

浙江工业大学

数据结构课程设计

2024/2025(1)



课设题目 大整数运算

学生姓名

学生学号

学生班级 软工2305

任课教师 毛国红

提交日期 2025.01

计算机科学与技术学院

目录

| | |
|---------------------|----|
| 一、实验内容及要求..... | 3 |
| 1.1 问题描述 | 3 |
| 1.2 基本要求 | 3 |
| 1.3 实现提示 | 3 |
| 1.4 运行结果要求 | 4 |
| 二、实验开发环境..... | 4 |
| 三、实验课题分析..... | 4 |
| 3.1 系统总体设计 | 4 |
| 3.2 系统功能设计 | 6 |
| 3.3 类的设计 | 6 |
| 3.3.1 类的设计 | 6 |
| 3.3.2 主要函数的设计 | 13 |
| 3.4 主程序的设计 | 20 |
| 四、调试分析..... | 21 |
| 4.1 实验的调试和测试 | 21 |
| 4.2 技术难点分析 | 21 |
| 4.3 调试测试错误分析 | 22 |
| 五、测试结果分析..... | 22 |
| 5.1 测试结果展示..... | 22 |
| 5.2 收获与不足..... | 23 |
| 六、附录：源代码..... | 24 |
| 6.1 list.h..... | 24 |
| 6.2 io.h..... | 27 |
| 6.3 io.cpp..... | 28 |

| | |
|-----------------------|----|
| 6.4 complex.h | 31 |
| 6.4 complex.cpp | 31 |
| 6.5 big.h | 32 |
| 6.6 big.cpp | 33 |
| 6.7 main.cpp | 43 |

数据结构课程设计(大整数运算系统)实验报告

一、实验内容及要求

1.1 问题描述

密码学分为两类密码：对称密码和非对称密码。对称密码主要用于数据的加/解密，而非对称密码则主要用于认证、数字签名等场合。非对称密码在加密和解密时，是把加密的数据当作一个大的正整数来处理，这样就涉及到大整数的加、减、乘、除和指数运算等，同时，还需要对大整数进行输出。请采用相应的数据结构实现大整数的加、减、乘、除和指数运算，以及大整数的输入和输出。

1.2 基本要求

1. 要求采用链表来实现大整数的存储和运算，不允许使用标准模板类的链表类(list)和函数。同时要求可以从键盘输入大整数，也可以文件输入大整数，大整数可以输出至显示器，也可以输出至文件。大整数的存储、运算和显示，可以同时支持二进制和十进制，但至少支持十进制。大整数输出显示时，必须能清楚地表达出整数的位数。测试时，各种情况都需要测试，并附上测试截图；要求测试例子要比较详尽，各种极限情况也要考虑到，测试的输出信息要详细易懂，表明各个功能的执行正确；

2. 要求大整数的长度可以不受限制，即大整数的十进制位数不受限制，可以为十几位的整数，也可以为 500 多位的整数，甚至更长；大整数的运算和显示时，只需要考虑正的大整数。如果可能的话，请以秒为单位显示每次大整数运算的时间；

3. 要求采用类的设计思路，不允许出现类以外的函数定义，但允许友元函数。主函数中只能出现类的成员函数的调用，不允许出现对其它函数的调用。

4. 要求采用多文件方式：.h 文件存储类的声明，.cpp 文件存储类的实现，主函数 main 存储在另外一个单独的 cpp 文件中。如果采用类模板，则类的声明和实现都放在.h 文件中。

5. 不强制要求采用类模板，也不要求采用可视化窗口；要求源程序中有相应注释；

1.3 实现提示

1. 大整数的加减运算可以分解为普通整数的运算来实现；而大整数的乘、除和指数运算，可以分解为大整数的加减运算。

2. 大整数的加、减、乘、除和指数运算，一般是在求两大整数在取余操作下的加、减、乘、除和指数运算，即分别求 $(a + b) \bmod n$, $(a - b) \bmod n$, $(a * b) \bmod n$, $(a / b) \bmod n$ 和

$(a^b) \bmod n$ 。其中 a^b 是求 a 的 b 次方，而 n 称之为模数。说明：取余操作(即mod操作) 是计算相除之后所得的余数，不同于除法运算的是，取余操作得到的是余数，而不是除数。如 $7 \bmod 5 = 2$ 。模数 n 的设定，可以为 2^m 或 10^m ， m 允许每次计算时从键盘输入。模数 n 的取值一般为 2^{512} (相当于十进制 150 位左右), 2^{1024} (相当于十进制200~300 位), 2^{2048} (相当于十进制 300~500 位)。为了测试，模数 n 也可以为 2^{256} , 2^{128} 等值。

3. 需要设计主要类有：链表类和大整数类。链表类用于处理链表的相关操作，包括缺省构造函数、拷贝构造函数、赋值函数、析构函数、链表的创建、插入、删除和显示等；而大整数类则用于处理大整数的各种运算和显示等。

1.4 运行结果要求

要求能实现大整数的加、减、乘、除和指数运算，以及大整数的输入和输出，实验报告要求有详细的设计思路、功能测试截图。

二、实验开发环境

硬件环境：

操作系统：Windows 11

处理器：AMD Ryzen 7 7840HS, 3.80 GHz

内存：16.0 GB (13.7 GB 可用)

系统类型：64 位操作系统, 基于 x64 的处理器

软件环境：

IDE：VS Code 1.96.2

编译器版本：gcc version 14.2.0 (Rev2, Built by MSYS2 project)

Python版本：Python 3.12.3 （用于计算结果校验）

三、实验课题分析

3.1 系统总体设计

大整数运算系统主要功能为：实现任意长度正整数的加法、减法、乘法、除法、指数运算、取模运算、数据文件读取、结果文件写入。通过用户和计算机之间在终端的交互，即计算机终端显示操作提示后，用户在键盘上输入任意长度的长整数/文件读取对应行数据并进行相应的运算，然后程序将结果输出显示至终端交互界面/写入输出文件中，既可以将内存中的运行结果显示在交互界面中，也可以实现和硬盘中文件的连接将文

件中内容显示在交互界面中。

本程序中设计了六个类：

Node类：节点类，用于生成数据存储的最小单元

list类：链表类，实现Node类双向连接，用于存储大整数和链表基本功能的实现。

Iterator类：迭代器类，提供类似于STL的访问和操作体验。

Big类：用于实现大整数即list类对象的加减乘除指数取模等功能实现。

complex类：用于存储和实现FFT带入单位根时需要的多项式的点值表示的复数结果。

IO类：用于快速读写整数，字符，字符数组与Big类。在文件读写时还可以调用fread/fwrite实现更快的读写。

系统的交互设计：

1. 用户进入界面，依次输入模的底数，模的指数，第一个运算数，运算符，第二个运算数。在长整数运算系统中数字字符限制在0~9，即仅支持十进制输入。但可以是任意长度，输入回车来结束输入。
2. **仅支持模底数为2或10**，当模底数为10时，模指数需要为0（代表不取模）或者为4的倍数，当模底数为2时，模指数需要为0（代表不取模）或者为8的倍数。
3. 用户也可以通过参数的形式从文件读入和写出，但是输入格式需要遵循一定的规则，其他限制同直接交互一样。
4. 本系统还实现了以毫秒为单位计算各个运算所运行的相应时间，并将程序所用的运行时间打印显示。实际测试除了除法，其他都比Python运行时间快。

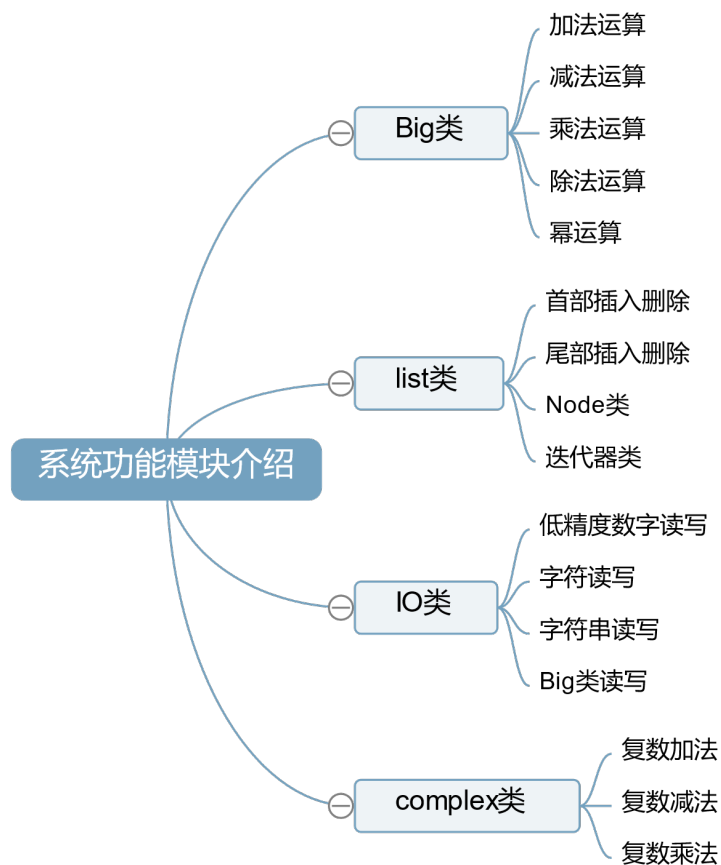


图 1 系统功能模块介绍

3.2 系统功能设计

1. main()函数系统功能调用模块

main()函数实现系统总体运行，调用其他各模块功能实现四则运算及指数运算。

根据调用时是否存在参数决定从文件还是交互页面读入。然后依次读入模底数，模指数，运算数1，运算符，运算操作，读入完成后开始计时，运算，然后输出运算结果和运行时间。

3.3 类的设计

3.3.1 类的设计

1. complex类

| 类名 | 成员函数和成员 | 功能 |
|---------|---|---------|
| complex | <code>constexpr double complex::real()</code> | 返回复数的实部 |
| | <code>constexpr double complex::imag()</code> | 返回复数的虚部 |
| | <code>complex complex::operator*(const complex &other) const</code> | 复数乘法 |
| | <code>complex operator-(const complex &other) const;</code> | 复数减法 |
| | <code>complex operator+(const complex &other) const;</code> | 加法 |
| | <code>complex(double x = 0.0, double y = 0.0);</code> | 构造函数 |
| | <code>double x;</code> | 实部 |
| | <code>double y;</code> | 虚部 |

表 1 complex类

2. IO类

| 类名 | 成员函数和成员 | 功能 |
|----|--|---|
| IO | <code>char buf[MAXSIZE]</code> | 用于 <code>fread</code> 输入的缓冲区 |
| | <code>char pbuf[MAXSIZE]</code> | 用于 <code>fwrite</code> 输出的缓冲区 |
| | <code>char gc();</code> | 基本等价于 <code>getchar()</code> 的其他读入函数的工具函数 |
| | <code>bool blank(char ch)</code> | 确认是否为一系列空白字符或EOF |
| | <code>void read(int &x);</code> | 读入数字 |
| | <code>void read(char *s);</code> | 读入字符串 |
| | <code>void read(char &c);</code> | 读入字符 |
| | <code>void read(Big &b);</code> | 读入字符串然后转换成Big类的封装 |
| | <code>void push(const char &c);</code> | 输出字符（其他一系列输出的工具函数） |

| | | |
|--|--|---|
| | <code>void write(int x, int width = -1, int base = 10);</code> | 根据确定的进制（2进制或10进制）和宽度（不足补0输出数字）输出数字 |
| | <code>void write(const char s[]);</code> | 输出字符串 |
| | <code>void write(const Big &b);</code> | 输出Big类，最高位不用补0，其他根据压位补0 |
| | <code>char *p1, *p2</code> | 控制输入缓冲区的左右指针，即当前一共读入到了p2位置，读入缓冲区但未给程序的为p1至p2之间，p1前已经读入给程序 |
| | <code>char *pp</code> | 控制输出缓冲区的指针 |
| | <code>IO();</code> | 构造函数 |
| | <code>~IO();</code> | 析构函数 |

表 2 IO类

3.list类

| 类名 | 成员函数和成员 | 功能 |
|------|--|-------------|
| list | <code>Node *head;</code> | 指向第一个元素的指针 |
| | <code>Node *tail;</code> | 指向最后一个元素的指针 |
| | <code>size_t siz;</code> | 当前元素个数 |
| | <code>list();</code> | 构造函数 |
| | <code>~list();</code> | 析构函数 |
| | <code>list &operator=(const list<T> &other)</code> | 赋值函数 |
| | <code>list(const list<T> &other)</code> | 复制构造函数 |
| | <code>size_t size() const</code> | 返回当前元素个数 |
| | <code>bool empty() const</code> | 返回当前链表是否为空 |
| | <code>void clear();</code> | 清空链表 |
| | <code>void push_back(const T &value);</code> | 在链表末尾插入一个元素 |
| | <code>void push_front(const T &value);</code> | 在链表首部插入一个元素 |

| | | |
|--|-----------------------------------|----------|
| | <code>void pop_back();</code> | 删除链表尾部元素 |
| | <code>void pop_front();</code> | 删除链表首部元素 |
| | <code>T &back() const</code> | 返回链表尾部元素 |
| | <code>T &front() const</code> | 返回链表首部元素 |

表 3 list类

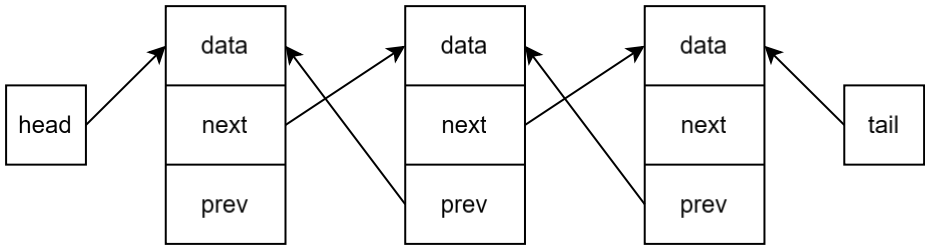


图 2 list 结构示意图

4.Node类

| 类名 | 成员函数和成员 | 功能 |
|------|---------------------------------------|-------------|
| Node | <code>T data;</code> | 数据 |
| | <code>Node *next;</code> | 指向下一个节点的指针 |
| | <code>Node *prev;</code> | 指向上一个节点的指针 |
| | <code>Node(const T &value)</code> | 构造函数（根据给定值） |

表 4 Node类

5.Iterator类

| 类名 | 成员函数和成员 | 功能 |
|----------|---|---------------------|
| Iterator | <code>Iterator(Node *node);</code> | 构造函数 |
| | <code>T &operator*() const</code> | 地址解析函数 |
| | <code>Iterator &operator++()</code> | 前缀自增运算符，使迭代器指向下一个位置 |
| | <code>Iterator operator++(int)</code> | 后缀自增运算符，使迭代器指向后一个位置 |

| | | |
|--|---|-----------------------------|
| | <code>Iterator &operator--()</code> | 前缀自增运算符，使迭代器指向前一个位置 |
| | <code>Iterator operator--(int)</code> | 后缀自增运算符，使迭代器指向前一个位置 |
| | <code>bool operator==(const Iterator &other) const</code> | 判断迭代器指向的位置是否相等 |
| | <code>bool operator!=(const Iterator &other) const</code> | 判断迭代器指向的位置是否不相等 |
| | <code>Iterator begin() const</code> | 返回指向 <code>head</code> 的迭代器 |
| | <code>Iterator end() const</code> | 返回空指针 |
| | <code>Iterator rbegin() const</code> | 返回指向 <code>tail</code> 的迭代器 |

表 5 Iterator类

6. Big 类

| 类名 | 成员函数和成员 | 功能 |
|-----|---|-----------------------------------|
| Big | <code>Big();</code> | 默认构造函数（赋值为0） |
| | <code>Big(int x);</code> | 构造函数（赋值为某个值，默认设置模底数为10，指数为0，即不取模） |
| | <code>Big(int modB, int modP);</code> | 构造函数（赋值为0并设置模底数和指数） |
| | <code>Big(int x, int modB, int modP);</code> | 构造函数（赋值为某个数并设置模底数和指数） |
| | <code>Big(const char number[], int modB = 10, int modP = 0);</code> | 构造函数（从字符串转换为Big类，并设置模底数和指数） |
| | <code>Big(const Big &other);</code> | 复制构造函数 |
| | <code>Big &operator=(const Big &other);</code> | 赋值函数 |
| | <code>Big &operator=(const int &x);</code> | 赋值函数（并从数转换为Big类） |
| | <code>Big operator+(const Big &b) const;</code> | 大整数加法 |
| | <code>Big &operator+=(const Big &b);</code> | 大整数加法赋值运算 |
| | <code>Big operator-(const Big &b) const;</code> | 大整数减法 |
| | <code>Big &operator--(const Big &b);</code> | 大整数减法赋值运算 |
| | <code>Big operator*(const Big &b) const;</code> | 基于快速傅里叶变换的大整数乘法 |
| | <code>Big &operator*=(const Big &b);</code> | 大整数乘法赋值运算 |
| | <code>Big operator/(const Big &b) const;</code> | 大整数倍增除法 |
| | <code>Big &operator/=(const Big &b);</code> | 大整数除法赋值运算 |
| | <code>void div2(int base = 10);</code> | 除以2，用于快速幂的工具函数，并根据当前进制确定借位的数 |
| | <code>int mod2() const;</code> | 模2，用于快速幂的工具函数 |

| | |
|--|------------------------------------|
| <code>bool iszero() const;</code> | 判断是否为0，用于快速幂的工具函数 |
| <code>Big operator^(Big b) const;</code> | 大整数快速幂 |
| <code>bool operator<(const Big &b) const;</code> | 大整数小于比较 |
| <code>bool operator==(const Big &b) const;</code> | 大整数等于比较 |
| <code>bool operator!=(const Big &b) const;</code> | 大整数不等于比较 |
| <code>bool operator>=(const Big &b) const;</code> | 大整数大于等于比较 |
| <code>bool operator<=(const Big &b) const;</code> | 大整数小于等于比较 |
| <code>bool operator>(const Big &b) const;</code> | 大整数大于比较 |
| <code>constexpr int base() const;</code> | 动态确定当前模的底数 |
| <code>constexpr int width() const;</code> | 动态确定当前每一位宽度 |
| <code>void checkMOD() const;</code> | 检查模数是否正确（底数为2时须为8的倍数，底数为10时须为4的倍数） |
| <code>void dec2bin();</code> | 十进制压位转二进制压位 |
| <code>list<i64> digits;</code> | 存储每一位上数的链表 |
| <code>int modB</code> | 模的底数 |
| <code>int modP</code> | 模的指数 |
| <code>void fft_recursive(complex *f, int len, int opt) const;</code> | 递归版快速傅里叶变换的实现 |
| <code>void fft(complex *f, int n, int opt) const;</code> | 常数更小的循环版快速傅里叶变换 |
| <code>void shift(int shift);</code> | 按进制左移右移，类似于二进制操作那样左移右移 |
| <code>void trim();</code> | 去除前导0 |
| <code>void mod();</code> | 根据当前模底数和指数取模 |
| <code>void carry();</code> | 处理加法和乘法运算导致的借位 |

表 6 Big类

3.3.2 主要函数的设计

我们约定下文时间复杂度中的 n 或者 m 为运算数的位数，即运算数为 10^n 或者 2^n 级别。

1. `Big operator+(const Big &b) const;`

加法函数，从前往后同步遍历两个 `Big` 类的所有位，并 `push_back` 到答案的 `Big` 类中。然后处理将较短的还没遍历完的类的后面的所有位也依次 `push_back`。然后再 `push_back` 一个 `0`，用于处理进位后位数增加的情况，然后依次处理进位，处理前导零（如果前面没有出现位数增加），然后处理取模。

时间复杂度 $O(n)$ 。

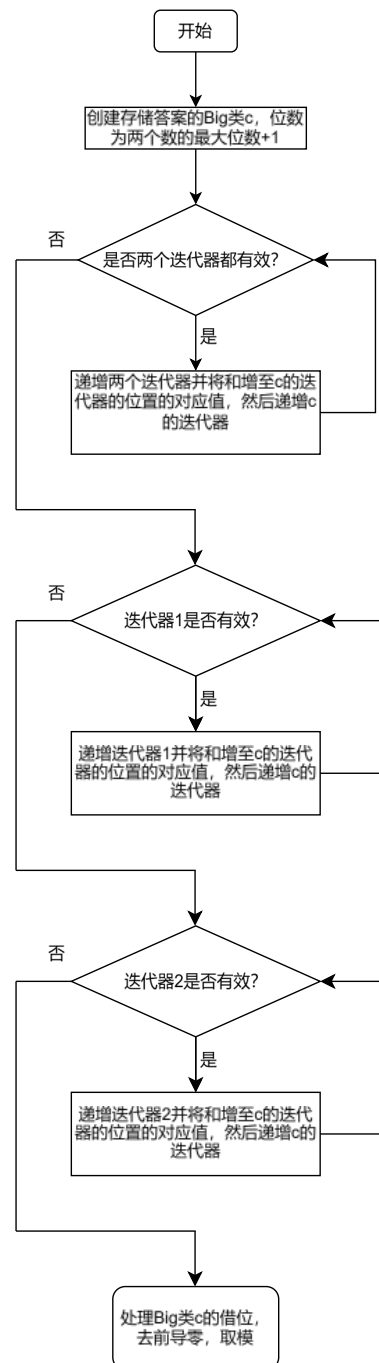


图 3 `Big::operator+` 流程图

2. Big operator-(const Big &b) const;

首先特判是否会导致负数。特别需要注意的是模意义下的负数我们可以转换为正数，因此只要在没有取模的时候特判一下，如果取模了就把小的数加上模数即可。

然后也是从低到高依次遍历每一位，如果出现需要借位的情况，就向更高位找到第一个不为 0 的位，然后把这两个位间都变成 $BASE - 1$ 。由于这样操作完后中间的位都不可能再借位了，每一位最多借一次，因此时间复杂度不会增大。

最终仍然需要处理可能的前导零。然后取模。

时间复杂度 $O(n)$ 。

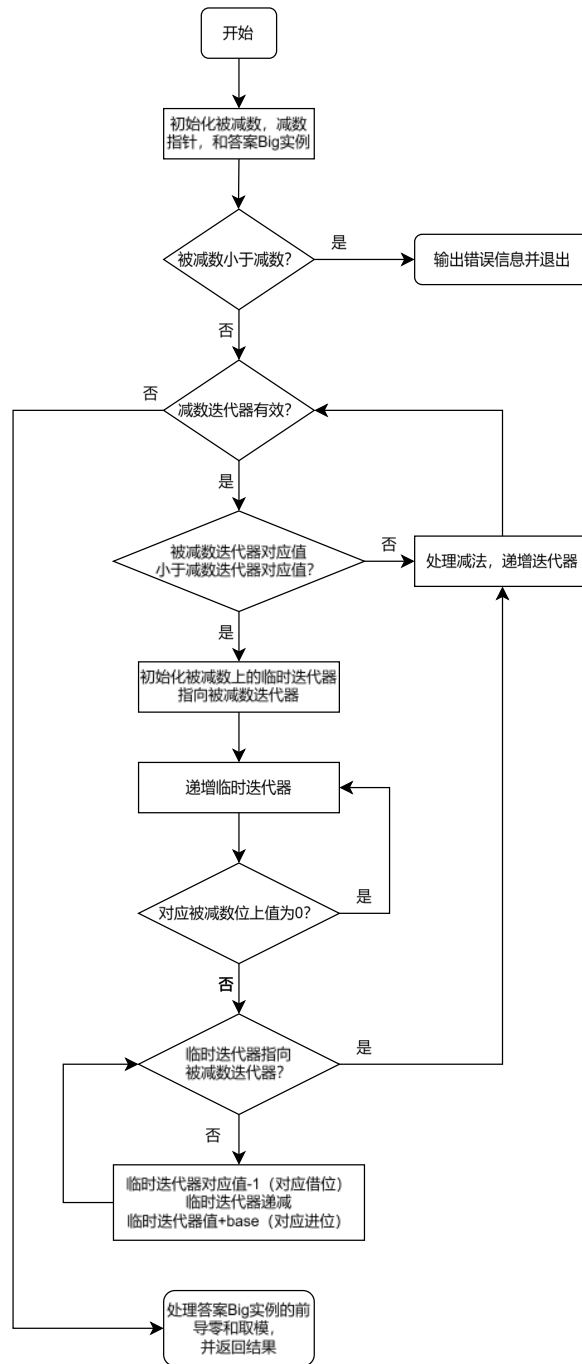


图 4 Big::operator-流程图

3. `Big operator*(const Big &b) const;`

首先，要使用 FFT，位数必须得是 2 的幂。考虑两个数的乘积的位数不多于两个数的位数之和，倍增找到最小的 n 使得其大于等于 $n + m$ ，这一步是 $O(\log n)$ 的。

然后，把用于存储点值表示的数组清空一下。因为需要随机访问，所以为了效率不得不使用数组，如果不清除，结果会被上一次运算的结果干扰。然后遍历链表，将值载入数组。

然后预处理一下用于位逆序置换的数组。这里实际上可以预处理，但由于我没有专门写一个 FFT 的类，因此找不到适合的预处理的地方，毕竟不能保证 `Big` 一定会被用来做 FFT。位逆序置换也就是根据 FFT 递归流程中实际上会将数放到二进制下翻转的数的位置，这个可以利用数位 `dp` 的思想：首先把除了最后一位之外的其他数翻转，然后如果最后一位是 1 就把 1 翻转到最开头。除了第一位数的其他数的翻转可以认为已经计算好了。

然后分别调用 FFT 将两个数转换成点值表示，然后相乘，再做一次 FFT 转换为多项式表示，就可以得到答案数组了，然后把数组转换成链表，然后依次处理进位，前导零和取模即可。

时间复杂度 $O(n \log n)$

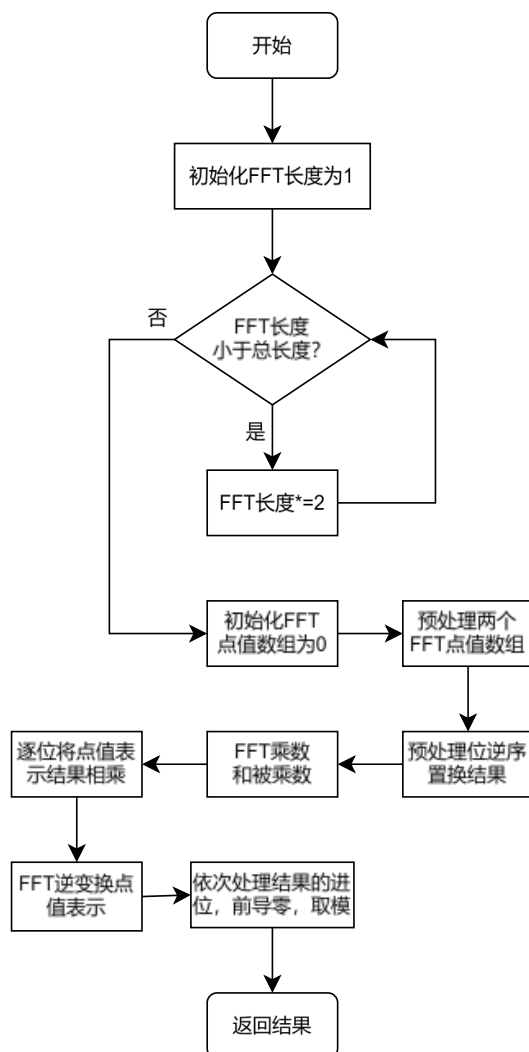


图 5 `Big::operator*`流程图

4. `Big operator/(const Big &b) const;`

首先，我们需要保证被除数大于等于除数，否则返回零。因此除法意义下的整除的取模应该没有什么特别的意义。

这里使用倍增。虽然按位除也可以但感觉倍增更自然。考虑我们实际上计算的是 a/b 后的十进制表达，因此可以按照二进制的思维，按位确定当前的被除数是否大于当前的除数的 2 的若干次幂倍。因为我们可以用二进制表达任何一个数，因此这样贪心是没有问题的。然后逐步从高位向低位确定即可。

也就是我们用当前的 $2^n b$ 来趋近于当前的 a ，并从高到低逐步减小 n ，如果当前的 n 满足 $2^n b \leq a$ ，就把答案加上 2^n 。

具体实现上先要找到最大的 n 满足 $2^n b \leq a$ ，只需要不断把 b 乘 2，直到刚刚比 a 大，这一步可以存储每一次扩大后的 b 和 2^n ，然后逐渐减少，加入答案，就可以了。

时间复杂度 $O(n^2)$ 。

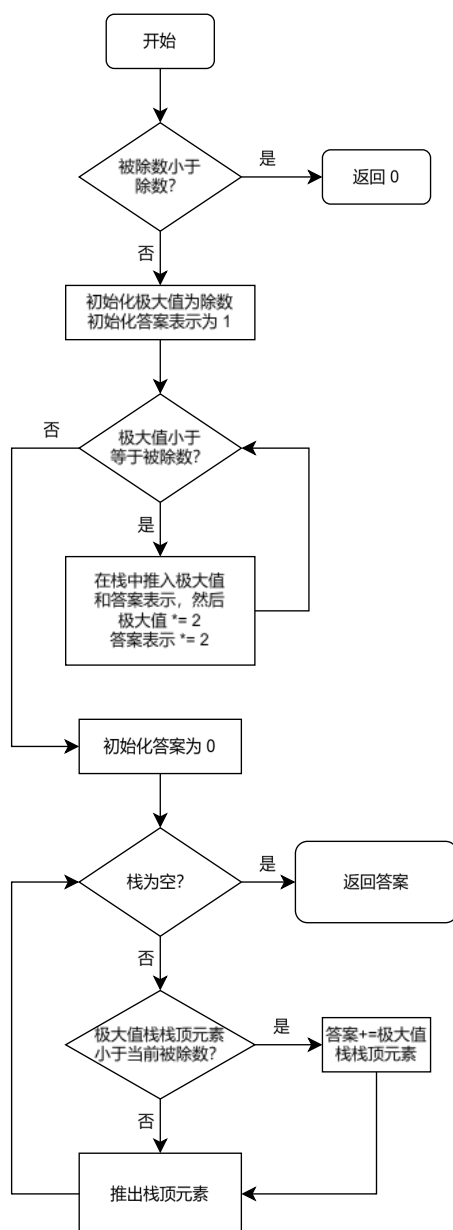


图 6 `Big::operator/`流程图

5. `Big operator^(Big b) const;`

这一步就是普通的快速幂，二进制按位确定 b ，逐位确定这一位是否需要额外乘上 a 。

时间复杂度的分析，我们考虑计算的结果是 10^{n10^m} ，取对数，位数也就是 $n10^m$ ，

考虑我们使用了快速幂，最后一步两个乘数的位数都是 $\frac{n10^m}{2}$ 级别，忽略常数，然后考虑乘法的时间复杂度，最后总的时间复杂度就是 $O(n10^m(\log n + m))$ 。

我们也可从积分的角度分析，复杂度是 $O(\sum_{k=0}^{m \log_2 10} 2^k n \log(2^k n))$ ，如果积分一下，结果也就是 $O(n10^m(\log n + m))$ 。 n 和 m 相等的时候，极限约是 n 和 m 都取6，此时就是 $O(3.6 \times 10^7)$ ，实测几秒内可以跑完，和理论符合的不错。

如果使用 $O(n^2)$ 的乘法，且使用快速幂，时间复杂度也来到了 $O(n^2 10^{2m})$ 。我们也可以利用积分的思想， $O(\sum_{k=0}^{m \log_2 10} 2^{2k} n^2)$ ，结果也是 $O(n^2 10^{2m})$ 。

如果还是暴力乘，那么时间复杂度来到 $O(\sum_{k=1}^{10^m} k n^2)$ ，把作为常数的 n^2 提出来，然后利用积分的思想，复杂度是 $O(n^2 10^{2m})$ 。

上面两个结果还是挺令人奇怪的，快速幂怎么没能够降低时间复杂度？因为快速幂到后面位数增长会十分恐怖，暴力乘的位数增长不大；也就是说低精度下快速幂能降低时间复杂度在于降低了乘法次数，而没有降低乘法规模。

换言之，快速幂通过增大乘数规模降低运算次数，在低精度下无论乘数规模多少乘法运算时间都是 $O(1)$ ，高精度下乘法运算时间随乘数规模迅速增加，且乘数规模随运算次数指数增大，因此快速幂对于降低整体时间复杂度是无效的。

但是在取模的意义下就很不同了，由于乘数规模不会过分扩大，此时快速幂远优于暴力连乘。复杂度来到 $O(md \log d)$ 。而指数过大时不取模也意义不大，因为根本没有哪里存的下那么大的数字。

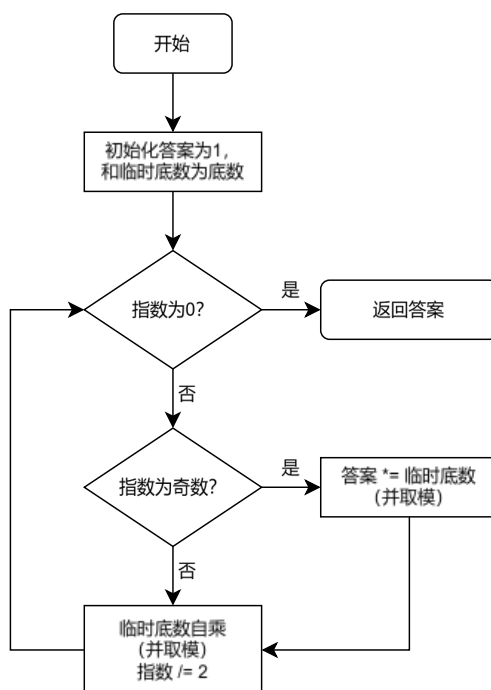


图 7 `Big::operator^` 流程图

6. `Big operator<(Big b) const;`

用于处理各种运算需要的大小比较，首先判断位数，位数不同直接返回结果。否则从高位向低位逐位比较即可。而定义了小于我们就可以基于此直接得到其他五个大小关系了。

时间复杂度 $O(n)$ 。

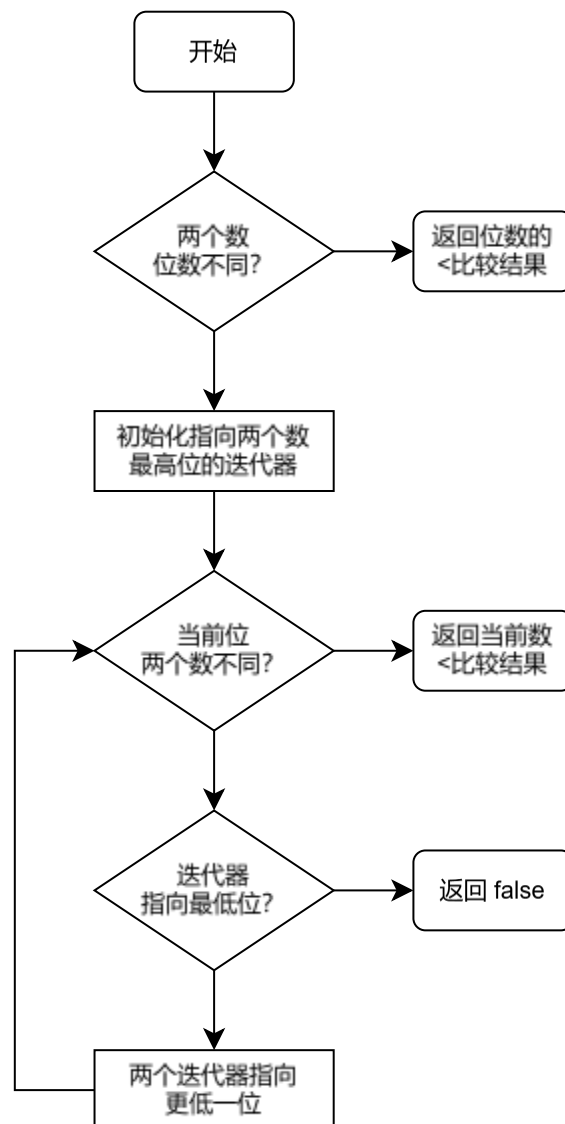


图 8 `Big::operator<`流程图

7. `void fft(complex *f, int n, int opt) const;`

首先根据预先计算的位逆序置换的结果，直接 `swap`。然后用类似于区间 `dp` 的思维，依次从小到大计算区间的结果，然后从小的合并到大的即可。具体实现上有一些可以优化常数的地方。也要注意由于右侧的值依赖于原先已经计算好的现在在左侧的值，因此需要先计算右边的，然后覆盖左边的才行。反正就是利用单位根性质。

时间复杂度 $O(n\log n)$ 。

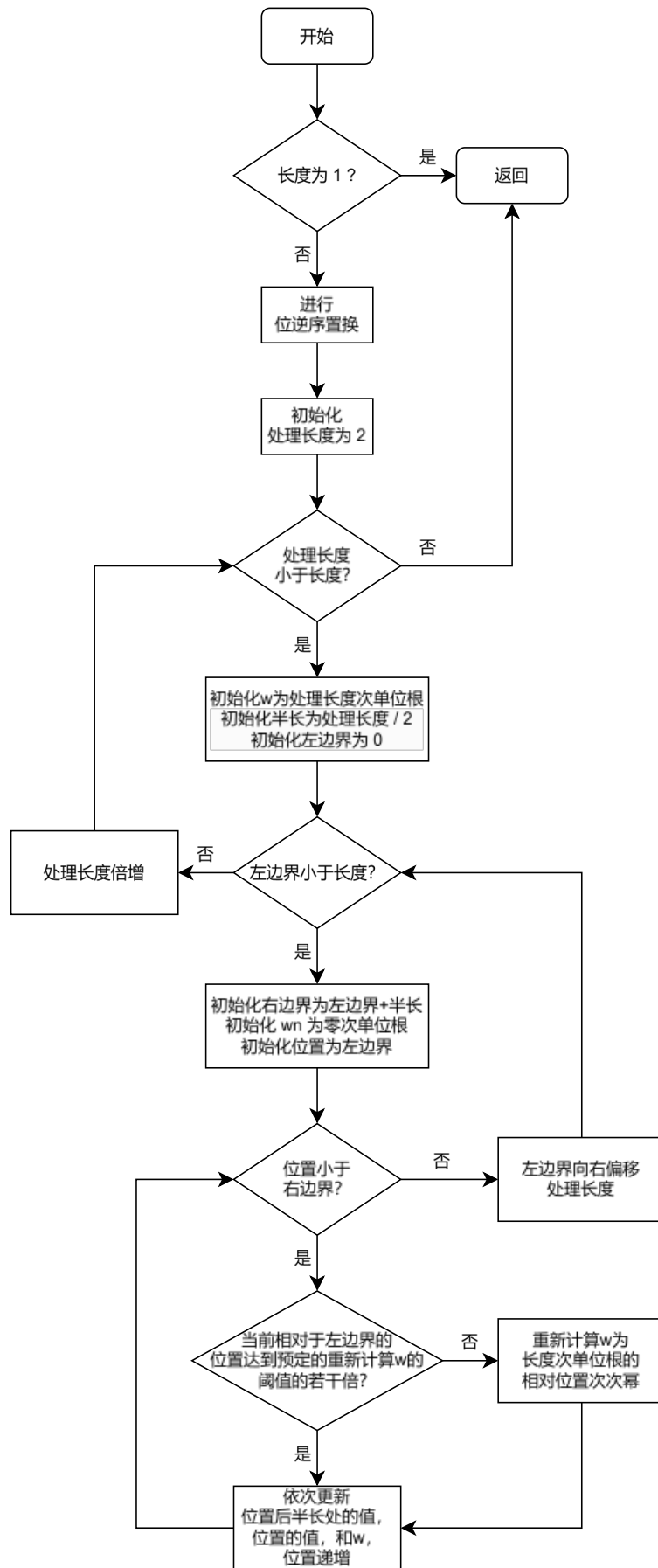


图 9 Big::fft流程图

3.4 主程序的设计

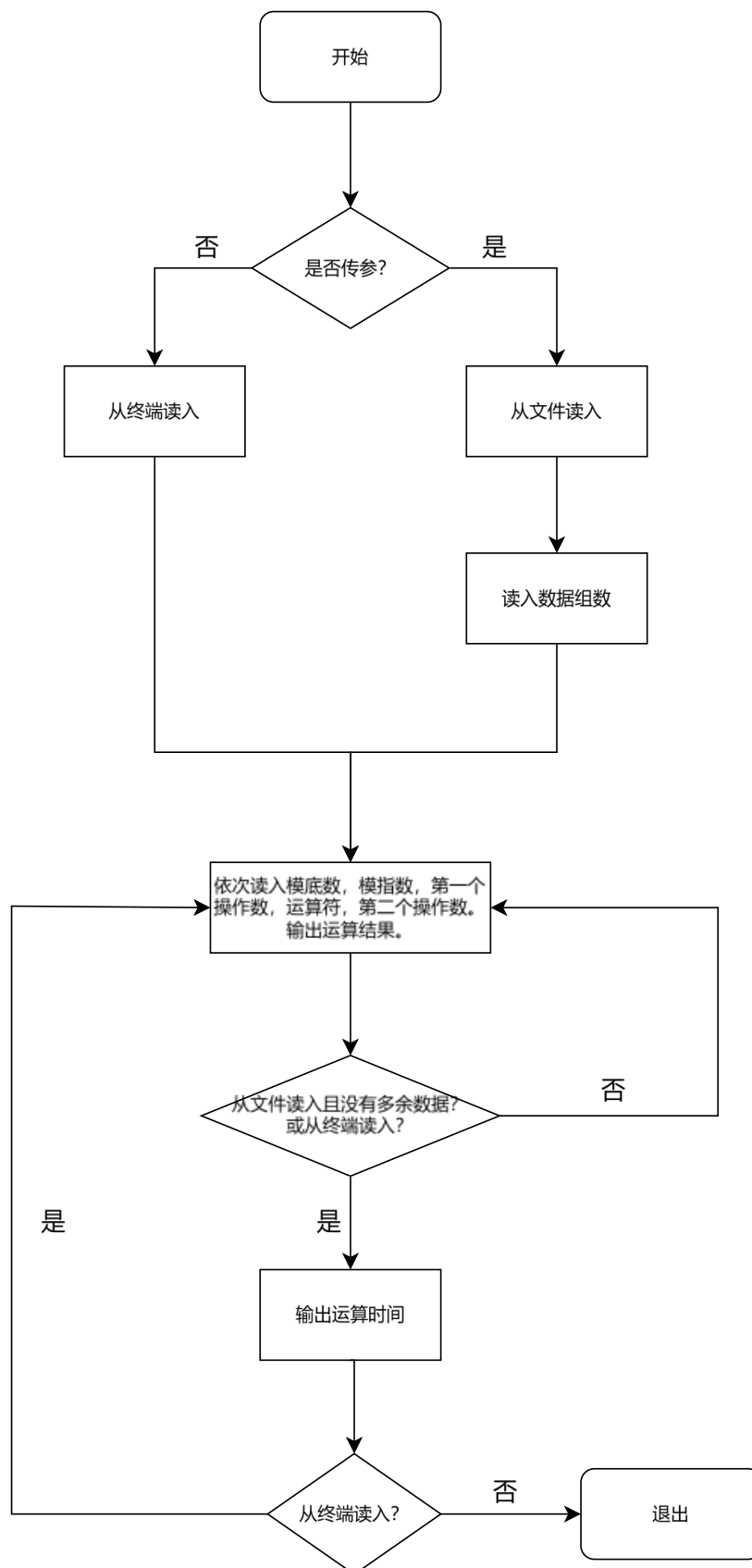


图 10 主程序流程图

四、调试分析

4.1 实验的调试和测试

1. 输入的形式：键盘输入或文件输入。键盘输入时根据提示依次输入模底数，模指数，运算数1，运算符，运算数2。从文件输入时需要首先输入测试数据组数，然后按上述格式输入。

4.2 技术难点分析

1. 普通的乘法运算速度过慢，时间复杂度达 $O(n^2)$ ，难以忍受。因此需要使用FFT来实现，可以做到 $O(n\log n)$ 。

2. 除法速度同样过慢，时间复杂度达 $O(n^2)$ ，虽然根据北京大学倪泽堃的著名论文《理性愉悦——高精度数值计算》可以在不依赖高精度实数的前提下做到 $O(n\log n)$ ，但实现难度过高，因此本程序没有加入。

3. 单向链表显然是不行的，考虑大小比较时必须从高位向低位遍历，而加法和乘法又必须从低位向高位遍历（需要处理进位），我们必须采用双向链表或数组。这里根据要求也只能使用双向链表。链表虽然不支持随机访问，但是链表相对于数组的好处在于可以很方便的从头部插入，具体到数上也就是很方便的进行shift操作。对于除法会很方便。因此采用双向链表，自头至尾依次存储低位到高位。

4. FFT的有递归实现和循环实现两种。虽然时间复杂度一样，都是 $O(n\log n)$ ，但是后者的常数更小，实际测试在 $n = 10^6$ 级别下，前者为268ms，后者为162ms，约快40%。

5. 存储高精度整数，优化常数必不可少的一部分是压位。如果使用普通的 $O(n^2)$ 的乘法，我们在使用long long的情况下可以压九位，也就是十亿进制，而不导致溢出。但是如果使用FFT，那么为了避免double的精度损失，在平衡使用long double与double的常数的优化下，我们最多只能压2位。

6. 有没有什么办法避免FFT中单位根计算时double的精度损失？是有的，只需要隔一定次数重新直接计算即可，表现在代码上就是。

```
if (!(i - 1) % 1024)
    w = complex(cos(2 * pi * (i - 1) / len),
                sin(2 * pi * (i - 1) / len) * opt);
```

在如是优化的情况下，我们可以把压位的位数增加到4位。实测没法再压更多位数了。同时为了便于取模，我们也要令压的位数最好是2的幂，这样只需要直接pop_back()即可。

7. 二进制取模。我们没法直接向上面对十进制取模那样对二进制取模。因为10中含有因数5，压位后没法直接pop_back()。因此我们需要将压位十进制转换为二进制再转换为压位二进制，同时压的位数也最好位2的幂。实际测试只能压8位，也就是10进制下的256进制，至于压16位的65536进制，虽然不会导致溢出，但似乎也存在double的精度丢失问题。而long double的运行效率远远慢于double，因此压2进制下的8位。

8.快速读写。我们可以不使用scanf和printf，只使用putchar和getchar来提高读写效率。实际上如果不考虑用户界面，我们还可以使用fread和fwrite来进一步提高读写效率。但由于二者本质上是冲突的，因此只能在编译期决定选择哪一种方法。后者没有回显不能用于用户从终端读写。

9.对二进制取模但是十进制输出，对十进制取模但是对二进制输出，由于采用了压位的万进制和256进制，两者都没什么优秀的解决方法，除了 $O(n)$ 的暴力转换，但是任务书并没有要求具体的输出是几进制，因此还是可以接受的。

10.FFT要求随机访问，因此为了运行效率存储点值表示时使用链表是肯定会大大降低效率的。

4.3 调试测试错误分析

1.一开始直接压9位来fft，小的时候还没问题，数字大的时候直接错了，会有这样奇怪的输出结果。后来发现最多压两位。再后来才发现是double精度损失，在洛谷上找到了解决方法，也就是隔一段时间重新计算，利用单位根性质。

```
1 - .
186005412622027504143107840506778880673505430807477221
189343787318847717780694840495597612143138784468174887
384101846708172789429271477601244135858350376109570040
```

图 11 错误debug

2.链表本来想写个兼容std::list的版本，但是弄半天也没弄出来end()指向最后一个节点的后一个节点，就还是让tail指向最后一个确定的元素了。

3.压位十进制转二进制的时候，由于是直接除，因此数组里存的实际上是低位到高位，但我仍然搞成了高位到低位的一般的读入形式，因此调了好久。

4.快速幂中判断数是否为0的时候也出现了奇怪的问题，gdb显示的数据和程序实际运行结果不同，明明判断条件应该没错误却一直在死循环。因此改成了直接判断是否存在某一位不为0，才解决了快速幂中死循环的问题。

5.多组数据运算的时候会出现奇怪的问题，后来发现应该是由FFT的数组没清空。

五、测试结果分析

5.1 测试结果展示

界面读写：包括了输出时间

```
-> # ./main.exe
请依次输入模底数与模指数，第一个运算数，运算符，第二个运算数：
10 0 213422 ^ 762311
```

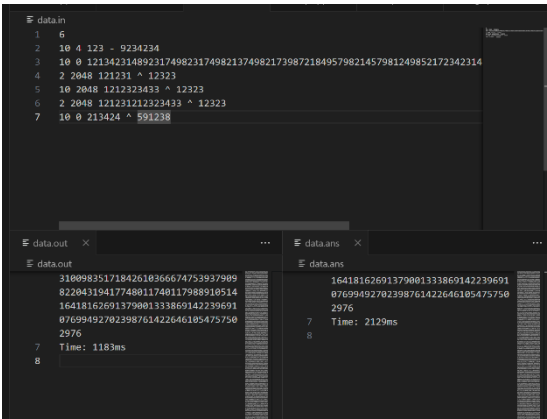
图 12 界面读写输入图

```
49667377129611041161300303745099663099723162651186213256390821635017410957563819508863293183254
21843386945023297568993849512211391771359682117320703526504354300543332359743879177195061102344
60254772282427536636656720650318666119645944135547718683421533148037033714399798599596861774820
18258304096157647646865414496292890156653155881975365683024526919450679689887349838877940816001
29357018102089732334496506592468904871829865632735751184474467210942588742213698316099146352699
25571206571385008998390028971121289413932958957436928
Time: 2602ms
请依次输入模底数与模指数，第一个运算数，运算符，第二个运算数：

```

图 13 界面读写输出图

文件读写：可以看到在结果完全一致的前提下，时间仅为 Python 的一半



The image shows a comparison of file I/O performance between C and Python. The top window, 'data.in', contains the same input as Figure 12. Below it, two windows show the output: 'data.out' (C) and 'data.ans' (Python). Both show identical results, but the C version took 1183ms while the Python version took 2129ms.

图 14 文件读写输入图

```
-> # ./main.exe data.in data.out && python test.py < data.in > data.ans && diff data.ans data.o
ut
7c7
< Time: 2129ms
---
> Time: 1183ms
```

图 15 文件读写输出图

5.2 收获与不足

- 1. 复习了FFT的写法，并且学习到了链表的高精度运算的实现。
- 2. 要有坚定的意志与永不言弃的精神，要善于变通，要用于挑战。
- 3. 不足在于高精度除法的实现实在没时间学，而且也很难读懂。而且FFT本身也可以有多种优化，不禁让我想到优化这条路的任重道远。

六、附录：源代码

6.1 list.h

```
#ifndef LIST_H
#define LIST_H

#ifdef _WIN32
    using size_t = unsigned long long;
#elif __linux__
    using size_t = unsigned long;
#else
    #error "Unsupported platform"
#endif

template <typename T> class list {
private:
    struct Node {
        T data;
        Node *next;
        Node *prev;
        Node(const T &value) : data(value), next(nullptr), prev(nullptr) {}
    };

public:
    Node *head;
    Node *tail;
    size_t siz;

    list() : head(nullptr), tail(nullptr), siz(0) {}
    ~list() { clear(); }
    list(const list<T> &other) : head(nullptr), tail(nullptr), siz(0) {
        for (const T &item : other) {
            push_back(item);
        }
    }
    list &operator=(const list<T> &other) {
        if (this != &other) {
            clear();
            for (const T &item : other) {
                push_back(item);
            }
        }
        return *this;
    }
};
```

```

}
size_t size() const { return siz; }
bool empty() const { return siz == 0; }

void clear() {
    Node *cur = head;
    while (cur) {
        Node *next = cur->next;
        delete cur;
        cur = next;
    }
    head = tail = nullptr;
    siz = 0;
}

void push_back(const T &value) {
    Node *newNode = new Node(value);
    if (!tail) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    siz++;
}

void push_front(const T &value) {
    Node *newNode = new Node(value);
    if (!head) {
        head = tail = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    siz++;
}

void pop_back() {
    if (!tail) {
        throw "pop_back on empty list";
    }
    Node *oldTail = tail;
    tail = tail->prev;
    if (tail) {

```

```

        tail->next = nullptr;
    } else {
        head = nullptr;
    }
    delete oldTail;
    siz--;
}

void pop_front() {
    if (!head) {
        throw "pop_front on empty list";
    }
    Node *oldHead = head;
    head = head->next;
    if (head) {
        head->prev = nullptr;
    } else {
        tail = nullptr;
    }
    delete oldHead;
    siz--;
}

T &back() const {
    if (!tail) {
        throw "back on empty list";
    }
    return tail->data;
}

T &front() const {
    if (!head) {
        throw "front on empty list";
    }
    return head->data;
}

class Iterator {
private:
    Node *cur;

public:
    Iterator(Node *node) : cur(node) {};

    T &operator*() const {
        if (!cur) {

```

```

        throw "Dereferencing a null iterator";
    }
    return cur->data;
}
Iterator &operator++() {
    if (cur) {
        cur = cur->next;
    }
    return *this;
}
Iterator operator++(int) {
    Iterator temp = *this;
    ++(*this);
    return temp;
}
Iterator &operator--() {
    if (cur) {
        cur = cur->prev;
    }
    return *this;
}
Iterator operator--(int) {
    Iterator temp = *this;
    --(*this);
    return temp;
}
bool operator==(const Iterator &other) const {
    return cur == other.cur;
}
bool operator!=(const Iterator &other) const { return !(*this == other); }
};

Iterator begin() const { return Iterator(head); }
Iterator end() const { return Iterator(nullptr); }
Iterator rbegin() const { return Iterator(tail); }
};
#endif

```

6.2 io.h

```

#ifndef IO_H
#define IO_H

#define MAXSIZE 1 << 22

```

```

#include "big.h"

class IO {
    char buf[MAXSIZE], *p1, *p2;
    char pbuf[MAXSIZE], *pp;

public:
    IO();
    ~IO();

    char gc();
    bool blank(char ch);
    constexpr bool isdigit(char ch) const;
    void read(int &x);
    void read(char *s);
    void read(char &c);
    void read(Big &b);
    void push(const char &c);
    void write(int x, int width = -1, int base = 10);
    void write(const char s[]);
    void write(const Big &b);
};

#endif

```

6.3 io.cpp

```

#define DEBUG 1 // 调试开关

```

```

// 快速读写，从文件读入时可以注释掉上面的 debug，但是交互的时候必须不注释，否则不会显示，
// 在数据量达到 1e8 级别时提升明显
// 在 1e6 范围内可以提升 10%左右

```

```

#include "io.h"
#include "big.h"
#include "big.cpp"

#include <cstdio>

#if DEBUG
IO::IO() {};
IO::~IO() {};
#else
IO::IO() : p1(buf), p2(buf), pp(pbuf) {}

```

```

IO::~IO() { fwrite(pbuf, 1, pp - pbuf, stdout); }
#endif
char IO::gc() {
    #if DEBUG // 调试, 可显示字符
        return getchar();
    #endif
    if (p1 == p2)
        p2 = (p1 = buf) + fread(buf, 1, MAXSIZE, stdin);
    return p1 == p2 ? ' ' : *p1++;
}

bool IO::blank(char ch) {
    return ch == ' ' || ch == '\n' || ch == '\r' || ch == '\t' || ch == -1;
}

constexpr bool IO::isdigit(char ch) const { return ch >= '0' && ch <= '9'; }

void IO::read(int &x) {
    double tmp = 1;
    bool sign = false;
    x = 0;
    char ch = gc();
    for (; !isdigit(ch); ch = gc())
        if (ch == '-') {
            write("Negative numbers are not allowed!\n");
            exit(-1);
        }
    for (; isdigit(ch); ch = gc())
        x = x * 10 + (ch - '0');
    if (ch == '.') {
        write("Floating point numbers are not allowed!\n");
        exit(-1);
    }
}

void IO::read(char *s) {
    char ch = gc();
    for (; blank(ch); ch = gc())
        ;
    for (; !blank(ch); ch = gc())
        *s++ = ch;
    *s = 0;
}

```

```

void IO::read(char &c) {
    for (c = gc(); blank(c); c = gc())
        ;
}

void IO::read(Big &b) {
    char s[MAXSIZE];
    read(s);
    b = Big(s, b.modB, b.modP);
}

void IO::push(const char &c) {
    #if DEBUG // 调试, 可显示字符
        putchar(c);
    #else
        if (pp - pbuf == MAXSIZE)
            fwrite(pbuf, 1, MAXSIZE, stdout), pp = pbuf;
        *pp++ = c;
    #endif
}

void IO::write(int x, int width, int modB) {
    static int sta[90];
    int top = 0;
    do {
        sta[top++] = x % modB, x /= modB;
        if (width != -1)
            width--;
    } while (x || width != -1 && width);
    while (top)
        push(sta[--top] + '0');
}

void IO::write(const char s[]) {
    int i = 0;
    while (s[i]) {
        push(s[i++]);
    }
}

void IO::write(const Big &b) {
    auto i = b.digits.rbegin();
    write(*i, -1, b.modB);
    if (i == b.digits.begin())

```

```

        return;
    i--;
    while (1) {
        write(*i, !b.width() ? -1 : b.width(), b.modB);
        if (i == b.digits.begin())
            break;
        i--;
    }
}

```

6.4 complex.h

// 简单的复数实现, 用于 fft

```

#ifndef COMPLEX_H
#define COMPLEX_H

class complex {
private:
    double x;
    double y;

public:
    complex(double x = 0.0, double y = 0.0);

    complex operator+(const complex &other) const;
    complex operator-(const complex &other) const;
    complex operator*(const complex &other) const;

    constexpr double real();
    constexpr double imag();
};
#endif

```

6.4 complex.cpp

```

#include "complex.h"

complex::complex(double x, double y) : x(x), y(y) {}

complex complex::operator+(const complex &other) const {
    return complex(x + other.x, y + other.y);
}

complex complex::operator-(const complex &other) const {
    return complex(x - other.x, y - other.y);
}

```



```

}

complex complex::operator*(const complex &other) const {
    return complex(x * other.x - y * other.y, x * other.y + y * other.x);
}

constexpr double complex::real() { return x; }
constexpr double complex::imag() { return y; }

```

6.5 big.h

```

#ifndef BIG_H
#define BIG_H

#include "complex.h"
#include "complex.cpp"
#include "list.h"
#include "list.cpp"

using u32 = unsigned int;
using i64 = long long;
using u64 = unsigned long long;

constexpr int WIDTH10 = 4;
constexpr int BASE10 = 1e4;

constexpr int WIDTH2 = 8; // 16 疑似会掉 double 精度
constexpr int BASE2 = 1 << 8;

class Big {
public:
    Big();
    Big(int x);
    Big(int modB, int modP);
    Big(int x, int modB, int modP);
    Big(const char number[], int modB = 10, int modP = 0);
    Big(const Big &other);

    Big &operator=(const Big &other);
    Big &operator=(const int &x);

    Big operator+(const Big &b) const;
    Big &operator+=(const Big &b);
    Big operator-(const Big &b) const;
    Big &operator-=(const Big &b);

```

```

Big operator*(const Big &b) const;
Big &operator*=(const Big &b);
Big operator/(const Big &b) const;
Big &operator/=(const Big &b);

void div2(int base = 10);
int mod2() const;
bool iszero() const;
Big operator^(Big b) const;

bool operator<(const Big &b) const;
bool operator==(const Big &b) const;
bool operator!=(const Big &b) const;
bool operator>=(const Big &b) const;
bool operator<=(const Big &b) const;
bool operator>(const Big &b) const;

friend class IO;

constexpr int base() const;
constexpr int width() const;
void checkMOD() const;
void dec2bin();
void bin2Dec();

private:
    list<i64> digits;
    int modB, modP;

    void fft_recursive(complex *f, int len, int opt) const;
    void fft(complex *f, int n, int opt) const;
    void shift(int shift);
    void trim();
    void mod();
    void carry();
};
#endif

```

6.6 big.cpp

```

#include "big.h"
#include "complex.h"
#include "io.h"

#include <cmath>

```

```

IO io;

const double pi = acos(-1);

complex f[MAXSIZE], g[MAXSIZE], sav[MAXSIZE];

int tr[MAXSIZE], buf[MAXSIZE];

Big::Big() { digits.push_back(0); }
Big::Big(int x) {
    digits.clear();
    if (!x) { // 避免链表为空导致的一系列内存问题
        digits.push_back(0);
    }
    while (x) {
        digits.push_back(x % base());
        x /= base();
    }
}
Big::Big(int modB, int modP) : modB(modB), modP(modP) {
    checkMOD();
}
Big::Big(int x, int modB, int modP) {
    this->modB = modB;
    this->modP = modP;

    checkMOD();

    digits.clear();
    if (!x) {
        digits.push_back(0);
    }
    while (x) {
        digits.push_back(x % base());
        x /= base();
    }
}

// 从字符串读入, 不使用 std::string 以提升性能
Big::Big(const char number[], int modB, int modP) {
    this->modB = modB;
    this->modP = modP;
}

```

```

checkMOD();

int tmp = 0, len = 0;
while (number[len]) {
    len++;
}
for (int r = len; r > 0; r -= WIDTH10) { // 十进制转压位十进制
    int l = std::max(r - WIDTH10, 0);
    tmp = 0;
    for (int i = l; i < r; i++) {
        tmp = tmp * 10 + number[i] - '0';
    }
    digits.push_back(tmp);
}
if (modB == 2) {
    dec2bin();
}
mod();
}

```

```

Big::Big(const Big &other) {
    digits = other.digits;
    modB = other.modB;
    modP = other.modP;
}

```

```

Big &Big::operator=(const Big &other) {
    digits = other.digits;
    modB = other.modB;
    modP = other.modP;
    return *this;
}

```

```

Big &Big::operator=(const int &x) {
    *this = Big(x, 10, 0);
    return *this;
}

```

// 简单的依次相加即可

// O(n)

```

Big Big::operator+(const Big &b) const {
    Big c(modB, modP);
    auto i1 = digits.begin(), i2 = b.digits.begin();
    while (i1 != digits.end() && i2 != b.digits.end()) {

```

```

        c.digits.push_back(*i1 + *i2);
        i1++, i2++;
    }
    while (i1 != digits.end()) {
        c.digits.push_back(*i1);
        i1++;
    }
    while (i2 != b.digits.end()) {
        c.digits.push_back(*i2);
        i2++;
    }
    c.digits.push_back(0);
    c.carry();
    c.trim();
    c.mod();
    return c;
};

Big &Big::operator+=(const Big &b) {
    *this = *this + b;
    return *this;
}

// 简单的借位即可
// O(n)
Big Big::operator-(const Big &b) const {
    Big a = *this;
    if (a < b) {
        io.write("Error: negative number\n");
        exit(-1);
    }
    auto i1 = a.digits.begin();
    auto i2 = b.digits.begin();
    while (i2 != b.digits.end()) {
        if (*i1 < *i2) {
            auto j = i1;
            j++;
            while (!*j)
                j++;
            while (j != i1) {
                --*j;
                j--;
                *j += base();
            }
        }
    }
}

```

```

        *i1 -= *i2;
        i1++, i2++;
    }
    a.trim();
    a.mod();
    return a;
};

Big &Big::operator--(const Big &b) {
    *this = *this - b;
    return *this;
}

// 乘法使用了循环实现的 fft，用了位逆序置换和蝶形运算优化，可以降低常数，
// 复杂度 O(nlogn)
Big Big::operator*(const Big &b) const {
    int n = digits.size(), m = b.digits.size();
    m += n, n = 1;
    while (n < m) // 长度必须为 2 的幂，因此在前面补零
        n <<= 1;
    // 我也不确定要不要这么做，但是初始化可以避免奇怪的问题
    for (int i = 0; i < n; i++) {
        f[i] = g[i] = 0;
    }
    int i = 0;
    for (auto it = digits.begin(); it != digits.end(); it++) {
        f[i++] = {double(*it), 0};
    }
    i = 0;
    for (auto it = b.digits.begin(); it != b.digits.end(); it++) {
        g[i++] = {double(*it), 0};
    }
    for (int i = 0; i < n; i++) {
        tr[i] = (tr[i >> 1] >> 1) | (i & 1 ? n >> 1 : 0);
        // 位逆序置换，数位 dp 的思想，提前放好位置
    }
    fft(f, n, 1);
    fft(g, n, 1);
    // 将多项式的系数表达转换为点值表达，从而降低运算复杂度
    for (int i = 0; i < n; i++) {
        f[i] = f[i] * g[i];
    }
    // 将多项式的点值表达转换为系数表达
    fft(f, n, -1);
    i = 0;
    Big c(modB, modP);

```

```

while (i < m) {
    c.digits.push_back(f[i++].real() / n + 0.5);
}
c.carry(); // 同样需要处理进位
c.trim();
c.mod();
return c;
};

Big &Big::operator*=(const Big &b) {
    *this = *this * b;
    return *this;
};
// 采用倍增，因为更快的  $O(n \log n)$  的算法实在来不及学了
// 复杂度  $O(n^2)$ 
Big Big::operator/(const Big &b) const {
    if (*this < b) {
        return Big(0, 10, 0);
    }
    Big x = b, y(1, modB, modP);
    // std::list<Big> s1, s2;
    list<Big> s1, s2;

    for (; x <= *this; x += x, y += y)
        s1.push_back(x), s2.push_back(y);

    x = *this;
    Big ans(0, 10, 0);

    for (; !s1.empty(); s1.pop_back(), s2.pop_back())
        if (s1.back() <= x) {
            x -= s1.back();
            ans += s2.back();
        }
    return ans;
};

Big &Big::operator/=(const Big &b) {
    *this = *this / b;
    return *this;
};
// 方便快捷幂
void Big::div2(int base) {
    int carry = 0;
    for (auto it = digits.rbegin(); --it) {

```

```

    int new_digit = *it + carry * base;
    *it = new_digit / 2;
    carry = new_digit % 2;
    if (it == digits.begin())
        break;
}
trim();
}
// 同样是方便快捷幂
int Big::mod2() const { return digits.front() % 2; }
bool Big::iszero() const {
    for (auto &i : digits) {
        if (i) {
            return false;
        }
    }
    return true;
}
// 朴素的快速幂实现
// 复杂度约为  $O(nm \log nm)$ ，但实际上常数很小，因此跑得还算快
// n, m 为底数和指数的位数，考虑最后一步的复杂度是两个 nm 位的数相乘，因此远远跑不满
Big Big::operator^(Big b) const {
    Big ans(1, modB, modP), a(*this);
    ans.mod();
    a.mod();
    while (!b.iszero()) {
        if (b.mod2())
            ans = ans * a, ans.mod();
        a = a * a, a.mod();
        b.div2(base());
    }
    ans.trim();
    ans.mod();
    return ans;
};

// 我们只需要一个 <
// 就可以得到其他所有偏序关系，
// 但是由于需要取模，因此大小比较基本上没有意义，仅仅用于避免出现负数
bool Big::operator<(const Big &b) const {
    if (digits.size() != b.digits.size()) {
        return digits.size() < b.digits.size();
    }
    for (auto i1 = digits.rbegin(), i2 = b.digits.rbegin();;) {

```



```

    if (*i1 != *i2) {
        return *i1 < *i2;
    }
    if (i1 == digits.begin())
        break;
    i1--, i2--;
}
return false;
}
bool Big::operator==(const Big &b) const {return !(*this < b) && !(b < *this);}
bool Big::operator!=(const Big &b) const { return !(*this == b); }
bool Big::operator>=(const Big &b) const { return !(*this < b); }
bool Big::operator<=(const Big &b) const { return *this < b || *this == b; }
bool Big::operator>(const Big &b) const { return !(*this <= b); }

```

// 递归实现的 **fft**，方便理解的初级版本，主要利用了单位根的性质和分治思想
 // 从而加速 **DFT**，复杂度 $O(n\log n)$

```

void Big::fft_recursive(complex *f, int len, int opt) const {
    if (len == 1)
        return;
    int mid = len / 2;
    for (int i = 0; i < len; i++) {
        sav[i] = f[i];
    }
    complex *fl = f, *fr = f + mid;
    for (int i = 0; i < mid; i++) {
        fl[i] = sav[i << 1], fr[i] = sav[i << 1 | 1];
    }
    fft_recursive(fl, mid, opt);
    fft_recursive(fr, mid, opt);
    complex w(cos(2 * pi / len), sin(2 * pi / len) * opt);
    complex c(1, 0);
    for (int i = 0; i < mid; i++) {
        sav[i] = fl[i] + c * fr[i];
        sav[i + mid] = fl[i] - c * fr[i];
        c = c * w;
    }
    for (int i = 0; i < len; i++) {
        f[i] = sav[i];
    }
}

```

// 常数更小的 **fft** 的循环写法，洛谷 **P1919** 比上面的版本快一倍，虽然时间复杂度相同

```

void Big::fft(complex *f, int n, int opt) const {

```

```

if (n == 1)
    return;
for (int i = 0; i < n; i++) {
    if (i < tr[i]) {
        std::swap(f[i], f[tr[i]]);
    }
}
for (int len = 2; len <= n; len <= 1) {
    complex wn(cos(2 * pi / len), sin(2 * pi / len) * opt);
    int mid = len >> 1;
    for (int l = 0; l < n; l += len) {
        int r = l + mid;
        complex w(1, 0), t;
        for (int i = l; i < r; i++) {
            // 因为 double 的精度损失, 为了压位, 我们需要重新计算准确的 wn 值
            // 如果不弄, 最多压 2 位, 也就是 100, 现在我们压 4 位
            if (!(i - l) % 1024)
                w = complex(cos(2 * pi * (i - l) / len),
                             sin(2 * pi * (i - l) / len) * opt);
            t = f[i + mid] * w;
            f[i + mid] = f[i] - t;
            f[i] = f[i] + t;
            w = w * wn; // 这里会导致 double 的精度损失
        }
    }
}
}

// 用于快速除法的工具函数, 但是来不及学了
void Big::shift(int shift) {
    if (shift >= 0) {
        for (int i = 0; i < shift; i++) {
            digits.push_front(0);
        }
    } else {
        shift = -shift;
        for (int i = 0; i < shift; i++) {
            digits.pop_front();
        }
    }
}

// 去除前导零
void Big::trim() {

```

```

    while (digits.size() > 1 && digits.back() == 0) {
        digits.pop_back();
    }
}

// 根据底数压位后就可以很方便的取模
void Big::mod() {
    while (modP && digits.size() * width() > modP) {
        digits.pop_back();
    }
}

// 处理借位
void Big::carry() {
    auto i = digits.begin(), j = ++digits.begin();
    while (j != digits.end()) {
        *j += *i / base();
        *i %= base();
        i++, j++;
    }
}

// 动态确定当前的 BASE 和 WIDTH
constexpr int Big::base() const {return modB == 2 ? BASE2 : BASE10;}
constexpr int Big::width() const {return modB == 2 ? WIDTH2 : WIDTH10;}

// 检查模数是否合法，当前仅支持模底数为 2 或 10，
// 当模底数为 2 时，模指数必须为 8 的倍数，当模底数为 10 时，模指数必须为 4 的倍数
void Big::checkMOD() const {
    if (modP % width() || modP == 1) {
        io.write(modP);
        io.write("\nError: modP must be 0 or a multiple of width\n");
        exit(-1);
    }
    if (modB != 10 && modB != 2) {
        io.write(modB);
        io.write("\nError: modB must be 10 or 2\n");
        exit(-1);
    }
}

// 压位十进制转压位二进制，因为我们只考虑了十进制输入，因此不能直接从二进制读入，需要从十进制转换

```

```

void Big::dec2bin() { // 改变内部压位规则，不取模
    int len = 0;
    while (!iszero()) {
        buf[len++] = mod2();
        div2(BASE10);
    }
    list<i64> digits2;
    for (int l = 0; l < len; l += WIDTH2) { // 由于前面是倒着的，因此这里从最低位到最高位
        int r = std::min(l + WIDTH2, len);
        int tmp = 0;
        for (int i = r - 1; i >= l; i--) {
            tmp = tmp * 2 + buf[i];
        }
        digits2.push_back(tmp);
    }
    digits = digits2;
}

```

6.7 main.cpp

```

#include "big.h"
#include "io.h"
#include "io.cpp"

#include <chrono>

signed main(int argc, char **argv) {
    if (argc > 1) {
        freopen(argv[1], "r", stdin);
        if (argc > 2) {
            freopen(argv[2], "w", stdout);
        }
        int t;
        io.read(t);
        auto start = std::chrono::high_resolution_clock::now();
        while (t--) {
            int b, p;
            char op;
            io.read(b);
            io.read(p);
            Big x(b, p), y(b, p);
            io.read(x);
            io.read(op);
            io.read(y);
            if (op == '+') {

```

```

        io.write(x + y);
    } else if (op == '-') {
        io.write(x - y);
    } else if (op == '*') {
        io.write(x * y);
    } else if (op == '/') {
        io.write(x / y);
    } else if (op == '^') {
        io.write(x ^ y);
    }
    io.push('\n');
}

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> elapsed = end - start;
io.write("Time: ");
io.write(elapsed.count());
io.write("ms\n");
} else {
    while (1) {
        io.write("请依次输入模底数与模指数，第一个运算数，运算符，第二个运算数:\n");

        int b, p;
        char op;
        io.read(b);
        io.read(p);
        Big x(b, p), y(b, p);
        io.read(x);
        io.read(op);
        io.read(y);

        auto start = std::chrono::high_resolution_clock::now();
        if (op == '+') {
            io.write(x + y);
        } else if (op == '-') {
            io.write(x - y);
        } else if (op == '*') {
            io.write(x * y);
        } else if (op == '/') {
            io.write(x / y);
        } else if (op == '^') {
            io.write(x ^ y);
        }
        io.push('\n');
        auto end = std::chrono::high_resolution_clock::now();

```

```
std::chrono::duration<double, std::milli> elapsed = end - start;
io.write("Time: ");
io.write(elapsed.count());
io.write("ms\n");
}
}
return 0;
}
```