

个人作业实验报告

拜重阳 v-chobai

1. 大整数运算

(1) 数据结构

每个大整数用 char 型数组表示，数组的每一个元素表示一位，用 char 是为了节省空间。

(2) 大整数类

类中私有成员如下：

Int start_id;

Int digits;

Char *data;

分别是大整数最高位在数组中的下标，最低位在数组中的下标+1，数组首指针

故 digits-start_id 即为大整数的位数；

公有成员简介如下：

Add

Minus

Multiply 基于手算方法实现大数乘法

Multi_byfft 基于 FFT 实现大数乘法

Multi_divide_conquer 基于分治法实现大数乘法

Divide

Compare 比较两个大整数的大小，基于位数和逐位比较

Print_result 打印一个大数

Clear 清除大数

(3) 算法

a. 加法

基于手算方法，从低位到高位逐位相加，最后得出结果。复杂度 $O(n)$

b. 减法

基于手算方法，从低位到高位逐位相减，最后得出结果。若为小数减大数，则反过来相减，并输出负号。复杂度 $O(n)$

c. 乘法

c1:基于手算方法，从第二个数低位到高位逐个与第一个数相乘，补0并全部相加得到结果。复杂度 $O(n^2)$

c2:分治法

把 a 从中间位数分为 a_1, a_2 ; 同理 b 分为 b_1, b_2 ; 做四次乘法 $a_1b_1, a_2b_2, a_1b_2, a_2b_1$, 给每个结果补适当的 0, 再相加。基于这个思想用分而治之的思想递归实现, 递归的最底层可以自己设计, 比如位数足够小时直接用两个 `int` 相乘, 或用基于手算的乘法。复杂度 $O(n^{1.58})$

c3: FFT

两个 10 进制大数相乘可以看作多项式系数向量卷积, 而卷积的傅里叶变换等于两个数分别傅里叶变换再相乘, 对相乘的结果再作傅里叶逆变换, 则得到了乘积的多项式表示的系数。

快速傅里叶变换 FFT 可以大大提高傅里叶变换的运算速度。该方法最终的复杂度为 $O(n \log n)$ 。

d. 除法

a/b , 对除数做补 0 处理。

每次做除法前将除数补 0 到小于 a 且与 a 最接近为止, 得到 b' 。然后 $a = a - b'$, 直到结果小于 a' 为止, 对这个结果做与 b 一样的补 0 操作, 再反复用 a 减.....直到某个结果小于除数 b 为止, 再对一系列的余数和商做整合, 即得到结果。

举例:

基本的思想是反复做减法, 看看从被除数里最多能减去多少个除数, 商就是多少。一个
一个减显然太慢, 如何减得更快一些呢? 以 7546 除以 23 为例来看一下: 开始商为 0。先减
去 23 的 100 倍, 就是 2300, 发现够减 3 次, 余下 646。于是商的值就增加 300。然后用 646
减去 230, 发现够减 2 次, 余下 186, 于是商的值增加 20。最后用 186 减去 23, 够减 8 次,
因此最终商就是 328。

(4) 一些优化

算法上: 对乘法使用了三种算法, 复杂度递减。Code 中只给了分治法和 fft 法的接口。

实现上:

- a. 由于加减法从低位向高位计算, 又因为数据结构是数组, 故容易想到的方法是算出逆序结果, 再反过来存入结果。为了加快速度, 去掉逆序拷贝花费时间, 故在数据结构中加入 `start_id` 用以记录起始 `id` (不一定是 `id=0`)。而结果的数组总是根据 a 和 b 的位数预先分配最大位数。

- b. 除法的补 0 和分治法的补 0 对于数组是耗时的事。如果长度超过数组长度，则需重新 new 空间再拷贝各位。为了减少这一问题，也预先分配可能最大空间并用 memset 全部置为 0，然后在补 0 时只需改变 digits 的值即可。

(5) 结果

输入\输出

按照要求的格式从 command window 输入输出，注意：+，-，/代表加，减，除，*代表 fft 大数乘法，#代表分治法的大数乘法

Performance

System

Processor:	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz 3.39 GHz
Installed memory (RAM):	8.00 GB (7.89 GB usable)
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

a: 600000digits b:600000digits

+:5ms

-:4ms

*(FFT):8ms

#(divide_conquer):too slow, unknown

/:too slow, unknown

a: 60000digits b:1000digits

+:0ms

-:0 ms

*(FFT) : 254 ms

#(divide_conquer) : 6613ms

/: 14000ms

2.求第 N 个素数

(1)数据结构

略。Bool 型数组 is_prime 存储一定范围内的整数是否为素数;

Int 型数组 prime, index 存储第 index 个素数。

(2)函数：

略

(3)算法

欧拉筛法。时间复杂度 $O(n)$

首先，先明确一个条件，任何合数都能表示成一系列素数的积。

然后利用了每个合数必有一个最小素因子，每个合数仅被它的最小素因子筛去正好一次。所以为线性时间。

代码中体现在：

```
if(i%prime[j]==0)break;
```

prime 数组 中的素数是递增的,当 i 能整除 prime[j] , 那么 $i \cdot \text{prime}[j+1]$ 这个合数肯定被 prime[j] 乘以某个数筛掉。

因为 i 中含有 prime[j], prime[j] 比 prime[j+1] 小。接下去的素数同理。所以不用筛下去了。

在满足 $i \% \text{prime}[j] == 0$ 这个条件之前以及第一次满足改条件时,prime[j]必定是 prime[j]*i 的最小因子。

该算法的优势是复杂度 $O(n)$,快于埃氏筛法的 $O(n \log \log n)$,但缺点是要想实现迅速,必须用额外的 int 数组建立第 i 个素数的索引,这将花费很大的空间。

注意,vs32 位编译环境可能无法 new 大概 20 亿的 bool 连续空间,为保险起见,请把编译环境设为 x64。

(4)优化

根据素数定理,第 n 个素数的上界是 $n \log n + n \log \log n$,故每次筛去合数上界是与 n 有关的,没必要总是筛到最大(第 1 亿个)素数。

(5)结果

Input N from command window

N=1,000,000

Prime:15485863 74ms

N=10,000,000

Prime:179424673 946ms

N=100,000,000

Prime: 2038074743 12310ms