深 圳 大 学 实 验 报 告

课程名称 算法导论

项目名称 实验一：大整数相乘

学　　院 计算机与软件学院

专　　业 计算机科学与技术

指导教师 刘刚

报 告 人 王 涛 学号 2060271033

实验时间 2020 年 11 月 1 日至 2020 年 12 月 10 日

实验报告提交时间 2020 年 12 月 10 日

教务处制

摘 要

大整数乘法是一个重要的计算工具，广泛应用于大规模科学计算、数论、密码学等多个领域。大整数乘法的主要算法有最简单的竖式乘法，基于分治算法思想，包括 karatsuba[1]算法，Toom-Cook[3]算法，以及利用快速傅里叶变换方法等，最近 Harvey[2]等人实现了一种时间复杂度为$O(n\log n)$的算法，达到了大整数乘法时间复杂度的理论下限。

本实验分别就普通大整数乘法、分治大整数乘法以及改进分治的大整数乘法（karatsuba 算法）进行了实验探究。利用主定理分析了各种算法的复杂度，通过编程实验测试实验结果证明了理论与实际的一致性。

关键词:**大整数乘法；分治算法；主定理法；时间复杂度**

目 录

1 实验要求

1) 编写普通大整数乘法

2) 编写基于分治的大整数乘法

3) 编写改进的基于分治的大整数乘法

4) 编写一个随机产生数字的算法，分别产生两个 10, 100, 1000, 10000, 100000 位的数字 a 和 b，该数字的每一位都是随机产生的

5) 用 4 所产生的数字来测试 1、2 和 3 算法的时间，列出运行时间表格

2 理论分析

2.1 普通大整数乘法

2.1.1 算法原理

给定两个长度分别为n的大整数，记为X和Y，现计算两个数相乘的结果，即:

$$Z = X \times Y \tag{1}$$

一个最直观的算法是直接利用乘法竖式算法则。如下图所示，分别将X中的每一位与Y中的每一位进行乘法运算，将运算结果累加得到最后的乘法运算结果。

$$\begin{array}{r} 3\ 9 \\ \times\ \ 4\ 5 \\ \hline 1\ 9\ 5 \\ 1\ 5\ 6\ \ \\ \hline 1\ 7\ 5\ 5 \end{array}$$

图 2.1 乘法竖式计算

2.1.2 复杂度分析

两个大整数乘法运算需要将X中的每一位与Y中的每一位进行乘法运算，所有结果移位累加，共n^2次乘法和加法运算，故其时间复杂度为：

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

2.2 基于分治的大整数乘法

2.2.1 算法原理

基于分治算法的思想，可将原始问题拆分为诸多子问题，通过求解子问题最终得到原问题的结果。

具体做法是将两个长度为n大整数X和Y分别分解为长度为$n/2$的两部分，即：

$$X = A \times 10^{n/2} + B$$
$$Y = C \times 10^{n/2} + D$$

那么，两个大数乘法运算可转化为：

$$\begin{aligned} X \times Y &= (A \times 10^{n/2} + B) \times (C \times 10^{n/2} + D) \\ &= AC \times 10^{n} + AD \times 10^{n/2} + BC \times 10^{n/2} + BD \\ &= AC \times 10^{n} + (AD + BC) \times 10^{n/2} + BD \end{aligned} \tag{2}$$

通过拆分原始大整数，可以将两个长度为n的大整数相乘运算转化为 4 个长度为$n/2$的大整数相乘运算，以及一些代价很小的移位和加法操作。通过这种分治方式，可以将原始大整数乘法的规模逐渐缩小，当问题规模达到计算机乘法可直接计算的长度后分治结束，为了便于算法分析，当规模降为 1，即乘数长度为 1 时结束分治，这样两个大整数乘法运算就分为很多个基本整数乘法运算。

2.2.2 复杂度分析

忽略上述算法中加法运算和移位运算带来的时间开销，当问题规模降到 1 时，计算的时间复杂度为$O(1)$两个n位大整数相乘，可得到如下递归式：

$$T(n) = \begin{cases} O(1), & \mathrm{n} = 1 \\ 4T(n/2) + O(n), & \mathrm{n} > 1 \end{cases}$$

当$n > 1$时, 根据 Master 定理计算上述递归式的时间复杂度，其中$a = 4, b = 2, f(n) = O(n)$，显然$\exists\ \varepsilon > 0$，使得

$$f(n) = O(n^{\log_b a - \varepsilon}) = O(n^{2-\varepsilon}) = O(n)$$

故上述递归式属于第一类，可得其时间复杂度为：

$$T(n) = O(n^{\log_2 4}) = O(n^2)$$

可以看出，基于分治的大整数乘法与普通的大整数乘法相比时间复杂度上并没有提升，但利用分治算法可以将原始的一个任务转化为多个子任务，这样为并行操作带来了可能，在资源利用等方面会更高效。

2.3 改进分治大数乘法

2.3.1 算法原理

通过 2.2 可以知道，简单的利用分治算法处理大整数乘法，对于单任务处理在算法时间复杂度上并没有改进。但在分治算法的基础上还可以进行进一步的改进，我们知道在上述分治算法中，主要影响时间的是 4 个乘法运算，通过某种方式减少乘法运算对于算法的时间复杂度会有提升。观察式(2)可以发现，可以将$(AD+BC)$更改运算形式从而减少一个乘法运算。将式(2)改写形式后可以得到下面两种结果。

形式一：

$$X\times Y=AC\times 10^{n}+((A-B)(D-C)+AC+BD)\times 10^{n/2}+BD \tag{3}$$

形式二：

$$X\times Y=AC\times 10^{n}+((A+B)(C+D)-AC-BD)\times 10^{n/2}+BD \tag{4}$$

通过改写运算形式，式(2)中的 4 个乘法运算：AC，AD，BC，BD，减少为 3 个乘法运算：AC，BD，$(A-B)(D-C)$或$(A+B)(C+D)$。理论上讲，式(3)和式(4)对于算法的改进是等价的。但在算法实现中，式(4)是要优于式(3)的。其原因在于，式(3)运算时不可避免会出现减数大于被减数的情况:$A<B$或$D<C$，这就意味着运算过程中需要引入符号，而且在大整数减法运算中也需要先判断两个数的绝对值大小。

如果采用式(4)的形式。因为$(A+B)(C+D)-AC-BD>0$在运算过程中不会出现负数，而且减法运算也一直是被减数大于减数。这就可以避免符号问题和减法的绝对值大小判断，从而在算法实现过程中会避免一些不必要的操作。

除此之外，需要注意的是，在分治过程中不可避免的会出现长度为奇数的大整数，这也就带来一个问题：X和Y的长度不等，这种情况下，为了保证AD和BC移位相同进行后续乘法变加减法的转换，需要坚持的原则就是拆分后B和D的长度需要对齐。

2.3.2 复杂度分析

通过改写计算形式，减少一个乘法后，改进分治算法的时间复杂度为：

$$T(n)=\begin{cases} O(1), & \mathrm{n}=1 \\ 3T(n/2)+O(n), & \mathrm{n}>1 \end{cases}$$

同理，由 Master 定理可得：

$$T(n)=O(n^{log_2 3})=O(n^{1.585})$$

通过上述改进，在不引进其他方法操作的前提下，分治算法的时间复杂度有了显著的降低。

3 实验过程

3.1 实验环境

- CPU：Intel(R) Core(TM) i7-9750H CPU @ 2.60 GHz x 12 cores：6
- 内存：2*8G, DDR4 2667Mhz 64bit
- 硬盘：1TB SSD PCI-E
- 操作系统：Windows 10 (x64)
- 编程语言：C++
- 编译器：g++(tdm-1) 5.1.0
- 集成开发环境：Visual Studio Code 1.51.1

3.2 算法实现

3.2.1 数据结构

算法实现过程中，分别尝试了 int 整型数组和 string 两个类型。

1）整型数组

```
typedef struct L_NUM
{
    int len;
    int *num;
} * LNUM;
```

整形数组可以分为定长和变长即通过指针动态操作内存空间。使用定长数组的优点是不必考虑内存操作，缺点是内存使用率不高，会造成一些内存空间的浪费。动态内存，可以通过 new 关键字开辟所需大小的空间，但需要手动使用 delete 关键字释放空间。在实验过程中发现这些操作也会带来一些时间开销。

2）string 类简介

C++中提供 string 头文件支持 string 数据类型，其本质是一个 char 数组，string 类定义隐藏了数组的性质，允许程序自动处理 string 的大小，避免了手动操作内存的一些弊端。还提供了很多内置函数，方便处理字符串。

本实验使用了上述两种数据类型进行了算法实现，经对比，整形数组在时间上优于 string，以两个 10000 位大整数乘法为例，在实验环境下普通大整数乘法使用整型数组所需时间 250ms±，string 所需时间为 350ms±，主要原因在于 string 每个是字符类型计算乘法时需要多两步减法；但在内存占用上 string 更占优势，而且 string 的灵活性更高，代码相对简洁，特别是在分治递归过程中，计算各种不定长度的大整数上更具优势。

本实验最终选择利用 string 来存储大整数，算法的各个处理函数中也主要处理的是 string 类型。

3.2.2 算法伪代码

在算法的伪代码中，分治法中的加减法为大整数加减法。

1）随机生成大整数算法

算法 1 生成随机大整数

Input: 大整数位数n

Output: 大整数X

```
function GenerateLargeNumber(n)
    X ← string[n]
    X[0] ← randomnumber(1, 9)
    index ← 0
    while index < n do
        X[index] ← randomnumber(0, 9)
        index ← index + 1
    end while
    return X
end function
```

2）普通大整数乘法

算法 2 普通大整数乘法

Input: 大整数X，大整数Y

Output: 运算结果Z

```
function VanillaMultiply(X, Y)
    Z ← string(len_X + len_Y)
    i ← len_Y − 1, j ← len_X − 1
    while i >= 0 do
        while j >= 0 do
            Z[i + j + 1] ← Z[i + j + 1] + Y[i] * X[j]
            j ← j − 1
        end while
        i ← i − 1
    end while
    i ← len_Z − 1
    while i > 0 do
        if Z[i] > 9 then
            Z[i − 1] ← Z[i]/10,  Z[i] ← Z[i]%10
        end if
    end while
    return Z
end function
```

3）基于分治的大整数乘法

算法 3 分治大整数乘法

Input: 大整数X，大整数Y

Output: 运算结果Z

function DACMULTIPLY(X, Y)
  if $len_X \leq threshold$ **or** $len_Y \leq threshold$ **then**
    return $VanillaMultiply(X, Y)$
  end if
  $A, B \leftarrow devide(X)$
  $C, D \leftarrow devide(Y)$
  $AC \leftarrow DACMultiply(A, C)$
  $AD \leftarrow DACMultiply(A, D)$
  $BC \leftarrow DACMultiply(B, C)$
  $BD \leftarrow DACMultiply(B, D)$
  $Z \leftarrow AC \times 10^n + (AD + BC) \times 10^{n/2} + BD$
  return Z
end function

4）改进的分治大整数乘法

算法 4 改进分治大整数乘法

Input: 大整数X，大整数Y

Output: 运算结果Z

function IMPROVEDDACMULTIPLY(X, Y)
  if $len_X \leq threshold$ **or** $len_Y \leq threshold$ **then**
    return $VanillaMultiply(X, Y)$
  end if
  $A, B \leftarrow devide(X)$
  $C, D \leftarrow devide(Y)$
  $AC \leftarrow ImprovedDACMultiply(A, C)$
  $BD \leftarrow ImprovedDACMultiply(B, D)$
  $ABCD \leftarrow ImprovedDACMultiply(A + B, C + D)$
  $Z \leftarrow AC \times 10^n + (ABCD - AC - BD) \times 10^{n/2} + BD$
  return Z
end function

3.3 实验细节

3.3.1 去零和移位求和

对于含有零比较多的大整数，在分治过程中可以通过去零操作快速缩短需计算的大整数长度，一个简单的例子如对于大整数：$X = 10000000000001$，将其拆分后可以得到$A = 1000000$和$B = 0000001$，若对B进行去零操作，可以得到$B = 1$从而减少了很多无用的分治递归，从而节省时间。

但需要注意的是，这种改进只有对一些计算特例有显著效果，但也带来了问题，通过去零操作会出现拆分的两个数长度差距过大的问题，当一个数的长度降为 1 后，另一个数的长度可能还很大，但此时已经无法继续分治递归，故无法进行基本的乘法操作。一个解决办法是在递归出口处使用普通的大数相乘算法。通过对比实验，发现通过增加去零操作后，一边情况下的时间消耗没有显著的增多和减少，但这样处理后，使用普通大数相乘作为递归基本运算，分治大整数乘法也可以不通过补零来计算长度不等的两个数。

除此之外，在进行诸如$A \times 10^n + B$类型的计算时，直观上看是在A后面补上 n 个 0 后与 B 进行大数相加。但补 0 操作对于动态数组或者字符串来说都会带来额外的时间开销，一个更优的方法是在计算方法上，通过计算技巧的改善免去补零的操作。具体来说，若B的长度大于 n，上例的计算结果的后 n 为一定是B的后 n 位，将其复制后剩下的高位在和 A 进行大数相加，如果B的长度不足 n 位，则结果将B复制到末尾，中间补零凑够 n 位与A拼接即可。

3.3.2 分治递归结束条件

上面讲了分治算法的递归出口，即基本运算采用普通大整数乘法，这就摆脱了基本运算长度限制在整型长度的限制。显然这种基本计算比起整型数组直接计算耗时会更多，但由于使用的数据结构是 string，会存在类型转换等额外的时间开销，所以对于 string 来说，使用普通大整数乘法作为递归结束的基本操作并不会带来特别多的时间开销，而且好处是算法更具灵活性，在不带来额外空间浪费的情况下计算任意长度的大整数乘法。

然而在实验中发现，当大整数长度降为 1 时结束递归计算速度会很慢，具体原因可能是由于调用的是普通大整数乘法作为基本运算，调用过程中，调用函数，创建变量等操作占据主导，基本乘法，加法的运算时间不再是主要耗时的操作。针对这个问题，设置了一个参数 threshold 代表结束递归时大整数的长度，通过探索在计算两个长度为 10000 的大整数乘法下不同 threshold 值的情况下算法的时间消耗。结果如下图所示，可以发现，threshold 值在 40 以内时，分治递归带来的时间开销要明显大于普通大整数乘法，此时直接结束递归进行普通大整数乘法进行计算会对分治算法的时间消耗有非常显著地降低。本实验中分治算法设置的分治结束长度 threshold 为 50。

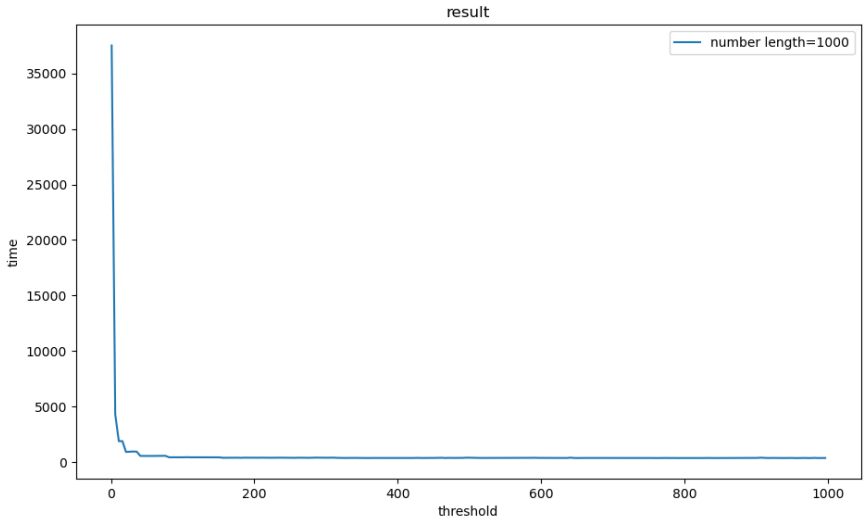

图 3.1 threshold 对运行时间的影响

3.3.3 程序计时

实验在 windows 操作系统下进行，通过引入 windows.h 头文件，使用 QueryPerformanceCounter()函数进行计时，相比 time.h 的 clock()函数，计时的结果更为精准，对于 10 位，100 位等大整数运算可以精确计时。

4 结果分析

上述实验中，分别实现了大整数乘法的普通算法，分治算法，改进的分治算法。为验证理论分析的正确性，我们运用所实现的算法进行了测试，并记录实验结果。具体而言，由于运行时间限制我们分别测试了两个长度为 10，100，1000，10000，100000，1000000 位大整数乘法，对于改进的分治算法增加测试了 10000000 位和 100000000 位。对于 10 位，100 位，1000 位的情况，由于运行耗时很短，会造成较大的误差，我们采用循环测试 1000 次取平均值的方法得到最后结果。测试所需的大整数均由大整数生成函数随机产生。最终测试实验结果如下表所示，其中 Vanilla，DAC，Improved DAC 分别代表大整数乘法的普通算法，分治算法，改进的分治算法，时间单位为毫秒(ms)。

表 4-1 三种算法运行结果

Length of large Number	Vanilla (ms)	DAC (ms)	Imporved DAC (ms)
10	0.0007691	0.0008739	0.0008367
100	0.041211	0.042144	0.041733
1,000	3.5319	4.6138	1.7158

10,000	350.5247	418.6627	69.1501
100,000	35333.5209	40056.0319	2672.1968
1,000,000	3410786.916	4455187.463	103587.818
10,000,000	-	-	3987544.697
100,000,000	-	-	153360969.04

在第 2 节中，我们分析了三种算法的时间复杂度，现设大整数乘法的基本运算所需时间为T，可以得到三种算法理论上计算长度为 n 位的大整数乘法所需要的时间t_1，t_2，t_3分别为：

$$t_1 = T_0 \times n^2$$

$$t_2 = T_1 \times n^2$$

$$t_3 = T_2 \times n^{1.585}$$

在分析上表之前，我们先看一下当大整数位数较小时算法运算时间的变化趋势。如下图所示，我们从 1 位开始增长步幅为 100 位，记录了前 50000 位计算时间的变化趋势，绘制出散点图，并绘制了理论上的变化曲线。通过结果可以发现，普通大整数乘法和普通分治乘法的时间增长变化趋势整体符合理论上的变化，由于实验环境算法实现等问题出现一些偏差。改进的分治算法很好的拟合了理论上的变化趋势。这说明算法在是现实上基本与理论分析相符，初步证明了理论分析的正确性。

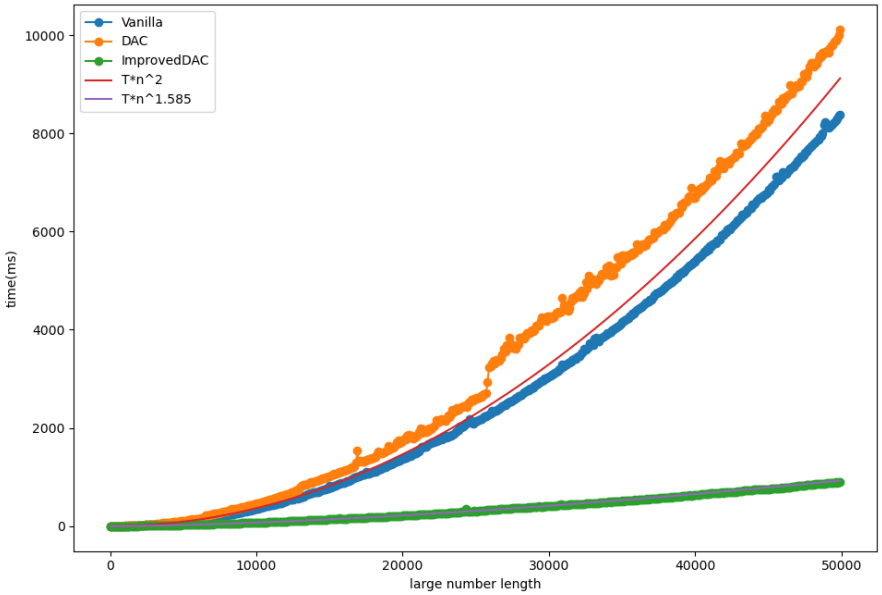

图 4.1 50000 位以内三种算法计算时间散点图

上面分析了较小位数的大整数乘法三个算法的正确性，现在我们根据表中数据分析，在位数很大时三个算法的正确性。由于表中各位数之间时间数量级差距过大，

直接在图中画出后不能看出区别，故采用取对数的方式绘制表中的散点图。具体做法是：

$$log_{10}t_0 = log_{10}(T_0 \times n^2) = 2log_{10}n + log_{10}T_0$$
$$log_{10}t_1 = log_{10}(T_1 \times n^2) = 2log_{10}n + log_{10}T_1$$
$$log_{10}t_2 = log_{10}(T_2 \times n^{1.585}) = 1.585log_{10}n + log_{10}T_2$$

通过上式可以看出，理论上取对数后表中数据应该呈直线分布，普通乘法和普通分治乘法的斜率为 2，改进分治算法的斜率为 1.585。我们将表中数据绘制散点图，如下图所示，直观来看数据呈直线分布，我们分别绘制了两条斜率分别为 2 和 1.585 的直线，可以看出数据分布基本拟合理论算出的直线斜率，这说明实际实验结果与理论分析在数据位数很大时也基本保持一致。

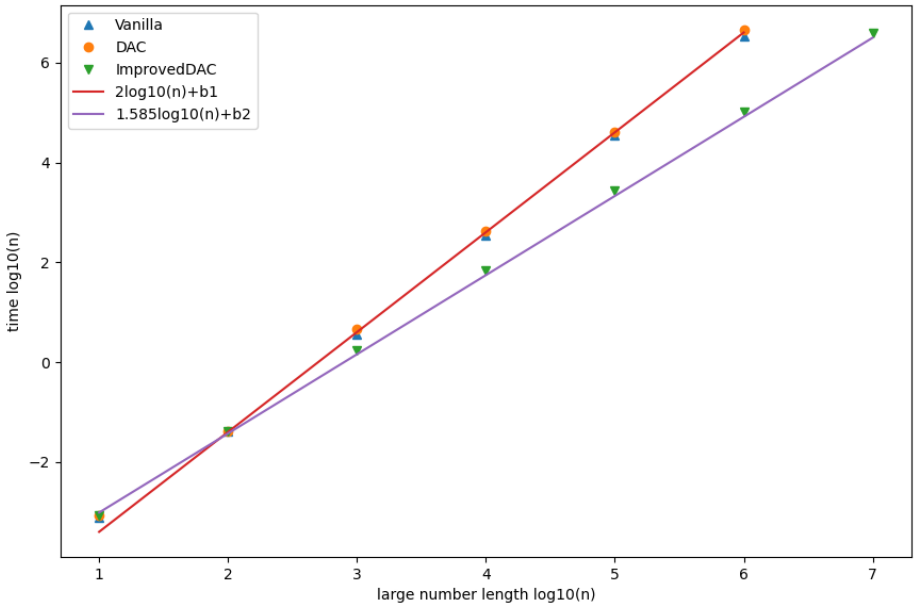

图 4-2 计算时间对数散点图

5 思考

1）在你的机器中，乘法是否比加法更费时？从哪里体现出来的？

在本机中计算 100000000 次基本乘法耗时为 110ms±，100000000 次基本加法的运算耗时同为 110ms±，可以看出对于编程语言的基本运算乘法和加法耗时是相同的。但是由于实验中分治算法涉及到大整数乘法和大整数加法，由于其计算复杂度不同普通大整数乘法时间复杂度为$O(n^2)$，普通大整数加法时间复杂度为$O(n)$，当大整数位数很大时，所以乘法是主要耗时的操作，加法可以忽略不计，所以减少乘法数量是优化算法的主要思路。

2）如果要做更大规模的乘法，比如 10 亿亿位的两个数的乘法，你有什么方法来解决这个问题？

对于更大规模的乘法而言，首先继续优化算法复杂度，如引入快速傅里叶变换等，但大整数乘法已经被证明其时间复杂度下限为$O(nlogn)$。单任务处理优化到理论极致之后，可以考虑的方式是并行处理。本实验中提到的分治方法，可以将原本大规模任务分为多个子任务，子任务可以并行处理，例如，分治法将规模为n的大数乘法分为 3 个规模为$n/2$的子任务，如果子任务并行处理那么可以将原本的串行处理 4 个子任务的时间降为处理 1 个子任务的时间。单机可以考虑多线程，规模更大的可以考虑多机分布式运算。

6 总结

通过本次实验，直观感受到了主定理在计算分治法时间复杂度时的作用。通过编程实验，验证了理论的正确性。同时对分治算法有了更深入的了解，重拾了 C++ 编程语言。通过查阅调研相关博客和文献了解了一些大整数乘法的基本算法。在实验过程中也发现了一些问题，比如使用 string 数据类型在 Visual Studio 开发环境下运行时间会非常长，比单纯用 g++慢几十倍。为了减少运行时间开销，从各种细节方面进行探索，在编程方面也有了要考虑算法效率的意识。

参考文献

[1] Karatsuba，Yu. Ofma． Multiplication of Many-Digital Numbers by Automatic Computers：Proceedings of the USSR Academy of Sciences，1962：293-294.

[2] Harvey D, Van Der Hoeven J. Integer multiplication in time O (n log n)[J]. 2019.

[3] Bodrato M, Zanoni A. Integer and polynomial multiplication: Towards optimal Toom-Cook matrices[C]//Proceedings of the 2007 international symposium on Symbolic and algebraic computation. 2007: 17-24.

[4] http://www.cplusplus.com/reference/string/ C++ string 类介绍

[5] https://matplotlib.org/contents.html matplotlib 绘图接口文档

[6] https://baike.baidu.com/item/%E4%B8%BB%E5%AE%9A%E7%90%86/3463232?fr=aladdin 主定理法

附录

程序源代码

见附件 code.zip

指导教师批阅意见：   成绩评定：   指导教师签字： 年　月　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。