

CS205 C/ C++ Programming - Project 1

Name: 叶璨铭(Ye CanMing)

SID: 12011404

CS205 C/ C++ Programming - Project 1

Part 1 - Analysis

1-1 The goal of this project

1-2 Some problems to consider

Part 2 - Code

Part 3 - Result & Verification

3-1 computation result correctness

3-1-1 basic cases

3-1-2 negative numbers

3-1-3 invalid input

3-1-4 big integer

3-2 time complexity experiment: 不同乘法算法的时间复杂度测试实验

3-2-1 实验模型: 乘法算法的输入模型与成本模型

3-2-1-1 输入模型

3-2-1-1 成本模型

3-2-2 实验原理: 目前有哪些常用的乘法算法? 他们的原理是什么?

3-2-2-1 平凡的算法: 简单但有效

3-2-2-2 常数优化: 压位高精度与二进制

3-2-2-3 分治算法: 机遇与挑战

3-2-3 实验器材: 倍率测试方法 (c++语言实现)

3-2-3-1 倍率测试

3-2-3-2 c++11 的chrono库 & Stopwatch类

3-2-3-3 进程测试 or 代码引入测试?

3-2-4 实验内容与结论: 不同乘法算法库的性能比较实验。

3-2-4-1 目前有哪些知名的大整数类库? 他们有哪些特点?

3-2-4-2 使用这些库之后, 如何编译代码?

3-2-4-3 实验数据处理

3-2-4-4 实验结论

Part 4 - Difficulties & Solutions

4-1 如何处理输入? 如何进行输出? 如何对程序进行测试?

4-1-1 io的选择

4-1-2 不同输入方式的处理: 命令行参数和运行时读参

4-2 如何对输入数据进行抽象?

4-3 如何高效实现乘法算法?

Part 5 - Summary

参考文献

Part 1 - Analysis

在这一部分, 我将分析本次project所要设计的程序所应该达到的目标, 并且提出一些具体的、不同层次的问题。这些问题不会在本部分即被提出解决方案, 我们将在第二、第三部分看到我们代码能够解决这些问题, 实现这些目标, 并且在Part4看到详细的解决方案与解释。

1-1 The goal of this project

我们需要使用C++程序设计语言来设计一个乘法器。什么是一个C++程序设计语言实现的乘法器？和Java语言设计的乘法器有何不同？用户将如何使用我们的乘法器？这是我们首先要考虑的问题。经过研究，一个C++设计的乘法器应该满足以下几点：

- 该乘法器必须是一个**系统程序级**的乘法器。

C++是一门偏重于系统程序设计的通用程序设计语言。所谓系统程序，就意味着其能够直接使用硬件资源，并且其性能受限于资源。¹ 拿我们的乘法器来说，我们需要想象，它是工作在一个计算速度特别慢的嵌入式系统上、或者我们抛给它的乘法计算任务是特别复杂的、规模很大的，在这样的情况下，我们的程序需要能够胜任系统程序的称号，保证在合理的时间内给出正确的结果。

既然有系统程序，相应的就有应用程序，应用程序调用系统程序来完成自己的任务。比如，一个规模庞大的科学计算程序，需要不断地使用乘法（大数乘法），我们需要保证如果这个程序多次使用我们的乘法器去帮它完成任务，我们的乘法器是值得信赖的。当然，在这种情况下很少说调用乘法器作为一个进程去计算，一般那个科学计算程序自己就得包含相应的代码，或者以动态链接库的形式调用我们（我们依然要保证代码是系统程序级的零开销），这里我们仅作简单的讨论。

- 该乘法器**正确使用**C++提供的**灵活且低开销的抽象机制**，能够让开发者优雅愉悦地进行维护、迭代。

如果说我们的乘法器不同于Java的乘法器是因为Java不适合编写系统程序级的乘法器，那么我们的C++乘法器异于C语言的乘法器就在于我们充分地、正确地使用了C++的抽象机制。对于乘法器而言，我们主要研究一下几个抽象：大整数类对于大整数的抽象；cout、cin对输入输出的抽象；string类对字符串的抽象。由于只上了两周课，还没有讲到C++的类，我主要从其api接口（使用方法）与java的风格的不同之处入手，做一些简单的讨论。

- 该乘法器是**安全可靠的**，可以处理其职责范围之内的异常情况。

如果输入的不是整数怎么办？我们需要抛出一个异常。由于我们是个乘法器，我们并不能知道为什么给我们的输入是错的，以及这种情况怎么处理，我们早点抛出异常，就能早点让知道怎么处理的上层程序去正确的处理异常。

此外，我们的内存管理也应该是正确的。比如对于类而言，应该遵循RAII原则。对于一个大整数类而言，我们通常是在栈上创建一个整数变量，然后使用它，那么当大整数销毁时，应该把它拥有的资源释放掉。

1-2 Some problems to consider

- 1.如何处理project文档中提到的，不同的输入方式？
- 2.如何对输入数据进行抽象？
- 3.如何高效实现乘法算法？

Part 2 - Code

```
//mul.cpp 版本1（版本2见 3-2-4）
#include <iostream>
#include "BigInt.hpp"
int main(int argc, char *argv[]){
    //preparation
    using namespace std;
    ios_base::sync_with_stdio(false); //make cin, cout faster //we don't need
    printf, scanf here.
    //we only want 2 arguments, or no argument from the command line.
    if (argc > 3){
        cout << "too many arguments." << endl;
        return -1;
    }
```

```

}
BigInt integerA, integerB;//default to be 0
switch (argc){
case 2:
    cout << "too few argument." << endl;
    return -1;
case 1:
    //if there is no argument, we ask for two integers.
    cout << "Please input two integers" << endl;
    cin >> integerA >> integerB;
    break;
default://equally case 3
    //first convert c-style strings to cpp standard strings,
    //then we use strings to construct big integers.
    integerA = string(argv[1]);
    integerB = string(argv[2]);
    break;
}
cout <<integerA<<" "<<integerB<<" = "<<integerA*integerB << endl; //
operator * has been overridden in class BigInt
return 0;
}

```

Note: BigInt.hpp is from [faheel/BigInt: Arbitrary-sized integer class for C++ \(github.com\)](https://github.com/faheel/BigInt)

Important:

- Since this isn't a dsaa class after all, plus we haven't learnt the concepts of cpp class, I would implement neither the best big integer multiplication algorithm nor the best-designed c++ class by myself.
- **For this project, I shall explain my preference between different libraries, and explain how to compile such libraries^[^3-2-4], in order to review and gain an thorough comprehension of the contents covered by the first two lectures.**
- also, I will try to implement a demo bigint.

Part 3 - Result & Verification

3-1 computation result correctness

3-1-1 basic cases

- compilation

```

[yecanming@Laptop10] - [/mnt/d/EnglishStandardPath/Practice_File/P_C_Cpp/P_Project1/build] - [2021-09-13 06:38:05]
[0] > ll
total 152K
-rwxrwxrwx 1 yecanming yecanming 40K Sep 10 17:20 BigInt.hpp
-rwxrwxrwx 1 yecanming yecanming 1.1K Sep 13 18:25 main.cpp
-rwxrwxrwx 1 yecanming yecanming 107K Sep 13 18:36 mul
[yecanming@Laptop10] - [/mnt/d/EnglishStandardPath/Practice_File/P_C_Cpp/P_Project1/build] - [2021-09-13 06:38:07]
[0] > g++ main.cpp -o mul

```

- case 1

```

[0] > ./mul
Please input two integers
2 3
2 * 3 = 6

```

- case 2

```

[0] > ./mul 2 3
2 * 3 = 6

```

- case 3

```
└─[0] ◇ ./mul 422 0
422 * 0 = 0
```

3-1-2 negative numbers

- case 1

```
└─[0] ◇ ./mul
Please input two integers
123 -123
123 * -123 = -15129
```

- case 2

```
└─[0] ◇ ./mul
Please input two integers
-123 -21334
-123 * -21334 = 2624082
```

3-1-3 invalid input

- case 1

```
└─[0] ◇ ./mul
Please input two integers
ycm 123
terminate called after throwing an instance of 'std::invalid_argument'
  what(): Expected an integer, got 'ycm'
[1] 481 abort ./mul
```

note: 这一异常是在构造BigInt时抛出的，我选用的这个BigInt类考虑到了这个情况。

如果它没考虑，我就要写个函数isInvalidNumber，然后先去掉首位的-+或者没有-+，然后查找字符串有没有非数字存在，最后和他一样：

```
else {
    throw std::invalid_argument("Expected an integer, got '\" + num + '\"");
}
```

注意，如果使用int，就没有这个好处，int被istream>>时，如果没有读到int，就会很奇怪的退出。

3-1-4 big integer

- case 1

```
└─[0] ◇ ./mul 7987978912378912937183781293712893721893712 2137987128937218937128937218937129837218937128937128937
7987978912378912937183781293712893721893712 * 2137987128937218937128937218937129837218937128937128937 = 170781961008880408
24399725686418861556625489724244913030840726442034003803343232903331363953544144
```

- case 2

3-2-1-1 成本模型

- 程序的运行时间取决于 基本操作（机器指令）的单条执行时间（由CPU等决定），与基本操作执行的频率（由代码逻辑决定）。
- 而成本模型则是重新定义基本操作，只考虑重要的基本操作的频率²
- 这里对于乘法，我们假定程序用string来存储大整数，那么我们的成本模型仅考虑对string的访问次数（包括读写）。

3-2-2 实验原理：目前有哪些常用的乘法算法？他们的原理是什么？

3-2-2-1 平凡的算法：简单但有效

最简单的算法就是模拟竖式计算乘法。

注意，实现的细节与竖式乘法略有不同：

- 竖式乘法要把中间结果抄一遍在中间，但是我们的算法可以直接对结果数组的相应位置加上相应的数
- 同时，我们可以最后才处理进位。
- 我们写竖式乘法时，纸和笔是无限的，但是我们的数组长度有限，需要合理的规划。

这种算法的时间复杂度是 $O(n^2)$ 。

3-2-2-2 常数优化：压位高精度与二进制

- 压位高精度是指，对于数组的存储进行优化。
 - 因为int或者unsigned int性能比较优秀，我们要迁就c/c++和有关的计算机环境，对整数的表示也是使用int数组
 - 而不是{0-9}数组，这就与我们的竖式乘法不一样。
 - **如果我们还是存0-9的数，无论是空间上（存储过长）还是时间上（问题的规模增加数倍）都不合适**
 - **如果我们分组来存，每一组存原本的4-5位**
 - 那么效率就会成倍提高。
 - 这种方法也可以理解成把10进制数转换为10000进制数。
 - 注意，时间复杂度的增长级别没有降低。
- 二进制是指，用二进制来存储整数。
 - 二进制乘法虽然也是竖式，但是其效率要高。
 - **原因在于，如果是0的话，可以直接continue**
 - **如果是1的话，可以把另一个整数移位之后加到结果上。**
 - 这样，遍历的次数就只剩下1的个数，然后复制、移位操作如果再CPU层面的话，是可以并行的，可能可以把n变成1
 - 这样我们效率有了一定的提高
 - 但是注意，**我们真的需要用1和0去表示二进制吗？**
- 什么转成二进制？本来都是二进制！
 - 注意，int本质上就是一个二进制的位模式
 - 而int被解释成int，只是十进制的显示而已，
 - 而刚才压位高精度中我们定义的10000进制，也只是一个定义，
 - 我们完全可以修改这个定义，改成2的多少次方。
 - 而string的数据也就是char的数组，char也是一个整数。³
 - 我们刚才为什么要努力做压位高精度，不就是为了让二进制的计算更加少、更加高效吗？
 - 如果我们用1和0的数组去表示，完全是错误的高层抽象。
- 二进制快在哪？
 - 我们所说的二进制，是压位二进制
 - 其修改的只是压位进制（数组相邻两个元素的基数比）的定义

- 其实质是利用“是一移一位，是零不用管”的策略，来提高效率的。
 - 而且我们还要考虑，转换成二进制，其实长度就变长了许多，我们的1不见得比原来的十进制的数要少
- 二进制慢在哪？
 - 用户输入的是10进制、要求输出的也是10进制
 - 我们如果用二进制来表示，我们需要做进制转换
 - 最简单的进制转换算法，那就是实现那个进制的高精度算法，然后一位一位加上去。
 - 那样，我们呈现结果的时候就要多算一次。
 - 但是，我们需要每次计算都呈现结果吗？
 - 显然不是的。
 - 我们只在计算结束的时候输出结果。**
 - 用户给我们的计算任务，一般都是复杂的，我们计算过程应该保证尽可能快，而呈现结果的一点代价，可以忽略不计。
- 延伸思考：bitset
 - c++还提供了个类，叫做bitset
 - 这个bitset是用集合去实现的，比如，1000100
 - 因为右数第2位、第6位是1，所以存为{2,6}
 - 如果要对这个位模式进行左移位，就变成{3,7}
 - 这样，对于很长的数，bitset的存储是高效的。
 - 如果用这种表示方法来抽象二进制的表示，是不是能实现一个更加高效的大整数呢？
 - 笔者将在之后尝试。

3-2-2-3 分治算法：机遇与挑战

- 以上的压位高精度和二进制优化，都不能改变 $O(n^2)$ 的实质。
- 而Karatsuba利用分治算法，成功解决了这一问题。
 - karatsuba算法把大数拆分成两部分，比如 $x=123$ $y=45$ ，从右数第一位分开
 - $x=12 \times 10 + 3$, $y=4 \times 10 + 5$
 - $xy = (12 \times 4) \times 100 + (12 \times 5 + 3 \times 4) \times 10 + 3 \times 5$
 - 因为加法是 $O(n)$ 的，可以当成常数了
 - 然后 12×4 等，就是递归的更小的乘法，需要做4个乘法
 - karatsuba改进后只需要三次乘法
 - 这就让递归方程式解出来的算法复杂度就不一样了，变快了。
- 然而我们注意到，这种算法需要多次加法，常数比较高。
- 所以我们需要进行实现，检查适合使用karatsuba算法的阈值。

3-2-3 实验器材：倍率测试方法 (c++语言实现)

3-2-3-1 倍率测试

- 在3-2-2中，我们已经分析已有算法的复杂度。我们注意到，他们都能够多项式时间内完成任务。
 - 因此，我们可以说，乘法算法的时间函数 $T(n) \sim an^b$ 。（ \sim 表示等价无穷大，见²），其中 b 由上述算法决定， a 由算法和机器都有关系。
- 我们不仅要从理论上了解、分析乘法的算法，接下来，我们还要通过实验，来验证我们的理论猜想。这样，我们才能给出结论，指导乘法器的开发。那么如何通过实验来测试一个算法的复杂度呢？
 - 我们可以这样，通过生成规模为不同的 n 的数据，然后测量程序运行的时间，最后通过统计方法，拟合 $T(n)$ ，从而确定 a 和 b 的值。
- 如何选定数据的规模呢？

- 我们采用一种叫做 **倍率测试** 的方法来选定数据的规模。这种方法是说，每次实验都选用上一次实验m倍的规模，然后如果我们的程序运行时间也增加了m倍，那么b的值就被测试出来是1，如果增加了m^2倍，那么b的值就被测试出来是2，换句话说，算法是n方的。
- 倍率测试的代码逻辑如下：

```
#include <iostream>
#include <string>
#include <sstream>
#include "../StopWatch.hpp"//implement later in 3-2-3-2
#include <gmpxx.h>
using namespace std;
long long time_trail(const string& val1, const string& val2){
    //implement later in 3-2-3-3
}
void doubling_test(int most_testing_seconds){
    Stopwatch sw; //最长测试时间
    for (int i = 1; sw.elapsedMicrosecond()*0.000001<most_testing_seconds;
i<=1) {
        stringstream val_str;
        for (int j = 0; j < i; ++j) {
            val_str<<"9";
        }
        cout<<i<<","<<time_trail(val_str.str(), val_str.str())<<endl;
    }
}
int main(int argc, char *argv[]){
    doubling_test(10);//只测试10s
}
```

- 其中，我们数据生成的方法是，生成n长度的9。
- 比如，测试规模是2，那么我们就测试99*99
- 测试规模是4，那么我们就测试9999*9999

3-2-3-2 c++11 的chrono库 & Stopwatch类

- **chrono** 是c++11引入的一个时间库，旨在解决程序计时出现的精度不足和不支持跨平台等问题。这个库实现了纳秒级的高精度时钟，并且对于时间点、时间长度有着比较好的定义。
- 我们使用chrono来实现一个StopWatch c++类，该类在被初始化时记录下时间，然后再被查看的时候告诉我们过去了多少时间。


```

#ifndef MUL_STOP_WATCH_HPP
#define MUL_STOP_WATCH_HPP

#include ...

using namespace std::chrono;
class Stopwatch{
private:
    time_point<high_resolution_clock> start_time;
public:
    inline
    Stopwatch():start_time{high_resolution_clock::now()} {}
    void reset(){
        start_time = high_resolution_clock::now();
    }
    inline
    long long elapsedMicroSecond()
    {
        return duration_cast<microseconds>(_Dur: high_resolution_clock::now() - start_time).count();
    }
};
#endif //MUL_STOP_WATCH_HPP

```

3-2-3-3 进程测试 or 代码引入测试？

- 进程测试，即把被测试的算法当做一个独立的进程。
 - 通过传递命令行参数，等待进程结束，
 - 我们可以测得进程运行的时间。
- 代码引入测试，即将乘法器的乘法函数或者大整数类复制或包含到测试器中。
 - 通过函数调用传递参数，等待函数执行结束，
 - 我们可以测得函数运行的时间。
- 无论是哪种测试，**其实质都是对乘法算法的用时进行测试。**
 - 如果我们不知道乘法器的源代码，或者没有相应库的dll文件，我们就不能通过函数调用来测试，只好通过进程测试。
 - 不过，现在我们可以知道乘法器的源代码。
 - 我们应该选用**误差小**的方法来进行测试，以保证实验的说服力。
- 误差分析：
 - 进程测试的代码如下

```

const string tested_path = "pwsh -Command \"../mul.exe\"; //pwsh指令
long long time_trail(const string& val1, const string& val2){
    Stopwatch sw;
    stringstream command;
    command <<tested_path<< " "<<val1<< " "<<val2<< " |out-null \\"; //out-
null阻止输出9*9 = 81这样的结果
    system(command.str().c_str());
    return sw.elapsedMicroSecond();
}

```

■ 实验数据如下：

1	2	4	8	16	32	64	128	256	512	1024	2048	4096
1545901	1279314	1210483	1212000	1212360	1216989	1185216	1170615	1206700	1207964	1294143	1301303	命令 行太 长。

- 可以看到，时间随着问题规模的增大并没有增大，而是保持稳定。
- 这说明，命令行、操作系统进程间通信等时间>>算法执行时间，导致算法的实验测定时间与问题规模无关。

- 因此，我们不能把进程测试方法的实验测定时间当做算法执行的时间。
- 代码引入测试的time_trail函数代码如下

```
long long time_trail(const string& val1, const string& val2){
    Stopwatch sw;
    BigInt(val1)*BigInt(val2);
    return sw.elapsedMicroSecond();
}
```

1	2	4	8	16	32	64	128
0	0	0	0	0	0	0	1990

- 注意到一开始速度太快，没法计时
- 所以，我们改进一下，让time_trail会重复执行*操作100次，并且舍弃掉前面的数据(9*9和99*99的差异不明显)。

3-2-4 实验内容与结论：不同乘法算法库的性能比较实验。

3-2-4-1 目前有哪些知名的大整数类库？他们有哪些特点？

- 在3-2-2中我们讨论了人们发明的各种大数乘法的算法，并进行了简单的原理分析。
 - 那么实际应用中，有哪些大整数类库比较出名呢？他们可以用吗？
- 经过搜索，我们找到如下结果
 - 其他语言
 - Java BigInteger类
 - 使用了二进制来表示数据
 - 根据数的大小来决定使用普通算法、Karatsuba算法、Toom-cook 算法中的一个。
 - 使用Java语言实现，没有用到native的cpp代码。^{4 5}
 - python Number类型
 - 使用了二进制来表示数据
 - 根据数的大小来决定使用普通算法、Karatsuba算法这两个中的一个。
 - 使用C语言实现⁶
 - c++ [faheel/BigInt: Arbitrary-sized integer class for C++\(github.com\)](https://github.com/faheel/BigInt: Arbitrary-sized integer class for C++(github.com)) (github排名高\star多\issue都有解决方法)
 - 使用十进制表示数据
 - 只是用Karatsuba算法，没有决策分析。
 - 使用cpp语言实现
 - c++ **The GNU MP Bignum Library** (知名自由软件开发计划GNU 开发)
 - 使用二进制表示数据
 - 根据数的大小，决定使用普通算法、karatsuba算法、不同的toom算法、乃至FFT算法。⁷
 - 使用c语言实现的c库，但是提供了cpp接口。
 - 可以看到，这个库考虑最为周全。

3-2-4-2 使用这些库之后，如何编译代码？

- fahell/BigInt:

Highlights

- No additional dependencies apart from the standard library.
- Modern C++ (compiles with C++11 / C++14 / C++17).
- No special compiling or linking required. Simply download the [single header file](#), include it in your code, and compile however you would.

Usage

1. Download the [single-include header file](#) to a location under your include path. Then `#include` it in your code:

```
#include "BigInt.hpp" // the actual path may vary
```

2. Create objects of the `BigInt` class, and do what you got to do!

```
BigInt big1 = 1234567890, big2;  
big2 = "9876543210123456789098765432101234567890";  
  
std::cout << big1 * big2 * 123456 << "\n";  
// Output: 1505331490682966620443288524512589666204282352096057600
```

- 可以看到，这种库叫做“single header file”的库，对于使用者来说非常方便。
- 简单地查了一下，其实开发的时候，他们写了很多文件，按照cpp开发规范写的。
- 但是最后release的时候，他们把不同的文件手工合并起来，形成一个.h文件。
- 这样，对于我来说，我没有任何编译的麻烦。
- include这个库，就和include一样自然。

- GNU MP Bignum Library

- 第一步：解压 gmp-6.2.1.tar.lz

```
sudo apt install lzip  
lzip -d gmp-6.2.1.tar.lz  
tar xvf gmp-6.2.1.tar
```

- 第二步：安装到linux⁸

```
./configure  
make  
sudo make install
```

可以看到，这个库很大气，configure检查了很久，仔细确认我的gcc支不支持各种年代的c语言的操作。然后make有很壮观，成堆的gcc命令闪烁在屏幕当中，让人不经佩服。

- 第三步：编译我的程序⁹

```
//mul.cpp 版本2(版本一见Part2)  
#include <iostream>  
#include "GMPxx.h"  
int main(int argc, char *argv[]){  
    //preparation  
    using namespace std;  
    ios_base::sync_with_stdio(false); //make cin\cout faster //we don't need  
    printf\scanf here.  
    //we only want 2 arguments, or no argument from the command line.  
    if (argc > 3){  
        cout << "too many arguments." << endl;
```

```

        return -1;
    }
    mpz_class integerA, integerB;
    switch (argc){
        case 2:
            cout << "too few argument." << endl;
            return -1;
        case 1:
            //if there is no argument, we ask for two integers.
            cout << "Please input two integers" << endl;
            cin >> integerA >> integerB;
            break;
        default://equally case 3
            //first convert c-style strings to cpp standard strings,
            //then we use strings to construct mpz_class.
            integerA = string(argv[1]);
            integerB = string(argv[2]);
            break;
    }
    cout <<integerA<<" * "<<integerB<<" = "<<integerA*integerB << endl; //
    //operator * has been overridden in mpz_class
    return 0;
}

```

#编译指令

```
g++ mul.cpp -lgmpxx -lgmp -o mul
```

这部分有些难度，官方的tutorial找了很久没找到。

- 总结：为什么gmp不是一个头文件，编译又那么复杂呢？
 - 问题就在于，gmp是一个巨大的工程。
 - 如果我们每次都要include gmp，然后把整个工程都编译一遍，那么我们项目编译的速度就是很慢的。
 - 反之，如果我们以动态链接库的形式，比如make install到Ubuntu系统，那么不仅我们不用再次编译，而且其他程序要用到gmp的话，也不用自己去定位gmpxx.h的位置。
 - opencv也是类似的。

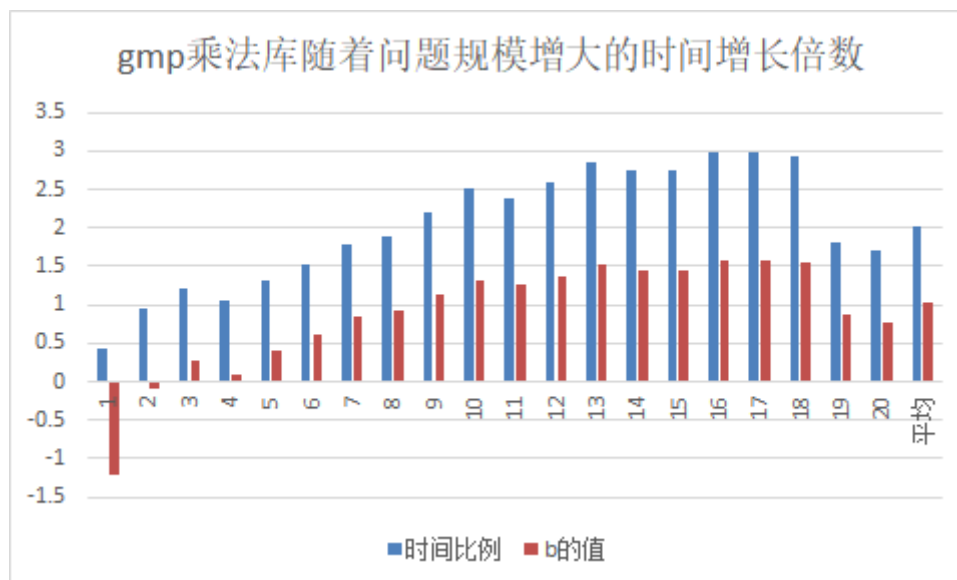
3-2-4-3 实验数据处理

前面我们已经给出了倍率测试的代码，现在我们编译并运行，记录实验数据，使用excel来处理数据。

- gmp的测试

问题规模 (n/位)	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
求解时间 (t/ms)	37	16	15	18	19	25	38	68	129	285	713	1707	4404	12572	34515	94703	283109	846142	2475611	4496290	7714063

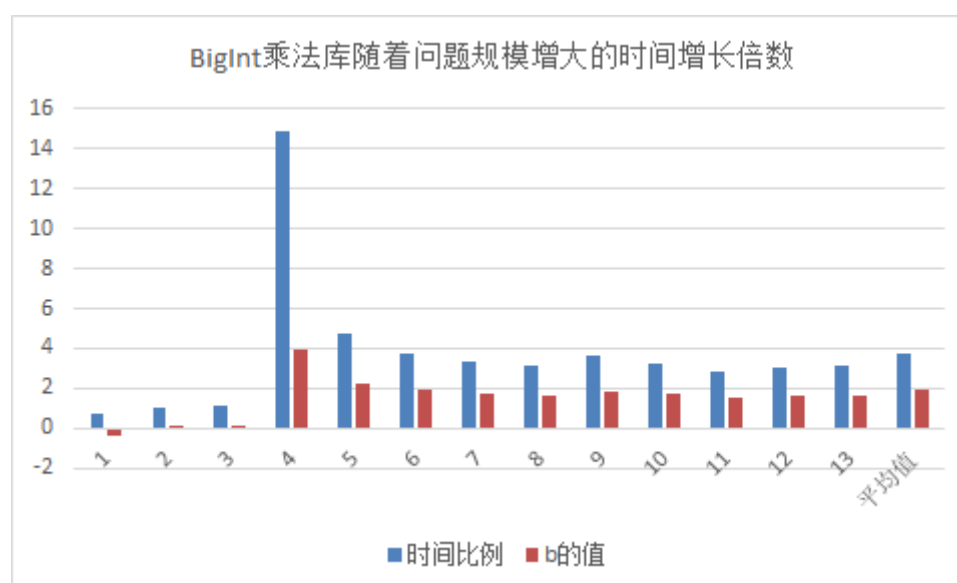
倍率计算：（时间比例是某次时间与上一次时间的比例，b的值则是log₂时间比例）



经过实验处理，我们可以认为 $b=1.02193$ ，而a的值不是我们研究的重点（如果要研究a的话，我们可以画出log-log图像，做直线拟合）。这个数值接近于1。

- BigInt的测试

问题规模 (n/位)	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
求解时间 (t/ms)	196	153	160	178	2652	12471	47111	156372	491495	1783774	5869792	16776494	50745462	161046600



经过实验处理，我们可以认为 $b=1.907088$ ，这个数值接近于2。

3-2-4-4 实验结论

- GMP库无论是常数还是算法增长数量级，都远远超过

Part 4 - Difficulties & Solutions

4-1 如何处理输入？如何进行输出？如何对程序进行测试？

4-1-1 io的选择

- 我们知道, C++兼容C, 因此即可以用printf、scanf, 又可以使用std::cout、std::cin。
 - 在网上论坛和平时的同学讨论中, 我们常常会听到这样一些说法 “printf是c语言的, c语言底层! 肯定快! 能用printf就用printf! ” “我是搞信息竞赛的, 之前有道题我算法写得太烂了, 过不了, 然后我试着把cout改成printf, 结果过了, 从此我就一直用printf。对了, scanf也别用, 比赛的时候手写一个int快读! ”
 - 这些说法固然有其计算机语言历史发展乃至实用主义的道理, 然而其对于C/C++语言特别是C++语言的设计目的的认识是片面的。
- c++设计的一个目的, 就在于所谓的**零开销原则**¹
 - **零开销原则**是 C++ 设计原则, 所说的是:
 1. 你无需为你所不用的付出。
 2. 你所用的正与你所能合理手写的效率相同。¹⁰
 - cin和cout作为c++的一个基本的语言特性或者说抽象机制 (对输入输出的抽象), 从语言的设计角度来说, 是不可能比你用更加底层的c语言性能差很多。如果存在一个程序员能够用手工的方法 (比如说“写个快读”), 取用更加底层的puts、gets去模拟cin>>i, 那么就会有很多人去效仿, 也就违背了C++程序设计语言设计的初衷。
- 说了这么多思想, 那么实际上C++的cin、cout表现如何呢?
 - 这里有一个简单的测评[Using scanf\(\) in C++ programs is faster than using cin? - Stack Overflow](#)
 - 可以看到, 如果不要C++兼容C的输入输出, 那么C++的效率更高。
 - 为什么cin、cout快? 主要原因是:
 - scanf、printf需要运行时解析字符串,再决定如何输出
 - cin、cout在编译时就可以确定调用哪个operator。
- 另外, 还有一种说法, 说printf的好处在于格式化字符串, 填入一个东西, 而cout只能各输各的。对于这种说法
 - 我认为有一定的道理。如果拿java去比较, java也认为printf很重要 (String都要有个format方法), 而java实现c++的cout这种“不同类型我都可以输出”的方法更加优雅, java是通过Object类的toString(), 基本类型使用工具类或者加上空字符串来转换、其他类继承重写来实现这一点的。
 - 但是, 你不能说纯粹的c++就没有这种格式控制的输入输出
 - 我们知道, 所谓的格式控制输出, 实际上最主要的用途应该是对int、float、double这样的基本类型的格式控制 (比如保留几位小数、几进制输出) 而言的。 (而不是实现“不同类型我都可以输出”)
 - 就是这样c++风格的一个解决方案。
 - 通过导入这个头文件, 我们可以设置cout下一个输出的进制、填充、精度等, 可以说是十分方便。¹¹

4-1-2 不同输入方式的处理: 命令行参数和运行时读参

- 命令行参数
 - 命令行参数输入, 也就是通过
 - ```
./mu1 [number1] [number2]
```
  - 的方式进行输入
  - 它的好处是方便进程间管道调用, 从而使得
    - 我们的程序能够为其他程序提供工具
    - 我们的程序可以被命令行测试.[^3-2-3-3]

- 运行时读参
  - 运行时读参则是没有命令行参数的两个数字的情况下，请求用户输入两个数字作为输入。
  - 这个过程如何加速，我们在4-1-1已经进行了探讨。
- 根据参数的数量进行调整
  - 如果参数有两个，那么就是命令行参数读入。
  - 如果是0个，那么就是运行时读参
  - 如果只有1个，或者多于三个，我们有责任提醒用户出现了错误，并且退出程序。
  - 注意事项：c/cpp的argc计数，记录的是包括自己路径（或者命令，不同编译器不同）的参数数，比上文所述的参数数量多一个。
- 了解了命令行参数和运行时读参之后，我们马上想到下一个问题：读入的参数是什么？
  - 我们知道，命令行参数是char\*类型的数组，表示一系列的参数，每一个参数是char\*类型的
  - 然而，我们在运行时读参的话，不一定要读入char\*类型。
  - 如何才能合理地处理这一情况？
  - 换句话说，我们如何对输入数据进行抽象？

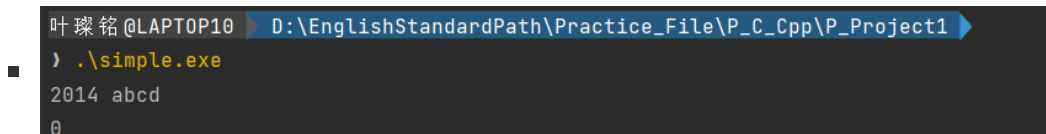
## 4-2 如何对输入数据进行抽象？

- 输入的数据，是两个等待被乘的整数，那么很自然我们想到用int类型去抽象它们。

```
//simple.cpp
#include <iostream>
int main(int argc, char *argv[]){
 using namespace std;
 int a, b = 0;
 cin>>a>>b;
 cout<<a*b;
}
```

- 然而，这种抽象方式有如下问题：

- 无法处理异常输入



```
叶臻铭 @LAPTOP10 D:\EnglishStandardPath\Practice_File\P_C_Cpp\P_Project1
> .\simple.exe
2014 abcd
0
```

- 对于异常输入，cin<<将b的值赋值为0，而不是报出一个异常推出程序。
    - 这就给出了错误的a\*b结果。
  - 无法让 命令行参数和运行时读参 两种方式统一
  - 无法处理超过int范围的整数
- 因此，我们需要先读入两个string，对string分析无误后，使用string来表示这两个数的数据。
  - 由于string的长度是无限的（受限于实际设备系统环境），而且不会有输入错误，我们就解决了上述问题。
    - 那么，我们如何对string表示的数来高效地进行乘法计算呢？
    - 注意，string不一定是最终的答案。我们的数据可以进一步被抽象为int数组、char数组、bitset，可以按照二进制，也可以十进制表示。因此，我们的讨论只是解决了输入的抽象。
  - 事实上，我们需要的是一个**抽象数据类型**。抽象数据类型将函数与数据关联起来<sup>2</sup>。具体来说，一个大整数类，可以把大数乘法、大数加法、大数除法、大数减法等函数与大整数的表示方式（比如string，也可以是数组）关联在一起，以类型的方式呈现在用户面前。
  - cpp的抽象数据类型和java类似，都是用类的概念去表示的。
  - 但是cpp的类和java的类差异很大。

- cpp的设计目标之一，就是让用户使用类定义的类型，仿佛就和使用内置类型一样，既没有api上的区别，也没有效率上的区别。<sup>1</sup>
- 为了达到这一目标，c++的很多设计都是和java不同的。

## 4-3 如何高效实现乘法算法？

见 3-2 的详细分析。<sup>12</sup>

## Part 5 - Summary

---

- 通过本次project,
  - 了解了大数乘法的不同算法及其时间复杂度
  - 学习了C++编程的一些思想
  - 了解了C++编程中如何编译使用了库的程序
- 得出结论，C++编程中遇到处理大数乘法的问题时，应该使用知名的GMP库。如果要自己实现的话，应该使用压位高精度存储数据，然后根据数的大小来选择最合适的算法来求解。
- 可以说，本次project很有收获，对以后的project的制作提供了经验。

## 参考文献

- 
1. 本贾尼·斯特劳斯特鲁普, 斯特劳斯特鲁普, 王刚, 等. C++程序设计语言[M]. 机械工业出版社, 2016. [↗](#) [↗](#) [↗](#)
  2. 算法：第四版/（美）塞奇威克，（美）韦恩 著；谢路云译. ——人民邮电出版社. (2020.5) [↗](#) [↗](#) [↗](#) [↗](#)
  3. 于仕琪. week1.pdf, week2课件.pdf [↗](#)
  4. java的BigInteger的乘法运算是用什么算法实现的？ - 知乎 (zhihu.com) [↗](#)
  5. jdk11的BigInteger类，查找native关键字，没有native出现。 [↗](#)
  6. cpython/longobject.c at main · python/cpython (github.com) [↗](#)
  7. Multiplication Algorithms (GNU MP 6.2.1) (gmp1ib.org) [↗](#)
  8. Installing GMP (GNU MP 6.2.1) (gmp1ib.org) [↗](#)
  9. C++ Interface General (GNU MP 6.2.1) (gmp1ib.org) [↗](#)
  10. 零开销原则 - C++中文 - API参考文档 (apiref.com) [↗](#)
  11. 标准库头文件 - cppreference.com [↗](#)
  12. 在这一部分，我们将按照确定理论模型、搜集资料分析、实验验证的思路，从浅入深，全面分析乘法算法的时间复杂度。最后，我们将给出结论，分析何时使用什么样的算法是更优的，从而为我们c++实现的系统级乘法器的高效性提供可靠的依据。 [↗](#)