CS219 Project 1: A Simple Calculator

蔡易霖 12312927

February 2025

1 分析与开始

拿到这个任务之后，我首先仔细阅读了任务的 requirements。开始时，我还认为实现一个计算器功能似乎并不复杂，直到题目中的这一段话引起了我的思考：

If you input some big numbers, what will happen? Please provide some possible solutions, and try to implement them.

```
./calculator 987654321 * 987654321
# The result should be 975461057789971041
```

众所周知，c 语言内置的整数类型中，最大的 long long 也只有 8 个字节的大小。如果输入的数超过了这个范围该怎么办呢？比较好的解决方法似乎是使用浮点数 float，double 或 long double，它们可以支持很大范围内的数值的的计算。

然而，c 语言内置的浮点数的浮点数的计算的精度没法得到保证。我不禁思考，有没有更好的方法来提高我的计算器的运算精度呢？

我带着这个问题去问 DeepSeek，得到的答案是高精度算法，也就是通过把大数进行拆分，变成一位一位的数进行储存，然后对拆分开来的每一位数进行运算，进而得到最终答案。有意思！但是这样拆分带来的结果是，当我进行运算时，进位和退位都需要我手动模拟。加减乘法倒还好，可是除法应该怎么办呢？

仔细询问 DeepSeek 后，我发现高精度下的这些运算都有一些很有趣的实现方法，弄得我心痒痒的。于是我决定使用高精度算法来实现计算器！

2 数据处理

2.1 数据的输入与转化

既然决定了使用高精度，我就不应该再使用 c 语言内置的整数或者浮点数类型储存用户输入的数据了。输入的类型是 char* 字符串，如果直接拿来运算的话十分麻烦。因此，我定义了一种数据类型 decimal_t 来存储数据。该类型具有两个字符串 integer 和 fraction，分别装着一个数的整数部分和小数部分，还具有一个 bool 类型变量 is_negative 记录该数是否为负数。我定义了函数 parse_number 来将输入的字符串转化为 decimal_t 的结构。如果使用科学计数法输入，我就用 convert_scientific 函数将输入的数转化为自然数，再将其存入 decimal_t 的结构中。由此就能实现输入的字符串到我运算时的数据结构的转换。

2.2 数据的合法性检测

有次调试计算器的时候，我不小心在按 1 的时候按成了 q，结果计算器居然正常地运行了，并且大言不惭地告诉我 $q + 1 = 0$。这个错误启示了我：输入数据的时候，用户有可能给出非法的输入，我因此应当判断出用户的非正确输入并且给出提示。因此，我在每次输入数据的时候会对输入的字符串进行一个循环，判断每一位是否有不属于数字，-，.，e，E 的字符。只要存在这些字符，用户就肯定输入了一个非法的数，因而此时会输出错误提示。

同时，我意识到如果用户的输入过长，字符串可能会因为过长而越界。因此，我给用户的输入设置了一个最大值 1000。当用户的输入超过 1000 位，程序就会返回错误提示。

2.3 数据的输出与输出模式切换

既然支持了科学计数法输入，我决定也支持科学计数法输出。但是为了方便用户观察结果，我没有对所有输出使用科学计数法，而只是对大于 1e10 和小于 1e-10 的数进行了科学计数法的转换。但即使使用了科学计数法，也存在一些不好表示的数。比如 1e100+1，精确的答案如果用科学计数法表示，也会是 1.000(很多 0)001e100，看起来并不美观，而如果截取了前面一些位数输出的准确性与精度又没能体现出来。

因此，我写了一个输出模式切换的功能。用户可以在交互模式下通过

mode precise 和 mode approximate 切换输出的精度。在 approximate 模式下，计算器会保留九位并且四舍五入；在 precise 模式下，计算器会尽可能输出它能计算的每一位的数值（所以在除法得到无限循环的小数输出的时候，场面会变得十分壮观）

3 运算的实现

用户的输入已经被包装成 decimal_t 类型了！为了让程序看起来更有条理，我规定了自己的加减乘除和开根号函数的输入都为 decimal_t 类型的指针，而输出都是 char 类型的指针。

3.1 简单的加减法实现

加减法的实现其实就像做一个竖式运算，首先要做的就是对齐加数与被加数的位数。我将对齐位数的操作封装在 align_decimal 函数里，在该操作后分别将两个数的小数位和整数位拼在一起，之后通过 add_string 函数将两个拼好的数进行加法操作，最后再在调整小数点的位置，然后我们就得到加法的结果啦！

至于减法，我一开始其实想的还蛮复杂，还想着要借位什么的要怎么实现。但是某个晚上，我幡然醒悟：减法不就是减数和被减数的相反数做加法吗！想到这一点，减法的问题也就迎刃而解了。

3.2 乘法基于 FFT 的实现

乘法运算其实也可以使用和加法类似的列竖式方法解决。然而，DeepSeek 给我介绍了另一种解决这个问题的办法：使用 FFT，即快速傅里叶变换做乘法。

简单介绍一下快速傅里叶变换，它是计算 DFT 的一种工具。所谓 DFT 指的是对形如这样的多项式带入 $x^n = 1$ 的 n 个根得到的值

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1$$

而 FFT 是通过类似分治算法对 DFT 进行拆分成奇数项和偶数项，进

而快速地计算 DFT。像这样：

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-i\frac{2\pi}{N}(2m)k} + e^{-i\frac{2\pi}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i\frac{2\pi}{N}(2m+1)k}$$

但这和乘法有什么关系呢？其实对于每个数，我们都可以将其写成度多项式的形式，比如：

$$12345 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$$

不难发现这个形式符合可以使用 DFT 的形式，因此我们可以对任意一个整数做 DFT 操作。这样做的意义是什么呢？DeepSeek 告诉我，根据卷积定理，**两个序列的卷积，其傅里叶变换等于它们傅里叶变换的点乘**。

这就意味着，要计算两个数的乘法，我们可以先对他们做 DFT，然后让他们相乘，再通过 DFT 的逆操作将他们转化回数字，就能得到乘法的结果。

这样做的好处是，传统乘法的时间复杂度是 $O(n^2)$，而通过 FFT 做的乘法由于应用了分治算法，计算的时间复杂度变成了 $O(nlogn)$，其效率得到了极大的提升。虽然传统乘法的时间复杂度在计算器中看起来已经足够，但是利用 FFT 计算乘法实在是非常炫酷！因此，我尝试了在我的计算机中使用 FFT 计算乘法。

在 DeepSeek 的帮助下，我写了 fft 函数，它可以正向和反向进行 DFT。由于 DFT 会生成复数结果，我使用了 <complex.h> 库以获得复数支持。正向 DFT 将我们输入的数转化成的多项式并通过二分法拆解到频域，得到一个新的复数数组。我们乘数和被乘数都进行 DFT，然后在 fft_multiple 函数中将得到的数组一一相乘。最后，我们再进行反向 DFT 将得到的多项式转化为整数，这样就能完成一个乘法的计算了！

3.3 牛顿迭代计算除法

除法的运算是我认为计算器中最巧妙的部分！我对计算除法的第一个思路是使用大除法，也就是找到对应的位置做减法，结果上就是把除法转化为做减法的过程。对于 n 位的数，这样做的一次减法要 n 次运算；对于 n 位的被除数，这样要做 m 次减法。因此大除法的时间复杂度是 $O(n^2)$。

有没有更好的算法呢？DeepSeek 给我介绍了使用牛顿迭代法计算除法。

牛顿迭代法其实是针对函数的。对于一个函数，如果要求一个 $f(x) = 0$ 的根，假如我们已经有了一个邻近这个根的 x_0，我们可以通过以下公式得

到一个更接近根的解：

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

现在我们来看除法的问题：如果我们想计算 a 与 b 相除，其实也就等效于计算 a 与 b 的倒数相乘。这个相乘可以调用我们之前写的乘法！现在的重点就是计算 b 的倒数。这时候就可以使用牛顿迭代法了。我们只需要估计出一个近似值，就可以想办法用牛顿迭代法近似。

如何估计出一个近似值呢？DeepSeek 给了我思路：可以通过不断地通过乘 2 或乘 0.5 的操作将 b 转化成一个属于范围 $[0.5, 1)$ 的数，并且记录做乘法的次数。然后通过公式

$$b = 1.48 - 0.96b_0$$

得到一个新的 b，再进行与刚刚同等次数的相同的乘法操作，就可以得到近似值。原理上，求倒数的过程就像是以 1 为轴做一个映射，而上述操作实现的操作能得到一个相同量级的结果。至于为什么是 1.48 和 0.96 而不是 1.5 和 1，我想是为了让特定输入下的模拟的 $f(x) = 0$ 根更接近正数的根，而不是负数的根，这样才能拟合出一个正数解，也就是我们想要的解。

关于牛顿迭代，如果我们直接求解 b，假如考虑方程

$$bx - 1 = 0$$

那么迭代式为

$$x = x_0 - \frac{bx_0 - 1}{b} = \frac{1}{b}$$

不难发现我们仍然需要做除法，所以这没有解决问题！但是如果我们考虑方程

$$\frac{1}{x} - b = 0$$

那我们就能得到迭代式

$$x = x_0 - \frac{\frac{1}{x_0} - b}{-\frac{1}{x_0^2}} = 2x_0 - bx_0^2$$

这时我们就只需要做乘法和减法就能进行拟合了！我们进行多次拟合后，得到的 x 就会是 b 的倒数！经过我的测试，在我的精度下，14 次拟合就已经绰绰有余，再与 a 相乘，我们就得到了 a/b 的值！

用牛顿迭代法计算除法的时间复杂度是 $O(nlogn)$，因为它都是用二分寻找近似值，并且调用的是 $O(nlogn)$ 乘法。当然，它的常数要比乘法大很多。这个除法通过调用已有的运算来进行运算，这一点实在是美妙!

当然，由于我之前的设计采用了 decimal_t* 类型输入 char* 类型输出，我不得不设计了一个由 char 转化为 decimal_t 的函数 change_format 来把 char 类型转化回 decimal_t 类型.....

噢对，被除数不能是 0，这一点我特判了一下。

3.4 牛顿迭代法开根号

牛顿迭代法开根号和做除法的思路其实是差不多的，无非都是找近似，再通过公式逼近

对数 a 开根号，在找近似的时候，若 $a_0 > 1$，我就不断令 $a = a_0/2$ 直到 $a_0^2 < a$; 若 $a_0 < 1$，我就不断令 $a = 2a_0$ 直到 $a_0^2 > a$。然而，我发现这样得到的近似值有可能更接近负根，这是我们不希望看到的。通过多次尝试，我发现令 $a = 0.7a_0$，令 $a = 1.5a_0$ 就可以避免负根，找到一个不错的近似值。

如果直接计算 $\sqrt{a}$，我们有方程

$$x^2 - a = 0$$

迭代式为

$$x = x_0 - \frac{x_0^2 - a}{2_0} = \frac{x_0}{2} + \frac{a}{2x_0}$$

我发现这样需要做除法，时间复杂度比较高。然而，如果我是计算 $\frac{1}{\sqrt{a}}$，有方程

$$\frac{1}{x^2} - \frac{1}{a} = 0$$

迭代式为

$$x = x_0 - \frac{\frac{1}{x_0^2} - \frac{1}{a}}{-\frac{2}{x^3}} = \frac{3}{2}x_0 - \frac{x_0^3}{2a}$$

这样我们就可以只用乘法得到 $\frac{1}{\sqrt{a}}$ 的值，最后只要再乘个 a 就能得到 $\sqrt{a}$ 的值了!

在我自认为写好之后，我发现我的程序会在输入为负数的时候陷入死循环，这让我感觉非常奇怪并且花了一个多小时寻找 bug 的来源，后来才幡然醒悟：明明就不能对负数开根号..... 总之我加上了个特判，会在开根号输入为负数时返回错误信息。

4 内存的泄露与修复

这部分是我编写计算器的整个过程再最为痛苦的一部分... 在此之前，我只是听说 c 和 c++ 语言的内存容易泄漏，而没有实际经历过。在上述功能基本实现后，根据 DeepSeek 的建议，我使用了-fsantilize=address 的编译选项来观察程序的内存情况。很不幸的是，第一次测试时，我发现我的程序有 270000byte 的内存泄漏，这对我而言是一个不小的打击。

DeepSeek 告诉我，内存的泄漏大多与指针有关，于是我开始仔细观察指针的相关事宜。很快我意识到一件事：

```
x = add_numbers(a,b);
```

我的代码中存在这大量这样的把一个指针赋值给另一个指针的操作。然而，此处的 x 原来指向的是另一个位置。被赋值了之后，原来的位置的地址就丢失了，进而导致可内存的泄漏。这是不合理的！（顺带一提，这是 DeepSeek 给我的框架中的代码，而且 DeepSeek 完全没发现有问题！）

于是我把那些代码都改成了这样

```
x_now = x;
x = add_numbers(a,b);
free(x_now);
```

在这个修改之后，我的内存泄漏果然减少了很多。在经过了又一轮排查之后，我发现了另一个问题。我给我的 decimal_t 原本类型写了一个释放内存的函数。它原本是这样的：

```
void free_decimal(decimal_t *num) {
if (num->integer) free(num->integer);
if (num->fraction) free(num->fraction);
}
```

我一直以为这个函数没有问题，因为追栈的时候明明没有看到这个函数，但在反复排查后我发现，虽然这个函数 free 了 decimal_t 的所有内部的指针，但是 decimal_t 本身也是个指针，而它是没有被 free 的！因此，为 i 把代码改成了这样

```
void free_decimal(decimal_t *num) {
if (num->integer) free(num->integer);
if (num->fraction) free(num->fraction);
free(num);
}
```

在长达两天的艰苦卓绝的修改之后,我的代码终于不会在加上-fsanitize=address的编译选项后报内存泄漏了!

5 结语 | 一点想说的话

这次的作业其实消耗了超乎我想象的时间。我复盘了一下，比较消耗时间的点一个是在学习新的东西，包括数据结构和一些数学方法；一个是修改自己的代码，因为我的代码是基于 DeepSeek 的模板进行修改的，但是改着改着，我发现 DeepSeek 的代码有一些不足，或者我想到一些更好的方法，就会修改代码。也是因此，我有过多次将大段代码删去进行重构的过程。

我想，这也能反映出我在代码的架构搭建上的不足，这一点希望我能在下一次的 project 中得到提升。想来这次 project 有点越写越上头的感觉，过程中不断地发现好方法，学习并应用的过程实在是令人感到愉悦。或许这就是代码的乐趣?