北京航空航天大學

《算法分析与设计》课程论文

大数乘法的算法分析与比较

作者姓名 张雨任

学　　号 15081070

学　　院 软件学院

班　　级 162113

提交日期 2018 年 1 月 25 日

目录

摘要

大数乘法是算法界的一个重要的研究和应用领域。大数乘法主要可以应用于密码学中，其运算性能也影响着许多公钥密码学的算法的性能，如 RSA 等。大数乘法在天文学等领域也有着重要的意义。本文主要分析当前大数乘法的主流算法，讨论它们的实现，比较它们之间的性能。大数乘法的主流算法主要有朴素算法、朴素分治算法、Karatsuba 分治算法、Toom-cook 三路算法、快速傅里叶变换（FFT）算法、快速数论变换（NTT）算法等。文章的最后，讨论了大数乘法的新方案，以及大数乘法在实际领域中的应用情况。

关键词：大数乘法；快速傅里叶变换；分治法；Toom-cook 三路算法

1 引言

1.1 研究背景

大数乘法在公钥密码学中的应用较为广泛和频繁。而大数乘法的使用是大量的，而且由于大数乘法的位数较多，因而提高大数乘法的性能是一个重要的课题。本文主要分析了当前主流的大数乘法的算法，对相关算法的性能进行分析，并且分析这些算法的主要实现思路。

本文的研究背景是针对当前主流的大数乘法，进行分析和比较，并且提出新的算法方案。

1.2 国内外研究现状

针对大数乘法的研究已经具有较长的历史。国内外学者对于大数乘法的研究也已经取得了不少成果。大数乘法的朴素算法模拟了笔算乘法的过程。

1962 年，Karatsuba 提出了 Karatsuba 分治算法[5]，该算法使用分治方法，使得大数乘法的时间复杂度低至$O(n^{1.585})$，这个算法最大的优点是实现简单，额外开销较少，因而成为 RSA 等公钥加密系统的理想算法。本文在节 2.3 中详细讲述该算法的实现与分析。

1963 年，Toom 基于 Karatsuba 分治算法的思想[6]，提出将大数均分为 k 段（k≥2）进行分治，1969 年，Cook 基于这种算法进行了改良，因此称为 Toom-cook k 路（k-way）算法，k 随着输入大数的规模 n 的增长而按一定规律增长。本文在节 2.4 中，取 k=3，详细讲述 Toom-cook 三路算法的实现与分析。其时间复杂度低至$O(n^{1.465})$。

最新的大数乘法算法当属 Fürer 算法[4]，算法的设计者是 2007 年由宾夕法尼亚州立大学的瑞士数学家马丁·费尔（Martin Fürer）。该算法是在分治的基础上，结合了快速傅里叶变换进行解决，它的时间复杂度能够达到$O(n\log n \cdot 2^{O(log^* n)})$，其中，$log^* n$是迭代对数。该算法也是当今渐近意义上最快的大数乘法算法，不过该算法在超大数乘法中的使用更多，实际适用的场景较少。

2 大数乘法主流算法的分析

大数乘法主要包括朴素算法、朴素分治算法、Karatsuba 分治算法、Toom-cook 三路算法、快速傅里叶变换（FFT）算法、快速数论变换（NTT）算法等。

2.1 朴素算法

2.1.1 算法思路分析

朴素算法实质上是模拟数乘竖式的运算方式，把两个大数的每一位对应相乘，之后再对对应的位进行相加，然后处理进位的情况，得到最终的结果。大数乘法的朴素算法和朴素多项式相乘比较类似，不同的是，大数乘法的朴素算法需要在最后进行进位等后续处理。

2.1.2 算法实现

2.1.2.1 伪代码

变量 a 和 b 代表相乘的两个大数，可以用数组表示每一位的情况，length 代表该大数的位数，变量 c 代表每次运算的进位大小，变量 ans 代表 a 和 b 相乘得到的大数。

```
NAIVE-LARGE-NUMBER-MULTIPLICATION (a, b)
1    for i ← 0 to b.length - 1
2        for j ← 0 to a.length - 1
3            ans[i+j] ← ans[i+j] + a[j] * b[i]
4        end for
5    end for
6    c ← 0
7    for i ← 0 to a.length + b.length - 1
8        k ← ans[i]
9        ans[i] ← (k + c) mod 10
10       c ← (k + c) / 10
11   end for
12   for i ← a.length + b.length downto 0
13       if ans[i] ≠ 0 then
14           ans.length ← i + 1
15       end if
16   end for
17   if ans.length = 0 then
18       ans[0] ← 0
19       ans.length ← 1
20   end if
21   return ans
```

2.1.2.2 算法实现分析

在代码 1 到 5 行中，对两个大数的每一位对应相乘，大数 a 的第 i 位和大数 b 的第 j 位相乘，则存到 ans 的第 i+j 位中。然后对应位相加，存放在 ans 中，此时 ans 的每一位不能保证小于等于 10，因此接下来处理进位问题。

在代码 6 到 11 行中，处理 ans 中每一位的进位情况。对 ans 的每一位对 10 取余，则能够得到该位的大小。对 ans 的每一位除以 10，得到当前位向前一位的进位的大小。

在代码 12 到 21 行中，计算 ans 的总位数，用 ans.length 来表示。最后单独处理一种特殊情况，ans 等于 0 时，长度为 1。

2.1.3 算法性能分析

伪代码中按位相乘的部分使用的是二层嵌套循环，若设大数 a 的长度为 n，大数 b 的长度为 m，则该部分的时间复杂度为$O(n \times m)$。接下来的后续处理，最高都是线性的时间复杂度，因此朴素算法的时间复杂度是$O(n \times m)$。

2.2 朴素分治算法

2.2.1 算法思路分析

分治法是算法中的一个重要思想，很多问题都能够巧妙的转化为分治法，从而降低问题的复杂程度。分治法在每层递归时都有三个步骤[1]：(1) 分解：分解原问题为若干子问题，这些子问题是原问题的规模较小的实例。(2) 解决：解决这些子问题，递归地求解各子问题。然而，若子问题的规模足够小，则直接求解。(3) 合并：合并这些子问题的解成原问题的解。

而使用朴素分治算法解决大数乘法时，首先进行分解，把原问题分解为规模更小的问题，对于 n 位的大数，把这个大数平分为长度为$\lceil n/2 \rceil$的两段，这样就得到 4 个新的大数。接下来处理递归地求解这 4 个大数两两之间的乘积即可。对于待求解的大数 $\overline{AB}$ 和 $\overline{CD}$ 的乘积，通过上述的分解，可以分解为 A、B、C、D 四个大数，则有：

$$\overline{AB} \times \overline{CD}$$

$$= (10^m \times A + B)(10^m \times C + D)$$

$$= 10^{2m} \times A \times C + 10^m \times (A \times D + B \times C) + B \times D$$

递归的限制条件，也就是最小规模，是当 n=1 时，大数的长度只有 1，那么直接返回两个数的乘积即可。

2.2.2 算法实现

2.2.2.1 伪代码

X 和 Y 是输入待相乘的大数，n 是 X 和 Y 的最大长度（长度不足 n 的在前面作补 0 处理）。

```
DIVIDE-CONQUER-MULTIPLICATION(X, Y, n)
1    if n = 1 then
2        return X × Y
3    else
4        m ← ⌈n / 2⌉
5        a ← ⌊X / 10^m⌋
6        b ← X mod 10^m
7        c ← ⌊Y / 10^m⌋
8        d ← Y mod 10^m
9        e ← DIVIDE-CONQUER-MULTIPLICATION(a, c, m)
10       f ← DIVIDE-CONQUER-MULTIPLICATION(b, d, m)
11       g ← DIVIDE-CONQUER-MULTIPLICATION(b, c, m)
12       h ← DIVIDE-CONQUER-MULTIPLICATION(a, d, m)
13       return 10^2m e + 10^m (g + h) + f
14   end if
```

2.2.2.2 算法实现分析

在代码 1 到 2 行中，处理大数为最小规模的情况，直接相乘即可。

在代码 3 到 8 行中，把两个大数分成等长的 4 个大数，相当于把原问题分解为规模较小的子问题。

在代码 9 到 14 行中，分别计算 4 个数之间的两两乘积，为最后的合并做准备。根据 2.2.1 中推导出的公式：

$$\overline{\mathrm{AB}} \times \overline{\mathrm{CD}} = 10^{2m} \times \mathrm{A} \times \mathrm{C} + 10^{m} \times (\mathrm{A} \times \mathrm{D} + \mathrm{B} \times \mathrm{C}) + \mathrm{B} \times \mathrm{D}$$

算出最后的乘积。

2.2.3 算法性能分析

分析分治算法的时间复杂度，一般采用求解递归式的方法。设整个算法的时间复杂度为$T(n)$。在代码 4 到 9 行中，分解大数的过程需要计算 10 的 m 次方，是一个线性的时间复杂度，即$O(n)$。在代码 10 到 13 行中，进行了 4 次递归调用，每一个递归调用的规模是上一次调用的一半，因此这部分的时间复杂度可以表示为$4T(n/2)$。所以，该分治算法的时间复杂度可以表示为

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

接下来，进行递归式的求解。根据主方法定理（Master Method）[1]，该式有$a = 4 \geq 1$和$b = 2 > 1$，满足存在$t = 1 > 0$，有

$$f(n) = O(n) = \Theta(n^{log_b a - t}) = \Theta(n^{log_2 4 - 1})$$

因此，根据主方法定理情况 1，$T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_2 4}) = O(n^2)$。

可以看到，朴素分治算法与朴素算法相比，性能上基本没有提升。

2.3 Karatsuba 分治算法

2.3.1 算法思路分析

2.2 中的朴素分治算法实际上并没有改进朴素算法的时间复杂度，两者都是$O(n^2)$。本节提供一个时间复杂度较低的 Karatsuba 分治算法。与 2.2 它提供的算法相同，先把原问题分解为规模更小的问题，对于 n 位的大数，把这个大数平分为长度为$\lceil n/2 \rceil$的两段，这样就得到 4 个新的大数。接下来进行递归处理的时候与 2.2 有所不同。对于 $\overline{AB} \times \overline{CD}$ 有如下推导：

$$\begin{aligned}
&\overline{AB} \times \overline{CD} \\
&= (10^m \times A + B)\ (10^m \times C + D) \\
&= 10^{2m} \times A \times C + 10^m \times (A \times D + B \times C) + B \times D \\
&= 10^{2m} \times A \times C + 10^m \times (A \times C + B \times D - A \times C + B \times C + A \times D - B \\
&\qquad \times D) + B \times D
\end{aligned}$$

$$= 10^{2m} \times A \times C + 10^{m} \times (A \times C + B \times D - (A \times C - B \times C - A \times D + B \times D)) + B \times D$$

$$= 10^{2m} \times A \times C + 10^{m} \times (A \times C + B \times D - (A - B) \times (C - D)) + B \times D$$

2.2 的算法需要递归 4 次计算 4 个乘积，而 2.3 的算法只需要递归 3 次计算 3 个乘积，因此在性能上有一定的提升。

2.3.2 算法实现

2.3.2.1 伪代码

```
DIVIDE-CONQUER-MULTIPLICATION-KARATSUBA (X, Y, n)
1   if n = 1 then
2       return X × Y
3   else
4       m ← ⌈n / 2⌉
5       a ← ⌊X / 10^m⌋
6       b ← X mod 10^m
7       c ← ⌊Y / 10^m⌋
8       d ← Y mod 10^m
9       e ← DIVIDE-CONQUER-MULTIPLICATION- KARATSUBA (a, c, m)
10      f ← DIVIDE-CONQUER-MULTIPLICATION- KARATSUBA (b, d, m)
11      g ← DIVIDE-CONQUER-MULTIPLICATION- KARATSUBA (a - b, c - d, m)
12      return 10^2m e + 10^m (e + f - g) + f
13  end if
```

2.3.2.2 算法实现分析

在代码 1 到 2 行中，处理大数为最小规模的情况，直接相乘即可。

在代码 3 到 8 行中，把两个大数分成等长的 4 个大数，相当于把原问题分解为规模较小的子问题。

在代码 9 到 13 行中，分别计算 4 个数之间的两两乘积，为最后的合并做准备。根据 2.3.1 中推导出的公式：

$$\overline{AB} \times \overline{CD}$$

$$= 10^{2m} \times A \times C + 10^{m} \times (A \times C + B \times D - (A - B) \times (C - D)) + B \times D$$

算出最后的乘积。

2.3.3 算法性能分析

分析分治算法的时间复杂度，一般采用求解递归式的方法。设整个算法的时间复杂度为$T(n)$。在代码 4 到 9 行中，分解大数的过程需要计算 10 的 m 次方，是一个线性的时间复杂度，即$O(n)$。在代码 10 到 12 行中，进行了 3 次递归调用，每一个递归调用的规模是上一次调用的一半，因此这部分的时间复杂度可以表示为$3T(n/2)$。所以，该分治算法的时间复杂度可以表示为

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

接下来，进行递归式的求解。根据主方法定理（Master Method）[1]，该式有$a = 3 \geq 1$和$b = 2 > 1$，满足存在$t \approx 0.585 > 0$，有

$$f(n) = O(n) = \Theta(n^{log_b a - t}) = \Theta(n^{log_2 3 - 0.585})$$

因此，根据主方法定理情况 1，$T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_2 3}) = O(n^{1.585})$。

可以看到，改进的分治算法，相比于普通的分治算法和朴素算法，在性能上有一定的提升。

2.4 Toom-cook 三路（3-way）算法

Karatsuba 分治算法的时间复杂度能够低至$O(n^{1.585})$，已经具有不错的性能，而同样是分治算法，Toom-cook 三路算法能够具有更好的性能。朴素分治算法和 Karatsuba 分治算法都是把大数分解为两部分，而 Toom-cook 三路算法把大数分解为三部分，从而降低每次递归的规模，达到提高性能的目的。

2.4.1 算法思路分析[3]

对于输入的两个大数 X 和 Y，可以表示为：

$$X = \overline{ABC} = A \cdot 10^{2p} + B \cdot 10^{p} + C$$

$$Y = \overline{DEF} = D \cdot 10^{2p} + E \cdot 10^{p} + F$$

则

$$X \times Y = \overline{\mathrm{ABC}} \times \overline{\mathrm{DEF}}$$

$$= A \cdot D \cdot 10^{4p} + (A \cdot E + B \cdot D)10^{3p} + (A \cdot F + B \cdot E + D \cdot C)10^{2p}$$

$$+ (B \cdot F + E \cdot C)10^{p} + C \cdot F$$

可以使用 5 个值表示上述变量：

$$A \cdot D = \frac{6G - 4H + I - 4J + K}{24}$$

$$A \cdot E + B \cdot D = \frac{-2H + I + 2J - K}{12}$$

$$A \cdot F + B \cdot E + D \cdot C = \frac{-30G + 6H - I + 16J - K}{24}$$

$$B \cdot F + E \cdot C = \frac{8H - I - 8J + K}{12}$$

$$C \cdot F = G$$

其中，

$$\mathrm{G} = \mathrm{C} \cdot \mathrm{F}$$

$$\mathrm{H} = (\mathrm{C} + \mathrm{B} + \mathrm{A}) \cdot (\mathrm{F} + \mathrm{E} + \mathrm{D})$$

$$\mathrm{I} = (\mathrm{C} + 2\mathrm{B} + 4\mathrm{A}) \cdot (\mathrm{F} + 2\mathrm{E} + 4\mathrm{D})$$

$$\mathrm{J} = (\mathrm{C} - \mathrm{B} + \mathrm{A}) \cdot (\mathrm{F} - \mathrm{E} + \mathrm{D})$$

$$\mathrm{K} = (\mathrm{C} - 2\mathrm{B} + 4\mathrm{A}) \cdot (\mathrm{F} - 2\mathrm{E} + 4\mathrm{D})$$

因此，算法进行 5 次递归调用，每次递归的规模是前一次调用的$\frac{1}{3}$，最终在进行合并即可。最终结果表述为：

$$X \times Y = \overline{\mathrm{ABC}} \times \overline{\mathrm{DEF}}$$

$$= \frac{6G - 4H + I - 4J + K}{24}10^{4p} + \frac{-2H + I + 2J - K}{12}10^{3p}$$

$$+ \frac{-30G + 6H - I + 16J - K}{24}10^{2p} + \frac{8H - I - 8J + K}{12}10^{p} + G$$

2.4.2 算法实现

2.4.2.1 伪代码

```
TOOM-COOK-MULTIPLICATION(X, Y, n)
1   if n = 1 then
2       return X × Y
3   else
4       m ← [n / 3]
5       a ← [X / 10^2m]
6       b ← [X / 10^m] mod 10^m
7       c ← X mod 10^m
8       d ← [Y / 10^2m]
9       e ← [Y / 10^m] mod 10^m
10      f ← Y mod 10^m
11      g ← TOOM-COOK-MULTIPLICATION(c, f, m)
12      h ← TOOM-COOK-MULTIPLICATION(c+b+a, f+e+d, m)
13      i ← TOOM-COOK-MULTIPLICATION(c+2b+4a, f+2e+4d, m)
14      j ← TOOM-COOK-MULTIPLICATION(c-b+a, f-e+d, m)
15      k ← TOOM-COOK-MULTIPLICATION(c-2b+4a, f-2e+4d, m)
16      return (6G-4H+I-4J+K) / 24 × 10^4m + (-2H+I+2J-K) / 12 × 10^3m + (-30G+6H-I+16J-
            K) / 24 × 10^2m + (8H-I-8J+K) / 12 × 10^m + G
17  end if
```

2.4.2.2 算法实现分析

在代码 1 到 2 行中，处理大数为最小规模的情况，直接相乘即可。

在代码 3 到 10 行中，把两个大数分成等长的 6 个大数，相当于把原问题分解为规模较小的子问题。

在代码 11 到 15 行中，分别计算 6 个数不同组合之间的乘积，为最后的合并做准备。根据 2.3.1 中推导出的公式：

$$\overline{\mathrm{ABC}} \times \overline{\mathrm{DEF}}$$

$$= \frac{6G-4H+I-4J+K}{24}10^{4p} + \frac{-2H+I+2J-K}{12}10^{3p}$$

$$+ \frac{-30G+6H-I+16J-K}{24}10^{2p} + \frac{8H-I-8J+K}{12}10^{p} + G$$

算出最后的乘积。

2.4.3 算法性能分析

分析分治算法的时间复杂度，一般采用求解递归式的方法。设整个算法的时间复杂度为$T(n)$。在代码 4 到 10 行中，分解大数的过程需要计算 10 的 m 次方，是一个线性的时间复杂度，即$O(n)$。在代码 11 到 15 行中，进行了 5 次递归调用，每一个递归调用的规模是上一次调用的$\frac{1}{3}$，因此这部分的时间复杂度可以表示为$5T(n/3)$。所以，该分治算法的时间复杂度可以表示为

$$T(n) = 5T(n/3) + O(n)$$

接下来，进行递归式的求解。根据主方法定理（Master Method）[1]，该式有$a = 5 \geq 1$和$b = 3 > 1$，满足存在$t \approx 0.465 > 0$，有

$$f(n) = O(n) = \Theta(n^{log_b a - t}) = \Theta(n^{log_3 5 - 0.465})$$

因此，根据主方法定理情况 1，$T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_3 5}) = O(n^{1.465})$。

可以看到，Toom-cook 三路算法，相比于 Karatsuba 分治算法，在性能上有一定的提升。

2.5 快速傅里叶变换（FFT）算法

2.5.1 算法思路分析

快速傅里叶变换本质上是用来计算多项式的乘积，而在 2.1.1 中提到了，多项式乘法和大数乘法是有着紧密联系的。所以我们也可以在大数乘法中使用快速傅里叶变换，来得到比较好的性能。

首先，通过离散傅里叶变换（DFT）把多项式从系数表达转换为点值表达。系数表达形如$y = f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$。点值表达形如$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})\}$，同时满足$y_0 = f(x_0)$，$y_1 = f(x_1)$，$y_2 = f(x_2)$，…，$y_{n-1} = f(x_{n-1})$。用这个点集就可以表示多项式。

点值表达的优点在于计算多项式乘法很方便。假设两个多项式的点值表达为$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})\}$和$\{(x_0, z_0), (x_1, z_1), (x_2, z_2), \ldots, (x_{n-1}, z_{n-1})\}$，设$y = A(x)$和$z = B(x)$，那么这两个多

项式的乘积$A(x)\cdot B(x)$可以等价地用点集，

$\{(x_0,y_0z_0),(x_1,y_1z_1),(x_2,y_2z_2),\ldots,(x_{n-1},y_{n-1}z_{n-1})\}$表示。

用系数表达来计算多项式乘积（类似于朴素算法）的时间复杂度是$O(n^2)$，用点值表达来计算多项式乘积的时间复杂度是$O(n)$。

本质上，快速傅里叶变换的基本思想就是，把大数的每一位看作是多项式的每一项，再通过“求值”操作把多项式从“系数表达”转换为“点值表达”的形式，这个操作的时间复杂度本应是$O(n^2)$，不过应用 FFT 和单位复数根的性质，可以把时间复杂度降低到$O(nlgn)$，然后再作“点值乘法”，计算出“点值表达”的乘积，再通过逆向离散傅里叶变换（IDFT）把“点值表达”的乘积转换为系数表达。如图 2.4.1-1 所示，可以看到，朴素算法的时间复杂度是$O(n^2)$，而快速傅里叶变换的时间复杂度是$\Theta(nlgn)$。

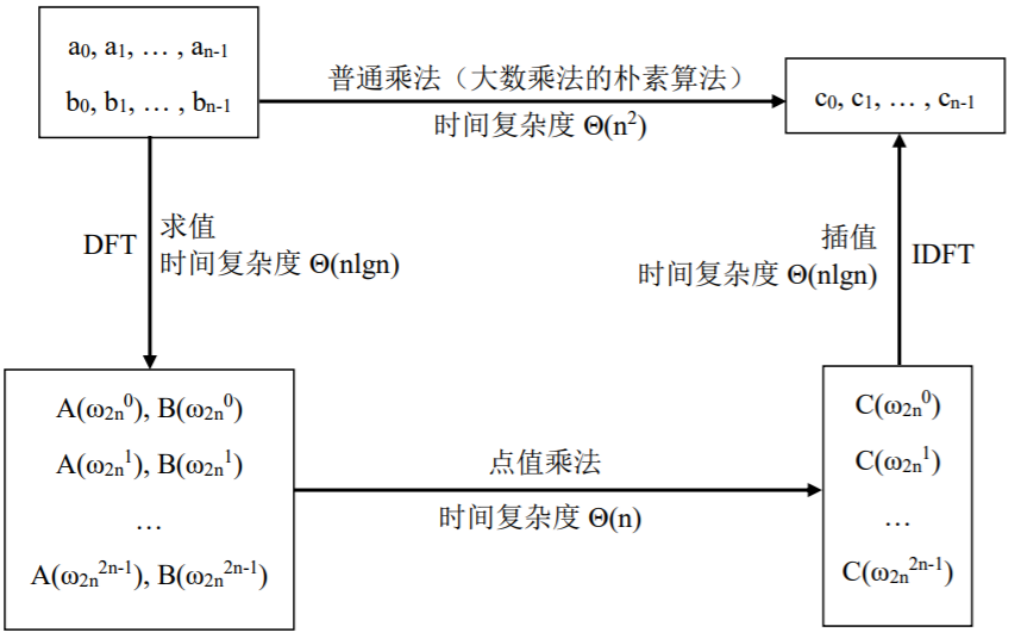

图 2.4.1-1

总结来说，快速傅里叶变换主要有以下四个要点：

(1) 使次数界（上界）增加一倍。$A(x)$、$B(x)$的长度扩充到$2n$。

(2) 求值。主要是求点值表示$A(x)$、$B(x)$的点值表示。

(3) 点乘。$C(x)=A(x)\cdot B(x)$。

(4) 插值。对$C(x)$进行插值，求出其系数表示。

对于大数乘法，再额外处理进位情况即可。

2.5.2 算法实现

2.5.2.1 伪代码

调用过程 FFT-MULTIPLICATION 执行大数乘法的快速傅里叶变换的算法，传入参数 a、b 代表待计算乘积的两个大数。过程返回的 ans 是大数 a、b 的乘积。

```
FFT-MULTIPLICATION(a, b)
1   n ← 1
2   loglen ← 0
3   while n < a.length + b.length
4       n ← n << 1
5       loglen ← loglen + 1
6   end while
7   FFT(a, loglen, n, 1)
8   FFT(b, loglen, n, 1)
9   for i ← 0 to n-1
10      ans[i] ← a[i] * b[i]
11  end for
12  FFT(ans, loglen, n, -1)
13  for i ← 0 to n-1
14      ans[i+1] ← ans[i+1] + ans[i] / 10
15      ans[i] ← ans[i] mod 10
16  end for
17  return ans
```

```
FFT(a, loglen, n, on)
1    BIT-REVERSE-COPY(a, A)
2    m ← 2
3    for s ← 1 to loglen
4         wm ← cos(2π×on/m)+i×sin(2π×on/m) // i means image here
5         for k ← 0 to n-1 by m
6              w ← 1
7              for j ← 0 to m/2-1
8                   t ← w * A[k+j+m/2]
9                   u ← A[k+j]
10                  A[k+j] ← u + t
11                  A[k+j+m/2] ← u -t
12                  w ← w * wm
13             end for
14        end for
15        m ← m << 1
16   end for
17   if on = -1 then
18        for i ← 0 to n-1
19             A[i] ← A[i] / len
20        end for
21   end if
22   return A

BIT-REVERSE-COPY(a, A)
1    n ← a.length
2    for k ← 0 to n-1
3         A[rev(k)] ← a[k]
4    end for
```

2.5.2.2 算法实现分析

调用过程 FFT-MULTIPLICATION 完成对两个大数的乘法，对两个大数，通过分别调用过程 FFT 分别进行离散傅里叶变换（DFT，参数 on 置为 1），然后通过过程 FFT-MULTIPLICATION 的代码 9 到 11 行，对得到的点值进行乘法。然后在对得到的乘积点集，调用过程 FFT，进行逆向离散傅里叶变换（IDFT，参数 on 置为-1）。最后，处理进位即可。

2.5.3 算法性能分析

DFT 和 IDFT 的时间复杂度都是$\Theta(\text{nlgn})$。在过程 FFT 中，首先调用了过程 BIT-REVERSE-COPY，迭代 n 次，每次在$O(\text{lgn})$内完成整数的逆序。因此，过程 BIT-REVERSE-COPY 的时间复杂度是$O(\text{nlgn})$。在过程 FFT 中，代码 5 到 14 行的第二层 for 循环的迭代次数是$\text{n/m} = \text{n}/2^s$次，代码 7 到 13 行的最内层循环的迭代次数是$\text{m}/2 = 2^{s-1}$次。因此，时间复杂度为：

$$\sum_{s=1}^{lgn} \frac{n}{2^s} \cdot 2^{s-1} = \sum_{s=1}^{lgn} \frac{n}{2} = \Theta(nlgn)$$

2.6 快速数论变换（NTT）算法

快速傅里叶变换（FTT）算法使用单位复数根来处理，由于每个单位复数根涉及到正弦函数和余弦函数，因此很多系数可能是浮点数，因此当数据量巨大的时候，在某些情况下，可能会造成误差。因此，可以考虑使用 NTT 算法避免大数据量时的误差。

快速傅里叶变换（FFT）使用单位复数根来进行采样和插值，因而能够把时间复杂度降低到$\Theta(\text{nlgn})$，但是单位复数根涉及到浮点数精度问题，因此快速数论变换（NTT）使用原根进行采样和插值，来避开浮点数的讨论。

单位复数根能够把时间复杂度降低到$\Theta(\text{nlgn})$，因为单位复数根具有消去引理和折半引理的性质。对于单位复数根 ω_n，有以下性质[1]：

(1) 消去引理：对任何整数 n≥0，k≥0，以及 d＞0，有

$$\omega_{dn}^{dk} = \omega_n^k$$

(2) 折半引理：如果 n＞0 为偶数，那么 n 个 n 次单位复数根的平方的集合就是 n/2 个 n/2 次单位复数根的集合。

(3) 特殊值性质：

$$\omega_n^n = 1$$

$$\omega_n^{n/2} = -1$$

特别是折半引理，折半引理对于用分治策略来对多项式的系数与点值表达进行相互转换是非常重要的，因为它保证递归子问题的规模只是递归调用前的一半，因而保证了时间复杂度低至Θ(nlgn)。

而快速数论变换（NTT）的成功在于，原根同样具有以上单位复数根所具有的性质，同时还能够避免浮点数的讨论。对于原根$g_n = g^{\frac{p-1}{n}}$，有以下性质：

(1) 消去引理：对任何整数 n≥0，k≥0，以及 d＞0，有

$$g_{dn}^{dk} = g^{\frac{dk(p-1)}{dn}} = g^{\frac{k(p-1)}{n}} = g_n^k$$

(2) 折半引理：如果 n＞0 为偶数，那么 n 个 n 次原根的平方的集合就是 n/2 个 n/2 次原根的集合。

$$(g_n^k)^2 = g_n^{2k} = g_{n/2}^k$$

$$(g_n^{k+n/2})^2 = g_n^{2k+n} = g_n^{2k} = g_{n/2}^k$$

从而，原根具有单位复数根所有必要性质，因此能够证明其正确性。

2.6.1 算法思路分析

首先，明确 NTT 在 Z_P（素数 P 阶循环群）进行。其次，可以把 NTT 与 FFT 进行类比，把 FFT 中的单位复数根$e^{-\frac{2\pi i}{N}} = \omega_N^{-1}$看作是 NTT 中的$g^{\frac{P-1}{N}}(mod\ \ P)$。因此，有$e^{-\frac{2\pi i}{N}} = \omega_N^{-1} = g^{\frac{P-1}{N}}(mod\ \ P)$。

因此，把 FFT 公式中$e^{-\frac{2\pi i}{N}}$部分替换为$g^{\frac{P-1}{N}}(mod\ \ P)$即可。

则数论变换（对应离散傅里叶变换，DFT）可表示为：

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj} = \sum_{j=0}^{n-1} a_j g^{jk}(mod\ \ P)$$

类似地，逆向数论变换（对应逆向离散傅里叶变换，IDFT）可表示为：

$$y_k = \frac{1}{n}\sum_{j=0}^{n-1} a_j \omega_n^{-kj} = \frac{1}{n}\sum_{j=0}^{n-1} a_j g^{-jk}(mod\ \ P)$$

在以上公式中，n 是 P-1 的一个因子且 P 是素数。FFT 中一般会把 n 置为 2 的整数次幂，因此素数 P 可以构造为$P = c \cdot 2^k + 1$，素数 P 可以选择费马素数。可以考虑选择，$P = (479 \ll 21) + 1$，$N = 1 \ll 18$，则模 P 的原值根为$G = 3$[2]。

2.6.2 算法实现

2.6.2.1 伪代码

调用过程 NTT-MULTIPLICATION 执行大数乘法的快速数论变换算法，传入参数 a、b 代表待计算乘积的两个大数。过程返回的 ans 是大数 a、b 的乘积。

```
NTT-MULTIPLICATION(a, b)
1   P = (479 << 21) + 1
2   n ← 1
3   loglen ← 0
4   while n < a.length + b.length
5       n ← n << 1
6       loglen ← loglen + 1
7   end while
8   NTT(a, loglen, n, 1)
9   NTT(b, loglen, n, 1)
10  for i ← 0 to n-1
11      ans[i] ← a[i] * b[i] mod P
12  end for
13  NTT(ans, loglen, n, -1)
14  for i ← 0 to n-1
15      ans[i+1] ← ans[i+1] + ans[i] / 10
16      ans[i] ← ans[i] mod 10
17  end for
18  return ans
```

```
NTT(a, loglen, n, on)
1   P ← (479 << 21) + 1
2   G ← 3
3   for i ← 0 to 19
4       wm[i] ← G^((P-1)/(1<<i)) mod P
5   end for
6   BIT-REVERSE-COPY(a, A)
7   a ← A
8   cnt ← 0
9   m ← 2
10  while m <= n
11      cnt ← cnt + 1
12      for j ← 0 to n-1 by m
13          w ← 1
14          for k ← j to j+m/2-1
15              u ← a[k] mod P
16              t ← w * a[k+m/2] mod P
17              a[k] ← (u+t) mod P
18              a[k+m/2] ← (u-t+P) mod P
19              w = w * wm[i] mod P
20          end for
21      end for
22      m ← m << 1
23  end while
24  if on = -1 then
25      for i ← 1 to n/2
26          swap(a[i], a[n-i])
27      end for
28      tmp ← n^(P-2) mod P
29      for i ← 0 to n-1
30          a[i] ← a[i] * tmp mod P
31      end for
32  end if
33  return a
```

```
BIT-REVERSE-COPY(a, A)
1   n ← a.length
2   for k ← 0 to n-1
3       A[rev(k)] ← a[k]
4   end for
```

2.6.2.2 算法实现分析

NTT 的实现和 FFT 类似。调用过程 NTT-MULTIPLICATION 完成对两个大数的乘法，对两个大数，通过分别调用过程 NTT 分别进行数论变换（参数 on 置为 1），然后通过过程 NTT-MULTIPLICATION 的代码 10 到 12 行，对得到的点值进行乘法。然后在对得到的乘积点集，调用过程 NTT，进行逆向数论变换（参数 on 置为-1）。最后，处理进位即可。

2.6.3 算法性能分析

数论变换和逆向数论变换的时间复杂度都是$\Theta(\mathrm{nlgn})$。在过程 NTT 中，首先调用了过程 BIT-REVERSE-COPY，迭代 n 次，每次在$O(\mathrm{lgn})$内完成整数的逆序。因此，过程 BIT-REVERSE-COPY 的时间复杂度是$O(\mathrm{nlgn})$。在过程 NTT 中，代码 12 到 21 行的 for 循环的迭代次数是$\mathrm{n/m} = \mathrm{n}/2^{s}$次，代码 14 到 20 行的最内层循环的迭代次数是$\mathrm{m}/2 = 2^{s-1}$次。因此，时间复杂度为：

$$\sum_{s=1}^{lgn} \frac{n}{2^{s}} \cdot 2^{s-1} = \sum_{s=1}^{lgn} \frac{n}{2} = \Theta(nlgn)$$

3 大数乘法主流算法的比较

前文主要分析了当前大数乘法的主流算法的具体实现与性能。表 3-1 对比了上述算法的性能与算法特点。

表 3-1 大数乘法主流算法的比较

	时间复杂度	特点
朴素算法	$\Theta(n^2)$	使用多位数笔乘的基本原理
朴素分治算法	$\Theta(n^2)$	Karatsuba 分治算法的基础
Karatsuba 分治算法	$\Theta(n^{1.585})$	通过减少递归的次数降低时间复杂度
Toom-cook 三路算法	$\Theta(n^{1.465})$	通过调整递归次数和每次递归的规模降低时间复杂度
快速傅里叶变换（FFT）算法	$\Theta(\text{nlgn})$	浮点数精度可能会导致误差
快速数论变换（NTT）算法	$\Theta(\text{nlgn})$	原根具有单位复数根一切必要的性质，同时避免了浮点数的讨论
Fürer 算法	$O(\text{nlogn} \cdot 2^{O(log^* n)})$	渐近意义上最快的算法，但适用于超大数乘法，应用场景有限

可以看到，每个算法都有自己不同的性能与特征，算法不能够用好坏来区分，只有针对当前应用场景合适与否之分。针对当前的应用场景，选择合适的算法，从而解决问题。比如，当数据量巨大的时候，应该选取时间复杂度较低的算法，当问题对结果的稳定性和精确性有较高的要求时，应该避免使用可能会出现精度问题的算法，当对于性能没有要求时，我们可以选择实现起来简单方便的算法。

4 大数乘法的应用与创新

4.1 大数乘法的应用

大数乘法的应用很广泛，在密码学有着重要的地位。RSA 算法是密码学界乃至算法界最重要的算法之一，是最广泛应用的公钥机制，既能用于数据的加密和解密，还可以用于数字签名。RSA 的算法实现中，要计算两个大素数 p、q 的乘积，以及两个大合数 p-1、q-1 的乘积。因此，大数乘法的性能很大程度上，影响着 RSA 算法的性能。

除此之外，大数运算在科研领域的应用也十分广泛。尤其在天文学、气象学等实验数据较多较为复杂的领域。在天文领域，数字很多都是巨大的，而天文领域又存在着众多的几何问题和大量的计算问题，因此不可避免的会遇到大数运算或大数乘法。

当今社会是大数据的时代，数据量会越来越大，而数据本身也不可避免的发生膨胀，因此大数运算在今后的计算相关学科可能也会越来越多的被提到。

4.2 大数乘法的创新

实际上，通过增加递归的次数，分治法大数相乘还可以继续优化。

如今很多大数乘法的创新算法都具有很好的性能。比如，可以考虑把每个大数拆成 100 段等长的大数，结合快速傅里叶变换进行优化，时间复杂度最低可以优化到$O(n^{1.149})$。经过上述处理，大数乘法的性能已经可以接近线性，当然，算法的实现也会更加复杂。

我认为，也可以考虑结合 Karatsuba 分治算法和快速傅里叶变换（FFT）算法，在克服 FFT 浮点数精度问题的同时，进一步提升算法的性能，同时保证实现起来较为简便。这种算法并没有分治到规模为 1 的边界情况，而是在一定规模下，进行 FFT，从而减少递归层数，减少递归造成的额外开销。当然，时间复杂度的降低或性能的提升绝对不是算法设计的唯一追求，我们还要针对不同的场景，来选择或设计不同的算法，从而更好地行使其功能。

在我看来，今后的主流大数乘法算法，一定是结合分治算法和快速傅里叶变换来应用的，它们之间在一定程度上能够互相弥补对方的不足之处。

5 结束语

大数乘法是应用广泛的重要课题，高性能解决大数乘法的算法主要可以分为分治法类、傅里叶变换类两大类，基于两者，或者两者结合，可以设计出高效的大数乘法算法。当然，时间复杂度只是评估算法性能的一个要素，在不同的情境下还是应该选用最合适的算法。而大数乘法不仅在密码学、天文学、气象学等领域有着重要的意义，在这个数据爆炸的时代，大数乘法的应用也会越来越广泛和频繁，因此，对于大数乘法的研究和应用，任重道远。

参考文献

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. 算法导论[M]. 北京: 机械工业出版社, 2013: 17, 53-54, 532.

[2] P、N、G 的取值参考 http://blog.csdn.net/acdreamers/article/details/39026505

[3] gmplib 高精度计算库官方文档 15.1.3 Toom 3-Way Multiplication

[4] Svyatoslav Covanov. Putting Fürer Algorithm into Practice [J]. 2015.

[5] Karatsuba A.A., Ofman Y. Multiplication of Many-Digital Numbers by Automatic Computers[J]. Doklady Akad. Nauk SSSR, 1962, 145(2): 293-294.

[6] Toom A.L., The complexity of a scheme of functional elements realizing the multiplication of integers[J]. Dokl. Akad. Nauk SSSR 1963, 150: 496-498