巨大数技术文档

目录

一、项目概述

1.1 项目背景

计算机对于数据的处理存在一定范围，当数据超出这个范围就无法正常表示以及运算，C 语言提供的各种数据类型，以 int 类型为例，可表示的有效范围是 $-217483648 \sim 2147483647$，而当涉及上千万位数据超过这个范围，参与运算的时候，C 语言提供的各数据类型显然无法满足，就需要设计一种数据结构来解决这些问题，也就是设计巨大数这个工具。

1.2 项目核心技术

整个项目核心技术主要有两个：微易码补码和万进制。

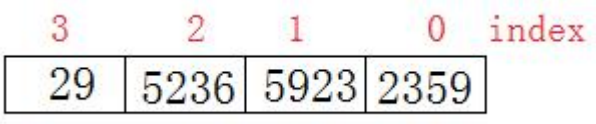
1.2.1 一万进制

解释万进制之前先讲述巨大数的存储部分，如何存储一个巨大数首先想到，字符串长度可以达到很长，但是通过字符串来表示巨大数，存在的弊端就是涉及字符和整型的转换，当巨大数位数达到上千上万位的时候，进行运算的时候，每一个字符去逐个运算效率非常低。所以字符串的方法不可取，采取的是万进制，用数组存储。

万进制：数组里面每个元素是一位，每逢一万进一，对比十进制不难理解。例如：29523659232359，可以写成如下

$$29523659232359=29\times10000^3+5236\times10000^2+5923\times10000^1+2359\times10000^0$$

存储到数组中为：是一个四位的巨大数。



1.2.1.1 万进制的优点

1> **存储效率高**

对比使用字符串来存储巨大数，使用万进制和数组来存储，每四个十进制数字是一位，存储效率相同，在物理空间上申请的大小并没有更加节省空间。

2> **计算效率高**

当两个巨大数进行运算的时候，字符串存储的方式每次只能运算一位十进制数字，而且还需要进行字符和整型的转换，同时当巨大数位数很大，字符串很长，计算时逐个计算每一位字符，极大了降低了整体效率。

而使用万进制和数组来存储，每一次运算可以处理四位的十进制数字，数组本身便是 int 类型，也不需要考虑类型的转换问题，

3> **进位时位数处理合适**

进行运算的时候，产生进位的情况，只需要将进位的数据保存下来，更高一位运算时候加上低位进位上来的数据即可。

万进制是最合适的进制原因：每一位存储的数据范围是 $0\sim9999$，在后续涉及乘法的时候，10000×10000 小于整型 int 的上限 2147483647，不会产生溢出的情况，而更大的进制 10 万相乘则会溢出，巨大数也是由十进制数字构成，而 10000 是 10 的次方倍，在转换进制的方面万进制提供了很好的便利，只需要每 4 位取出一个十进制数字存放到数组中即可，所以说 10000 是最合适的进制。

1.1.2 微易码补码

为了解决巨大数减法运算时产生借位的问题，模仿计算机补码设计一种微易码补码，将加减法问题统一转换为加法问题，巨大数的正负符号位单独拿出来表示正为 0，负为 1。

正数巨大数每一位的微易码补码为：该数本身，负数巨大数每一位的微易码补码为：9999-该数。

例如：

巨大数源码	452 3232 6921	-2639 5263 8462
微易码补码（0 非负，1 负）	0，452 3232 6921	1，7360 4736 1537

注：巨大数的数值部分和符号位部分单独处理。

微易码转换公式可表示为如下，x 表示巨大数一位的数据。

$$f(x)=\begin{cases}x & x\geq 0\\ 9999-x & x<0\end{cases}$$

微易码补码可以实现减法的证明放在后面讲解巨大数加减法具体的后面。

综上所述，可以给出巨大数结构体：

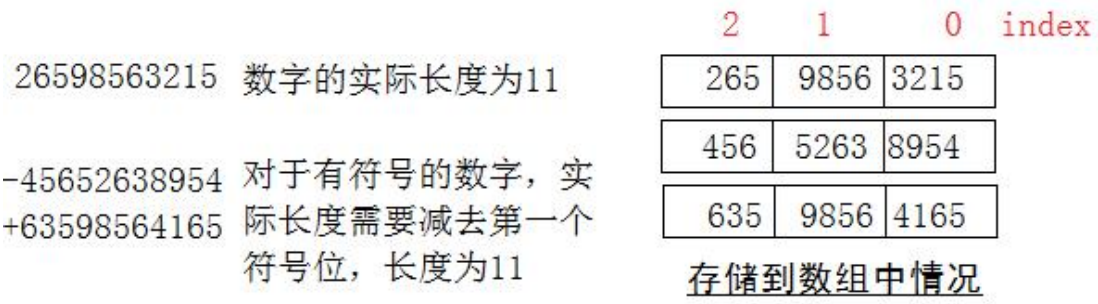
```
typedef struct HUGENUMBER {
    char number_sign;        //标记正负符号
    int *number_array;       //指向存储巨大数的数组
    int number_count;        //数组中元素个数
}HUGENUMBER;
```

存储例如

巨大数	2696323265659626	-59233598962
符号	0	1
存储数组	2696, 3232, 6565, 9626	592, 3359, 8962
数组长度	4	3

数据如何存储解释：巨大数存储在 txt 文件下，打开存储巨大数的文件之后，

先得到文件中数据的长度 file_length，假设文件提供的数据都是合法字符，不存在字母的情况，这里需要注意的是文件中可能存放数据第一个字符可能是' -' ,' +' ，巨大数的实际长度就需要相应进行变化，如果读到的第一个字符是 0-9 之间说明没有正负号字符，反之则存在正负号字符需要对上面得到的 file_length 减一。根据得到的文件长度就可以去申请存放巨大数数组空间，数组的长度 number_count = (file_length + 3) / 4,然后把巨大数高位放在数组高位，低位放在数组低位，存放的时候先存放巨大数最高位的数据，剩下数字都可以每 4 位取得存放在数组中。例如 234 5268 5982，先存放高位的 234，剩下数字都可以每 4 位取得。如下图：



1.3 开发环境

1.3.1 开发工具

Sublime Text3，vc6.0 编译环境。

1.3.2 操作系统

Windows 7。

1.3.3 开发语言

C 语言。

二、项目功能

2.1 提供对存储在文件中的巨大数的读取功能；

对存储在 txt 文件中的数据进行读取，并将读取到的所有数据分析，再分别存储到巨大数结构体中各个成员，为以后巨大数运算做基础。

2.2 提供对巨大数的显示和输出功能；

在对巨大数进行操作之后，可以将得到的新的巨大数显示在屏幕上。

2.3 提供将巨大数写入文件中的功能；

进行巨大数的运算之后，得到的新的巨大数显示在屏幕上的同时，将得到的答案存入一个文件。

2.4 提供对两个巨大数进行四则运算的功能；

在读取巨大数的基础之上，对得到的巨大数可以进行四则运算。

三、工具使用说明

3.1 提供函数.

函数名称	功能概述
addHugeNumber	得到两个整数巨大数相加结果
subHugeNumber	得到两个巨大数相减的结果
mulHugeNumber	得到两个巨大数相乘的结果，
divHugeNumber	得到两个巨大数相除的结果
getMecCode	根据传递进来的符号得到传递进来数字的微易码补码，正数微易码补码就是

	该数本身，负数微易码补码就是 9999-该数
indexOf	在加法运算中判断当前循环次数是否超过数组长度，超过返回 0，未超过则返回对应下标数组值
getHighestNum	得到巨大数最高位的一个数字，注意不是一位数字，为巨大数除法部分，放大除数服务
compareHighestNum	从 getHighestNum 函数得到最高位的一个数字并比较巨大数最高位的数字大小
getMinDivResult	通过计算得到除数可以放大的最大倍数，放大倍数是 10 的某次方
getHugenumerLength	得到巨大数的有效数据长度，不含高位无效数字 0
eliminateZeros	去除巨大数高位无效数据 0 的方法，先通过 getHugenumerLength 函数得到巨大数实际长度，再重新申请空间赋值，销毁原空间达到去除 0 的目的

四、操作方法说明

4.1 准备巨大数

在巨大数.c 文件同级目录下建立一个存放巨大数的 txt 文件，文件中不得出现非法字符，正数的表示可以加符号‘+’，也可以默认不添加符号，再去调用读取函数，将文件中的数据存储到巨大数结构体中去，该结构体实例就可以当做一个巨大数，进行后续的运算等操作。

4.2 巨大数运算（具体代码实现）

4.2.1 加减法的实现

①申请加法结果的空间，申请的结构体中数组的长度是被加数和加数中数组最长的那个还多申请一个空间。

②把已经申请好结果的数组长度当做循环次数，开始两个巨大数的相加，先通过 `int indexOf(HUGENUMBER myNumber, int current_Index)` 函数得到相加应该提供的值，再将返回值传进 `int getMecCode(int value, char number_sign)` 函数得到该巨大数该位的微易码补码。

③开始两个巨大数的相加，从低位开始向高位计算，结果数组值是当前位的被加数+加数+进位再和 10000 取余，进位数据是每次相加的结果除以 10000。

④将最后一次进位数据和两个加数的符号位这三个数据进行异或得到结果的符号位，并将最后一次进位数据加到结果的最低位。

⑤将得到的结果转为原码。

减法的实现：被减数-减数=被减数+（-减数），只需要将减数的符号位置反，再将被减数和减数传进加法的函数即可达到减法的目的。

微易码补码的证明：微易码补码的引入将加减法统一成加法来解决，加法出现的所有情况有：1、正数+正数，2、正数+负数，3、负数+负数。不妨设被加数为x，加数为y，即$x+y$。

①正数+正数（$x \geq 0, y \geq 0$）

由于正数的微易码补码为该数本身，实际上微易码补码并没有在这里使用，运算的时候还是源码进行运算，不需要证明。

②正数+负数（$x \geq 0, y < 0$）

在这种情况下结果是非负数或者负数

$$x + y = x + (9999 - |y|) = 9999 + (x - |y|)$$

1、当$x - |y| > 0$，运算结果为正，所以$9999 + (x - |y|) > 9999$，此次加法运算产生进位数据$carry_data = 1$，按照设计的加法法则，符号位 0^1^1 = 0，结果为正，将进位数据加至最低位和 10000 取余便是最终结果，结果便是源码。

$$\begin{aligned}&(9999 + (x - |y|) + carry\_data)\%10000\\&= (9999 + (x - |y|) + 1)\%10000 = (10000 + (x - |y|))\%10000\\&= x - |y| = x + y\end{aligned}$$

2、当$x - |y| \leq 0$，运算结果为负（此种设计的运算法则把 0 归为负数，所以数值 0 的符号位为 1），所以$9999 + (x - |y|) \leq 9999$，此次加法运算不产生进位数据$carry_data = 0$，按照设计的加法法则，符号位 0^1^0 = 1，结果为负，将进位数据加至最低位和 10000 取余便是最终结果。

$$\begin{aligned}&(9999 + (x - |y|) + carry\_data)\%10000\\=\ &(9999 + (x - |y|) + 0)\%10000\\=\ &(9999 + (x - |y|)\end{aligned}$$

此时得到的结果$(9999 + (x - |y|))$为负数，是微易码补码的形式，需要转换为源码为：$9999 - (9999 + (x - |y|)) = -x + |y|$，$-x + |y|$即是运算结果的数值部分，添上符号位完整结果为：$-(-x + |y|) = x - |y| = x + y$。

③负数+负数（$x < 0, y < 0$）

$$x + y = 9999 - |x| + 9999 - |y| = 9999 \times 2 - |x| - |y|$$

在加法运算中结果申请的数组长度是被加数和加数中最长的还多申请一个数组空间，进行相加时，长度不够加的巨大数前面补 0，转换为补码即为 9999，所以两个负数相加时最后一次相加一定产生进位，$carry\_data = 1$。

$$(9999\times 2-|x|-|y|+carry\_data)\%10000=(9999\times 2-|x|-|y|+1)\%10000$$
$$=(10000+9999-|x|-|y|)\%10000$$
$$=9999-|x|-|y|$$

两个负数相加结果为负数，所以得到$9999-|x|-|y|$为补码，转换为源码为$9999-(9999-|x|-|y|)=|x|+|y|$即是运算结果的数值部分，添上符号位完整结果为：$-(|x|+|y|)=-|x|-|y|=x+y$。

把补码来实现的减法看做一个钟表，加多少就是顺时针转多少，减就是逆时针转多少，比如 3-2=1 点，也等于 3+（12-2）=13 点，看成一个圈，向前跑多少也等于向后跑圈周长减去向前跑的距离，在钟表原点 0 点的时候，向前转 12 刻就等于向后转 0 刻。这样子的角度去理解补码。

4.2.2 乘法的实现

①乘法的实现原理和十进制的实现相同，按位相乘即可。两个乘数相乘的积申请的数组长度：两个乘数的长度之和。

②开始两层循环的乘法，相乘得到积是一个把每次相乘结果加起来的过程，就需要定义一个每次生成积的开始位置，作用域在循环体中。积的符号位是两个乘数符号位按位异或，负负得正，正正得正，负正得负。

③调用去除高位无效零的函数将积简化。

4.2.3 除法的实现

除法的一个思路是：通过放大除法接近被除数，再让被除数去减除数，相减的次数乘以放大倍数，加起来就是商。

①比较被除数和除数，如果被除数小于除数直接返回结果 0

②确定除数可以放大的最大倍数，同时从高位遍历两个巨大数，

1、 如果除数的第一个最高位数字大于被除数最高位第一个数字，说明最大倍数是$10^{\Delta length-1}$， $\Delta length$ 为两个巨大数长度之差。

2、 如果除数的第一个最高位数字小于被除数最高位第一个数字，说明

最大倍数是$10^{\Delta length}$。

3、 如果除数的第一个最高位数字等于被除数最高位第一个数字最高位数字继续重复 1，2。

④不断的执行减法的过程，减法的次数乘以当前的倍数累加起来就是商，直到被除数小于除数完成除法。

另外一个解决巨大数除法的思路是通过试商去找最终结果商，给定一个商的最小最大可能，在这个区间里面不断逼近商，问题便是一个一维搜索问题，可选二分查找法，把每次得到的区间中点和除数相乘，再和被除数做差值，直到差值绝对值小于被除数，说明此时区间中点就是最终结果，迭代停止。

$$\begin{cases} f(x) = divsor \ * \ x \\ |\ f(try\,\mathrm{Re}sult) - dividend\ | < divsor \ \&\ \& \ f(try\,\mathrm{Re}sult) < dividend \end{cases}$$

用二分法搜索商完成的除法消耗的时间要大于用减法实现的除数，因为减法实现的除法不过多涉及巨大数的相乘，二分搜索算法，每次取得到区间中点都需要进行一次乘法再和被除数做差值，通过差值的正负和差值的绝对值大小来确定是否停止迭代，时间主要消耗在了试商的相乘部分。而减法中不涉及过多的乘法，只在放大除数处进行一次乘法，如果去掉这一次乘法放大数据，还可以进一步提高性能。在试商的思路上无论用什么搜索算法都没办法避免相乘。

这里是把除法运算的结果表示成一个商和余数的结果，如果需要表示成小数形式，那么就继续放大余数作为被除数，再让被除数重复上述减法去迭代，可以持续提高精度，同时记录一个权重，最终结果表示的时候应是一个科学计数法，何时停止就给定一个精度保留小数点后几位，整数除法运算表示成商和余数更合理，而涉及浮点数的除法运算应该表示为小数更合理。

4.3 巨大数的显示

在屏幕上输出一个巨大数，打印符号位和所有的数据。