



求 $N!$ 的高精度算法

本文中的算法主要针对 Pascal 语言

这篇文章的内容

- ◆ 你了解高精度吗？
- ◆ 你曾经使用过哪些数据结构？
- ◆ 你仔细思考过如何优化算法吗？

在这里，你将看到怎样成倍提速
求 $N!$ 的高精度算法





关于高精度

- ◆ Pascal中的标准整数类型
- ◆ 高精度算法的基本思想

Pascal 中的标准整数类型

数据类型	值域
Shortint	-128 ~ 127
Byte	0 ~ 255
Integer	-32768 ~ 32767
Word	0 ~ 65535
Longint	-2147483648 ~ 2147483647
Comp	-9.2e18 ~ 9.2e18

Comp 虽然属于实型，实际上是一个 64 位的整数



高精度算法的基本思想

- ◆ Pascal 中的标准整数类型最多只能处理在 $-2^{63} \sim 2^{63}$ 之间的整数。如果要支持更大的整数运算，就需要使用高精度
- ◆ 高精度算法的基本思想，就是将无法直接处理的大整数，分割成若干可以直接处理的小整数段，把对大整数的处理转化为对这些小整数段的处理



数据结构的选择

- ◆ 每个小整数段保留尽量多的位
- ◆ 使用Comp类型
- ◆ 采用二进制表示法



每个小整数段保留尽量多的位

◆ 一个例子：计算两个 15 位数的和

➤ 方法一

- 分为 15 个小整数段，每段都是 1 位数，需要 15 次 1 位数加法

➤ 方法二

- 分为 5 个小整数段，每段都是 3 位数，需要 5 次 3 位数加法

➤ 方法三

- `Comp` 类型可以直接处理 15 位的整数，故 1 次加法就可以了

➤ 比较

- 用 `Integer` 计算 1 位数的加法和 3 位数的加法是一样快的
- 故方法二比方法一效率高
- 虽然对 `Comp` 的操作要比 `Integer` 慢，但加法次数却大大减少
- 实践证明，方法三比方法二更快



使用 Comp 类型

- ◆ 高精度运算中，每个小整数段可以用 Comp 类型表示
- ◆ Comp 有效数位为 $19 \sim 20$ 位
- ◆ 求两个高精度数的和，每个整数段可以保留 17 位
- ◆ 求高精度数与不超过 m 位整数的积，每个整数段可以保留 $18-m$ 位
- ◆ 求两个高精度数的积，每个整数段可以保留 9 位
- ◆ 如果每个小整数段保留 k 位十进制数，实际上可以认为其只保存了 1 位 10^k 进制数，简称为高进制数，称 1 位高进制数为单精度数



采用二进制表示法

- ◆ 采用二进制表示，运算过程中时空效率都会有所提高，但题目一般需要以十进制输出结果，所以还要一个很耗时的进制转换过程。因此这种方法竞赛中一般不采用，也不在本文讨论之列



算法的优化

- ◆ 高精度乘法的复杂度分析
- ◆ 连乘的复杂度分析
- ◆ 设置缓存
- ◆ 分解质因数求阶乘
- ◆ 二分法求乘幂
- ◆ 分解质因数后的调整



高精度乘法的复杂度分析

- ◆ 计算 n 位高进制数与 m 位高进制数的积
 - 需要 $n*m$ 次乘法
 - 积可能是 $n+m-1$ 或 $n+m$ 位高进制数

连乘的复杂度分析 (1)

◆ 一个例子：计算 $5*6*7*8$

➤ 方法一：顺序连乘

- $5*6=30$, $1*1=1$ 次乘法 共 6 次乘法
- $30*7=210$, $2*1=2$ 次乘法
- $210*8=1680$, $3*1=3$ 次乘法

➤ 方法二：非顺序连乘

- $5*6=30$, $1*1=1$ 次乘法 共 6 次乘法
- $7*8=56$, $1*1=1$ 次乘法
- $30*56=1680$, $2*2=4$ 次乘法

特点： n 位数 $*m$ 位数 $=n+m$ 位数

连乘的复杂度分析 (2)

◆ 若“ n 位数 $\times m$ 位数 $= n+m$ 位数”，则 n 个单精度数，无论以何种顺序相乘，乘法次数一定为 $n(n-1)/2$ 次

➤ 证明：

- 设 $F(n)$ 表示乘法次数，则 $F(1)=0$ ，满足题设
- 设 $k < n$ 时， $F(k)=k(k-1)/2$ ，现在计算 $F(n)$
- 设最后一次乘法计算为“ k 位数 $\times (n-k)$ 位数”，则
- $F(n)=F(k)+F(n-k)+k(n-k)=n(n-1)/2$ （与 k 的选择无关）



设置缓存 (1)

◆ 一个例子：计算 $9*8*3*2$

➤ 方法一：顺序连乘

- $9*8=72$, $1*1=1$ 次乘法
 - $72*3=216$, $2*1=2$ 次乘法
 - $216*2=432$, $3*1=3$ 次乘法
- 共 6 次乘法

➤ 方法二：非顺序连乘

- $9*8=72$, $1*1=1$ 次乘法
 - $3*2=6$, $1*1=1$ 次乘法
 - $72*6=432$, $2*1=2$ 次乘法
- 共 4 次乘法

特点： n 位数 * m 位数可能是 $n+m-1$ 位数

设置缓存 (2)

◆ 考虑 $k+t$ 个单精度数相乘 $a_1 * a_2 * \dots * a_k * a_{k+1} * \dots * a_{k+t}$

- 设 $a_1 * a_2 * \dots * a_k$ 结果为 m 位高进制数 (假设已经算出)
- $a_{k+1} * \dots * a_{k+t}$ 结果为 1 位高进制数
- 若顺序相乘, 需要 t 次“ m 位数 * 1 位数”, 共 mt 次乘法
- 可以先计算 $a_{k+1} * \dots * a_{k+t}$, 再一起乘, 只需要 $m+t$ 次乘法

在设置了缓存的前提下, 计算 m 个单精度数的积, 如果结果为 n 位数, 则乘法次数约为 $n(n-1)/2$ 次, 与 m 关系不大

- 设 $S = a_1 a_2 \dots a_m$, S 是 n 位高进制数
- 可以把乘法的过程近似看做, 先将这 m 个数分为 n 组, 每组的积仍然是一个单精度数, 最后计算后面这 n 个数的积。时间主要集中在求最后 n 个数的积上, 这时基本上满足“ n 位数 * m 位数 = $n+m$ 位数”, 故乘法次数可近似的看做 $n(n-1)/2$ 次



设置缓存 (3)

◆ 缓存的大小

- 设所选标准数据类型最大可以直接处理 t 位十进制数
- 设缓存为 k 位十进制数，每个小整数段保存 $t-k$ 位十进制数
- 设最后结果为 n 位十进制数，则乘法次数约为
- $k/(n-k) \sum_{i=1..n/k} i = (n+k)n/(2k(t-k))$ ，其中 k 远小于 n
- 要乘法次数最少，只需 $k(t-k)$ 最大，这时 $k=t/2$
- 因此，缓存的大小与每个小整数段大小一样时，效率最高
- 故在一般的连乘运算中，可以用 **Comp** 作为基本整数类型，每个小整数段为 9 位十进制数，缓存也是 9 位十进制数



分解质因数求阶乘

◆ 例： $10! = 2^8 * 3^4 * 5^2 * 7$

- $n!$ 分解质因数的复杂度远小于 $n \log n$ ，可以忽略不计
- 与普通算法相比，分解质因数后，虽然因子个数 m 变多了，但结果的位数 n 没有变，只要使用了缓存，乘法次数还是约为 $n(n-1)/2$ 次
- 因此，分解质因数不会变慢（这也可以通过实践来说明）
- 分解质因数之后，出现了大量求乘幂的运算，我们可以优化求乘幂的算法。这样，分解质因数的好处就体现出来了



二分法求乘幂

◆ 二分法求乘幂，即：

- $a^{2n+1} = a^{2n} * a$
- $a^{2n} = (a^n)^2$
- 其中， a 是单精度数

◆ 复杂度分析

- 假定 n 位数与 m 位数的积是 $n+m$ 位数
- 设用二分法计算 a^n 需要 $F(n)$ 次乘法
- $F(2n) = F(n) + n^2$ ， $F(1) = 0$
- 设 $n = 2^k$ ，则有 $F(n) = F(2^k) = \sum_{i=0..k-1} 4^i = (4^k - 1) / 3 = (n^2 - 1) / 3$

◆ 与连乘的比较

- 用连乘需要 $n(n-1)/2$ 次乘法，二分法需要 $(n^2-1)/3$
- 连乘比二分法耗时仅多 50%
- 采用二分法，复杂度没有从 n^2 降到 $n \log n$

二分法求乘幂之优化平方算法

◆ 怎样优化

➤ $(a+b)^2 = a^2 + 2ab + b^2$

➤ 例: $12345^2 = \underline{123}^2 * 10000 + \underline{45}^2 + 2 * \underline{123} * \underline{45} * 100$

➤ 把一个 n 位数分为一个 t 位数和一个 $n-t$ 位数, 再求平方

◆ 怎样分

➤ 设求 n 位数的平方需要 $F(n)$ 次乘法

➤ $F(n) = F(t) + F(n-t) + t(n-t)$, $F(1) = 1$

➤ 用数学归纳法, 可证明 $F(n)$ 恒等于 $n(n+1)/2$

➤ 所以, 无论怎样分, 效率都是一样

➤ 将 n 位数分为一个 1 位数和 $n-1$ 位数, 这样处理比较方便


二分法求乘幂之复杂度分析

◆ 复杂度分析

- 前面已经求出 $F(n)=n(n+1)/2$ ，下面换一个角度来处理
- $S^2=(\sum_{(0 \leq i < n)} a_i 10^i)^2 = \sum_{(0 \leq i < n)} a_i^2 10^{2i} + 2 \sum_{(0 \leq i < j < n)} a_i a_j 10^{i+j}$
- 一共做了 $n+C(n,2)=n(n+1)/2$ 次乘法运算
- 普通算法需要 n^2 次乘法，比改进后的慢 1 倍

◆ 改进求乘幂的算法

- 如果在用改进后的方法求平方，则用二分法求乘幂，需要 $(n+4)(n-1)/6$ 次乘法，约是连乘算法 $n(n-1)/2$ 的三分之一



分解质因数后的调整 (1)

◆ 为什么要调整

- 计算 $S=2^{11}3^{10}$ ，可以先算 2^{11} ，再算 3^{10} ，最后求它们的积
- 也可以根据 $S=2^{11}3^{10}=6^{10}*2$ ，先算 6^{10} ，再乘以 2 即可
- 两种算法的效率是不同的

分解质因数后的调整 (2)

◆ 什么时候调整

- 计算 $S = a^{x+k}b^x = (ab)^xa^k$
- 当 $k < x \log_a b$ 时, 采用 $(ab)^xa^k$ 比较好, 否则采用 $a^{x+k}b^x$ 更快
- 证明:
 - 可以先计算两种算法的乘法次数, 再解不等式, 就可以得到结论
 - 也可以换一个角度来分析。其实, 两种算法主要差别在最后一步求积上。由于两种方法, 积的位数都是一样的, 所以两个因数的差越大, 乘法次数就越小
 - \therefore 当 $a^xb^x - a^k > a^{x+k} - b^x$ 时, 选用 $(ab)^xa^k$, 反之, 则采用 $a^{x+k}b^x$ 。
 - $\therefore a^xb^x - a^k > a^{x+k} - b^x$
 - $\therefore (b^x - a^k)(a^x + 1) > 0$
 - $\therefore b^x > a^k$
 - 这时 $k < x \log_a b$



总结

◆ 内容小结

- 用 `Comp` 作为每个小整数段的基本整数类型
- 采用二进制优化算法
- 高精度连乘时缓存和缓存的设置
- 改进的求平方算法
- 二分法求乘幂
- 分解质因数法求阶乘以及分解质因数后的调整

◆ 应用

- 高精度求乘幂（平方）
- 高精度求连乘（阶乘）
- 高精度求排列组合



结束语

求 $N!$ 的高精度算法本身并不难，但我们仍然可以从多种角度对它进行优化。

其实，很多经典算法都有优化的余地。我们自己编写的一些程序也不例外。只要用心去优化，说不准你就想出更好的算法来了。

也许你认为本文中的优化毫无价值。确实是这样，竞赛中对高精度的要求很低，根本不需要优化。而我以高精度算法为例，不过想谈谈如何优化一个算法。我想说明的只有一点：算法是可以优化的。