武汉大学计算机学院
本科生实验报告

数据结构实验报告
实验四：大整数计算器

专 业 名 称 ： 计算机科学与技术

课 程 名 称 ： 数据结构

指 导 教 师 ： 安　　扬

学 生 学 号 ： 2017301500061

学 生 姓 名 ： 彭 思 翔

学 生 班 级 ： 计科二班

上 机 环 境 ： Visual Studio Code

二〇一八 年 11 月

一、实验题目

实验三：大整数计算器

【问题描述】

实现大整数（200 位以内的整数）的加、减、乘、除运算。

【基本要求】

设计程序实现两个大整数的四则运算，输出这两个大整数的和、差、积、商及余数。

【实现提示】

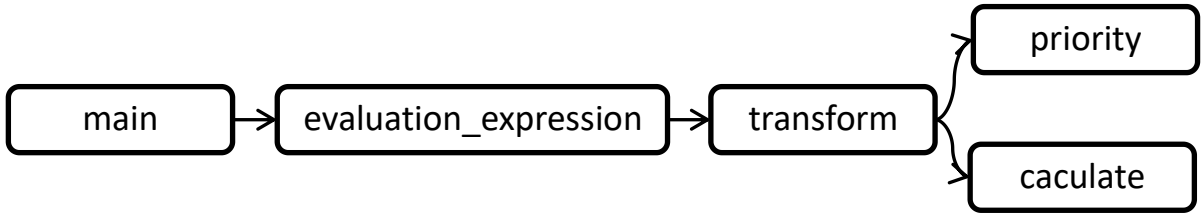
由于整型数据存储位数有限，因此引入串的概念，将整型数据用字符串进行存储，利用字符串的一个字符存储大整数的一位数值，然后根据四则运算规则，对相应位依次进行相应运算，同时保存进位，从而实现大整数精确的运算。

二、实验目的

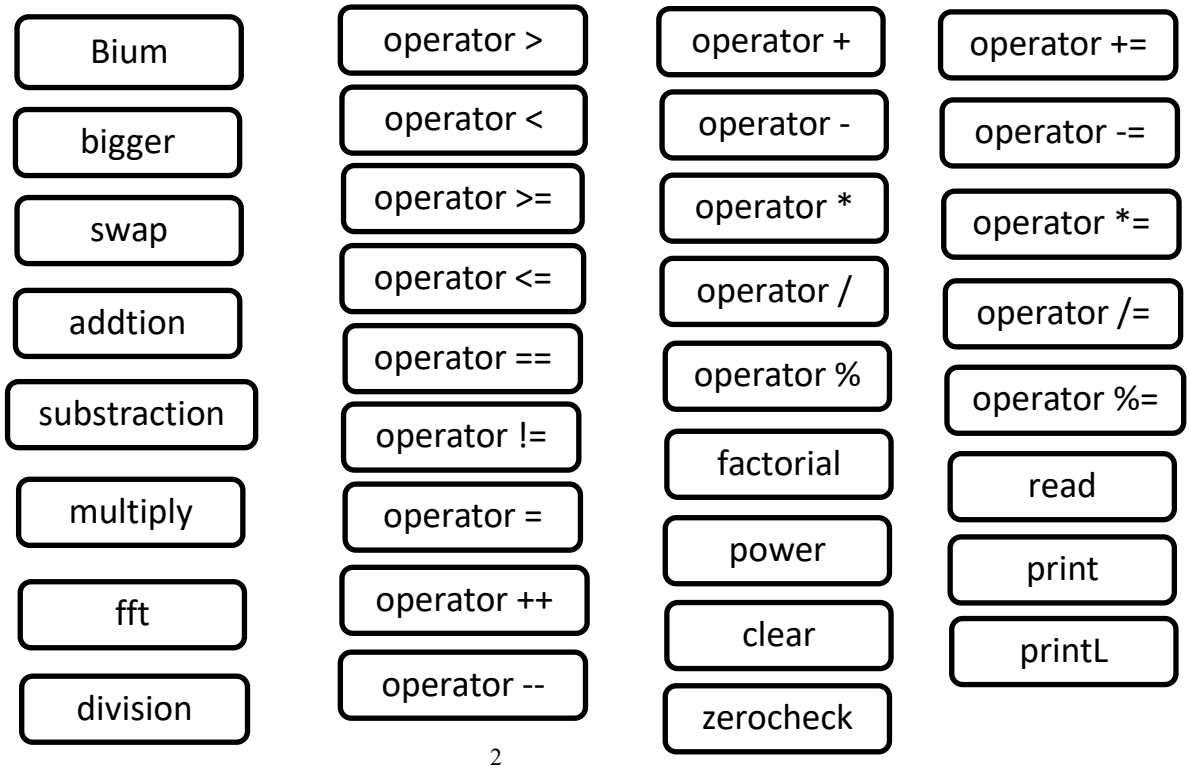
深入掌握串的使用，掌握高精度问题的求解。

三、实验项目程序结构

表达式计算过程



Bium 类的方法



四、实验项目中各文件函数功能描述

【过程函数】

```
int priority(char);                    //Compare the priority of operator
void transform(char*);                 //Get the value of the expression
void evaluation_expression();          //Read a expression and Get the value of the expression
Bium calculate(Bium&, Bium&, char);    //Do calculate
```

【Bium 方法函数】

```
class Bium {
    public:
        Bium();                                          // Structure for nothing
        Bium (Bium &);                                   // Structure for Bium
        Bium (const int);                                // Structure for int
        Bium (const char *);                             // Structure for string

        int bigger(const Bium&, const Bium&);
        //Compare if former is bigger than later,if they are the same return 2

        void swap(Bium&, Bium&);                    //Swap two Biums
        void addtion(Bium&, Bium&, Bium&);          //Store the second plus the third in the first
        void multiply(Bium&, Bium&, Bium&);         //Store the second times the third in the first
        void division(Bium&, Bium&, Bium&);
        //Store the second divides the third in the first, and left the remainder in the second
        void subtraction(Bium&, Bium&, Bium&);
        //Store the second minus the third in the first, after two have the same sg
        void fft(complex<double>*, int*, int, int); //Fast Fourier transform

        bool operator > (const Bium&);              //Compare if former is bigger than later
        bool operator < (const Bium&);              //Compare if former is smaller than later
        bool operator >= (const Bium&);             //Compare if former is not smaller than later
        bool operator <= (const Bium&);             //Compare if former is not bigger than later
        bool operator == (const Bium&);             //Compare if former and later are the same
        bool operator != (const Bium&);             //Compare if former and later are not the same

        Bium operator + (Bium&);                //Reload +
        Bium operator - (Bium&);                //Reload -
        Bium operator * (Bium&);                //Reload *
        Bium operator / (Bium&);                //Reload /
        Bium operator % (Bium&);                //Reload %

        Bium factorial();               //Caculate factorial
        Bium power(Bium&);              //Calculate the nth power, n is the parameter

        Bium& operator = (const int&);                      //Reload = for int
```

```
        Bium& operator = (const Bium&);                    //Reload = for Bium
        Bium& operator = (const char*&);                   //Reload = for string

        Bium& operator += (Bium&);                         //Reload +=
        Bium& operator -= (Bium&);                         //Reload -=
        Bium& operator *= (Bium&);                         //Reload *=
        Bium& operator /= (Bium&);                         //Reload /=
        Bium& operator %= (Bium&);                         //Reload %=

        Bium& operator ++ ();                              //Reload pre ++
        Bium& operator -- ();                              //Reload pre --
        Bium operator ++ (int);                            //Reload post ++
        Bium operator -- (int);                            //Reload post --

        read ();                                  //Read the Bium
        clear ();                                 //Clear the Bium
        print ();                                 //Print the Bium
        printL ();                                //Print the Bium
        zerocheck();                              //Remove the zero in the front of the Bium
};
```

五、算法描述

【数据结构】

Bium 类：表示大整数，len 为长度，sg 为该数符号，char*为每一位的数字。为了计算方便，char*内的每个数字字符向前平移了 48 个 ascall 码。

数字字符	原 ASCALL 码	平移后 ASCALL 码
0	48	0
1	49	1
2	50	2
3	51	3
4	52	4
5	53	5
6	54	6
7	55	7
8	56	8
9	57	9

类里面为了方便计算，重载了运算符+、-、*、/、%的运算，并加入了阶乘!和幂运算^的方法，重载了各类赋值运算、比较运算和前置后置的自加自减。详细可以见函数功能描述。

```
class Bium {
    private:
```

```
        char num[MAXLEN];        //num[] for the number,
        int len, sg;             //len is the lenth of number, sg is the sign
}
```

栈：用于中缀表达式转后缀表达式，有两个，一个是运算字符栈，一个是数字栈。因为相对简单所以没有单独用结构体或类定义。

```
char stack1[MAXSTACK];
Bium stack2[MAXSTACK];
int top1 = 0, top2 = 0;
```

【设计思路】

（1）表达式求值：由于输入是表达式，所以我们要处理表达式的计算。我们都知道，平常我们所见的表达式是中缀表达式，即操作符以中缀形式位于操作数之间。

```
格式:"操作数 1 操作符 操作数 2"
12 * (3 + 4) - 6 + 8 / 2; // 中缀表达式
```

如果表达式是以后缀表达式表示的，那么只需要依次读取表达式，遇见数字压入数字栈中，遇见字符则从栈中取出两个数计算并将结果压回栈中。

```
格式："操作数 操作符"
12 3 4 + * 6 - 8 2 / +; //后缀表达式
```

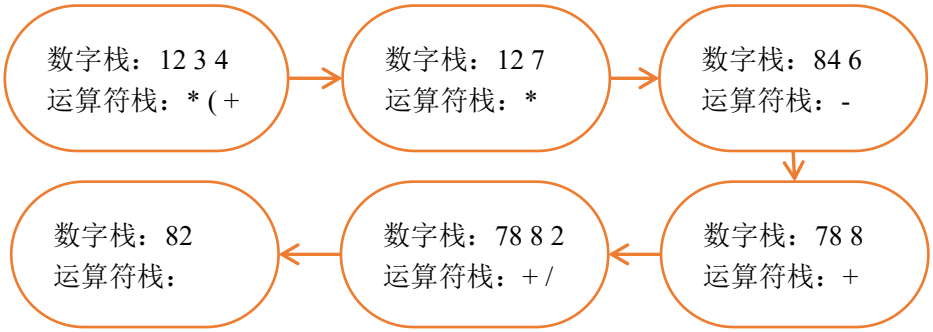
那么如何将中缀转为后缀呢？从中缀表达式中从左往右依次取出数据，并维护一个运算符栈：

符号	操作
数字	加入后缀队列
(	直接入栈
)	把栈里的操作符依次出栈并插入到后缀序列后面，直到遇到'('并弹出栈
其他	如果操作符的优先级比栈顶的符号优先级高，则入栈；当操作符优先级等于或小于栈顶符号优先级，则将栈顶出栈并插入到后缀序列的后面，直到此操作符比栈顶优先级高，将此操作符入栈。
如果中缀队列里的数据已经读取完毕，记录操作符的栈里，还有操作符的话，依次出栈插入到后缀序列的后面。在表达式最外层加上一对括号可以省略这一步骤。	

其中运算符的优先级为(阶乘不涉及两个数可在数字处直接处理)：

符号	操作
(	0
)	1
+ -	2
* / %	3
^	4

为了优化常数，可以在中缀转后缀的同时维护数字栈计算后缀表达式。整个过程大致如下：



时间复杂度 O(ExpressionLenth)

```
int n = strlen(ch);
top1 = top2 = 0;
ch[n++] = ')';
stack1[top1++] = '('; //Add parentheses to the outer layer to fully evaluate the expression
for (int i = 0; i < n; i++) {
    if (ch[i] >= '0' && ch[i] <= '9') 数进数字栈，注意多位数处理;
    else {
        if (ch[i] == '!') {计算阶乘;continue;}
        if (ch[i] == '-' && ch[i - 1] != ')' && (i == 0 || ch[i - 1] < '0' || ch[i - 1] > '9'))
        {下一个数是负数;continue;}
        if (ch[i] == '(') {ch[i]进运算符栈;continue;}
        while (stack1[top1 - 1] != '(' && priority(stack1[top1 - 1]) >= priority(ch[i]))
            根据运算符栈顶计算数字栈头两个数并将结果放回数字栈;
        if (ch[i] == ')') top1--; else ch[i]进运算符栈;
    }
}
```

（2）比较运算符重载：设置一个 bigger(a, b)函数判断 a,b 大小，a>b 返回 1，a<b 返回 0，a==b 返回 2。利用这个函数可以重载>、<、>=、<=、==、!=的运算。bigger 中首先判断两数符号，符号相等则判断两数长度，长度相等则依次判断每一位的位数。

时间复杂度 O(N)

```
if (a.sg * b.sg < 0) return a.sg > b.sg;
else if (a.sg > 0)
    if (a.len != b.len) return (a.len > b.len);
    else 判断每一位;
else if (a.sg < 0)                    //Absolute value a < b if they are negative
    if (a.len != b.len) return (a.len < b.len);
    else 判断每一位;
```

（3）赋值运算符重载：对于 Bium 类赋值给 Bium 类，则直接把每个属性覆盖；对于 int 赋值给 Bium 类，则将 int 每一位拆分存入；对于 char*赋值给 Bium 类，则将 char*每一位 char 左平移 48 位 ASCALL 码存入。由于重载了赋值，所以在比较运算时可以直接将 Bium 类与 int 型或 char*型比较，因为 c++编译器底层在做比较运算的时候是先将两侧用赋值换作同类型再进行比较。

时间复杂度 O(N)

```
this->sg = s.sg, this->len = s.len;
for (int i = 0; i < len; i++) this->num[i] = s.num[i];
```

```
符号处理; this->len = 0;
if (a != 0) while (a) this->num[this->len] = a % 10, a /= 10, this->len++;
else this->num[0] = 0, this->len = 1;
```

```
符号处理; this->len = strlen(s);
for (int i = 0; i < this->len; i++) this->num[i] = s[this->len - i - 1] - '0';
```

（4）加减运算：从 0 开始将每一位相加减，并将每一位产生的进位或借位传递给下一位。注意结果的符号以及减法只能绝对值大的减绝对值小的。

时间复杂度 O(N)

```
c.sg = a.sg, c.len = max(a.len, b.len), int up = 0;
for (int i = 0; i < c.len; i++)
    c.num[i] = (i >= a.len? 0: a.num[i]) + (i >= b.len? 0: b.num[i])+ up,
    up = c.num[i] / 10, c.num[i] %= 10;
继续进位;
```

```
c.len = a.len, up = 0;
for (int i = 0; i < c.len; i++) {
    c.num[i] = a.num[i] - ((i >= b.len ? 0: b.num[i]) + up);
    up = 0; while (c.num[i] < 0) c.num[i] += 10, up++;
}
```

（4）乘法运算：朴素的乘法运算是采用公式 $C_{i+j}=\sum(A_i\times B_j)$ 枚举两个数的每一位暴力计算，这样的时间复杂度是 O(N^2)的。对于题目要求而言是够用了。但是由于不满足这种简单实现，想进一步了解高精度乘法的快速傅里叶变换 FFT 解法。我们先来了解一些概念，理解为什么可以这么做，再来介绍方法。

我们在概率论的课堂上学到了连续型卷积的定义：

$$h(n)=\int f(m)g(n-m)dm$$

与此类似可以定义**离散型卷积**：

$$h[n]=\sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

我们可以看到，乘法运算 $C_{i+j}=\sum (A_i \times B_j)$ 和上面的离散型卷积的定义是一致的，也就是说乘法运算就是一种离散型的卷积。

下面将引入时域和频域。

一个数如 132 的幂级数可以表示成多项式 $f(x)=2x^0+3x^1+1x^2$，x=10 的时候。我们将 x 分别取 0,1,2 可以得到三个点(0，2)，(1，6)，(2，12)。x 取三个不同值 x_1, x_2, x_3 的时候，已知 f(x)系数通过函数的带入可以唯一确定三个点；于此同时，已知三个点通过高斯消元可以解得函数 f(x)的系数。也就是说，f(x)和三个点坐标都可以唯一确定一个多项式。

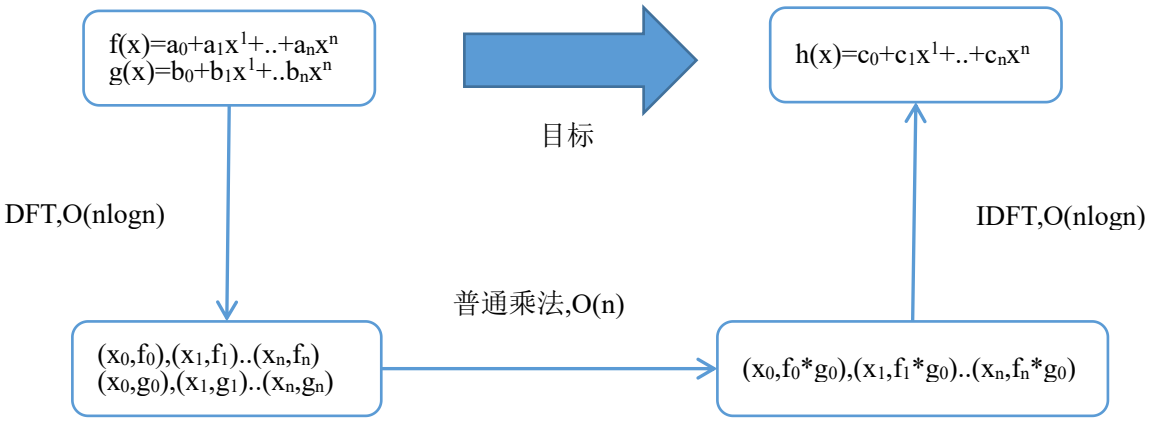
假设多项式最高次为 n，我们把 n+1 个系数如 2，3，1 表示多项式称为多项式的**系数表示法**；把 n+1 个横坐标不同的点如(0，2)，(1，6)，(2，12)表示多项式成为**点值表示法**。在信号领域，系数表示法相当于**频域**表示，点值表示法相当于**时域**表示。

系数表示法：$f(x)=a_0+a_1x^1+..+a_nx^n$
点值表示法：$(x_0,f_0),(x_1,f_1)..(x_n,f_n)$

在乘法运算中，如果已经已经知道了两个数多项式 f,g 的关于 $x_0, x_1.....x_n$ 点值表示法，那么相乘只需要把 $f(x_0), f(x_1).....f(x_n)$ 与 $g(x_0), g(x_1).....g(x_n)$ 对应相乘为 $f(x_0)\times g(x_0), f(x_1)\times g(x_1).....f(x_n)\times g(x_n)$ 即可得到积的点值表示法。这里的时间复杂度是 O(n)。

但是我们已知的是系数表示法，暴力从系数表示法转换为点值表示法时间复杂度是 $O(n^2)$ 并没有什么改变。不用担心，**离散傅里叶变换 DFT**(Discrete Fourier Transform)可以做到 O(nlogn)从频域转换为时域，而**离散傅里叶逆变换 IDFT**(inverse DFT)可以做到 O(nlogn)从时域转换为频域，只需要 $x_0, x_1.....x_n$ 取特定的值。计

算 DFT 和 IDFT 的算法称为**快速傅里叶变换算法 FFT**（Fast Fourier Transformation）。这样整个算法的思路就很清楚了：



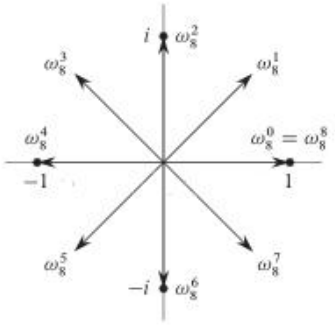
事实上，所有的离散型卷积都可以这么做，这就是**卷积定理**，即函数卷积的傅立叶变换是函数傅立叶变换的乘积。

在介绍 FFT 算法之前，先了解一下复数知识。

在复平面上任何一个复数 z 都能表示成为一个向量，即：$z=r(\cos\theta+i\sin\theta)$ 其中 r 是 z 的模长，θ 是向量与 x 轴的夹角，称之为幅角。根据欧拉公式 $e^{i\theta}=\cos\theta+i\sin\theta$ 则有 $z=r\times e^{i\theta}$，可以推出 $(\cos\theta+i\sin\theta)^{\alpha}=(\cos\alpha\theta+i\sin\alpha\theta)$ 这是棣莫弗公式。

在系数表达法到点值表达法的转换过程中，n 个不同 x 的取值是任意的。但是 DFT 的在频域到时域的时候用的是 x 是 $x^n=1$ 的所有复数解，这些解是 $e^{2k\pi i/n},k=0,1,...,n-1$，我们定义 $\omega_n^k=e^{2k\pi i/n}$。即 DFT 中带入的不同的 x 的值分别为 $\omega_n^0,\omega_n^1,\omega_n^2,...,\omega_n^{n-1}$，他们是 1 的**单位根**，其中 ω_n^1 为**本原根**。这些解有特殊的性质。

根据欧拉公式，$\omega_n=\cos\frac{2\pi}{n}+i\sin\frac{2\pi}{n}$，再根据棣莫弗公式我们会发现单位根平分复平面。



由棣莫弗公式，我们很容易证得一下公式：

$$公式1:(\omega_{2n}^{k})^{2}=\omega_{n}^{k}$$

$$公式2:\omega_{\alpha n}^{\alpha k}=\omega_{n}^{k}$$

$$公式3:\omega_{n}^{j+k}=\omega_{n}^{(j+k)\pm n}$$

$$公式4:\omega_{n}^{-k}=\omega_{n}^{-k}=1/\omega_{n}^{k}$$

我们终于可以介绍 FFT 算法了！

既然 DFT 要选取单位根作为 x 的值，需要得到所有 $y_k=f(\omega_n^k)=\sum_{j=0}^{n-1}a_j\omega_n^{kj}$ 。怎么快速求得呢？对于一个多项式 $f(x)$，我们根据项的指数分成两个部分：

$$a_0+a_2x^2+a_4x^4+\cdots+a_{n-2}x^{n-2}$$

$$a_1x+a_3x^3+a_5x^5+\cdots+a_{n-1}x^{n-1}$$

令：

$$f^{[0]}(x)=a_0+a_2x+a_4x^2+\cdots+a_{n-2}x^{n/2-1}$$

$$f^{[1]}(x)=a_1+a_3x+a_5x^2+\cdots+a_{n-1}x^{n/2-1}$$

可得，

$$f(x)=f^{[0]}(x^2)+xf^{[1]}(x^2)$$

而根据公式 1，对于求 $f(\omega_n^k)$ 的值就转化为了求 $f^{[0]}(\omega_{n/2}^k)$ 和 $f^{[1]}(\omega_{n/2}^k)$

这样，我们把一个次数界为 n 的多项式求值问题转换成了两个个次数界为 n/2 的多项式求值问题。按照一下公式递归就可以得到解：

$$f(\omega_n^k)=f^{[0]}(\omega_{n/2}^k)+\omega_n^k f^{[1]}(\omega_{n/2}^k),n>1$$

$$f(\omega_n^k)=a_0,n=1$$

由于单位根的对称性，由公式 3 和 4 $\omega_n^{n-k}=\omega_n^{-k}=-\omega_n^k$ 可以这样计算点值减少时间常数：

$$f(\omega_n^k)=f^{[0]}(\omega_{n/2}^k)+\omega_n^k f^{[1]}(\omega_{n/2}^k)$$

$$f(\omega_n^{k+(n/2)})=f^{[0]}(\omega_{n/2}^k)-\omega_n^k f^{[1]}(\omega_{n/2}^k)$$

递归的层数为 logN 层，而每层都要计算对应的 N 个值，所以时间复杂度为 O(NlogN)即可实现：

```
complex<double>* fft(int* f) {
    n = strlen(f);
    if (n == 1) return f[0];
    complex<double>* y[n];
    int* f1[n/2],f2[n/2];
    for (int i = 0; i < n; i+=2)                    //调整递归系数
        f1.append(f[i]),f2.append(f[i+1]);
    y1 = fft(fx1), y2 = fft(fx2);                   //递归
    w = 1;
    wn = complex(cos(2*PI/n),sin(2*PI/n));          //本原根
    for (int i = 0; i < n/2; i++) {
        y[i] = (y1[i%(n/2)] + w * y2[i%(n/2)]);     //递归的还原
        y[i + (n/2)] = (y1[i%(n/2)] - w * y2[i%(n/2)]);
        w = w*wn;                                   //构造复数的单位根
```

```
    }
    return y;
}
```

我们完成了 O(NlogN)的频域到时域的变换，也可以 O(N)相乘算出积的时域，那么怎么 O(NlogN)从时域到频域变换呢？我们观察一下刚刚从系数表示到点值表示的变换，发现它本质上是一个矩阵相乘 $y = M \bullet a$：

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

要使点值表示变为系数表示，只需使点值乘以 M 的逆矩阵，即 $a = M^{-1} \bullet y$。而我们会发现其中的 M 矩阵就是一个**范德蒙矩阵**，范德蒙矩阵一定有逆，因此我们可以得到：

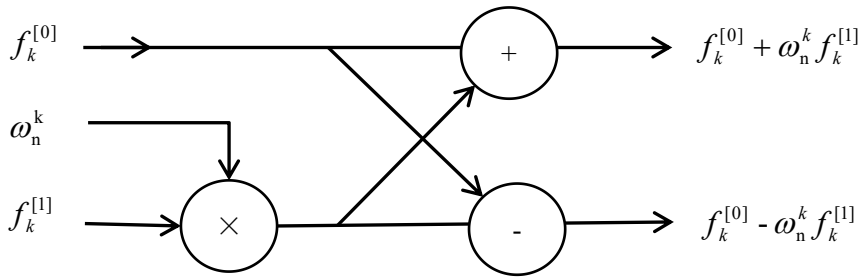
$$M^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \cdots & \omega_n^{-2(n-1)} \\ 1 & \omega_n^{-3} & \omega_n^{-6} & \omega_n^{-9} & \cdots & \omega_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \omega_n^{-3(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{pmatrix} \times \frac{1}{n}$$

所以只需要把单位根用公式 4 换成 $\omega_n^{-0}, \omega_n^{-1}, \omega_n^{-2}, \ldots, \omega_n^{-(n-1)}$，并把最终结果都除以 n，IDFT 还可以用刚刚 FFT 的算法求解。

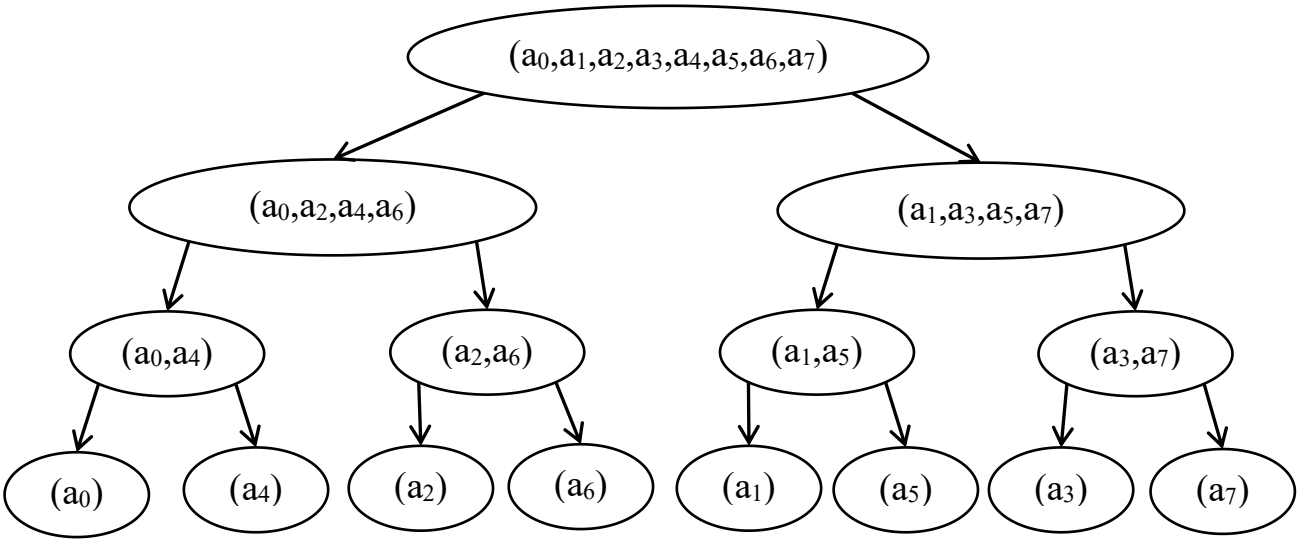
```
complex<double>* fft(int* f, bool inverse) {
    ...
    y1 = fft(fx1, inverse), y2 = fft(fx2, inverse);      //递归
    w = 1;
    wn = 1 / complex(cos(2*PI/n),sin(2*PI/n));           //本原根换成1/wn
    ...
}

y=fft(f,1);
for (int i = 0; i < n; i++) y = y / n;
```

这样我们已经基本完成了多项式乘法求解的算法，为了优化时间复杂度和空间复杂度，我们将刚刚递归的 FFT 改为迭代的 FFT。

观察递归的公式，发现它进行的是以下这样一个一个**蝴蝶操作**：



我们按照一定次序进行蝴蝶操作就可以迭代求解，什么顺序呢？我们之前按照指数的奇偶分别进行了递归，用 n=8 模拟一下顺序：



只需从叶子节点不断向上使用蝴蝶操作，最终得到根节点的值，就完成了 DFT 的迭代。通过观察，这个叶子节点顺序是一个**位逆序置换**，即是说上面的秩序 0, 4, 2, 6, 1, 5, 3, 7，其二进制为 000, 100, 010, 110, 001, 101, 011, 111，把二进制各位逆序后，得到顺序的序列 000, 001, 010, 011, 100, 101, 110, 111。为了得到一般情况下的位逆序置换，注意到最低为 0 的数就是去掉最低位反转后最高位置 0，最低为 1 的数就是去掉最低位反转后最高位置 1. 即设 k 位逆序置换后的位置为 rev(k)：

末位为 0： $\mathrm{rev}(k) = (\mathrm{rev}[k >> 1] >> 1)$

末位为 1： $\mathrm{rev}(k) = (\mathrm{rev}[k >> 1] >> 1) \mid (n >> 1)$

如此便可以循环递推求出位逆序置换的位置，在 FFT 开始前将 $a_{0..n-1}$ 中的数置换即可迭代实现。因为做了 DFT 和 IDFT，所以置换两次又会回到原序列。

FFT 迭代实现如下：

```
void fft(complex<double>* f, int* pos, int len, int on) {
//f is a sequence of point values, pos is the position of the bit reverse order, len is the length
of the sequence, and on is judged as DFT or IDFT
    complex<double> temp;
```

```
    for(int i = 0; i < len; i++)              //Bit reverse order replacement
        if(i < pos[i]) 交换 f[i]和 f[pos[i]]
    for(int i = 1; i < len; i <<= 1) {        //Enumerate each layer for butterfly operations
      complex<double> wn(cos(on * PI / i), sin(on * PI / i));
      for(int j = 0; j < len; j += (i << 1)) {
          complex<double> wi(1, 0);
          for(int k = j; k < j + i; k++) {
                complex<double> u = f[k], v = f[k + i] * wi;
                f[k] = u + v;
                f[k + i] = u - v;
                wi *= wn;
          }
      }
    }
    if(on == -1)                              //Remember to divide the IDFT result by n
      for(int i = 0; i < len; i++) f[i] /= len;
}
```

从而整体的乘法实现如下，注意多项式的长度必须是 2 的正整数次幂，不然 FFT 没法分治，所以我们扩展多项式的项至 2^{len}，多余的项系数为 0：

```
void Bium::multiply(Bium& c, Bium& a, Bium& b) {
    int len = 1;
    int pos[MAXLEN] = {0};
    while (len < a.len + b.len) len <<= 1;
    complex<double> f1[len], f2[len];
    for(int i = 0; i < len; i++)     //Reverse order replacement
       pos[i] = (i & 1) ? (pos[i >> 1] >> 1) | (len >> 1) : (pos[i >> 1] >> 1);
    for(int i = 0; i < len; i++) {   //Change to plural form
       f1[i].real(i < a.len? a.num[i] : 0); f1[i].imag(0);
       f2[i].real(i < b.len? b.num[i] : 0); f2[i].imag(0);
    }
    fft(f1, pos, len, 1);            //Convert to point value representation
    fft(f2, pos, len, 1);
    for(int i = 0; i < len; i++) f1[i] *= f2[i];
    fft(f1, pos, len, -1);
    c.len = len, c.sg = a.sg * b.sg;
    int up = 0;
    for(int i = 0; i < c.len; i++) {
       c.num[i] = int(f1[i].real() + 0.5 + up) % 10;
       up = int(f1[i].real() + 0.5 + up) / 10;          //Carry for the number
    }
   while (up) c.num[c.len++] = up % 10, up /= 10;
}
```

时间复杂度 O(NlogN)

（5）除法运算：将除数 b 的位置与被除数 a 对齐，记为 b^1，若被除数减去 b^1 大于等于 0，则相减并将对应位置上的商加 1。如此往复，直到不能减为止，b^1 降一位为 b^2，按上面的方法算下一位商，直至 $b^n=0$。余数就是被减数在不断被减后剩下的值。

时间复杂度 O(N^2)

```
for (int i = a.len - 1; i >= 0; i--) //Align the divisor with the dividend
    if (a.len - i <= b.len) b.num[i] = b.num[i - (a.len - b.len)];
    else b.num[i] = 0;
b.len = a.len; c.len = 0;
for (int i = a.len - 1; i >= t.len - 1; i--) {
    while (a >= b) {
        a = a - b, c.num[b.len - t.len] += 1;
        if (c.len == 0) c.len = b.len - t.len + 1;
    }
    for (int j = 0; j < i; j++) b.num[j] = b.num[j + 1];
    b.num[i] = 0;
    b.len--;
}
```

（6）阶乘运算：对于 n!，暴力从 1 乘到 n。对于位数为 N 的数，每次乘法要 O(NlogN)的时间复杂度，此运算耗时很长。

时间复杂度 $O(10^N \times NlogN)$

（7）幂运算：使用**快速幂**算法计算。算法思想是把指数换成二进制，假设要算 a^b，例如 6^5，我们把 b 转换成二进制 101 即 $6^{101_2} = 6^{1\times2^0+0\times2^1+1\times2^2} = 6^{2^0} \times 6^{2^2}$，由此计算每次 b>>1 遍历 b 的二进制每一位，a 迭代平方，若 b 当前的末尾位数为 1 则乘上 a 的贡献。一般来说快速幂的时间复杂度是 O(logN)的，但是大数运算中乘法需要 O(NlogN)，b 除以 2 需要 O(N)，所以整体时间复杂度为 O((NlogN+N)logN)约为 $O(Nlog^2N)$。

时间复杂度 $O(Nlog^2N)$

```
c = 1;
while (b > 0) {
    if (b % two == 1) c *= a;
    a *= a;
    b /= 2;
}
```

六、实验数据和实验结果分析

运行结果良好。

```
(35+(-69))*(98-7)%10!/2^5
= 96
请按任意键继续. . .

12345678987654321*98765432123456789*586^89
= 26853925150742554292724195550681825073823935702947683424950047017895563034097409350
22824773416886304372179948632694066472470939339752308330270015902787371928439784200 4
25939386099249965724447842549141573916932783625797713146615491147254036241289096691 5
16294087767353529959197337190 4
请按任意键继续. . .
```

七、实验体会

这次大整数计算器花了我挺长时间，我尝试用类来构造大整数并重载了运算符，同时运用了比普通算法更加精妙的算法，使效率得到了较大提升，只是整个计算器虽然用到了字符串，但是把字符 ASCALL 码向左平移了 48 位当作整型用了，有一些不符合题目要求。然而这样更方便处理，希望谅解。报告也让我花了很长时间来阐述快速傅里叶变换算法，希望我讲清楚了。我觉得复变分析在计算机领域的应用也挺广的嘛，最近感觉我们院的数学教得还是浅了一些。然而学海无涯，只有提高自己得自学能力才可以再众多知识中灵活应对。

八、参考文献

https://blog.csdn.net/dream_1996/article/details/78126839
https://blog.csdn.net/ice__snow/article/details/52733968
https://blog.csdn.net/ljhandlwt/article/details/51999762
https://blog.csdn.net/ripped/article/details/70241716
https://baike.baidu.com/item/%E5%8D%B7%E7%A7%AF%E5%AE%9A%E7%90%86/10440902?fr=aladdin
算法导论（原书第 3 版）Thomas H.Cormen/Charles E.Leiserson/Ronald L.Rivest/Clifford Stein