CS205 · C/C++ Program Design

A Simple Calculator

Monad

September 26, 2022

目录

1 分析

1.1 阅读需求

先看 Project 的例子，它们包含了很多需要实现的功能的解释。

首先是一个最简单的 example：

Example 1

```
$ ./mul 2 3
2 * 3 = 6
```

这个项目的最基本的框架就是通过命令行参数传入两个数字，然后输出它们的乘积。

Example 2

```
$ ./mul 3.1416 2
3.1416 * 2 = 6.2832
```

嗯，还需要支持**小数**。

Example 3

```
$ ./mul 3.1416 2.0e-2
3.1416 * 0.02 = 0.062832
```

嗯，甚至还有**科学计数法**。

Example 4

```
$ ./mul a 2
The input cannot be interpreted as numbers!
```

然后对于**非法输入**，还需要检测出来并且报错。这个也挺合理的，毕竟一个整天会崩的程序不是好程序。

Example 5

```
$ ./mul 1234567890 1234567890
```

然后题目给了一个情况，如果输入一些非常大的整数，应该怎么处理？我大概想了 3 种思路：

1. 第一种思路就是，如果检测到输入的数字过大（比如超过了 `int` 的范围），就报错。当然，这个“超出范围”的判断，除了输入的参数之外，也要对乘积的结果做判断。

   对于前者，实现方法蛮简单的。标准库中提供了 `std::stoi`, `std::stol` 等函数，可以将字符串解析为数字。当这个数字超过可表示范围的时候，就会抛出 `std::out_of_range` 异常，捕获这个异常并报错即可。

   对于后者，不妨令乘积的结果为 `x = a * b`，如果乘法没有溢出（即结果是正确的），那么 `x / a == b` 一定成立，否则就意味着发生了溢出。（当然这里还需要先检查一下 `a` 是否为 `0`，不然就寄了。）

2. 另一种思路就是，对于范围过大的数，解析成 `double`，借助 `double` 的科学计数法特性，把范围过大的数舍弃一定的精度，只保留一定的有效位数，也是可以接受的。同时，这也是很多系统的计算器对于大数的处理方式。

   对于前两种方案，可以在 `https://gist.github.com/YanWQ-monad/b7eb7a6c7e2d8c170b8d1163c39b2ce5` 找到实现。不过由于是原型机，所以就没有加很多的细节处理。

3. 最后一种思路是，一不做二不休，直接干上**高精度**。这也是本项目使用的思路。

然后还有一个小问题，如果用户输入 `1e1000000000000` 的话，计算器将会输出一个 10^{12} 位的大整数，几乎不可能输出完。但是其中大部分都是 `0`，信息熵很小，在这种情况下用科学计数法输出十分的合适。所以这里加了一个是否输出科学计数法的选项，让用户来选择。

更具体地，就是：

- `./mul A B -scientific`，以科学计数法输出
- `./mul A B -scientific 5`，以科学计数法输出，且尾数的小数部份精度为 5 位
- `./mul A B -s 5`，缩写，功能与上一条相同

所以综上所述，我们要做一个支持**小数**的**高精度**乘法，并且能以**普通方式**或**科学计数法**进行输入输出的计算器。

1.2 需求实现

1.2.1 高精度

首先我们先解决高精度，高精度从高精度整数开始，小数可以由整数拓展得到。

对于高精度整数的**存储**，我们可以用一个 `char` 数组把数字的每一位储存起来，即

```
number = " 9  7  8  1  9  7  5  3  3  2  9  5  2 "
           ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
char[] = { '9', '7', '8', '1', '9', '7', '5', '3', '3', '2', '9', '5', '2' }
```

然后在 `char[]` 里面存字符会有点不方便，因为后面做运算的时候都要减个 `'0'` 转成 0 到 9，何不在存储的时候就储存 0 到 9 的数呢？那改进之后就是这样子储存：

```
number = " 9  7  8  1  9  7  5  3  3  2  9  5  2 "
           ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
char[] = { 9,  7,  8,  1,  9,  7,  5,  3,  3,  2,  9,  5,  2 }
```

这样我们就完成了高精度整数的储存。为了令程序更优雅，这里用 `vector<char>` 来储存，也可以在一定程度上避免在指针操作上寄掉。

至于解析和输出，也挺简单的，大概就是：

```cpp
string s = "9781975332952";  // <- input
vector<char> digits;  // <- output
transform(s.begin(), s.end(), back_inserter(digits), [](char c) { return c - '0'; });
```

```cpp
vector<char> digits{ 9, 7, 8, 1, 9, 7, 5, 3, 3, 2, 9, 5, 2 };  // <- input
string s;  // <- output
transform(digits.begin(), digits.end(), back_inserter(s), [](char c) { return c + '0'; });
```

1.2.2 高精度乘法

接下来，就是乘法部分了。一般的高精度乘法用模拟竖式计算的方法来实现，但是这个方法有个弊端，就是它的时间复杂度是 $\Theta(n^2)$，有亿点点大。

所以，这里决定使用 FFT 来进行乘法运算。

高精度整数的多项式表示

定理. *对于任何一个高精度数字，都有多项式与之对应*

首先，我们可以把一个 n 位的高精度整数记为一个 n 项的多项式：

$$f(x) = \sum_{i=0}^{n} a_i x^i$$

其中 a_i 表示储存数字的数组 `array` 中，**从右往左**（即从低位到高位）数第 i 个元素的值。例如，对于整数 `9781975332952`，其多项式为：

$$\begin{array}{rl} \texttt{array[] = \{} & \texttt{9, 7, 8, 1, 9, 7, 5, 3, 3, 2, 9, 5, 2 \}} \\ & \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow \\ f(x) = & 9x^{12}+7x^{11}+8x^{10}+1x^{9}+9x^{8}+7x^{7}+5x^{6}+3x^{5}+3x^{4}+2x^{3}+9x^{2}+5x+2 \end{array}$$

再者，对于任意一个多项式 $f(x)$，将 $x = 10$ 代入到 $f(x)$ 中，就可以得到其对应的高精度整数。

高精度整数乘法与多项式乘法

所以如果我们要计算数字 a, b 乘积 c，我们可以将 a, b 分别用多项式 $f(x), g(x)$ 表示，然后计算

$$h(x) = f(x)\, g(x)$$

再将 $h(x)$ 还原回高精度整数即可。

证明. 将 $x = 10$ 代入等式两边，则由上式，有 $h(10) = f(10)\, g(10)$ 成立。

且 $a = f(10)$，$b = g(10)$，则有 $c = h(10) = f(10)\, g(10) = ab$ 成立。

多项式卷积优化

要计算

$$h(x) = f(x)\, g(x)$$

的话，最直接的方法就是直接把系数都乘起来。但是这样将系数相乘的时间复杂度是 $\Theta(n^2)$，和暴力竖式乘法相比，并没有什么优势。

但是这里多项式乘法 $h(x) = f(x)\, g(x)$，属于多项式卷积，可以用 FFT 优化。

FFT 优化是什么东西呢?

首先，一个多项式有两种表示方法，一种是**系数表示法**，即 $f(x) = \sum_{i=0}^{n} a_i x^i$，另一种是**点值表示法**，即用 x-y 平面上的、在这个 n 项多项式上的 n 个不同的点来表示代表这个多项式。

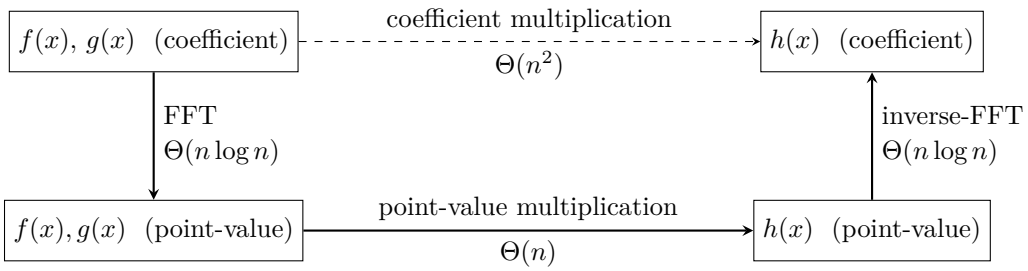
定理. 给定 $n+1$ 个点 $(x_0, y_0), \cdots, (x_n, y_n)$，一定有唯一的一个多项式与之对应（即 $f(x_i) = y_i$），且其次数不大于 n（即最多有 $n+1$ 项）。

由于是代数领域的命题，这里就不展开证明了[1]。

然后如果我们在系数表示中做多项式乘法，就需要 $\Theta(n^2)$ 的时间。但是如果我们在点值表示法中做乘法，我们只需要把两个多项式的点值两两相乘即可（假设 $\{x_i\}$ 相同），只需要 $\Theta(n)$ 的时间，非常优秀。

既然点值表示法那么方便，怎么把系数表示法转成点值表示法呢? 如果我们暴力代入求解，那就是 $\Theta(n^2)$，还是摆脱不了。但是幸运的是，我们有优秀的**快速傅里叶变换**，它通过取特殊的 x_i 值，并且运用了一些数学定理，使得它可以在 $\Theta(n \log n)$ 的时间内可以把系数表示法转成点值表示法。然后也有**快速傅里叶逆变换**，可以从点值表示转回系数表示。

所以大致的流程就是：



[1] 详细证明可参考：`https://en.wikipedia.org/wiki/Polynomial_interpolation#Interpolation_theorem`

这个流程就有点“曲线救国”的味道，经过一条更曲折的道路来达到优化的目的。

经过 FFT 优化之后，时间复杂度就下降到 $\Theta(n \log n)$ 了，非常的不错。经过实际测试，在长度 $n = 10^6$ 时，做乘法所需的时间为约 250 ms。[2]

还有一点需要注意的是，两个数乘起来之后，由于 FFT 运算前需要补足 2^k 位，所以做完乘法之后可能会出现很多前导 `0`，可以将其去掉。

1.2.3 压位

设计思路

想想感觉上面的思路，数组中的一个位置只存储了一位数字，无论是在存储方面还是在运算方面，都有点浪费。所以我们可以压一下位。

压位的意思就是，可以数组中的一个位置可以存放多个数字。比如说，一个元素（即数组中的一个位置）储存 2 位数字的话，就是这个样子：

```
number = "  9   78  19  75  33  29  52 "
           ↓   ↓   ↓   ↓   ↓   ↓   ↓
array[] = {  9,  78, 19, 75, 33, 29, 52 }
```

可见压位之后，数组的长度减小了，那么对应的多项式长度也减小了，进而做乘法时速度也会加快。但是压多少位比较合适呢，或者说，**最多能压多少位**？

对于传统的竖式乘法，超过 9 位的时候，两个元素相乘就很可能会超过 `int64_t` 的范围，所以最多能压 9 位。对于 FFT 来说，我们可以从其原理入手分析。

不妨设 $a = f(x) = \sum_{i=0}^{n} a_i x^i$，$b = g(x) = \sum_{i=0}^{n} b_i x^i$，那么

$$a \cdot b = f(x) \cdot g(x) = \Big(\sum_{i=0}^{n} a_i x^i\Big) \cdot \Big(\sum_{i=0}^{n} b_i x^i\Big) = \sum_{i=0}^{2n-1} \Big(\sum_{j=\max\{i-n,0\}}^{\min\{n,i\}} a_i b_{i-j}\Big) x^i = \sum_{i=0}^{2n-1} c_i x^i$$

不妨令 a_i, b_i 的最大值为 10^k，其中 k 表示压了多少位。那么有

$$c_i = \sum_{j=\max\{i-n,0\}}^{\min\{n,i\}} a_i b_{i-j} \le \big(\min\{n,i\} - \max\{i-n,0\}\big) \cdot 10^k \cdot 10^k \le n 10^{2k}$$

在 FFT 实现中，c_i 为虚数，一般用 `complex<double>` 来存储。然后 `double` 的有效数字为 2^{53}，即 15.95 位数字。也就是说，我们需要令 $n10^{2k} \le 10^{15.95}$。

这里我取了 $k = 4$ 作为最终的压位长度。首先是因为 $k = 4$ 时可以支持长度 $n \le 10^{7.95}$ 的整数运行结果准确，这个长度对本项目的计算器来说已经足够了。其次是 $k = 4$ 的时候，10^4 恰好可以装入到 `short` 中，可以更好地利用缓存，提高效率。

实现

在实际讲实现之前，我想先说明一下，为什么压位不这样压：

```
number = "  97  81  97  53  32  95  2 "
           ↓   ↓   ↓   ↓   ↓   ↓   ↓
array[] = {  97, 81, 97, 53, 32, 95, 2 }
```

即为什么多出来的一两位，要扔在前面而不是扔在后面。因为如果扔在后面的话，后面那个 `2`，到底是 `02` 还是 `2` 就会引起歧义，如果引入数字总长度之类的东西来处理这个情况，就会把代码变得更复杂。所以为了便于处理，扔在前面是一个比较好的选择。

[2] 测试平台为 MacBook Air, M1 chip, 16G memory

$$
\begin{array}{rccccccc}
\text{number} = \text{"} & 9 & 78 & 19 & 75 & 33 & 29 & 52\ \text{"} \\
& \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\texttt{array[] = \{} & \texttt{9,} & \texttt{78,} & \texttt{19,} & \texttt{75,} & \texttt{33,} & \texttt{29,} & \texttt{52 \}}
\end{array}
$$

由于数组索引 `0` 在左边，所以我们需要一开始就计算好第 0 个元素对应字符串中多少个字符，例如上面的例子中，就是 1 位。

那，怎么计算这个值呢？其实这个也不难，直接取模一下就好了，不过需要注意的是，我们需要的结果的值域是 $[1, k]$，而不是普遍的 $[0, k)$，所以还需要做一点转换。不过这个同样也不难，只需要一些 trick 就可以了，`(length - 1) % k + 1` 即可。

在实现中，就先把原字符串中每一位先“扔”入到一个临时变量中，然后等到收集完一个元素所需的数字之后，在将其 `push_back` 到数组中即可。

```
string number = "9781975332952";

size_t j = (number.length() - 1) % kDigitWidth + 1;  // 第 0 个元素需要多少个字符
uint16_t value = 0;  // temporary value
for (char digit : number) {
    value = value * 10 + (digit - '0');
    if ((--j) == 0) {  // 当收集完了，就 push_back
        digits_.push_back(value);
        value = 0;
        j = kDigitWidth;  // 然后重置计数器
    }
}
```

然后转回字符串也很简单，将每个元素分别转成字符串之后，再加到一起即可。由于将每个元素转成字符串的时候，是先从个位到高位的，和最终字符串的顺序——从高位到低位——相反，所以我们需要一个临时缓冲区，正序插入，反序输出。

```
vector<uint16_t> digits{ 9, 7819, 7533, 2952 };

string s;
char buffer[kDigitWidth];
for (uint16_t element : digits) {
    for (char &digit : buffer) {
        digit = static_cast<char>(element % 10 + '0');
        element /= 10;
    }

    // 反序添加到 s 中
    for (int i = kDigitWidth - 1; i >= 0; --i)
        s.push_back(buffer[i]);
}
```

然后转成字符串这个地方，没有特别处理前导 `0`，也就是说，对于上述例子，它会得到 `s = "0009781975332952"`。这是因为在这里进行特殊处理并不优美，再者我们也不一定要在这里就删掉前导 `0`，可以交给 `BigDecimal` 来处理。（在调用方 `BigDecimal` 的输出中，直接用 `string_view` 去除一段前缀就可以了，没有什么 cost。）

性能

实际测试下来，加上压位之后，$n = 10^6$ 的运行时间从 250 ms 降到了约 40 ms [3]，还是非常理想的。

于是，高精度整数部分到这里就说完了，然后这里给出 `BigInteger` 类的声明，具体实现在代码部分：

[3] 测试平台为 MacBook Air, M1 chip, 16G memory

```cpp
class BigInteger {
 private:
    bool positive_;
    vector<uint16_t> digits_;

 public:
    BigInteger();
    explicit BigInteger(string_view number);  // 从字符串解析构造

    void trim_leading_zeros();     // 删除前导的 0 元素

    bool is_positive() const;
    string get_number_string() const;  // 转为 string

    BigInteger operator*(const BigInteger &other) const;
}
```

1.2.4 高精度小数

说了那么多，结果全都是在说整数，那小数怎么搞嘛。别急，支持小数比乘法还简单。

首先不难发现，把小数中的小数点去掉，就和普通的整数没有什么区别。所以就可以用“**整数 + 小数点**”的思路来表示一个小数。

更具体地，我们用下面的式子来表示一个小数

$$\texttt{decimal} = m \cdot 10^n$$

其中 m 为高精度整数，表示上面的“整数”；n 也是整数，表示上面的“小数点”。并且指数部分 n 一般也不会大得特别离谱，所以这里就用普通的整数类型来存储了。

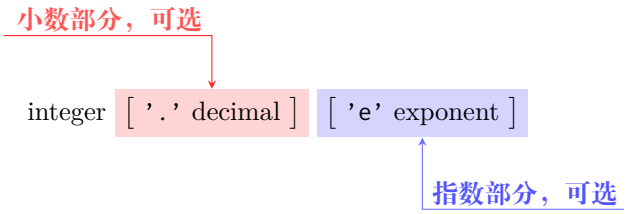
所以小数部分就可以用这么一个 class 来表示，具体实现在代码部分也会给出。

```cpp
class BigDecimal {
 private:
    BigInteger mantissa_;
    int64_t exponent_;

 public:
    explicit BigDecimal(BigInteger &&mantissa, int64_t exponent);  // 从 n, m 构造
    explicit BigDecimal(string_view number);  // 解析

    BigDecimal operator*(const BigDecimal &other) const;
}
```

1.2.5 输入输出

科学计数法 输入

科学计数法的格式可以这样表示：



我们需要先把 `.` 和 `e` 的位置找出来，然后基于这两个字符的位置，把三个部分提取出来。

提取完之后，把 integer + decimal 扔给 `BigInteger` 解析，然后指数 `exponent_` 就是 exponent - len(decimal)。

值得一提的是，当没有指数部分的时候，它就是一个普通的小数（在此基础上，如果没有小数点部分的话，就是一个普通的整数）。换句话说，这个解析输入的方式，是**兼容一般的小数和整数**的，就可以避免分类讨论让程序变得更复杂。

输入的合法性检查

虽然用正则表达式检查输入是否合法十分地方便，用 `^(-?)(\d*)(\.\d+)?(e(\+|-)?\d+)?$` 检查一下就完事了，甚至还能顺便提取出数字的每个部分。

但是在 C++ 里面，标准库的正则表达式有个弊端，就是它太慢了，以至于 “it is currently faster to launch PHP to execute a regex than it is to use `std::regex`” [4]。在 StackOverflow 的一个页面中[5]，列出了 C++ 标准库的正则表达式与其它实现或其它语言的正则表达式的速度差距，然后其中 C++ 普遍比其它实现慢十倍至一百倍。

其次，用正则表达式检查错误，它只能告诉你 “错了”，但是不能告诉你 “错在哪里”，这就对输出一个友善的错误提示有很大障碍。

再者，为了一个小小的错误检测而使用正则表达式，势必会给程序带来不必要的体积膨胀，不符合 Simple is Beautiful 的原则。~~当然这也是我为什么不喜欢 Electron 的原因。~~

所以我最终的解决方案是，在解析数字的同时，检查其中的每个数字是否在 `'0'` 到 `'9'` 范围内，并且检查指数部分（如有）是否超出范围等。这样首先不用多写很多代码，也可以在检查的时候分清楚不同的错误情况，以给用户生成更恰当的错误描述。

非科学计数法 输出

然后对于非科学计数法输出，首先令 k 为需要输出的整数部分的长度（此处先忽略前导和后导零的影响，具体的处理可看代码），即 $k = \texttt{exponent\_} + \text{len}(\texttt{mantissa\_})$。然后我们需要对 k 和 `mantissa_` 的长度 l 的相对大小关系进行分类讨论：

- 当 $l > k$ 时，即需要输出的整数部分的长度大于 `mantissa_` 长度，需要往整数部分后面补 `0`，即如 `1017`*000*。
- 当 $l \leq 0$ 时，整数部分为 “空”，即需要输出一个 `0`。然后在小数部份，可能需要填前导 `0`（需要填充 $-l$ 个），即如 *0.00*`1017`。
- 当 $0 < l \leq k$ 时，是最正常的情况，只需要控制好小数点的位置即可。例如：`10.17`。

科学计数法 输出

对于科学计数法输出，就简单不少，先输出一位整数，再输出小数点以及小数部份，指数部分就直接瞎算一下然后输出就 OK 了。大概就是：

```cpp
int64_t precision = stream.precision();
int64_t exponent = mantissa_view.length() + decimal.exponent_ - 1;

stream << mantissa_view[0];  // 科学计数法的整数部分只有 1 位

// 如果没小数的话 (即有效数字只有 1 个, 或用户指定精度为 0 位), 就不用输出小数部份了
if (precision > 0 && mantissa_view.length() > 1)
    stream << '.' << mantissa_view.substr(1, precision);
char exponent_sign = exponent >= 0 ? '+' : '-';
stream << 'e' << exponent_sign << abs(exponent);  // 指数部分
```

[4] The Day The Standard Library Died: https://cor3ntin.github.io/posts/abi/#why-do-we-want-to-break-abi
[5] https://stackoverflow.com/questions/70583395/why-is-stdregex-notoriously-much-slower-than-other-regular-expression-librarie

1.3 项目特色

1. **高精度支持**：高精度整数和小数，可以支持长度高达 10^7 的整数，保证不会出现任何小数被截断的情况。
2. **运算速度快**：在 FFT 的加持下，长度为 $n = 10^7$ 的数相乘，能在 0.5 秒内得出结果[6]。
3. **友善的错误提示**：可以提示是输入出现了哪种问题，比如说出现了非数字、指数部分错误等等。
4. **科学计数法输出**：对于一些超大的整数（如 `1e1000000000` 等），将全部数位都输出会花费很多的时间。此时用户可以选择用科学计数法输出。当然，用科学计数法输出也是全精度的，在没有用户的允许，不会私自截断小数。
5. **有帮助信息**：当你不会使用该程序，或者是忘记某些 options 的时候，可以通过 `--help` 参数打印帮助信息。

2 代码

2.1 核心代码展示

`source.cpp` 中的源代码有注释，建议前往那里观看代码。

```cpp
#include "header_required.h"  // placeholder
using namespace std;

class number_parse_error : exception {
 public:
    const char *reason_;
    explicit number_parse_error(const char *reason) : reason_(reason) {}
};

class FFTContext {
    vector<complex<double>> omega_, omega_inverse_;

 public:
    uint32_t n_, k_ = 0;

    explicit FFTContext(const uint32_t m);
    void transform(vector<complex<double>> &a, const vector<complex<double>> &omega) const;
    void dft(vector<complex<double>> &a) const;
    void inverse_dft(vector<complex<double>> &a) const;
};

class BigDecimal;

class BigInteger {
 public:
    constexpr static int kDigitWidth = 4;
    constexpr static int kDigitRange = 10000;

 private:
    bool positive_;
    vector<uint16_t> digits_;

 public:
    BigInteger() : positive_(true), digits_() {}

    explicit BigInteger(string_view number) : positive_(true) {
        if (!number.empty() && number[0] == '-') {
            positive_ = false;
            number.remove_prefix(1);
        }
        number.remove_prefix(min(number.size(), number.find_first_not_of('0')));
```

[6] 当然，这也是在 MacBook Air, M1 chip, 16G memory 上测得的

```
        if (find_if_not(number.begin(), number.end(),
                        [](char digit) { return '0' <= digit && digit <= '9'; }) != number.end()) {
            throw number_parse_error("not digit (0 to 9)");
        }

        decltype(digits_)::value_type j = (number.length() - 1) % kDigitWidth + 1;
        digits_.reserve(number.length() / kDigitWidth + 1);
        size_t value = 0;
        for (char digit : number) {
            value = value * 10 + (digit - '0');
            if ((--j) == 0) {
                digits_.push_back(value);
                value = 0;
                j = kDigitWidth;
            }
        }
    }

    void trim_leading_zeros() {
        auto non_zero = find_if_not(digits_.begin(), digits_.end(), [](auto digit) { return digit == 0; });
        digits_.erase(digits_.begin(), non_zero);
    }

    bool is_positive() const {
        return positive_;
    }

    string get_number_string() const {
        string s;
        s.reserve(digits_.size() * kDigitWidth);
        char buffer[kDigitWidth];
        for (auto element : digits_) {
            for (char &digit : buffer) {
                digit = static_cast<char>(element % 10 + '0');
                element /= 10;
            }
            for (int i = kDigitWidth - 1; i >= 0; --i)
                s.push_back(buffer[i]);
        }
        return s;
    }

    BigInteger operator*(const BigInteger &other) const {
        BigInteger result;
        result.positive_ = !positive_ ^ other.positive_;

        FFTContext context(digits_.size() + other.digits_.size());
        vector<complex<double>> lhs(context.n_), rhs(context.n_);
        copy(digits_.rbegin(), digits_.rend(), lhs.begin());
        copy(other.digits_.rbegin(), other.digits_.rend(), rhs.begin());

        context.dft(lhs);
        context.dft(rhs);
        transform(lhs.begin(), lhs.end(), rhs.begin(), lhs.begin(), [](auto x, auto y) { return x * y; });
        context.inverse_dft(lhs);

        result.digits_.reserve(context.n_);
        decltype(digits_)::value_type carry = 0;
        for (const auto &p : lhs) {
            carry += static_cast<decltype(carry)>(round(p.real()));
            result.digits_.push_back(static_cast<decltype(digits_[0])>(carry % kDigitRange));
            carry /= kDigitRange;
        }
        reverse(result.digits_.begin(), result.digits_.end());
        result.trim_leading_zeros();

        return result;
```

```
    }

    friend ostream &operator<<(ostream &stream, const BigDecimal &decimal);
};

class BigDecimal {
 private:
    BigInteger mantissa_;
    int64_t exponent_;

 public:
    explicit BigDecimal(BigInteger &&mantissa, int64_t exponent) : mantissa_(mantissa), exponent_(exponent) {}

    explicit BigDecimal(string_view number) {
        if (!number.empty() && number[0] == '+')
            number.remove_prefix(1);

        auto riter = find_if(number.rbegin(), number.rend(), [](char c) { return c == 'e' || c == 'E'; });
        auto e_pos_iter = riter == number.rend() ? number.end() : number.end() - (riter - number.rbegin()) - 1;
        size_t e_pos = e_pos_iter - number.begin();
        size_t dot_pos = find(number.begin(), e_pos_iter, '.') - number.begin();

        string_view part1 = number.substr(0, dot_pos);
        string_view part2 = e_pos == dot_pos ? "" : number.substr(dot_pos + 1, e_pos - dot_pos - 1);

        string mantissa_part;
        mantissa_part.reserve(part1.length() + part2.length());
        mantissa_part.append(part1).append(part2);
        mantissa_ = BigInteger(mantissa_part);

        if (e_pos_iter == number.end()) {
            exponent_ = 0;
        } else {
            char *str_end;
            errno = 0;
            exponent_ = strtoll(number.substr(e_pos + 1).data(), &str_end, 10);
            if (errno == ERANGE)
                throw number_parse_error("exponent out of range");
            if (str_end != number.end())
                throw number_parse_error("invalid exponent");
        }
        exponent_ -= static_cast<int64_t>(part2.length());
    }

    BigDecimal operator*(const BigDecimal &other) const {
        BigInteger mantissa = mantissa_ * other.mantissa_;
        int64_t exponent = exponent_ + other.exponent_;
        return BigDecimal(std::move(mantissa), exponent);
    }

    friend ostream &operator<<(ostream &stream, const BigDecimal &decimal);
};

ostream &operator<<(ostream &stream, const BigDecimal &decimal) {
    if (decimal.mantissa_.digits_.empty())
        return stream << '0';

    string mantissa_part = decimal.mantissa_.get_number_string();
    string_view mantissa_view = mantissa_part;
    mantissa_view.remove_prefix(mantissa_view.find_first_not_of('0'));
    int64_t integer_length = static_cast<int64_t>(mantissa_view.length()) + decimal.exponent_;
    mantissa_view.remove_suffix(mantissa_view.size() - mantissa_view.find_last_not_of('0') - 1);

    if (!decimal.mantissa_.positive_)
        stream << '-';

    if (stream.flags() & std::ios_base::scientific) {
```

```
        int64_t precision = stream.precision();
        int64_t output_exponent = integer_length - 1;

        stream << mantissa_view[0];
        if (precision > 0 && mantissa_view.length() > 1)
            stream << '.' << mantissa_view.substr(1, precision);
        char exponent_sign = output_exponent >= 0 ? '+' : '-';
        stream << 'e' << exponent_sign << abs(output_exponent);
    } else {
        if (integer_length <= 0) {
            stream << '0';
        } else {
            int64_t view_length = min(integer_length, static_cast<int64_t>(mantissa_view.length()));
            stream << mantissa_view.substr(0, view_length);
            mantissa_view.remove_prefix(view_length);
            integer_length -= view_length;

            fill_n(ostream_iterator<char>(stream), integer_length, '0');
        }

        if (!mantissa_view.empty()) {
            stream << '.';
            fill_n(ostream_iterator<char>(stream), -integer_length, '0');
            stream << mantissa_view;
        }
    }

    return stream;
}

struct options {
    bool scientific = false;
    int64_t scientific_precision = -1;
};

options parse_options(int argc, char *argv[]);
void print_help(const char *executable);

int main(int argc, char *argv[]) {
    if (argc < 3) {
        cerr << "You input less numbers than expected, use \"--help\" for help" << endl;
        return 1;
    }

    options option = parse_options(argc, argv);
    if (option.scientific) {
        cout << std::scientific;
        if (option.scientific_precision != -1)
            cout.precision(option.scientific_precision);
    }

    try {
        BigDecimal multiplier(argv[1]), multiplicand(argv[2]);
        BigDecimal result = multiplier * multiplicand;

        cout << multiplier << " * " << multiplicand << " = " << result << endl;
    } catch (number_parse_error &e) {
        cerr << "The input cannot be interpreted as numbers: " << e.reason_ << endl;
        return 1;
    } catch (exception &e) {
        cerr << "Unknown exception: " << e.what() << endl;
        return 1;
    }

    return 0;
}
```

统计数据

```
$ cloc mul.cpp
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.01 s (177.3 files/s, 81559.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                              1             77             81            302
-------------------------------------------------------------------------------
```

2.2 关于源码

本源代码用 `-Wall -Wextra` 选项下编译，没有任何编译警告。

代码风格遵循 Google C++ Style Guides。除了有一处地方（FFT 位反转）用特殊对齐会更好看之外，其余代码均能通过 cpplint.py 检验。

然后代码中尽可能使用 C++ 17 的迭代器相关函数和高阶函数，如 `find_if`, `back_inserter` 等，来代替传统 for 循环。这样做，能使代码能更好的具有自解释能力，更加清晰易懂。

举个例子，下面代码需要从 `number` 中找出非数字的字符，两段代码相比，显然后者更加的语义化、更可读。

```cpp
string number = "...";

// 片段 1
int pos = 0;
for (; pos < number.length(); pos++)
    if (!isdigit(number[pos]))
        break;

// 片段 2
auto iter = find_if_not(number.begin(), number.end(), [](char c) { return isdigit(c); });
```

当然了，对于一些核心代码，也写有详细的注释。平均每 5 行代码（空行除外），就有 1 行是注释。

另外对于在堆上分配空间的容器，如 `string`, `vector`，都会采取 `reserve` 或 `resize` 的方式，根据已知长度提前分配内存，避免后续插入时使容器多次扩容而影响性能。

3 结果与验证

3.1 运行展示

由于我不太喜欢在 PDF 中放图片，所以这里就把运行结果直接粘贴到此处，不贴图片了。不过可以保证的是，下面的所有运行结果都是程序的直接输出，不经任何修改。

Output of Examples

```
$ ./mul 2 3
2 * 3 = 6

$ ./mul 3.1416 2
3.1416 * 2 = 6.2832

$ ./mul 3.1416 2.0e-2
3.1416 * 0.02 = 0.062832

$ ./mul a 2
The input cannot be interpreted as numbers: not digit (0 to 9)

$ ./mul 1234567890 1234567890
1234567890 * 1234567890 = 1524157875019052100
```

科学计数法输出

```
$ ./mul 1234.5 0.06789 -s
1.2345e+3 * 6.789e-2 = 8.381020e+1

$ ./mul 1234.5 0.06789 -s 3
1.234e+3 * 6.789e-2 = 8.381e+1
```

负数和零

```
$ ./mul -998 1017
-998 * 1017 = -1014966

$ ./mul -1017 -996
-1017 * -996 = 1012932

$ ./mul -0 0
0 * 0 = 0

$ ./mul 1234.5 -0
1234.5 * 0 = 0
```

炒饭

```
$ ./mul -54.e+5 1234
-5400000 * 1234 = -6663600000

$ ./mul .23456e-1 1234
0.023456 * 1234 = 28.944704

$ ./mul . 1
0 * 1 = 0

$ ./mul 1e123456789012345678901 1234
The input cannot be interpreted as numbers: exponent out of range

$ ./mul 9999e1.1 1234
The input cannot be interpreted as numbers: invalid exponent

$ ./mul 2 3 4
Unrecognized option: 4
Maybe you input more numbers than expected

$ ./mul 2
You input less numbers than expected, use "--help" for help

$ ./mul 我爱 C++
The input cannot be interpreted as numbers: not digit (0 to 9)
```

3.2 结果验证

对于**程序基本逻辑验证**，使用 Google Test 框架进行测试。具体测试代码可翻阅 `mul_test.cpp`，这里贴出其中的一部分：

```
TEST(BigDecimalTest, ParsingTest) {
    EXPECT_EQ(big_decimal_string(BigDecimal("12345.6789")), "12345.6789");
    EXPECT_EQ(big_decimal_string(BigDecimal(".6789")), "0.6789");
    EXPECT_EQ(big_decimal_string(BigDecimal("123e+6")), "123000000");
    EXPECT_EQ(big_decimal_string(BigDecimal("123e-6")), "0.000123");
    EXPECT_EQ(big_decimal_string(BigDecimal("-5432.1e-01")), "-543.21");
}

TEST(BigDecimalTest, InvalidInputTest) {
    EXPECT_THROW({ BigDecimal("victorica"); }, number_parse_error);
    EXPECT_THROW({ BigDecimal("1234e10000000000000000000000"); }, number_parse_error);
    EXPECT_THROW({ BigDecimal("12.34e11.11"); }, number_parse_error);
    EXPECT_THROW({ BigDecimal(" 中文"); }, number_parse_error);
}

TEST(BigDecimalTest, MultiplicationTest) {
    BigDecimal lhs("-99.8"), rhs("10.17");
    EXPECT_EQ(big_decimal_string(lhs * rhs), "-1014.966");
}
```

以及测试的运行结果：

Test

```
$ ./mul_test
Running main() from /Users/monad/Projects/CppProject1/build/_deps/googletest-src/googletest/src/gtest_main.c
[==========] Running 6 tests from 3 test suites.
[----------] Global test environment set-up.
[----------] 1 test from FFTContextTest
[ RUN      ] FFTContextTest.IdentityTest
[       OK ] FFTContextTest.IdentityTest (0 ms)
[----------] 1 test from FFTContextTest (0 ms total)

[----------] 1 test from BigIntegerTest
[ RUN      ] BigIntegerTest.ParsingTest
[       OK ] BigIntegerTest.ParsingTest (0 ms)
[----------] 1 test from BigIntegerTest (0 ms total)

[----------] 4 tests from BigDecimalTest
[ RUN      ] BigDecimalTest.ParsingTest
[       OK ] BigDecimalTest.ParsingTest (0 ms)
[ RUN      ] BigDecimalTest.SanityTest
[       OK ] BigDecimalTest.SanityTest (0 ms)
[ RUN      ] BigDecimalTest.InvalidInputTest
[       OK ] BigDecimalTest.InvalidInputTest (0 ms)
[ RUN      ] BigDecimalTest.MultiplicationTest
[       OK ] BigDecimalTest.MultiplicationTest (0 ms)
[----------] 4 tests from BigDecimalTest (0 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 3 test suites ran. (0 ms total)
[  PASSED  ] 6 tests.
```

对于**结果正确性**验证，如果只需要很简单地验证一下结果正确并不困难，因为 Python 整数自带高精度，直接将两者相比即可。

Simple Verification

```
$ ./mul 1234567890123456 1234567890123456
1234567890123456 * 1234567890123456 = 1524157875323881726870921383936

$ python3
Python 3.9.9 (main, Nov 21 2021, 03:16:13)
[Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1234567890123456 * 1234567890123456
1524157875323881726870921383936
>>>
```

但是假如我们要验证长度为 10^6 级别的大整数，就有一点尴尬了。再者，如果我们想要测试多几组测试数据的话，就要反反复复地复制粘贴和比较，多少都觉得有点麻烦。这时候终端就有点力不从心了，我们需要一些更高效的测试方法。

前面说到，Python 的整数是自带高精度的，十分方便。于是我们就能考虑能不能用 Python，完成自动生成随机大整数，调用 `./mul` 进行计算，再进行比较这三个步骤，并且也可以很方面地进行压力测试。

这个脚本的实现细节会在下面第 4 部分的“困难与解法”中给出。

不过无论如何，用此脚本对 `./mul` 在整数长度为 10^6 时进行的长达 10 个小时的测试，全部通过。

3.3 性能

使用 Google Benchmark 框架进行性能测试，其中测试平台为 MacBook Air, M1 chip, 16G memory，结果如下：

Benchmark（元信息输出有删减）

```
$ ./mul_benchmark
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
BM_BigIntegerParsing/10                669 ns          673 ns       945129
BM_BigIntegerParsing/100               746 ns          747 ns       936117
BM_BigIntegerParsing/1000             1542 ns         1542 ns       453368
BM_BigIntegerParsing/10000            9076 ns         9076 ns        77275
BM_BigIntegerParsing/100000          94900 ns        94907 ns         7415
BM_BigIntegerParsing/1000000        947032 ns       947083 ns          743
BM_BigIntegerParsing/10000000      9852723 ns      9849125 ns           72
BM_BigDecimalParsing/10                710 ns          714 ns       979117
BM_BigDecimalParsing/100               853 ns          856 ns       818254
BM_BigDecimalParsing/1000             1906 ns         1909 ns       365684
BM_BigDecimalParsing/10000           11304 ns        11303 ns        62349
BM_BigDecimalParsing/100000         125442 ns       125412 ns         5713
BM_BigDecimalParsing/1000000       1158017 ns      1157883 ns          609
BM_BigDecimalParsing/10000000     12199303 ns     12199328 ns           58
BM_BigIntegerMultiply/10               882 ns          885 ns       788226
BM_BigIntegerMultiply/100             2434 ns         2427 ns       288572
BM_BigIntegerMultiply/1000           16324 ns        16327 ns        42822
BM_BigIntegerMultiply/10000         354609 ns       354627 ns         1977
BM_BigIntegerMultiply/100000       3582737 ns      3582821 ns          196
BM_BigIntegerMultiply/1000000     36854338 ns     36854053 ns           19
```

注：`BM_BigIntegerMultiply` 只测试到 10^6，和上面的 10^7 的不一样，~~不要被绕进去子~~。

4 困难与解法

4.1 错误处理

错误处理永远是一个老大难问题。由于返回错误码的方式比较繁琐，而且在构造函数中使用错误码，会造成代码语义不清晰的情况。而且考虑到这是一个 C++ 项目，就选择了 exception 异常处理。

由于这是一个应用（而不是一个库），不需要控制错误的粒度，所以具体的实现方式比较简单，哪里出错了就在哪里 throw 一个异常，然后在 `main` 函数中捕获并输出错误信息即可。

4.2 如何进行自动化测试

前面我们提到过，自动化测试是比手动测试好的。那在 Python 中，要怎么调用 `./mul` 来测试呢?

一个很明显的方案就是，可以模拟用户调用 `./mul`，即用 `subprocess` 库直接执行 `./mul`，然后收集输出，和标准结果比较。但是这样有个问题，这样做只能对程序进行整体测试，较难控制测试的粒度。换句话说，如果我们想要测试整数乘法对不对，就非常的尴尬，因为程序的输入输出并不直接由提供 `BigInteger`，如果硬要加上这个功能的话，会导致代码膨胀以及破坏程序的行为。

幸运的是，在本学期的一个编译原理课程的仓库中，发现了一个解决方案[7]（~~不要问我是怎么发现的，我没选这门课，它只是出现在了我 GitHub timeline 里面，然后好奇地点进去看了一下~~）。

它的方法是，把程序编译成一个 `.so` 动态链接库，然后用 Python 的 `ctypes` 去加载和使用它。

[7] https://github.com/sqlab-sustech/CS323-2022F/blob/main/lab1/ll_test.py

但是这里又有个问题，我们这个项目是 C++ 写的，C++ 的类和函数并不能直接在 .so 中导出，所以我们还要写一个 wrapper class 和 wrapper functions，用它们来代理 C++ 的东西。

你可能会问，诶不对啊，加上这些功能难道就不会破坏程序的行为了吗？嗯，确实是不会的，我们可以新建一个 abi.cpp 文件，在它里面写 wrapper，然后这个文件只会在测试的时候编译到 .so，并不会编译到 mul 中，所以不必担心。而且 abi.cpp 可以直接使用到 mul.cpp 中的类，所以可以自由地通过接口测试自己想测试的东西，不用拘泥于输入输出。

所以我们就只需要在 abi.cpp 里面写上一些 wrapper class 和函数即可，大致内容如下所示：

```
extern "C" {

struct c_biginteger {
    BigInteger *inner;
};

struct c_bigdecimal {
    BigDecimal *inner;
};

char buffer[10000000];  // used in to_string

c_biginteger* create_biginteger(const char *number);
c_biginteger* integer_multiplication(c_biginteger *lhs, c_biginteger *rhs);
char* integer_string(c_biginteger *integer);
void delete_integer(c_biginteger *ptr);

c_bigdecimal* create_bigdecimal(const char *number);
c_bigdecimal* decimal_multiplication(c_bigdecimal *lhs, c_bigdecimal *rhs);
char* decimal_string(c_bigdecimal *ptr);
void delete_decimal(c_bigdecimal *ptr);

}
```

然后再用 Python 的 ctypes 库调用即可，差不多是这样子：

```
class BigInteger(ctypes.Structure):
    def __init__(self, address):
        super(BigInteger, self).__init__()
        self.address = address

    @staticmethod
    def new(argument):
        if address := lib.create_biginteger(ctypes.c_char_p(argument.encode())):
            return BigInteger(address)
        raise ABIException("Maybe number parse error")

    def __mul__(self, other):
        return BigInteger(lib.integer_multiplication(self.address, other.address))

    def __repr__(self):
        return lib.integer_string(self.address).decode()

    def __del__(self):
        lib.delete_integer(self.address)

lib = ctypes.CDLL('libmul_abi.so')
lib.create_biginteger.restype = ctypes.POINTER(BigInteger)
lib.integer_multiplication.restype = ctypes.POINTER(BigInteger)
```

```
lib.integer_string.restype = ctypes.c_char_p
```

然后 BigDecimal 也类似。之后就可以用 new 来解析新建，a * b 来做乘法，repr(obj) 来转换成 str 了。接下来的部分，就是随便写个随机数生成，然后用 decimal 库进行高精度小数运算，并把它的结果与自己程序的结果进行比较。

然后跑起来之后它就会自己持续地测试，~~大就可以去睡大觉子~~，十分舒服。

4.3 记一个调优

在写完 benchmark 之后，就直接跑了下性能测试，结果发现，BigInteger 的解析需要 12 ms，然而 BigDecimal 的解析需要高达 35 ms。理论上这两者不应该有如此大的差异才对。

一开始我还以为在 BigDecimal 解析的时候，构建 mantissa_part 的时候没有 reserve 内存，导致多了一次内存分配导致的，不过优化了一下，只少了 1 ms，发现并不全是它的问题。

然后我又翻了一遍，最终锁定在这么一行：

```
auto iter = find_if(number.begin(), number.end(), [](char c) { return tolower(c) == 'e'; });
```

这段代码是用来搜索 e 的位置，考虑到大写的 E 也可以接受，当时想着 tolower 貌似更可读一点，就用了 tolower 把它转成小写再判断。但是实际上，我貌似高估了 C++ 编译器（至少是 macOS 上自带的 clang 编译器）的优化，它并不能很好地优化这个 tolower，所以可能导致效率偏低。

如果把它改成简单的判断，即

```
auto iter = find_if(number.begin(), number.end(), [](char c) { return c == 'e' || c == 'E'; });
```

它的效率就大为提升，从 35 ms 优化到了约 18 ms，直接对半砍了，非常不错。

事后将两种代码对比后发现[8]，编译器对上一种代码确实优化不是很好（即使开了 -O2），tolower 的调用没有内联，导致增加了不必要的跳转和指令，而且 tolower 内部可能也有一些分支判断，使执行速度下降。而后者直接把 tolower 的函数调用砍掉了，直接跟 'e' 和 'E' 做比较，性能大为改善。

然后知道了这个内幕之后，顺手又把 BigInteger 中的 isdigit 调用优化成了手动判断是否在 '0' 到 '9'，使得 BigInteger 的解析速度从 12 ms 优化到了 10ms 左右。

而且上面的代码还有一个优化能加，对于一些比较大的数，它的 'e' 都是出现在字符串的右侧的（如果有 'e' 的话），所以我们从右往左找可以期望更快地找到 'e'，即改成

```
auto iter = find_if(number.rbegin(), number.rend(), [](char c) { return c == 'e' || c == 'E'; });
```

然后再加上上面提到的对 mantissa_part 的优化，成功地把 BigDecimal 的解析速度优化到了 13 ms。[9]

5 总结

首先，我感觉~~这次的 Project 看起来还是挺简单的（低情商）~~，这次 Project 可以自由发挥的部分非常多（高情商）。如果只需要完成题目所提及的部分，而且想摸鱼的话，十几二十行就可以搞定这次 Project 了。当然了，我肯定不会止步于此。由于这次 Project 自由性很高，所以我把我能想到的东西，比如高精度、FFT 等，都写了进去。看着加进去的东西都 work 了，还是挺开心的。

然后，这是大概是我第一次很认真地写项目，或者说，是第一次以一种新的方式写 Project。之前写代码，基本上都是写个自己看的，并且自认为写的代码，自解释能力较强，就基本没写过注释。但是这一次，是写代码给别人看，所以我就在代码的自解释方面又努力了一把，而且也给一些可能比较模糊的代码，加了不少注释。现在回头再翻翻这些注释，心中还是蛮自豪的。值得一提的是，这次的 Project，也是我第一次大规模使用 find_if 等高阶函数进行编程，虽然性能不一定会十分的优秀，但是就如同其它语言的函数式编程一样，写起来非常舒服，可读性也比较好。

[8] https://godbolt.org/z/rrKfYraW5

[9] 本优化内的所有时间均在 MacBook Air, M1 chip, 16G memory 平台下、数字长度 $n = 10^7$、且一定有 'e' 的条件下测得

话说回来，虽然这次的 Project，题目上的要求给得比较少，但是我不知道为什么，居然给这个项目投入了将近 30 小时的时间，可能是因为自己确实喜欢写代码，一写起来就停不下来了。然后主要代码（即 `mul.cpp`），也写了将近 500 行，大小高达 19 KB。在开发的时候，为了方便，还提前自学了 CMake 的使用方法、Google Test 及 Google Benchmark 的简单使用，还有如何用 Python 调用 C++ 进行测试等等，都让我获益良多。

最后，这个项目也会放到 GitHub 开源，仓库已经创建好了，地址在 `https://github.com/YanWQ-monad/SUSTech_CS205_Projects`。在本项目的 deadline 之后，我会把本项目的整个开发环境（即包括但不限于 `CMakeLists.txt`、一些其它的测试文件，而不仅仅是一个源代码和一个 report），都 push 上去，~~欢迎大家 star~~。