LAB 5

马颢宸 23307110426

2025/4/28

2.1

2.1.4

1

做出如下三处修改

- 添加`const`，以保护`value_`不被修改

```
    size_t size() const{ //保护成员变量不被修改
        return 1 + value_.size();
    }
```

- 添加`const`，以保护`num`对象不被修改

```
std::ostream& operator<<(std::ostream& os, const Bignum& num); //num对象是只读的常量引用，防止引用修改
```

- 添加`std::size_t`参数，符合C++对自定义字面量的标准定义

```
inline namespace literals
{
    Bignum operator""_bn(const char* num,std::size_t); //添加std::size_t参数，以严格遵循C++标准定义自定义字面量运算符
}
```

2

见代码

程序成功截屏如下：

```
D:\Archive\subject\two two\OOP\lab\lab5\lab5\bin>"D:/Archive/subject/two two/OOP/lab/lab5/lab5/bin/
bignum.exe"
Running 18 tests...
Test: Bignum_create_zero ... OK
Test: Bignum_assign_normal ... OK
Test: Bignum_add_normal ... OK
Test: Bignum_sub_normal ... OK
Test: Bignum_mul_normal ... OK
Test: Bignum_div_normal ... OK
Test: Bignum_add_n_n ... OK
Test: Bignum_sub_n_n ... OK
Test: Bignum_mul_n_n ... OK
Test: Bignum_div_n_n ... OK
Test: Bignum_add_p_n ... OK
Test: Bignum_sub_p_n ... OK
Test: Bignum_mul_p_n ... OK
Test: Bignum_div_p_n ... OK
Test: Bignum_add_n_p ... OK
Test: Bignum_sub_n_p ... OK
Test: Bignum_mul_n_p ... OK
Test: Bignum_div_n_p ... OK

Test Summary:
  Passed: 18
  Failed: 0
```

3

对`Get`函数进行扩展定义，以生成符号组合不同的数据进行测试。

详情见代码

4

1. 当前实现效率

对四个算法进行时间复杂度分析如下：

- **+**:
  - 时间复杂度：`O(n)`，其中 `n` 是两个字符串中较长的那个的长度。
- **-**:
  - 时间复杂度：`O(n)`，其中 `n` 是两个字符串中较长的那个的长度。
- ***:
  - 时间复杂度：`O(m * n)`，其中 `m` 和 `n` 是两个字符串的长度。
  - 使用逐位相乘的基础算法，效率低（尤其是当m、n较大时）。
- **/**:
  - 时间复杂度：`O(n^2)`，其中 `n` 是被除数的长度。
  - 使用逐步减去除数的算法，效率低（尤其是当被除数和除数的位数较大时）。

---

2. 优化方法 & 实现难点

对效率较低的`*`和`/`进行优化

(1)*

- 优化方法
  - **Karatsuba算法**：时间复杂度 `O(n^1.585)`
  - **快速傅里叶变换（FFT）**：时间复杂度 `O(n log n)`

- 实现难点
  - `Karatsuba`算法需要将字符串分割成两部分并递归计算，逻辑较复杂。
  - `FFT` 需要将字符串转换为多项式形式，涉及复数运算和逆变换，逻辑复杂。

(2) /

- 优化方法
  - **二分法**：时间复杂度`O(n^2*log(n))`
  - **牛顿迭代法**：时间复杂度`O(n^2*log(log(n)))`
- 实现难点
  - 二分法每次迭代都需要进行大整数乘法
  - 牛顿迭代法实现复杂，需要处理初始值和精度问题

(3) 更高效的数据结构

- 因为当前实现使用 `std::string` 存储大整数，字符串操作（如反转、截取）可能会带来额外的开销。
- 优化方法
  - 使用 `std::vector<int>` 存储每一位数字，避免字符串操作的开销。
  - 每个元素存储一位或多位数字，减少存储空间和运算次数。
- 实现难点
  - 需要修改所有函数以适配新的数据结构。

(4) 并行化运算

- 因为当前实现是单线程的，无法利用多核 CPU 的计算能力。
- 优化方法
  - 对于乘法和除法，可以将计算任务分解为多个子任务并行执行。
  - 使用 C++ 的多线程库（如 `std::thread`）或 GPU 加速（如 CUDA）。
- 实现难点
  - 需要设计合理的任务分解和结果合并策略。
  - 可能引入线程安全问题，调试难度较大。

---

3. 性能测试&比较

可以通过随机生成的随即位数的大整数来确保测试数据覆盖所有情况，设立正确的比较指标，并利用合适的测试工具进行分析比较。

测试指标：

- 执行时间：相同输入下的运行时间。
- 内存使用：内存占用情况。

测试工具：

- C++ `<chrono>` 库：测量执行时间。
- 使用 Valgrind 或 Visual Studio Profiler：分析内存使用情况。
- Google测试框架：GTest