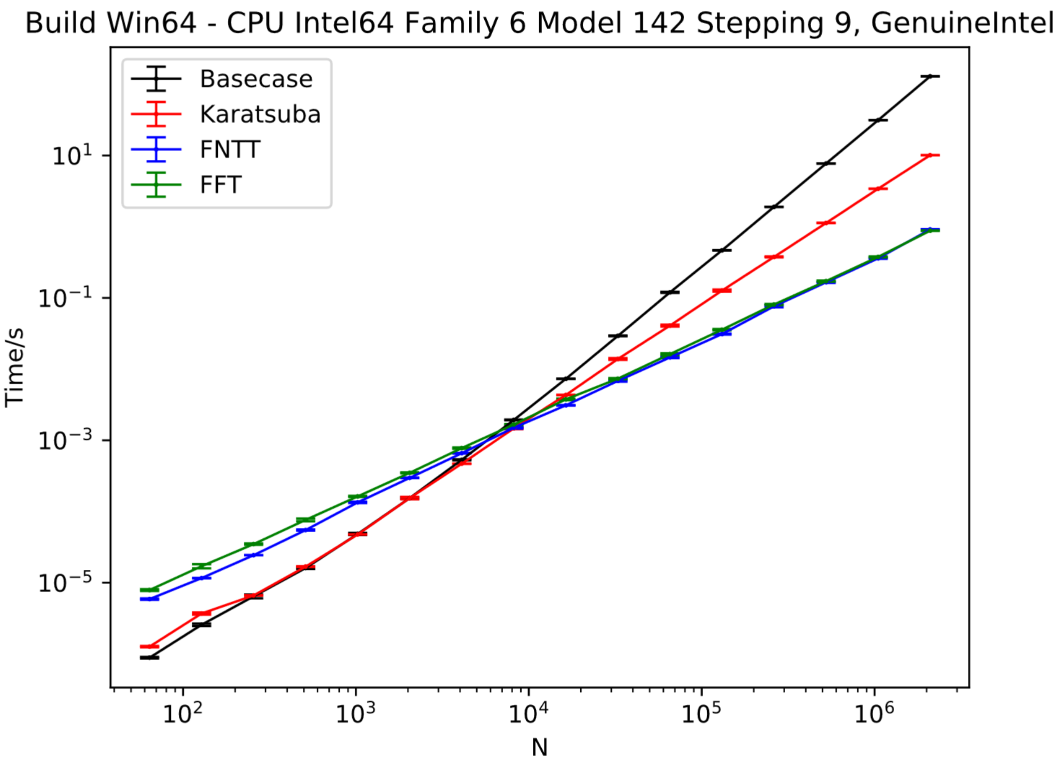
另一方面我们注意到，除了复单位根之外，满足以上性质的还有模$2^k s + 1$下的原根g，用原根幂g^s取代$e^{\frac{i\pi}{2^{m-1}}}$，我们便得到了快速数论变换(FNTT)。基于快速数论变换的整数乘法避免了浮点数误差带来的问题，但是由于其在模$2^k s + 1$下进行，因此一般情况下不能唯一确定乘法的结果，仅有当$2^k s + 1$为素数、$n < 2^k$且$nr^2 < 2^k s + 1$时结果才是唯一确定的。

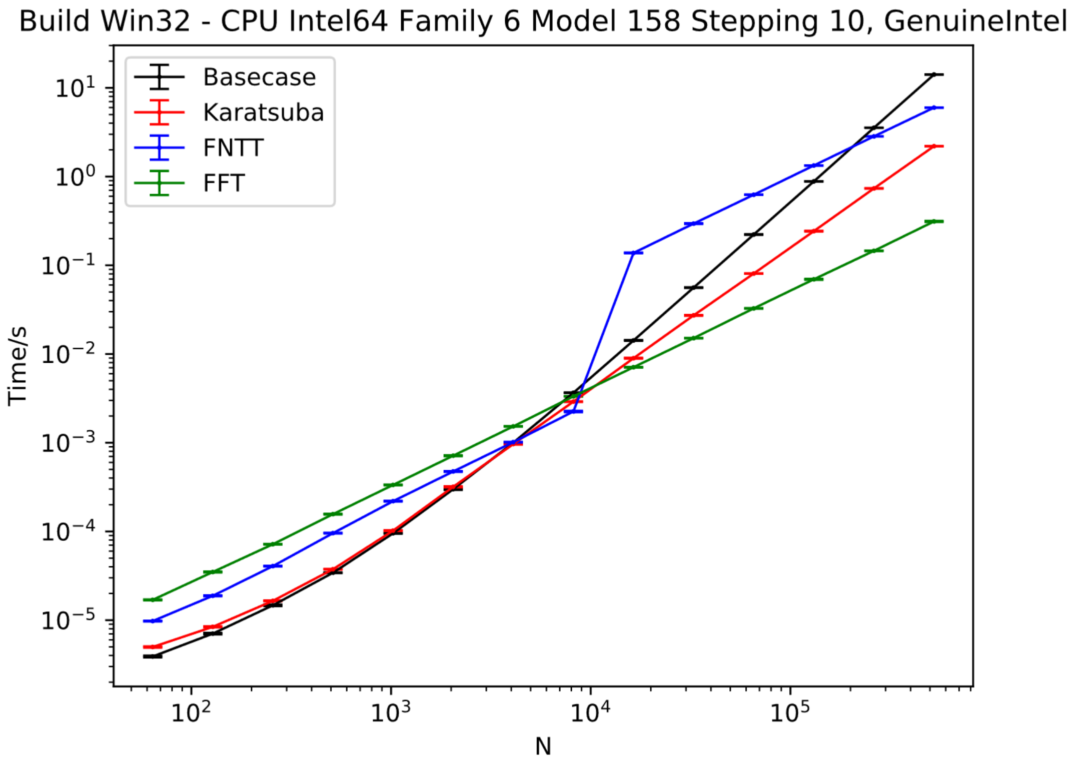
为了突破这样的限制，Strassen 等人通过选取模数$2^k + 1$并使 k 足够大以保证计算过程中不产生超过模数的 c，Strassen 算法的时间复杂度为$\left(n\log n\log^{(2)} n\right)$[4]。

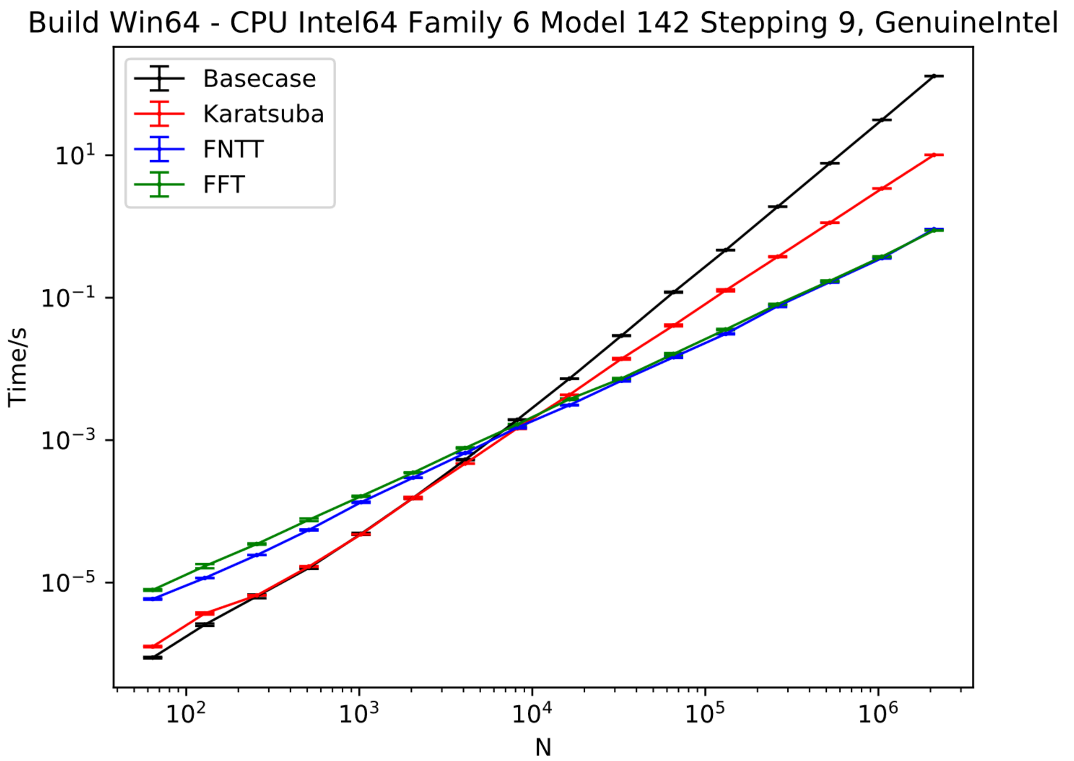
4. 基本乘法实际算法效率对比

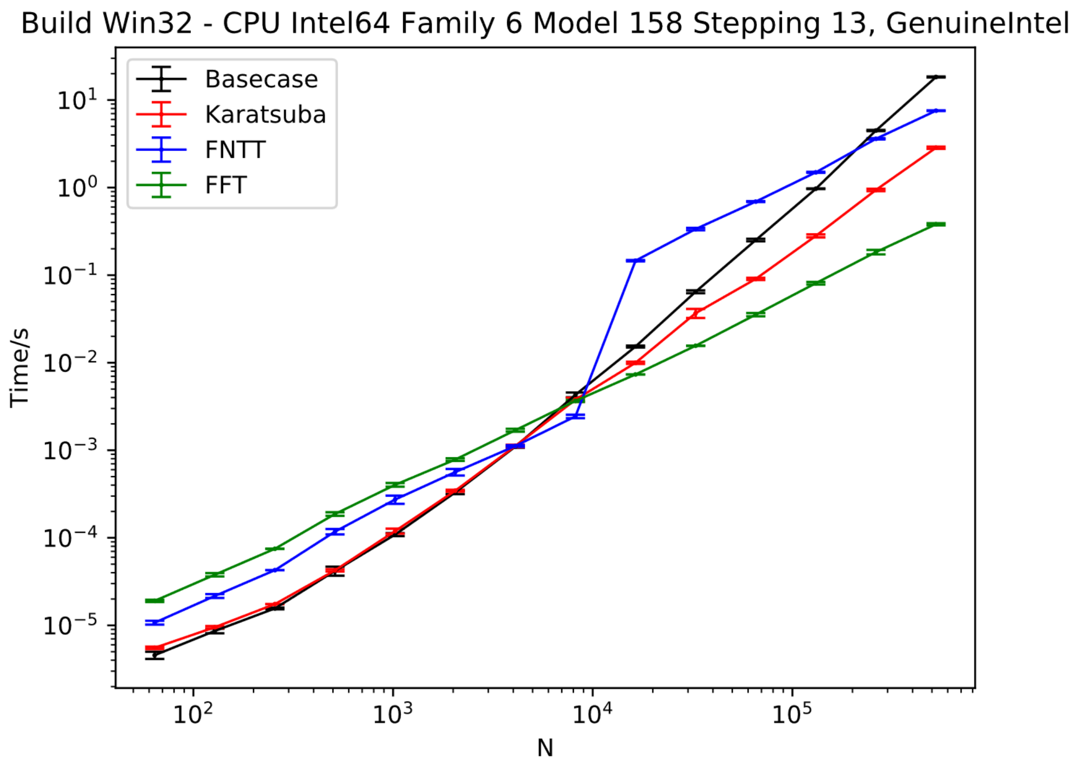
为了保证计算效率及稳定性，程序使用 C 语言编写，并分别于 MinGW-5.1.0-Win32 与 MinGW-8.1.0-Win64 下编译，在三台不同的电脑上运行并收集数据，绘制成如下图：

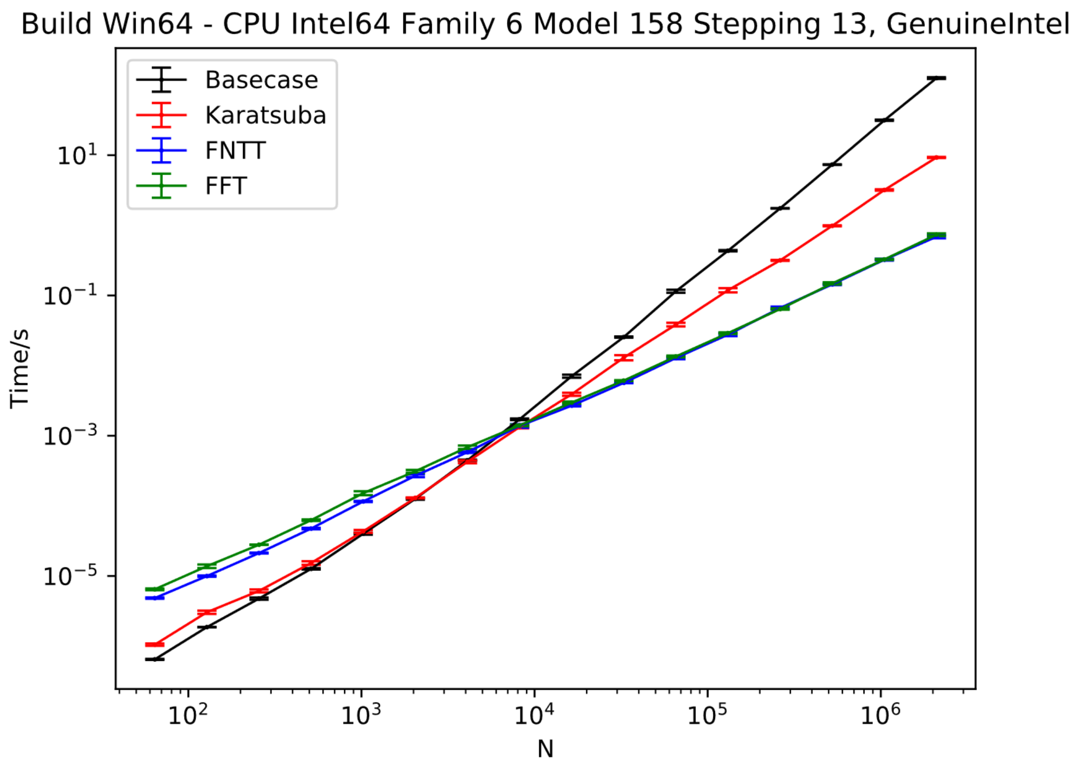

Build win32 - CPU Intel64 Family 6 Model 142 Stepping 9, GenuineIntel
Basecase
Karatsuba
FNTT
FFT
Time/s
N


Build Win64 - CPU Intel64 Family 6 Model 142 Stepping 9, GenuineIntel
Basecase
Karatsuba
FNTT
FFT
Time/s
N


Build Win32 - CPU Intel64 Family 6 Model 158 Stepping 10, GenuineIntel
Basecase
Karatsuba
FNTT
FFT
Time/s
N


Build Win64 - CPU Intel64 Family 6 Model 142 Stepping 9, GenuineIntel
Basecase
Karatsuba
FNTT
FFT
Time/s
N





（横轴计量单位是整型个数，FNTT 在 Win32 下速率发生阶跃系因为模数超过了 32 位整型的限制造成）

总体来说，设备导致的差异除了纵轴相应乘以一个系数之外差别不大。可以得到的结果是，对于 32 位系统下< 2^{4096}（1200 位十进制数）、64 位系统下< 2^{1024}（300 位十进制数）的数，经过合理优化的基础乘法算法仍然是最快的算法。一旦达到千位数级别，Karatsuba 算法便开始渐渐超过基础乘法算法，直到大约2^{8192}（2500 位十进制数）时再次被 FNTT 算

法超越。FNTT 算法相比于 FFT 算法在小数据范围表现良好，而在大数据范围则逐渐趋同乃至被略微超过（$2^{2097152}$大约 63 万位数），这可能系因为 FNTT 需要进行较多取模运算。

可知，在一般的数据范围内基础乘法算法或者 Karatsuba 算法是合适的，而对于超大数据可以考虑 FNTT 和 FTT。

5．迈向更快的乘法算法

2007 年 Fürer 发表了以其名字命名的算法，其工作在复多项式模域，具体来说是$\left(\frac{\mathcal{P}(\mathbb{R})}{(x^r-1)\mathcal{P}(\mathbb{R})}\right)$ 域。算法的原理依然是基于快速傅里叶变换的思想，具有时间复杂度$O(n\log n\cdot 16^{\log^* n})$ [5]。之后多次算法的优化均是基于快速傅里叶变换，差别只是在选取的工作域不同。2019 年 Harvey 发表的算法，原理仍然基于快速傅里叶变换，但其工作域进一步变为多元复多项式模域（事实上是 1729 维！这也使得该算法尽管渐进意义上更优，却没有任何实际的适用范围）[6]。至此，有关寻找乘法算法的努力算是告一段落，接下来的工作集中在证明上。

6．总结

作为计算领域的最基础算法之一，尽管乘法算法的极限看上去已经达到，对常数的优化和进一步的证明工作依然不会停止。乘法算法发展的经历也提醒我们$O(n\log n)$的极限仍然有可能被新的算法突破，尤其是随着科学技术的更新，新计算模型下一切尚未可知。尽管历史悠久，乘法算法亦必将在未来的计算机上大放光彩。

【参考文献】

[1] Hartnett, Kevin (2019-04-14). "Mathematicians Discover the Perfect Way to Multiply". *Wired*. ISSN 1059-1028. Retrieved 2019-04-15.

[2] A. Karatsuba and Yu. Ofman (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". *Proceedings of the USSR Academy of Sciences*. 145: 293–294. Translation in the academic journal Physics-Doklady, 7 (1963), pp. 595–596

[3] D. Knuth. *The Art of Computer Programming*, Volume 2. Third Edition, Addison-Wesley, 1997. Section 4.3.3.A: Digital methods, pg.294

[4] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen", *Computing* 7 (1971), pp. 281–292.

[5] M. Fürer (2007). "Faster Integer Multiplication" *Proceedings of the 39th annual ACM Symposium on Theory of Computing* (STOC), 55–67, San Diego, CA, June 11–13, 2007, and SIAM Journal on Computing, Vol. 39 Issue 3, 979–1005, 2009.

[6] Harvey, David; Van Der Hoeven, *Joris* (2019-04-12). Integer multiplication in time O(n log n).