**Karatsuba乘法实验报告**

李浩然

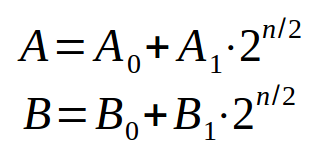
PB13210007

**实验目的**

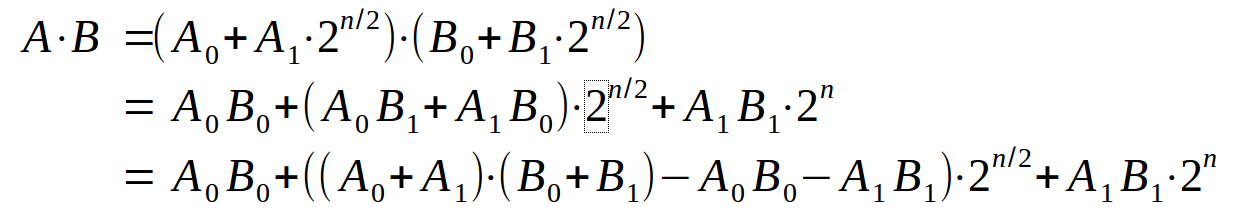
* 理解Karatsuba乘法算法流程
* 实现Karatsuba算法

**实验原理**

假设A, B两个数在二进制表示下位数小于n，将它们表示为：

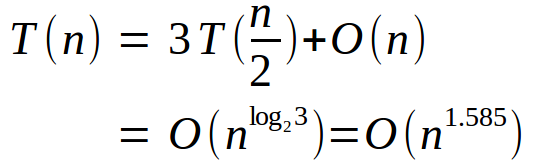


则可以被表示为：



观察可得，为了得到A, B相乘的结果，仅需进行3次小规模的乘法运算和6次小规模的加法运算即可。

加法运算和乘法运算的复杂度差一个数量级，是线性时间的复杂度，所以最终复杂度为：



**实验环境**

* Ubuntu 16.04 64-bit
* C++ 11 Standard
* No -o options set during compiling.

**实验内容**

1. 进制转换

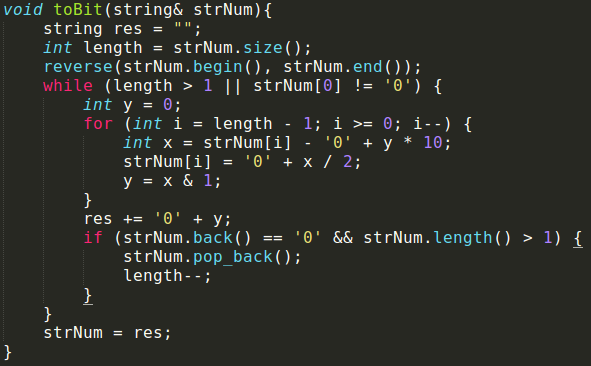
读入和输出的大整数都是以十进制的形式呈现的，但在实验原理中的公式推导是建立在二进制的基础上的。所以在进行乘法运算之前与得到结果之后，需要进行十进制和二进制之间的进制转换。

十进制到二进制的转换：

* **算法流程：**

1. 将十进制数进行一次数位反转，使得存储空间的低对应数位的低。
2. 从高位开始每位除2，记录余数信息。
3. 将最后一位的余数加到结果的最高位
4. 如果输入的数等于0则返回结果，否则回到2继续进行。

* **源代码:**

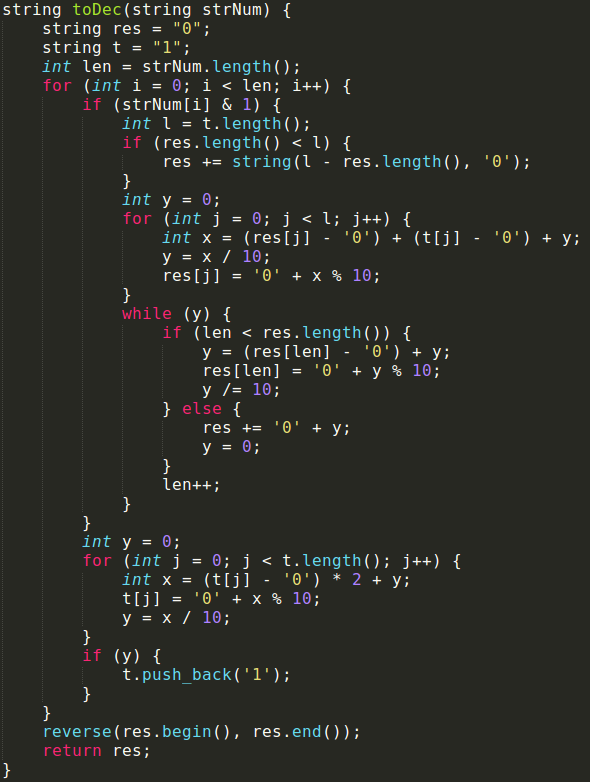


二进制到十进制的转换：

* **算法流程：**

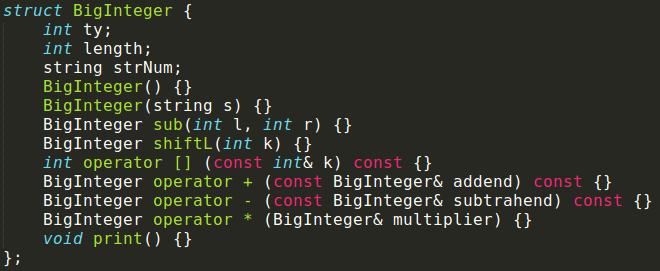
1. 用一个变量t记录2的0次幂（等于1）。
2. 从低位开始扫描二进制数。
3. 如果扫描到的数是1，在结果中加上t。
4. 如果扫描完，执行下一步。否则t乘上2，返回第三步继续扫描下一位。
5. 将结果数位反转并返回。

* **源代码：**



1. 大整数表示方式

前面已经提到过，本次实验用字符串表示大整数，字符串的低位存储大整数的低位。同时，在大整数的结构体中还保留了一个ty变量用来后续实验用bitset或者数组存储数位时候的运算效率。

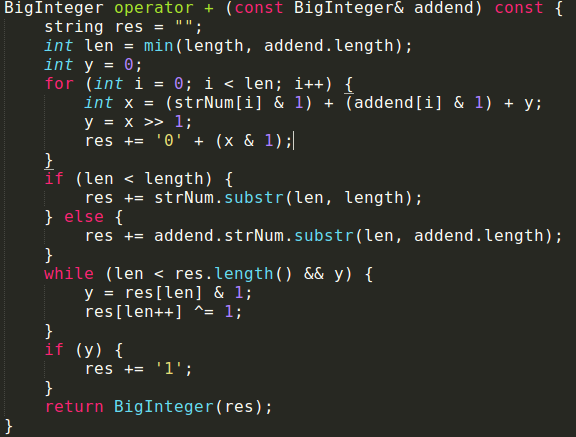


1. 大整数运算
   1. 加法

**算法流程：**

1. 从低位开始扫描。
2. 将对应位上的两个数和进位相加，结果保留最低比特，记录进位（高比特位）。
3. 扫描到较小的加数长度继续执行下一步，否则返回第二部扫描高一位。
4. 将两个加数相差大的位数直接连接到结果后。
5. 如果有进位信息，从扫描结束处开始向高位扫描。
6. 如果当前位是1，则当前位置0并继续扫描，否则进位信息清零。
7. 如果还有进位信息，在结果最后附加一个1。

**源代码：**

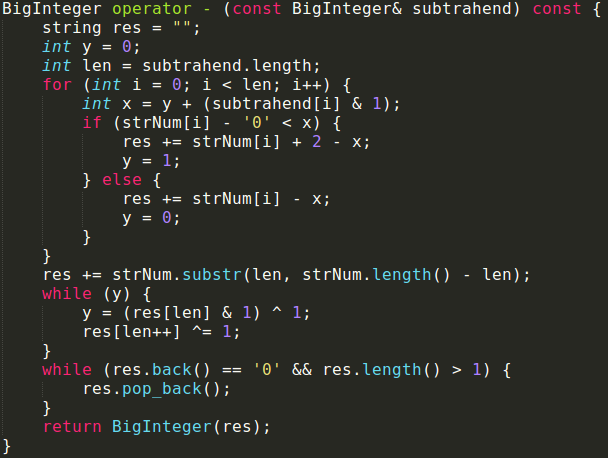


* 1. 减法

**算法流程：**

1. 从低位开始扫描。
2. 每次减法运算后记录进位信息。
3. 扫描完减数最高位以后， 将被减数和减数之间的相差位数附加到结果高位。
4. 判断是否还有进位信息，如果有则从减数最高位继续向高位扫描。
5. 将结果高位的0数位删除。

**源代码：**

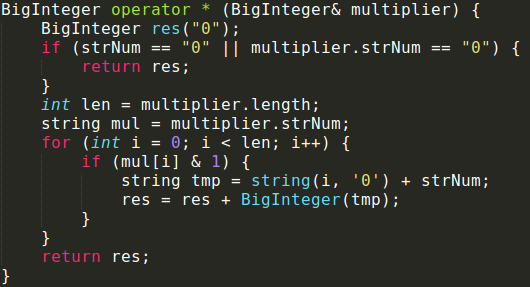


* 1. 乘法

**算法流程：**

因为大整数是以二进制的形式储存的，所以乘法的流程和二进制到十进制的转换流程类似，原理都是将乘数拆成2的若干次幂，分别乘上被乘数，再将结果相加。而被乘数的2的若干次幂可以通过一个循环依次求得，在循环的过程中可以顺带将结果累加，得到最终的乘积。

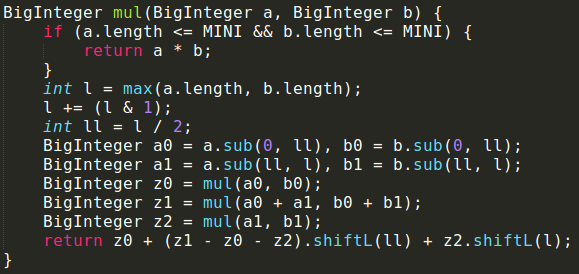
**源代码：**



1. Karatsuba乘法

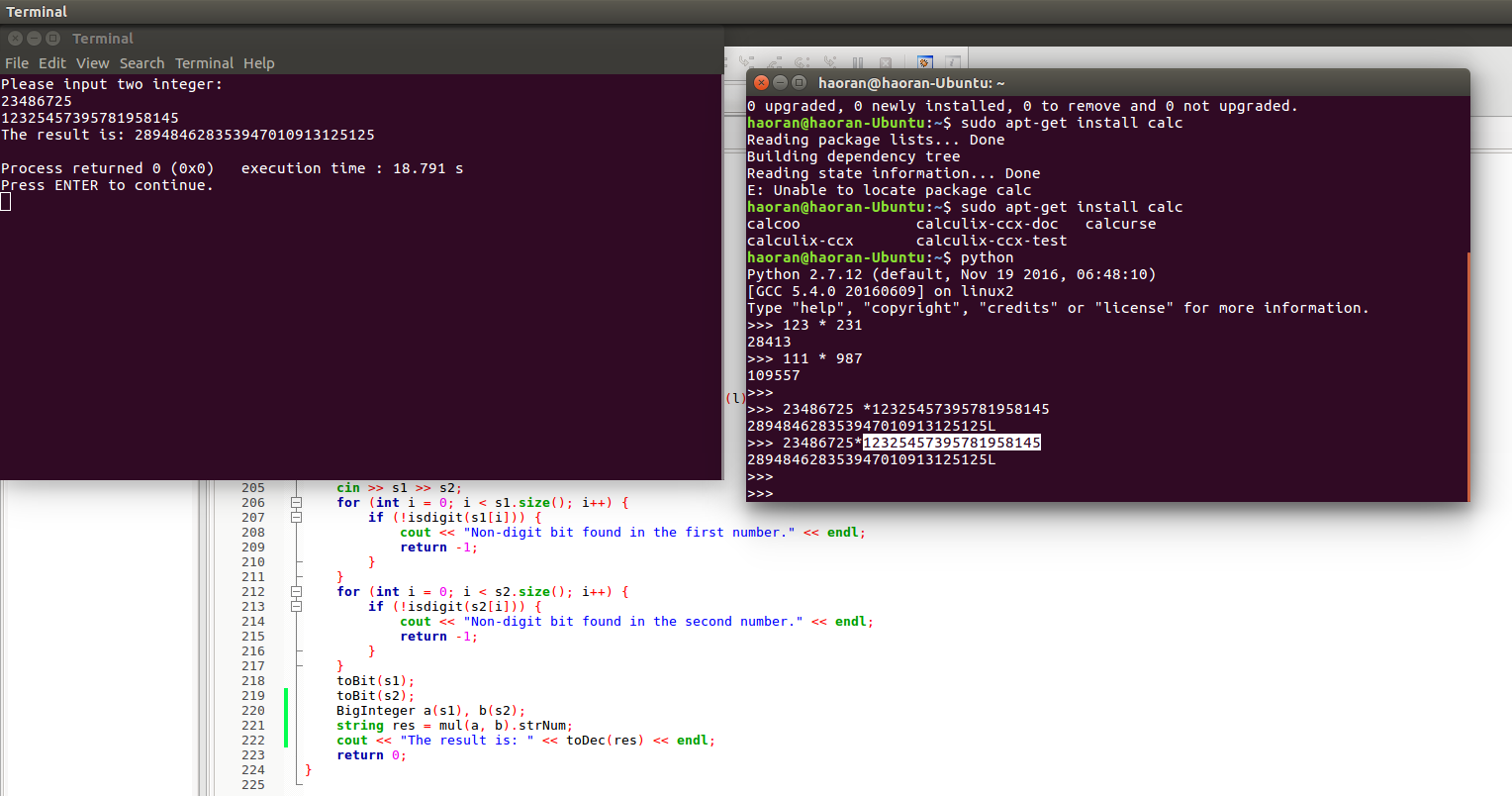
算法的主题部分大致如实验原理中所描述，但在最开始的时候会做一个判断，如果两个大整数的规模足够小，那么可以直接调用乘法运算，节约常数开销。

**源代码：**



**实验结果**

第一步先手动输入两个大整数计算结果，并将结果与python计算的结果进行比较，记录如下：



可以看出两个程序得到的结果相同，所以程序正确。

再调整MINI参数，观察程序在不同的常数优化下的表现：

(其中n表示两个乘数的十进制长度，MINI表示直接调用乘法的最大规模，MINI=1意味着没有常数优化，测试时间的单位均为秒)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | Origin | MINI=1 | MINI=10 | MINI=100 |
| 10 | 2.70E-05 | 0.000149 | 4.00E-05 | 2.50E-05 |
| 100 | 0.000975 | 0.00519 | 0.00139 | 0.00093 |
| 1000 | 0.0934 | 0.194 | 0.0512 | 0.0374 |
| 10000 | 8.54 | 7.61 | 1.87 | 1.39 |
| 100000 | 935 | 298 | 81.4 | 60.4 |
| 1000000 | INF | 2.40E+04 | 1.58E+04 | 7.70E+03 |

可以看出，Karatsuba算法相比于朴素的乘法有较大的优化效果，同时常数优化可以将运行时间下降到未加之前的30%。

**实验总结**

* Karatsuba可以有效优化大整数乘法的时间效率。
* Karatsuba具有较大的常数，未加常数优化的原始算法的时间复杂度相比于平方时间复杂度的算法没有明显的优势。
* 加上常数优化的Karatsuba算法可以将运行时间优化到平方算法的20%以下。
* 实验代码仍有较大的改进空间：
  + 大整数运算中涉及到许多字符串的处理，这部分运算的时间复杂度较高。
  + 字符串每一位的存储的信息只有一个比特，造成了较大的空间浪费，可以考虑用bitset，但是bitset必须在参与运算之前确定长度，一般性太差。另一种解决方案是将二进制大整数的连续若干位压成一个32位或者64位的整型，这样不仅节约了存储空间，还能实现二进制加法的超前进位，可以想见，能引入较大的时间效率优化。