基础操作

我们设计了`base_data_t`类，用于最基础运算，它将支持固定长度的基本运算，分别是加减乘除取余。这些实现类似于硬件设计的加法器，乘法器等，无符号大整数将通过这些基础操作设计。设计这个类的一个好处是可以修改这个基础运算的实现，而不用修改大整数的代码。

我们设计了四种运算，分别是`add sub mul divmod`，他们的参数列表与返回类型也均相同

```
pair<bint_t,bint_t> add(const base_data_t n,const bint_t carry=0) const
```

该类方法将接受一个同类型参数，以及上一轮计算产生的进位或借位，运算过后将返回两个值，第一个返回值是运算结果，第二个是进位或借位。这种设计在后面的使用中可以发现相当得方便。

为了支持大整数的比较操作，该类也实现了比较操作。

无符号大整数类

即`ubgn`，该类主要的运算时加减乘除，以及比较等。

加减法

我们将`+=`设计为最基础的加法，将两个存储无符号数的数组进行相加即可。

`-=`设计为最基础的减法，需要注意，该类的减法必须使**左操作出大于等于右操作数**

乘法

`mul_base`作为最基础的乘法，用于实现大整数乘以一个基本单位的乘法，这个乘法操作将在原对象上执行

`mul_meta`基于`mul_base`实现普通的竖式乘法，显然，其时间复杂度为$O(mn)$，m为左操作数长度，n为右操作数长度

`mul_karatsuba`基于`mul_meta`实现了karatsuba算法，它的时间复杂度为$O(n^{\log_2 3})$，我们选取在数组长度大于等于512时进行分治运算，因为512在我们的几次试验中表现最好。由于karatsuba算法最好将两个操作数长度置为相同，因此，在操作数长度不同时，我们选择拆分长度长的操作数。

乘法运算将总产生新的对象，因此，`*=`与`*`在性能上没有区别

除法

`divmod_base`方法将在原对象上进行除法操作，它将除以一个基本单位的右操作数。该函数即实现了短除法，它主要用于将该大整数转换成十进制数。

由于普通除法实现难度过大，我们目前暂未实现。

有符号大整数类

基于无符号大整数，额外增加一个符号标志，用于实现有符号大整数。

由于实现简单，不做过多的介绍。其中，需要注意运算结果得到0时，符号需要修改为正数，否则将出现-0

除了基本的运算操作，我们增加了幂运算方法`pow`，它使用快速幂算法实现

另外我们也实现了大整数向基本整数类型的强制类型转换

因为c++11支持用户自定义字面值常量，我们设计了后缀为`bgn`的数将产生一个`bgn`对象

我们没有支持自增自减操作，也没有提供左移右移（设计为私有方法）运算接口，因为对于大整数这些操作并不必须。

程序正确性测试

事实上我们并不需要测试所有操作来验证正确性，因为`pow`运算会同时调用乘法加法，乘法又分为`karatsuba`算法与竖式乘法，而输出会调用除法，因此，只需测试$3^{1000}-2^{1000}$即可同时验证完加减乘除，将结果与python进行对比即可知道输出是否正确

C++

```
auto x=(3bgn).pow(1000)-(2bgn).pow(1000);
cout<<x<<endl;
```

Python

```
print(3**1000-2**1000)
```

我们将编译完成的C++程序输出与python进行对比

```
test.exe > c++result.txt
py test.py > pyresult.txt
diff c++result.txt pyresult.txt
```

另外需要测试的就是边界情况，测试符号是否正确等，比较简单。

性能测试

我们在测试发现，基本与python执行速度大致相当，在阶乘方面速度甚至比python还快，这是由于阶乘右操作数为基本类型，将会调用`mul_base`。

在超长整数乘法方面，略慢于python，毕竟python解释器经过十几年的发展，而我们只进行了3天的编码工作。

在转化成十进制字符串方面，我们的算法耗时大致是python的两倍。

上述的测试均来自于计算并输出$100000!$与$100000!*100000!$