装

订

线

大数运算（基础版）
实现报告

班级：计算机一班
姓名：王哲源
学号：1652228

完成日期：2017.6.6

装

订

线

1.题目及其描述

题目主要目的在于通过类的实现以及运算符重载，重新定义一种 bigint 类，以实现当运算数超过 int 甚至 long long 范围时仍能进行各类运算并得出正确结果。同时 bigint 类也支持直接的读入与输出。

其应支持以下几种操作：

1.1 bigint 类的初始化

当定义 bigint 类时，自动对类进行初始化

1.2 bigint 类的构造

使类能够支持在定义时从已有数据进行直接构造

1.3 bigint 类的流读入与流输出

使 bigint 类可以通过输入流\输出流\文件流的方式读入\输出数据

1.4 赋值运算

使 bigint 类可以通过 = 实现直接的赋值，且能实现连等

1.5 双目四则运算

使 bigint 类可以支持 + - * \ % 运算，并同时支持复杂的四则混合运算

注：同时支持 += -= *= /= %= 的运算

1.6 单目自增\自减运算

使 bigint 类可以支持 ++ -- 的自增\减运算（包括前置\后置两种方式），同时得到的正确结果可以进行下一步的运算

1.7 比较类双目运算

支持两 bigint 类间 > < >= <= != == 的比较运算

2.整体设计思路

以下对于程序的一些设计思路进行概述

2.1 bigint 类的存储

由于 bigint 类超出了 int 的存储范围，因此这里可以使用数组进行存储。而鉴于其位数的不确定，因此采用动态分配空间的方式来定义数组会更好

2.2 bigint 类的读入与输出

由于没有类型能直接存储下 bigint 类所表示的数，因此读入方式采用字符串方式读入后按位填入数组中，这样输出时只需要按位输出数字即可

2.3 bigint 类的四则运算

可以想到的最简便的方法就是按四则运算时的列竖式进行计算。这样操作起来简便，但缺点也很明显，就是当进行乘除运算时的效率会也会非常低下。

至于自加\减\乘\除的运算，只需要直接对 this 指针所指的类进行修改并返回即可

而自增\减则可以用+1 来替代

2.4 bigint 类的大小比较

类似实际比较两数大小,bigint 类的大小比较只需要先符号,再位数,最后逐位的方式即可。判断相等时也是采用同样的方法

3.主要功能实现

以下对整个程序的具体实现进行较为详细的介绍。由于进阶版在乘除时要采用快速傅里叶变换等高级算法，普通算法的时间过于低下，这里就以普通版的思路进行表述

3.1 bigint 类存储

如 2.1 所述，同时为防止空间的过度浪费，bigint 类下这边包含三个成员，分别是指向存储数组的 num 指针，符号标志 f，存储位数的 len，以及存储块数的 block。存储块以 1024 为一个单位，类型为 long long 类型，每次按恰好大于需要存储的位数进行申请，在不足时重新申请，在过剩时自行缩小大小，以达到节省空间的目的。

同时，为了最大利用储存数组，这里采用压位 10^9 的方式，将 10^9 位存储于一个数组下标中，在保证加法不超过 long long 的存储范围同时最大化利用数组。为了方便运算，存储采用由低位向高位存储

3.1.1 Realloc 函数

该函数用于加法（乘法）进位时对空间不足的数组进行扩容。采用的方法是开一个 oldbuffer 的 bigint 类通过构造函数对原有数据进行拷贝，然后将原空间释放后对 block+1 进行空间的重新申请。再将 oldbuffer 中的数据拷贝回来

3.1.2 Freealloc 函数

该函数用于减法（除法）执行后对多余的空间数组进行释放。采用的方法是以一个 block 为单位从数组头向后遍历，直至遇到非 0 时，将多余的 block 释放，具体操作方式同 3.1.1

3.2 数据的读入与输出

数据输出较为简单,根据 3.1 的存储特性只需要从高位向低位逐位输出。这里应当特别注意，当一个数组下标内不满 9 位时，应当用 0 补全前位！

而数据的读入由于位数无法确定，因此借用 string 类来暂存数据，在确定位数后申请所需空间，再每隔 9 位对数据进行存入

3.3 bigint 类的构造与赋值

bigint 类的构造与赋值分为两种，一种是 bigint 类间的，这一种较为简单，直接对 num 类下所存数据进行 memcpy，同时对其他成员直接赋值即可。

还有一种是 long long 类赋给 bigint 类。由于 long long 类所能表达的数据远小于一个 block 所能存储的数据，因此这里只需要申请一个 block，之后按位填入。

同时，为了归一化类型，当 long long 为 0 时，也将其当作一位数进行处理，便于后续的运算

3.3 bigint 类的加减法

对于 bigint 类的加减法，首先可以根据符号的情况进行分类：当符号相同时，进行的是正常的加减法，而当符号不同时，对于加法可以直接转化为减法，而对于减法也可以直接转化为加法。

3.3.1 加法的具体实现

bigint 类的加法采用的是列竖式的思路，符号可以在开始时便确定。而当进行加法时，先选出两者中长度更长的一者构造返回类，之后对对长度短的一者进行逐位相加。当以上步骤执行完毕后，进行进位操作，一直进位到首位且首位无法继续进位为止。特别注意，进位时可能会超出当前 bigint 类所能存储的最大位数，因此需要上述 Realloc 函数对数组进行扩容

3.3.2 减法的具体实现

减法同样是采用列竖式的思路，但具体步骤与加法不太相同。符号首先通过绝对值最大的一方确定，并以此为基准构造返回类，之后由低位向高位进行减法，不足时借位同列竖式一样。由于已经保证是大数减小数，因此不用担心由于借位造成的越界问题。在减法执行完毕之后，为了防止出现过多的前置 0 占用空间，因此需要执行上述 Freealloc 函数对数组进行适当的缩减

3.4 乘法的具体实现

乘法的符号判断较为简单，同号为正异号为负，而计算同样采用列竖式的思路，但动态空间只需要先以固定一方为准即可。这得益于乘法的列竖式计算机制，每一次运算实际上为一次错位加法，最少只会计算一者（这里以 a*b 中的 b 为例）的定长，因此每次只需要以一个指针标记当前填到哪一位即可，不足时再扩大即可，只需要注意该指针在进行下一位计算时若未发生进位需要同时+1 即可。

3.5 除法的具体实现

除法采用的也是列竖式的思路，由于除法不存在交换律，因此一开始先判断是否被除数的绝对值小于除数，若是则直接返回 0 即可。

接下来先将对除数的低位填 0 的方式将两数通过补位的方法进行对其，并预先存储好答案的起始填入为止。之后每次判断被除数是否大于等于除数，若是则对答案当前位+1，并使被除数减去除数，反复至被除数小于除数时，将除数整体右移一位，重复以上操作直至除数无法继续右移（即比原数长度更短）为止，即可获得商，同时余数也存储于被除数中。

同样，为了节省空间，这里需要对答案进行 Freealloc 再返回

3.6 比较运算的实现

比较运算相对四则运算更为简单，只需要遵从符号，长度，逐位的顺序进行对比即可。这里

就不进行赘述了

4. 调试过程中遇到的问题

4.1 正负零的情况

在加减运算的检验中就发现了有正负零的情况出现，通过逐条功能发现是当通过符号判断为负后，运算时若答案为 0，负号符仍存在，这会对输出时的判断造成干扰。因此这里将转换构造函数中加入了一条判断，即若 len 为 1 且 num[0]为 0，则强制 f 为 1

4.2 乘法出现越界

在测试乘法可行性时发现，若两数较大时，答案中会出现多个负号，因此怀疑为乘法运算过程中出现了数值超范围的情况导致越界。

经过分析发现，由于乘法的累加可能存在若干项，若将累加进位留至最后进行，则有极大可能出现数值超出范围的情况。因此最后改为每进行完一层乘法，立即执行一次进位，以防数值过大导致的答案出错

5. 总结部分

原认为高精度的四则运算是十分简单的程序，但实际编写时发现细节极多，且对于动态内存分配的高精度运算也是第一次涉及，因此编写时低级错误也十分多。同时碍于列竖式的效率问题，最终只能编写基础版（FFT 没实现过动态内存的版本，同时除法的方法也没学习过），因此自己的知识和编写能力都有待进一步提高。

6. 附件：程序（仅为包含函数实现部分）

```
bigint::bigint()
{
    num = NULL;
    len = block = f = 0;
}
bigint::~bigint()
{
    delete[] num;
    num = NULL;
    len = block = f = 0;
}
bigint::bigint(const bigint &rhs)
{
    len = rhs.len;
    block = rhs.block;
    f = rhs.f;
    int size = block*Each_Block;
    if (len == 1 && rhs.num[0] == 0)
        f = 0;
    num = new(nothrow)LL[size];
    if (num == NULL)
    {
        puts("No Free Memory");
        exit(-1);
    }
    memcpy(num, rhs.num, size);
} bigint::bigint(LL n)
{
    block = 1;
    f = n >= 0 ? 0 : 1;
    n = abs(n);
    num = new(nothrow)LL[Each_Block];
    if (num == NULL)
    {
        puts("No Free Memory");
        exit(-1);
    }
    if (!n)
        len = 1, num[0] = 0;
    else
```

```
        {
            while (n)
                num[len++] = n%Max, n /= Max;
        }
        sFor(i, len, Each_Block)
            num[i] = 0;
    }

    void bigint::reset()
    {
        delete[] num;
        num = NULL;
        len = block = 0;
    }
    bigint& bigint::operator =(const bigint &rhs)
    {
        (*this).reset();
        len = rhs.len;
        block = rhs.block;
        f = rhs.f;
        int size = block*Each_Block;
        num = new(nothrow)LL[size];
        if (num == NULL)
        {
            puts("No Free Memory");
            exit(-1);
        }
        memcpy(num, rhs.num, size);
        return *this;
    }

    istream& operator >>(istream &it, bigint &rhs)
    {
        rhs.reset();
        char ch;
        string n = "";
        while (!isdigit(ch = it.get()))
            if (ch == '-')
                break;
        if (ch == '-')
            rhs.f = 1;
        else
            n += ch;
        while (isdigit(ch = it.get()))
            n += ch;
        rhs.len = n.length() / Maxn + !!(n.length() % Maxn);
        int size;
        while ((size = (rhs.block*Each_Block)) < rhs.len) //////////括号啊大哥。。
            ++rhs.block;
        rhs.num = new(nothrow) LL[size];
        if (rhs.num == NULL)
        {
            puts("No Free Memory");
            exit(Error);
        }
        int p = 0;
        for (int i = n.length() - 1; i >= 0;)
        {
            LL v = 0, m = 1;
            int cnt = 0;
            for (; i >= 0 && cnt < Maxn; --i, ++cnt)
                v += (n[i] - 48)*m, m *= 10;
            rhs.num[p++] = v;
        }
        sFor(i, rhs.len, size)
            rhs.num[i] = 0;
        return it;
    }
    ostream& operator <<(ostream &it, bigint &rhs)
    {
        if (rhs.f && !(rhs.len == 1 && rhs.num[0] == 0))
            it.put('-');
        it << rhs.num[rhs.len - 1];
        opFor(i, rhs.len - 2, 0)
            it << setw(Maxn) << setiosflags(ios::right) << setfill('0') << rhs.num[i];
        return it;
    }

    bigint bigint::operator -()const
    {
        bigint c(*this);
        c.f ^= 1;
        return c;
    }

    void Upper(bigint &rhs)
    {
        int p = 0;
        while (p<rhs.len)
        {
            if (rhs.num[p] >= Max)
            {
                if (p + 1 == rhs.len)
                {
                    if (rhs.len == rhs.block*Each_Block)
                        Realloc(rhs);
                    ++rhs.len;
                }
                LL tmp = rhs.num[p];
                rhs.num[p] = tmp%Max;
```

```
                rhs.num[p + 1] += tmp / Max;
            }
            ++p;
        }
    }
    void Realloc(bigint &newbuffer)
    {
        bigint oldbuffer(newbuffer);
        delete[] newbuffer.num;
        int              size              =
(++newbuffer.block)*Each_Block;
        newbuffer.num     =     new(nothrow)
LL[size];
       memcpy(newbuffer.num, oldbuffer.num,
size - Each_Block);
        sFor(i, size - Each_Block, size)
            newbuffer.num[i] = 0;
    }
    void Freealloc(bigint &newbuffer)
    {
        int  res_block  =  0,  res_len  =
newbuffer.len;
        opFor(i, newbuffer.block - 1, 0)
        {
            int limit = i*Each_Block;
            for  (;  res_len  >=  limit;
--res_len)
                if (newbuffer.num[i])
                    break;
            if (res_len != limit - 1)
                res_block = i + 1;
        }

        if (!res_block)
            res_block = 1, newbuffer.len = 1;
        else
            newbuffer.len = res_len;
        if (res_block == newbuffer.block)
            return;
        bigint oldbuffer(newbuffer);
        delete[] newbuffer.num;
        int size = res_block*Each_Block;
        newbuffer.num     =     new(nothrow)
LL[size];
       memcpy(newbuffer.num, oldbuffer.num,
size);
    }

    bigint operator +(const bigint &a, const
bigint &b)
    {
        bigint c;
        if (a.f^b.f)
        {
            c = a.f ? b - (-a) : a - (-b);
            return c;
        }
        int len = Max(a.len, b.len);

        if (a.len > b.len)
        {
            c = a;
            opFor(i, b.len - 1, 0)
                c.num[i] += b.num[i];
        }
        else
        {
            c = b;
            opFor(i, a.len - 1, 0)
                c.num[i] += a.num[i];
        }
        Upper(c);
        return c;
    }
    bigint operator -(const bigint &a, const
bigint &b)
    {
        bigint c;
        if (a.f^b.f)
        {
            c = a.f ? -((-a) + b) : a + (-b);
            return c;
        }
        if ((!a.f && a > b) || (a.f && -a < -b))
        {
            c = a;
            c.f = 0;
            sFor(i, 0, b.len)
            {
                c.num[i] -= b.num[i];
                if (c.num[i] < 0)
                    c.num[i]    +=    Max,
--c.num[i + 1];
            }
        }
        else
        {
            c = b;
            c.f = 1;
            sFor(i, 0, a.len)
            {
                c.num[i] -= a.num[i];
                if (c.num[i] < 0)
                    c.num[i]    +=    Max,
--c.num[i + 1];
            }
        }
        Freealloc(c);
        return c;
    }
    bigint& bigint::operator +=(const bigint
&rhs)
```

```
    {
        return (*this = (*this) + rhs);
    }
    bigint& bigint::operator -=(const bigint
&rhs)
    {
        return (*this = (*this) - rhs);
    }
    bigint bigint::operator ++(int)
    {
        bigint old(*this);
        if (f)
        {
            if (len == 1 && num[0] == 1)
                f ^= 1, num[0] = 0;
            else
            {
                --num[0];
                int p = 0;
                while (num[p] < 0)
                    num[p]    +=    Max,
--num[++p];
                Freealloc(*this);
            }
        }
        else
        {
            ++num[0];
            if (num[0] >= Max)
                Upper(*this);
        }
        return old;
    }
    bigint bigint::operator --(int)
    {
        bigint old(*this);
        if (!f)
        {
            if (len == 1 && num[0] == 0)
                f ^= 1, num[0] = 1;
            else
            {
                --num[0];
                int p = 0;
                while (num[p] < 0)
                    num[p]    +=    Max,
--num[++p];
                Freealloc(*this);
            }
        }
        else
        {
            ++num[0];
            if (num[0] >= Max)
                Upper(*this);
        }
        return old;
    }
    bigint& bigint::operator ++()
    {
        if (f)
        {
            if (len == 1 && num[0] == 1)
                f ^= 1, num[0] = 0;
            else
            {
                --num[0];
                int p = 0;
                while (num[p] < 0)
                    num[p]    +=    Max,
--num[++p];
                Freealloc(*this);
            }
        }
        else
        {
            ++num[0];
            if (num[0] >= Max)
                Upper(*this);
        }
        return *this;
    }
    bigint& bigint::operator --()
    {
        if (!f)
        {
            if (len == 1 && num[0] == 0)
                f ^= 1, num[0] = 1;
            else
            {
                --num[0];
                int p = 0;
                while (num[p] < 0)
                    num[p]    +=    Max,
--num[++p];
                Freealloc(*this);
            }
        }
        else
        {
            ++num[0];
            if (num[0] >= Max)
                Upper(*this);
        }
        return *this;
    }

    bigint operator *(const bigint &a, const
bigint &b)
    {
        bigint c;
        c.f = a.f^b.f;
```

```
        int limit = Each_Block*b.block;
        c.block = b.block;
        c.num = new(nothrow) LL[limit];
        if (c.num == NULL)
        {
            puts("No Free Memory");
            exit(Error);
        }
        sFor(i, 0, limit)
            c.num[i] = 0;
        sFor(i, 0, a.len)
        {
            sFor(j, 0, b.len)
            {
                if (i + j >= limit)
                    Realloc(c), limit +=
Each_Block;
                c.num[i + j] += a.num[i] *
b.num[j];
            }
            if (c.len < b.len + i)
                c.len = b.len + i;
            Upper(c);
        }
        return c;
    }
    bigint operator /(const bigint &a, const
bigint &b)
    {
        if (b.len == 1 && b.num[0] == 0)
        {
            puts("The divisor cannot be 0");
            exit(Error);
        }
        if ((a.f && b.f && a > b) || (!a.f
&& !b.f && a < b) || (a.f && !b.f && -a < b) ||
(!a.f && b.f && a < b))
        {
            bigint c(0);
            return c;
        }
        bigint ta(a), tb(a), c(a);
        c.f = a.f^b.f;
        ta.f = tb.f = 0;
        opFor(i, a.len - 1, 0)
            c.num[i] = 0;
        int p = a.len - 1;
        sFor(i, 0, b.len)
            tb.num[p--] = b.num[b.len - i -
1];
        int base = p;
        opFor(i, p, 0)
            tb.num[i] = 0;
        opFor(i, a.len - b.len, 0)
        {
            while (ta >= tb)
                ++c.num[i], ta -= tb;
            if (base < 0)
                break;
            --tb.len;
            sFor(j, base, tb.len)
                tb.num[j] = tb.num[j + 1];
            --base;
        }
        Freealloc(c);
        return c;
    }
    bigint operator %(const bigint &a, const
bigint &b)
    {
        if (b.len == 1 && b.num[0] == 0)
        {
            puts("The divisor cannot be 0");
            exit(Error);
        }
        if ((a.f && b.f && a > b) || (!a.f
&& !b.f && a < b) || (a.f && !b.f && -a < b) ||
(!a.f && b.f && a < b))
        {
            bigint c(a);
            return c;
        }
        bigint ta(a), tb(a), c(a);
        c.f = a.f^b.f;
        ta.f = tb.f = 0;
        opFor(i, a.len - 1, 0)
            c.num[i] = 0;
        int p = a.len - 1;
        sFor(i, 0, b.len)
            tb.num[p--] = b.num[b.len - i -
1];
        int base = p;
        opFor(i, p, 0)
            tb.num[i] = 0;
        opFor(i, a.len - b.len, 0)
        {
            while (ta >= tb)
                ++c.num[i], ta -= tb;
            if (base < 0)
                break;
            --tb.len;
            sFor(j, base, tb.len)
                tb.num[j] = tb.num[j + 1];
            --base;
        }
        Freealloc(ta);
        ta.f = a.f;
        return ta;
    }
    bigint& bigint::operator *=(const bigint
&rhs)
    {
```

```
        return (*this = *this*rhs);
    }
    bigint& bigint::operator /=(const bigint
&rhs)
    {
        return (*this = *this/rhs);
    }
    bigint& bigint::operator %=(const bigint
&rhs)
    {
        return (*this = *this%rhs);
    }

    bool operator >(const bigint &a, const
bigint &b)
    {
        if (a.f || b.f)
        {
            if (a.f && b.f)
                return -a < -b;
            else
                return !a.f;
        }

        if (a.len != b.len)
            return a.len > b.len;
        opFor(i, a.len - 1, 0)
            if (a.num[i] != b.num[i])
                return a.num[i] > b.num[i];
        return 0;
    }
    bool operator <(const bigint &a, const
bigint &b)
    {
        if (a.f || b.f)
        {
            if (a.f && b.f)
                return -a > -b;
            else
                return a.f;
        }

        if (a.len != b.len)
            return a.len < b.len;
        opFor(i, a.len - 1, 0)
            if (a.num[i] != b.num[i])
                return a.num[i] < b.num[i];
        return 0;
    }
    bool operator >=(const bigint &a, const
bigint &b)
    {
        if (a.f || b.f)
        {
            if (a.f && b.f)
                return -a <= -b;
            else
                return !a.f;
        }

        if (a.len != b.len)
            return a.len > b.len;
        opFor(i, a.len - 1, 0)
            if (a.num[i] != b.num[i])
                return a.num[i] > b.num[i];
        return 1;
    }
    bool operator <=(const bigint &a, const
bigint &b)
    {
        if (a.f || b.f)
        {
            if (a.f && b.f)
                return -a >= -b;
            else
                return a.f;
        }

        if (a.len != b.len)
            return a.len < b.len;
        opFor(i, a.len - 1, 0)
            if (a.num[i] != b.num[i])
                return a.num[i] < b.num[i];
        return 1;
    }
    bool operator ==(const bigint &a, const
bigint &b)
    {
        if (a.f^b.f || a.len != b.len)
            return 0;
        sFor(i, 0, a.len)
            if (a.num[i] != b.num[i])
                return 0;
        return 1;
    }
    bool operator !=(const bigint &a, const
bigint &b)
    {
        if (a.f^b.f || a.len != b.len)
            return 1;
        sFor(i, 0, a.len)
            if (a.num[i] != b.num[i])
                return 1;
        return 0;
    }
```