理性愉悦：高精度数值计算

倪泽堃

Feb 2011

Part I

大整数的四则运算与开平方

基础算法

以下算法如果不会请回去上NOIP 基础课:

- n 位大整数相加减: $O(n)$
- n 位大整数与小整数的乘除: $O(n)$
- 暴力n 位大整数相乘除: 上一行的直接推广, $O(n^2)$

FFT: 引入

给定一个多项式函数

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N = \sum_m a_m x^m,$$

如何求$S_0 = \sum_m a_{2m}$ 与$S_1 = \sum_m a_{2m+1}$?

把$f(x)$ 写成等价形式$f(x) = \sum_m a_{2m} x^{2m} + \sum_m a_{2m+1} x^{2m+1}$,
那么选取1 与-1, 得

$$f(1) = \sum_m a_{2m} + \sum_m a_{2m+1} = S_0 + S_1$$

$$f(-1) = \sum_m a_{2m} - \sum_m a_{2m+1} = S_0 - S_1$$

通过解方程易得$S_0 = \frac{1}{2}[f(1) + f(-1)]$ 与$S_1 = \frac{1}{2}[f(1) - f(-1)]$.

FFT: 引入

给定一个多项式函数

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N = \sum_m a_m x^m,$$

如何求$S_0 = \sum_m a_{2m}$ 与$S_1 = \sum_m a_{2m+1}$?

把$f(x)$ 写成等价形式$f(x) = \sum_m a_{2m} x^{2m} + \sum_m a_{2m+1} x^{2m+1}$,
那么选取1 与-1, 得

$$f(1) = \sum_m a_{2m} + \sum_m a_{2m+1} = S_0 + S_1$$

$$f(-1) = \sum_m a_{2m} - \sum_m a_{2m+1} = S_0 - S_1$$

通过解方程易得$S_0 = \frac{1}{2}[f(1) + f(-1)]$ 与$S_1 = \frac{1}{2}[f(1) - f(-1)]$.

FFT: 引入

现在问题变难一些：如何求$S_0 = \sum_m a_{4m}$，$S_1 = \sum_m a_{4m+1}$，$S_2 = \sum_m a_{4m+2}$ 与$S_3 = \sum_m a_{4m+3}$?

把$f(x)$ 写成$f(x) = \sum_m a_{4m}x^{4m} + \sum_m a_{4m+1}x^{4m+1} + \sum_m a_{4m+2}x^{4m+2} + \sum_m a_{4m+3}x^{4m+3}$，选取$x^4 = 1$ 的四个根1, i, -1, $-i$, 得

$$f(1) = S_0 + S_1 + S_2 + S_3, f(i) = S_0 + iS_1 - S_2 - iS_3$$
$$f(-1) = S_0 - S_1 + S_2 - S_3, f(-i) = S_0 - iS_1 - S_2 + iS_3$$

在这里解出各S_k. 但是不必硬解方程，事实上，只要在四个等式两端分别乘一个系数使得S_k 前的系数变成1，再将四式相加，那么最终其它项都被消去，只剩$4S_k$.

FFT: 引入

现在问题变难一些：如何求$S_0=\sum_m a_{4m}$，$S_1=\sum_m a_{4m+1}$，$S_2=\sum_m a_{4m+2}$与$S_3=\sum_m a_{4m+3}$?

把$f(x)$写成$f(x)=\sum_m a_{4m}x^{4m}+\sum_m a_{4m+1}x^{4m+1}+\sum_m a_{4m+2}x^{4m+2}+\sum_m a_{4m+3}x^{4m+3}$，选取$x^4=1$的四个根1, i, -1, $-i$，得

$$f(1)=S_0+S_1+S_2+S_3, f(i)=S_0+iS_1-S_2-iS_3$$
$$f(-1)=S_0-S_1+S_2-S_3, f(-i)=S_0-iS_1-S_2+iS_3$$

在这里解出各S_k. 但是不必硬解方程，事实上，只要在四个等式两端分别乘一个系数使得S_k前的系数变成1，再将四式相加，那么最终其它项都被消去，只剩$4S_k$.

FFT: 引入

例: 求S_1. 将四式两边分别乘1, $-i$, -1, i, 得

$$\begin{aligned}
f(1) &= +S_0 && +S_1 && +S_2 && +S_3 \\
-if(i) &= -iS_0 && +S_1 && +iS_2 && -S_3 \\
-f(-1) &= -S_0 && +S_1 && -S_2 && +S_3 \\
if(-i) &= +iS_0 && +S_1 && -iS_2 && -S_3 \\
&+)+0 && +4S_1 && +0 && +0
\end{aligned}$$

因此$S_1 = \frac{1}{4}[f(1) - if(i) - f(-1) + if(-i)]$.

类似可以得出$S_0 = \frac{1}{4}[f(1) + f(i) + f(-1) + f(-i)]$,
$S_2 = \frac{1}{4}[f(1) - f(i) + f(-1) - f(-i)]$,
$S_3 = \frac{1}{4}[f(1) + if(i) - f(-1) - if(-i)]$.

FFT: 引入

例: 求S_1. 将四式两边分别乘1, $-i$, -1, i, 得

$$
\begin{array}{lllll}
f(1) & =+S_0 & +S_1 & +S_2 & +S_3 \\
-if(i) & =-iS_0 & +S_1 & +iS_2 & -S_3 \\
-f(-1) & =-S_0 & +S_1 & -S_2 & +S_3 \\
if(-i) & =+iS_0 & +S_1 & -iS_2 & -S_3 \\
 & +)+0 & +4S_1 & +0 & +0
\end{array}
$$

因此$S_1 = \frac{1}{4}[f(1) - if(i) - f(-1) + if(-i)]$.

类似可以得出$S_0 = \frac{1}{4}[f(1) + f(i) + f(-1) + f(-i)]$,
$S_2 = \frac{1}{4}[f(1) - f(i) + f(-1) - f(-i)]$,
$S_3 = \frac{1}{4}[f(1) + if(i) - f(-1) - if(-i)]$.

FFT: 引入

下面将问题一般化. 首先需要一些复数知识.

De Moivre 定理: $(\cos x + i \sin x)^n = \cos(nx) + i \sin(nx)$

$x^n = 1$ 有n 个根$\epsilon^0, \epsilon^1, \ldots, \epsilon^{n-1}$, 其中$\epsilon = \cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n}$, 这样

$$\epsilon^m = \cos\frac{2m\pi}{n} + i \sin\frac{2m\pi}{n}$$

很容易发现这n 个根在复平面上等分单位圆, 如图($n = 5$, from Wikipedia):

FFT: 引入

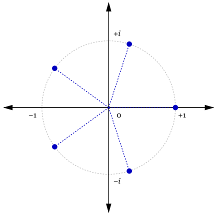
下面将问题一般化. 首先需要一些复数知识.

De Moivre 定理: $(\cos x + i\sin x)^n = \cos(nx) + i\sin(nx)$

$x^n = 1$ 有n 个根$\epsilon^0, \epsilon^1, \dots, \epsilon^{n-1}$, 其中$\epsilon = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$, 这样

$$\epsilon^m = \cos\frac{2m\pi}{n} + i\sin\frac{2m\pi}{n}$$

很容易发现这n 个根在复平面上等分单位圆, 如图($n = 5$, from Wikipedia):



FFT: 引入

进入问题. 假如要求$S_0 = \sum_m a_{nm}$, $S_1 = \sum_m a_{nm+1}$, ..., $S_{n-1} = \sum_m a_{nm+(n-1)}$, 那么将$f(x)$ 写成

$$f(x) = \sum_{k=0}^{n-1} \sum_m a_{nm+k} x^{nm+k}$$

将$x^n = 1$ 的n 个根$1, \epsilon, \ldots, \epsilon^{n-1}$ 代入, 那么$f(\epsilon^m) = \sum_{k=0}^{n-1} S_k \epsilon^{mk}$. 由刚刚的经验, 欲求$S_k$, 将每个等式中$S_k$ 前系数变成1, 即对所有的m, 将$f(\epsilon^m)$ 所在等式两端乘ϵ^{-mk}, 这样将n 个等式相加即得$nS_k = \sum_{m=0}^{n-1} \epsilon^{-mk} f(\epsilon^m)$, 即

$$S_k = \frac{1}{n} \sum_{m=0}^{n-1} \epsilon^{-mk} f(\epsilon^m).$$

FFT: 引入

进入问题. 假如要求$S_0 = \sum_m a_{nm}$, $S_1 = \sum_m a_{nm+1}$, ..., $S_{n-1} = \sum_m a_{nm+(n-1)}$, 那么将$f(x)$ 写成

$$f(x) = \sum_{k=0}^{n-1} \sum_m a_{nm+k} x^{nm+k}$$

将$x^n = 1$ 的n 个根$1, \epsilon, \ldots, \epsilon^{n-1}$ 代入, 那么$f(\epsilon^m) = \sum_{k=0}^{n-1} S_k \epsilon^{mk}$. 由刚刚的经验, 欲求$S_k$, 将每个等式中$S_k$ 前系数变成1, 即对所有的m, 将$f(\epsilon^m)$ 所在等式两端乘ϵ^{-mk}, 这样将n 个等式相加即得$nS_k = \sum_{m=0}^{n-1} \epsilon^{-mk} f(\epsilon^m)$, 即

$$S_k = \frac{1}{n} \sum_{m=0}^{n-1} \epsilon^{-mk} f(\epsilon^m).$$

FFT: 定义

在此基础上理解FFT 就不难了.

定义DFT 是如下一种变换：对于数列$a_0, a_1, \ldots, a_{n-1}$，令其生成多项式为$f_a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$，那么DFT 就是将$x^n = 1$ 的根$1, \epsilon, \ldots, \epsilon^{n-1}$ 依次代入$f_a(x)$ 所得的值，即

$$a'_k = f_a(\epsilon^k) = \sum_{m=0}^{n-1} a_m \epsilon^{mk}.$$

而定义IDFT 为其逆变换，即通过$a'_0, a'_1, \ldots, a'_{n-1}$ 反求回原数列．在这里，由于$S_k = \sum_m a_{nm+k} = a_k$，直接应用之前的方法得

$$a_k = \frac{1}{n}\sum_{m=0}^{n-1} a'_m \epsilon^{-mk}.$$

FFT，就是快速求解DFT 和IDFT 的一种算法.

FFT: 定义

在此基础上理解FFT 就不难了.

定义DFT 是如下一种变换：对于数列$a_0, a_1, \ldots, a_{n-1}$，令其生成多项式为$f_a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$，那么DFT 就是将$x^n = 1$ 的根$1, \epsilon, \ldots, \epsilon^{n-1}$ 依次代入$f_a(x)$ 所得的值，即

$$a'_k = f_a(\epsilon^k) = \sum_{m=0}^{n-1} a_m \epsilon^{mk}.$$

而定义IDFT 为其逆变换，即通过$a'_0, a'_1, \ldots, a'_{n-1}$ 反求回原数列. 在这里，由于$S_k = \sum_m a_{nm+k} = a_k$，直接应用之前的方法得

$$a_k = \frac{1}{n} \sum_{m=0}^{n-1} a'_m \epsilon^{-mk}.$$

FFT，就是快速求解DFT 和IDFT 的一种算法.

FFT: 定义

在此基础上理解FFT 就不难了.

定义DFT 是如下一种变换：对于数列$a_0, a_1, \ldots, a_{n-1}$，令其生成多项式为$f_a(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$，那么DFT 就是将$x^n = 1$ 的根$1, \epsilon, \ldots, \epsilon^{n-1}$ 依次代入$f_a(x)$ 所得的值，即

$$a'_k = f_a(\epsilon^k) = \sum_{m=0}^{n-1} a_m \epsilon^{mk}.$$

而定义IDFT 为其逆变换，即通过$a'_0, a'_1, \ldots, a'_{n-1}$ 反求回原数列. 在这里，由于$S_k = \sum_m a_{nm+k} = a_k$，直接应用之前的方法得

$$a_k = \frac{1}{n} \sum_{m=0}^{n-1} a'_m \epsilon^{-mk}.$$

FFT，就是快速求解DFT 和IDFT 的一种算法.

FFT: 定义

在此基础上理解FFT 就不难了.

定义DFT 是如下一种变换：对于数列$a_0, a_1, \ldots, a_{n-1}$，令其生成多项式为$f_a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$，那么DFT 就是将$x^n = 1$ 的根$1, \epsilon, \ldots, \epsilon^{n-1}$ 依次代入$f_a(x)$ 所得的值，即

$$a'_k = f_a(\epsilon^k) = \sum_{m=0}^{n-1} a_m \epsilon^{mk}.$$

而定义IDFT 为其逆变换，即通过$a'_0, a'_1, \ldots, a'_{n-1}$ 反求回原数列. 在这里，由于$S_k = \sum_m a_{nm+k} = a_k$，直接应用之前的方法得

$$a_k = \frac{1}{n} \sum_{m=0}^{n-1} a'_m \epsilon^{-mk}.$$

FFT，就是快速求解DFT 和IDFT 的一种算法.

FFT: 实现与意义

这样一变换的优越性在于，求解多项式乘法的时候方便了.

譬如要求解一个n 次多项式$f_a(x)$ 与一个m 次多项式$f_b(x)$ 相乘而得的多项式$f_c(x)$，由于结果多项式是$n+m$ 次的，那么先对$a_0, a_1, \ldots, a_n, 0, \ldots, 0$ 做长度不小于$n+m+1$ 的DFT，再对$b_0, b_1, \ldots, b_m, 0, \ldots, 0$ 做同样长度的DFT. 此时所求的$f_c(x)$ 对应的$c_0, c_1, \ldots, c_{n+m}, 0, \ldots, 0$ 经该长度的DFT 变换后的序列已经可以在$O(n+m)$ 的时间内求出了:

$$c'_k = f_c(\epsilon^k) = [f_a(x)f_b(x)]|_{x=\epsilon^k} = f_a(\epsilon^k)f_b(\epsilon^k) = a'_k b'_k$$

这样再对这样的序列进行IDFT 变换后即可得$f_c(x)$ 的每个系数.

FFT: 实现与意义

这样一变换的优越性在于，求解多项式乘法的时候方便了.

譬如要求解一个n 次多项式$f_a(x)$ 与一个m 次多项式$f_b(x)$ 相乘而得的多项式$f_c(x)$，由于结果多项式是$n+m$ 次的，那么先对$a_0, a_1, \ldots, a_n, 0, \ldots, 0$ 做长度不小于$n+m+1$ 的DFT，再对$b_0, b_1, \ldots, b_m, 0, \ldots, 0$ 做同样长度的DFT. 此时所求的$f_c(x)$ 对应的$c_0, c_1, \ldots, c_{n+m}, 0, \ldots, 0$ 经该长度的DFT 变换后的序列已经可以在$O(n+m)$ 的时间内求出了:

$$c'_k = f_c(\epsilon^k) = [f_a(x)f_b(x)]|_{x=\epsilon^k} = f_a(\epsilon^k)f_b(\epsilon^k) = a'_k b'_k$$

这样再对这样的序列进行IDFT 变换后即可得$f_c(x)$ 的每个系数.

FFT: 实现与意义

且慢! 通过DFT 和IDFT 的定义容易发现, 如果暴力进行长度为n 的(I)DFT, 那么时间复杂度为$O(n^2)$. 显然, 如果这么按刚刚的步骤做, n 次多项式乘法的时间复杂度仍然是$O(n^2)$, 没有改观. 于是FFT 就要登场了.

FFT 有很多种, 这里主要介绍一种较易实现的, 适用于长度为2 的幂(可以推广到所有可表为若干小质数的积的数) 的算法. 首先DFT 和IDFT 形式极为相似, 只是ϵ 上的指数互为相反数, 以及IDFT 多了个$\frac{1}{n}$, 因此两者可以适用同一种优化方法. 不妨分析DFT 的过程.

FFT: 实现与意义

且慢！通过DFT 和IDFT 的定义容易发现，如果暴力进行长度为n 的(I)DFT，那么时间复杂度为$O(n^2)$。显然，如果这么按刚刚的步骤做，n 次多项式乘法的时间复杂度仍然是$O(n^2)$，没有改观。于是FFT 就要登场了。

FFT 有很多种，这里主要介绍一种较易实现的，适用于长度为2 的幂(可以推广到所有可表为若干小质数的积的数) 的算法。首先DFT 和IDFT 形式极为相似，只是ϵ 上的指数互为相反数，以及IDFT 多了个$\frac{1}{n}$，因此两者可以适用同一种优化方法。不妨分析DFT 的过程。

FFT: 实现与意义

假设做长度为n 的DFT，当$k < n/2$ 时，对a_m 分奇偶

$$
\begin{aligned}
a'_k &= \sum_{m=0}^{n-1} a_m \epsilon^{mk} = \sum_{m=0}^{n/2-1} a_{2m} \epsilon^{2mk} + \sum_{m=0}^{n/2-1} a_{2m+1} \epsilon^{(2m+1)k} \\
&= \sum_{m=0}^{n/2-1} a_{2m} \epsilon^{2mk} + \epsilon^k \sum_{m=0}^{n/2-1} a_{2m+1} \epsilon^{2mk} \\
&= \sum_{m=0}^{n/2-1} a_{2m} \epsilon_{n/2}^{mk} + \epsilon^k \sum_{m=0}^{n/2-1} a_{2m+1} \epsilon_{n/2}^{mk} = a'_{\text{偶}_k} + \epsilon^k a'_{\text{奇}_k}
\end{aligned}
$$

其中，$\epsilon_{n/2}$ 表示次数为$n/2$ 时对应的ϵ，$\epsilon^2 = \epsilon_{n/2}$ 是因为

$$
\epsilon_{n/2} = \cos\frac{2\pi}{\frac{n}{2}} + i\sin\frac{2\pi}{\frac{n}{2}} = \cos\frac{4\pi}{n} + i\sin\frac{4\pi}{n} = \epsilon^2
$$

FFT: 实现与意义

当 $k \geq n/2$ 时

$$
\begin{aligned}
a'_k &= \sum_{m=0}^{n/2-1} a_{2m}\epsilon_{n/2}^{mk} + \epsilon^k \sum_{m=0}^{n/2-1} a_{2m+1}\epsilon_{n/2}^{mk} \\
&= \sum_{m=0}^{n/2-1} a_{2m}\epsilon_{n/2}^{m(k-n/2)}\epsilon_{n/2}^{mn/2} + \epsilon^k \sum_{m=0}^{n/2-1} a_{2m+1}\epsilon_{n/2}^{m(k-n/2)}\epsilon_{n/2}^{mn/2} \\
&= \sum_{m=0}^{n/2-1} a_{2m}\epsilon_{n/2}^{m(k-n/2)} + \epsilon^k \sum_{m=0}^{n/2-1} a_{2m+1}\epsilon_{n/2}^{m(k-n/2)} \\
&= a'_{\text{偶}_{k-n/2}} + \epsilon^k a'_{\text{奇}_{k-n/2}}
\end{aligned}
$$

FFT: 实现与意义

这样一个长度为n 的DFT 转化为长度为$n/2$ 的, 其下标为偶数的子序列与其下标为奇数的子序列的DFT 问题, 求得两子序列的DFT 后, 只要$O(n)$ 时间就能求得原序列的DFT.

但是$n/2$ 仍然为2 的幂, 故可一直分治下去直到$n = 1$, 此时直接有$a_0' = a_0$. 至此这个算法的运作过程描述完毕.

时间复杂度分析: 设用该算法做长度为n 的DFT 时时间复杂度为$T(n)$, 那么$T(n) = 2T(n/2) + O(n)$, 解得$T(n) = O(n \log n)$.

因此这种FFT 是一种可在$O(n \log n)$ 时间内求解序列DFT 的优秀算法. 这样, n 次多项式乘法就可以在$O(n \log n)$ 时间内完成.

FFT: 实现与意义

这样一个长度为n 的DFT 转化为长度为$n/2$ 的，其下标为偶数的子序列与其下标为奇数的子序列的DFT 问题，求得两子序列的DFT 后，只要$O(n)$ 时间就能求得原序列的DFT.

但是$n/2$ 仍然为2 的幂，故可一直分治下去直到$n = 1$，此时直接有$a'_0 = a_0$. 至此这个算法的运作过程描述完毕.

时间复杂度分析: 设用该算法做长度为n 的DFT 时时间复杂度为$T(n)$，那么$T(n) = 2T(n/2) + O(n)$，解得$T(n) = O(n \log n)$.

因此这种FFT 是一种可在$O(n \log n)$ 时间内求解序列DFT 的优秀算法. 这样，n 次多项式乘法就可以在$O(n \log n)$ 时间内完成.

FFT: 实现与意义

但是这和大整数的运算有何关系呢?

假设某n 位大整数A 的十进制表示为$\overline{a_{n-1}a_{n-2}\cdots a_0}$ $(0 \leq a_k < 10)$ (负数考虑其绝对值), 那么令$f_a(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$ 后立即就有$A = f_a(10)$. 于是求两个大整数A 和B 的积就转化为求解它们对应的生成多项式$f_a(x)$ 与$f_b(x)$ 的积, 然后再将10 代入. 当然求得的多项式每一项未必满足< 10, 有一个进位的问题, 不过这显然可以在$O(n)$ 时间内解决.

至此, 两个n 位大整数的乘积就可以在$O(n \log n)$ 时间内完成.

FFT: 实现与意义

但是这和大整数的运算有何关系呢?

假设某n 位大整数A 的十进制表示为$\overline{a_{n-1}a_{n-2}\cdots a_0}$ $(0 \le a_k < 10)$ (负数考虑其绝对值), 那么令$f_a(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$ 后立即就有$A = f_a(10)$. 于是求两个大整数A 和B 的积就转化为求解它们对应的生成多项式$f_a(x)$ 与$f_b(x)$ 的积, 然后再将10 代入. 当然求得的多项式每一项未必满足< 10, 有一个进位的问题, 不过这显然可以在$O(n)$ 时间内解决.

至此, 两个n 位大整数的乘积就可以在$O(n \log n)$ 时间内完成.

FFT: 实现与意义

实际实现上，一般需要预处理$x^n = 1$ 的n 个根$1, \epsilon, \ldots, \epsilon^{n-1}$ $(\epsilon^{-k} = \epsilon^{n-k})$. 另外，如果直接另开临时数组存储序列的奇偶子序列，内存使用会达到$O(n \log n)$，需要优化，不过这比较容易，具体就不讲了. 压位优化在FFT 中仍然有效，但要注意整个FFT 算法都是用实数，最后取整得到乘积多项式时有精度误差问题.

最后，这个算法有非递归的实现，始终在一个数组上操作，速度较快. 我用在某处得到的一个非递归实现卡线AC SPOJ VFMUL，代码长度2KB+.

牛顿迭代法

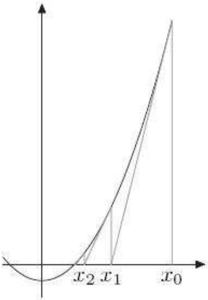
下面为了探求两个大整数相除的快速算法，需要了解牛顿迭代法的知识.

对于给定的一个函数$f(x)$，要求一个$f(x)=0$ 的根，可选择一个邻近这个根的点作为初始点x_0，然后求$f(x)$ 于$(x_0, f(x_0))$ 处切线与x 轴的交点，此交点一般将更加接近根，如图所示(from Wikipedia):

牛顿迭代法

下面为了探求两个大整数相除的快速算法，需要了解牛顿迭代法的知识.

对于给定的一个函数$f(x)$，要求一个$f(x)=0$ 的根，可选择一个邻近这个根的点作为初始点x_0，然后求$f(x)$ 于$(x_0, f(x_0))$ 处切线与x 轴的交点，此交点一般将更加接近根，如图所示(from Wikipedia):



牛顿迭代法

根据导数的知识，可得迭代式如下：

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

每次以x 代x_0 后继续迭代，一般最终x_0 将趋于所求的根，并且更进一步的分析表明，在满足某些条件的情况下，迭代是二阶收敛的，也就是每迭代一次，有效位数几乎增加一倍．这是很诱人的，因为这样达到n 位精度只需要$O(\log n)$ 的迭代次数．

这样，就给快速求解两大整数相除以及大整数开平方提供了一个思路．

牛顿迭代法: 应用

如果实现了高精度实数运算, 那么可以直接运用牛顿迭代法求解大整数A 的倒数. 假如考虑方程$Ax-1=0$, 那么迭代式为

$$x = x_0 - \frac{Ax_0 - 1}{A} = \frac{1}{A}$$

很显然, 它并没有解决问题. 但是如果考虑这样的方程$\frac{1}{x}-A=0$, 那么迭代式为

$$x = x_0 - \frac{\frac{1}{x_0} - A}{-\frac{1}{x_0^2}} = 2x_0 - Ax_0^2$$

可以看出, 这里只有减法和乘法, 因此估计一个较好的初值(可取一个比$\frac{1}{A}$ 稍小的数), 就能迭代得到$\frac{1}{A}$. 得到$\frac{1}{A}$ 后, 再乘B, 取整后调整, 即可得到$\lfloor B/A \rfloor$.

但是, 这里用到了不希望出现的高精度实数运算, 那么需要寻找一种替代方法. 下面提供一种实现的思路.

牛顿迭代法: 应用

如果实现了高精度实数运算, 那么可以直接运用牛顿迭代法求解大整数A 的倒数. 假如考虑方程$Ax-1=0$, 那么迭代式为

$$x = x_0 - \frac{Ax_0 - 1}{A} = \frac{1}{A}$$

很显然, 它并没有解决问题. 但是如果考虑这样的方程$\frac{1}{x} - A = 0$, 那么迭代式为

$$x = x_0 - \frac{\frac{1}{x_0} - A}{-\frac{1}{x_0^2}} = 2x_0 - Ax_0^2$$

可以看出, 这里只有减法和乘法, 因此估计一个较好的初值(可取一个比$\frac{1}{A}$ 稍小的数), 就能迭代得到$\frac{1}{A}$. 得到$\frac{1}{A}$ 后, 再乘B, 取整后调整, 即可得到$\lfloor B/A \rfloor$.

但是, 这里用到了不希望出现的高精度实数运算, 那么需要寻找一种替代方法. 下面提供一种实现的思路.

牛顿迭代法: 应用

假如A 有n 位, 那么我们希望求得$A' = \lfloor 10^{2n}/A \rfloor$, 然后计算$\lfloor BA'/10^{2n} \rfloor$ 后调整, 即得$\lfloor B/A \rfloor$. 不过这里面有个A' 的舍入误差问题, 为了保证最后调整次数是$O(1)$ 的, 那么A 相对B 位数不能太少, 假设B 有m 位, 我们需要使得$m \leq 2n$, 这样可以证明最后调整的时候, 误差不超过10. 这很简单, 假如$m > 2n$, 两者同时左移若干位即可.

下面的问题就是求解$\lfloor 10^{2n}/A \rfloor$. 设前一次迭代求解的是$A'_k = \lfloor 10^{2k}/A_k \rfloor$ (其中A_k 是A 的前k 位组成的数), 那么这一次的迭代式为

$$A'^*/10^{2n} = 2A'_k/10^{2k}/10^{n-k} - A(A'_k/10^{2k}/10^{n-k})^2$$

也就是

$$A'^* = 2A'_k \cdot 10^{n-k} - AA'^2_k/10^{2k} \approx 2A'_k \cdot 10^{n-k} - \lfloor AA'^2_k/10^{2k} \rfloor$$

牛顿迭代法: 应用

假如A 有n 位, 那么我们希望求得$A' = \lfloor 10^{2n}/A \rfloor$, 然后计算$\lfloor BA'/10^{2n} \rfloor$ 后调整, 即得$\lfloor B/A \rfloor$. 不过这里面有个A' 的舍入误差问题, 为了保证最后调整次数是$O(1)$ 的, 那么A 相对B 位数不能太少, 假设B 有m 位, 我们需要使得$m \le 2n$, 这样可以证明最后调整的时候, 误差不超过10. 这很简单, 假如$m > 2n$, 两者同时左移若干位即可.

下面的问题就是求解$\lfloor 10^{2n}/A \rfloor$. 设前一次迭代求解的是$A'_k = \lfloor 10^{2k}/A_k \rfloor$ (其中A_k 是A 的前k 位组成的数), 那么这一次的迭代式为

$$A'^{*}/10^{2n} = 2A'_k/10^{2k}/10^{n-k} - A(A'_k/10^{2k}/10^{n-k})^2$$

也就是

$$A'^{*} = 2A'_k \cdot 10^{n-k} - AA'^2_k/10^{2k} \approx 2A'_k \cdot 10^{n-k} - \lfloor AA'^2_k/10^{2k} \rfloor$$

牛顿迭代法: 应用

求得A'^* 当然还没完, 这只是迭代的结果, 并不是$\lfloor 10^{2n}/A \rfloor$ 的准确值. 误差分析表明, 当$k > n/2$ 时, 误差不超过100 (可取$k = \lfloor (n+2)/2 \rfloor$), 于是还要在求得余数$10^{2n} - AA'^*$ 后对A'^* 进行次数为$O(1)$ 的调整(为了降低常数, 可以二分误差). 这样就迭代地求解出了$\lfloor 10^{2n}/A \rfloor$. 边界条件是$n \le 2$, 此时$A$ 在小整数范围内, 可以直接求解结果.

求解出$\lfloor 10^{2n}/A \rfloor$ 后与B 相乘, 再在计算余数后进行和上面类似的调整, 即可求解出$\lfloor B/A \rfloor$.

时间复杂度分析: 假设计算$\lfloor 10^{2n}/A \rfloor$ 时复杂度为$T(n)$, 那么$T(n) = T(k) + O(n \log n) \approx T(n/2) + O(n \log n)$, 可推出$T(n) = O(n \log n)$; 最后一步复杂度仍为$O(n \log n)$, 因此整个计算$2n$ 位大整数与n 位大整数相除的复杂度为$O(n \log n)$.

于是我们得到了一个和FFT 复杂度相同的, 用来计算两大整数相除的优秀算法. 当然, 隐藏在时间复杂度后面的常数较FFT 与两大整数相乘大了不少.

牛顿迭代法: 应用

求得A'^* 当然还没完, 这只是迭代的结果, 并不是$\lfloor 10^{2n}/A \rfloor$ 的准确值. 误差分析表明, 当$k > n/2$ 时, 误差不超过100 (可取$k = \lfloor (n+2)/2 \rfloor$), 于是还要在求得余数$10^{2n} - AA'^*$ 后对A'^* 进行次数为$O(1)$ 的调整(为了降低常数, 可以二分误差). 这样就迭代地求解出了$\lfloor 10^{2n}/A \rfloor$. 边界条件是$n \leq 2$, 此时$A$ 在小整数范围内, 可以直接求解结果.

求解出$\lfloor 10^{2n}/A \rfloor$ 后与B 相乘, 再在计算余数后进行和上面类似的调整, 即可求解出$\lfloor B/A \rfloor$.

时间复杂度分析: 假设计算$\lfloor 10^{2n}/A \rfloor$ 时复杂度为$T(n)$, 那么$T(n) = T(k) + O(n \log n) \approx T(n/2) + O(n \log n)$, 可推出$T(n) = O(n \log n)$; 最后一步复杂度仍为$O(n \log n)$, 因此整个计算$2n$ 位大整数与n 位大整数相除的复杂度为$O(n \log n)$.

于是我们得到了一个和FFT 复杂度相同的, 用来计算两大整数相除的优秀算法. 当然, 隐藏在时间复杂度后面的常数较FFT 与两大整数相乘大了不少.

牛顿迭代法: 应用

牛顿迭代法还可以用来解决大整数开平方的问题. 不过假如直接计算$\sqrt{A}$, 考虑方程$x^2 - A = 0$, 迭代式为

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

中间出现了除法, 这并不好. 换一个方程$\frac{1}{x^2} - \frac{1}{A} = 0$, 迭代式为

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

还是有除法. 不过可以发现, 如果计算$\sqrt{1/A}$, 迭代式中就没有除法了, 这是很好的事. 得到$\sqrt{1/A}$ 后乘A 就可以得到$\sqrt{A}$.

这按道理需要用高精度实数, 不过还是可以像刚才那样, 假设A 有$2n-1$ 或$2n$ 位, 迭代计算$A' = \lfloor 10^{2n}\sqrt{1/A} \rfloor$, 然后从$\lfloor AA'/10^{2n} \rfloor$ 出发调整即可. 这样n 位大整数开平方也可以在$O(n \log n)$ 时间内完成了. (其实最糟糕的是误差分析. . .)

牛顿迭代法: 应用

牛顿迭代法还可以用来解决大整数开平方的问题. 不过假如直接计算$\sqrt{A}$, 考虑方程$x^2 - A = 0$, 迭代式为

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

中间出现了除法, 这并不好. 换一个方程$\frac{1}{x^2} - \frac{1}{A} = 0$, 迭代式为

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

还是有除法. 不过可以发现, 如果计算$\sqrt{1/A}$, 迭代式中就没有除法了, 这是很好的事. 得到$\sqrt{1/A}$ 后乘A 就可以得到$\sqrt{A}$.

这按道理需要用高精度实数, 不过还是可以像刚才那样, 假设A 有$2n-1$ 或$2n$ 位, 迭代计算$A' = \lfloor 10^{2n}\sqrt{1/A} \rfloor$, 然后从$\lfloor AA'/10^{2n} \rfloor$ 出发调整即可. 这样n 位大整数开平方也可以在$O(n \log n)$ 时间内完成了. (其实最糟糕的是误差分析. . .)

Part II

高精度实数与复数运算

高精度实数的存储

高精度实数存储的主要思路就是科学记数法，不过具体来说有许多实现方法，比如IEEE 754 标准的单精度与双精度实数类型采取$1.F \times 2^{E-bias}$ 的格式存储实数($E = 0$ 时较特殊，采取$0.F \times 2^{1-bias}$ 格式)，另外还特别规定了正负无穷大和NaN.

不过在这里为了叙述方便，也为了直接套上之前大整数运算的体系，就采取$N \times 10^E$ 的格式记录高精度实数，其中N 是首位非零的n 位大整数，E 是指数．但是零需要特殊处理，比如令$N = 0$．以下运算都假设实数非0．(假如有0 也很好特判...)

高精度实数的运算

加法: 假如两数的指数不一致, 将较小者的指数调到与较大者相同, 再将两者的N 相加. 如果所得$\geq 10^n$, 还需舍一位精度.

- 例: $9987 \times 10^4 + 1984 \times 10^2 \approx 9987 \times 10^4 + 20 \times 10^4 = 10007 \times 10^4 \approx 1001 \times 10^5$

减法: 和加法一样, 不过最后不是舍去精度, 而是可能需补零.

- 例: $1000 \times 10^4 - 9985 \times 10^3 \approx 1000 \times 10^4 - 999 \times 10^4 = 1 \times 10^4 = 1000 \times 10^1$

乘法: N 相乘, 指数相加, 再舍去精度.

- 例: $(1111 \times 10^3) \times (9999 \times 10^4) = 11108889 \times 10^7 \approx 1111 \times 10^{11}$

高精度实数的运算

加法: 假如两数的指数不一致, 将较小者的指数调到与较大者相同, 再将两者的N 相加. 如果所得$\geq 10^n$, 还需舍一位精度.

- 例: $9987 \times 10^4 + 1984 \times 10^2 \approx 9987 \times 10^4 + 20 \times 10^4 = 10007 \times 10^4 \approx 1001 \times 10^5$

减法: 和加法一样, 不过最后不是舍去精度, 而是可能需补零.

- 例: $1000 \times 10^4 - 9985 \times 10^3 \approx 1000 \times 10^4 - 999 \times 10^4 = 1 \times 10^4 = 1000 \times 10^1$

乘法: N 相乘, 指数相加, 再舍去精度.

- 例: $(1111 \times 10^3) \times (9999 \times 10^4) = 11108889 \times 10^7 \approx 1111 \times 10^{11}$

高精度实数的运算

加法：假如两数的指数不一致，将较小者的指数调到与较大者相同，再将两者的N 相加．如果所得$\geq 10^n$，还需舍一位精度．

- 例：$9987 \times 10^4 + 1984 \times 10^2 \approx 9987 \times 10^4 + 20 \times 10^4 = 10007 \times 10^4 \approx 1001 \times 10^5$

减法：和加法一样，不过最后不是舍去精度，而是可能需补零．

- 例：$1000 \times 10^4 - 9985 \times 10^3 \approx 1000 \times 10^4 - 999 \times 10^4 = 1 \times 10^4 = 1000 \times 10^1$

乘法：N 相乘，指数相加，再舍去精度．

- 例：$(1111 \times 10^3) \times (9999 \times 10^4) = 11108889 \times 10^7 \approx 1111 \times 10^{11}$

高精度实数的运算

除法: 先将被除数的N 左移n 位, 再将两者的N 相除, 指数相减.

- 例: $(1111 \times 10^3)/(9999 \times 10^4) = (11110000 \times 10^{-1})/(9999 \times 10^4) \approx 1111 \times 10^{-5}$

开平方: 先将N 左移n 位, 通过右移将指数调成偶数, 再将N 开平方, 指数减半.

- 例: $\sqrt{1111 \times 10^3} = \sqrt{11110000 \times 10^{-1}} = \sqrt{1111000 \times 10^0} \approx 1054 \times 10^0$

这样高精度实数体系通过大整数运算构建完毕, 且时间复杂度和n 位大整数运算相同.

高精度实数的运算

除法: 先将被除数的N 左移n 位, 再将两者的N 相除, 指数相减.

- 例: $(1111 \times 10^3)/(9999 \times 10^4) = (11110000 \times 10^{-1})/(9999 \times 10^4) \approx 1111 \times 10^{-5}$

开平方: 先将N 左移n 位, 通过右移将指数调成偶数, 再将N 开平方, 指数减半.

- 例: $\sqrt{1111 \times 10^3} = \sqrt{11110000 \times 10^{-1}} = \sqrt{1111000 \times 10^0} \approx 1054 \times 10^0$

这样高精度实数体系通过大整数运算构建完毕, 且时间复杂度和n 位大整数运算相同.

高精度实数的运算

除法: 先将被除数的N 左移n 位, 再将两者的N 相除, 指数相减.

- 例: $(1111 \times 10^3)/(9999 \times 10^4) = (11110000 \times 10^{-1})/(9999 \times 10^4) \approx 1111 \times 10^{-5}$

开平方: 先将N 左移n 位, 通过右移将指数调成偶数, 再将N 开平方, 指数减半.

- 例: $\sqrt{1111 \times 10^3} = \sqrt{11110000 \times 10^{-1}} = \sqrt{1111000 \times 10^0} \approx 1054 \times 10^0$

这样高精度实数体系通过大整数运算构建完毕, 且时间复杂度和n 位大整数运算相同.

高精度复数

高精度复数体系构建就很简单了.

存储：相同精度的实部与虚部

加减乘除：套复数运算律. 值得一提的是，乘法可以只通过三次实数乘法完成：$(a+bi)(c+di) = (ac-bd)+(ad+bc)i = (ac-bd)+[(a+b)(c+d)-ac-bd]i$; 除法也可类似进行优化.

开平方：复数有两个平方根，并且对于不是非负实数的复数不定义算术平方根. 当复数不为负实数时，可以通过以下公式计算其实部非负的那个平方根：

$$\sqrt{a+bi} = \sqrt{(r+a)/2} \pm i\sqrt{(r-a)/2}, r = \sqrt{a^2+b^2}$$

其中虚部符号由b 的符号决定.

显而易见，这些运算和相同精度的高精度实数具有相同的复杂度(当然常数要大几倍).

Part III

高精度实基本初等函数计算

泰勒展开：理论基础

这一节的主要内容是泰勒展开计算实基本初等函数. 显然假如不了解泰勒展开, 以下内容都无从谈起.

对于一个函数$f(x)$, 假如它在$x = x_0$ 处和邻近有$n+1$ 阶导数, 则定义多项式

$$f_n(x) = f(x_0) + f'(x_0)(x - x_0) + \ldots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

它的特点是它在$x = x_0$ 处的$0, 1, \ldots, n$ 阶导数与$f(x)$ 相同. 那么这个多项式在x_0 附近将非常接近$f(x)$, 其误差可以通过Lagrange 余项估计:

$$f(x) - f_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1} \tag{1}$$

其中ξ 是在x 与x_0 之间的一个数.

(1) 式即为带Lagrange 余项的泰勒公式.

泰勒展开: 理论基础

下面是几个实基本初等函数的泰勒公式:

$$e^x = 1 + x + \frac{x^2}{2} + \ldots + \frac{x^n}{n!} + \frac{e^{\theta x} x^{n+1}}{(n+1)!}$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots + \frac{(-1)^{n-1} x^n}{n} + \frac{(-1)^n x^{n+1}}{(n+1)(1+\theta x)^{n+1}}$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots - \frac{x^{4n-1}}{(4n-1)!} + \frac{\sin(\theta x) x^{4n}}{(4n)!}$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4!} + \ldots - \frac{x^{4n-2}}{(4n-2)!} + \frac{\cos(\theta x) x^{4n}}{(4n)!}$$

$$\arctan x = x - \frac{x^3}{3} + \ldots - \frac{x^{4n-1}}{4n-1} + \frac{\cos^{4n} y_{\theta x} \sin(4n y_{\theta x}) x^{4n}}{4n}$$

其中$\theta \in (0,1)$, $y_{\theta x} = \arctan(\theta x)$.

泰勒展开: 应用

下面说说e^x 的计算.

假如$x = 0$, 那么已有$e^x = 1$; 假如$x < 0$, 那么有$e^x = 1/e^{-x}$. 因此下面假定$x > 0$.

当$x \le 1$ 时, 作误差分析如下:

$$\left| e^x - \left(1 + x + \frac{x^2}{2} + \ldots + \frac{x^n}{n!} \right) \right| = \left| \frac{e^{\theta x} x^{n+1}}{(n+1)!} \right| \le \left| \frac{e^x}{(n+1)!} \right|$$

从最后的式子可以看出, 放大后的相对误差和x 无关, 因此可以在转化成对数后预先用低精度实数运算得出, 因此达到所需的相对误差需要计算的项数估计就可以得到了.

当$x > 1$ 时, 可以通过实数的存储形式得到$x = x^* \times 10^E$, 其中$x^* < 1$. 这样

$$e^x = e^{x^* \times 10^E} = \left(e^{x^*} \right)^{10^E}$$

于是转化为求e^{x^*} 后求一次快速幂.

泰勒展开：应用

下面说说e^x 的计算.

假如$x = 0$, 那么已有$e^x = 1$; 假如$x < 0$, 那么有$e^x = 1/e^{-x}$. 因此下面假定$x > 0$.

当$x \leq 1$ 时, 作误差分析如下:

$$\left| e^x - \left(1 + x + \frac{x^2}{2} + \ldots + \frac{x^n}{n!} \right) \right| = \left| \frac{e^{\theta x} x^{n+1}}{(n+1)!} \right| \leq \left| \frac{e^x}{(n+1)!} \right|$$

从最后的式子可以看出, 放大后的相对误差和x 无关, 因此可以在转化成对数后预先用低精度实数运算得出, 因此达到所需的相对误差需要计算的项数估计就可以得到了.

当$x > 1$ 时, 可以通过实数的存储形式得到$x = x^* \times 10^E$, 其中$x^* < 1$. 这样

$$e^x = e^{x^* \times 10^E} = \left(e^{x^*} \right)^{10^E}$$

于是转化为求e^{x^*} 后求一次快速幂.

泰勒展开：应用

下面说说e^x 的计算.

假如$x = 0$, 那么已有$e^x = 1$; 假如$x < 0$, 那么有$e^x = 1/e^{-x}$. 因此下面假定$x > 0$.

当$x \leq 1$ 时, 作误差分析如下:

$$\left| e^x - \left(1 + x + \frac{x^2}{2} + \ldots + \frac{x^n}{n!}\right) \right| = \left| \frac{e^{\theta x} x^{n+1}}{(n+1)!} \right| \leq \left| \frac{e^x}{(n+1)!} \right|$$

从最后的式子可以看出, 放大后的相对误差和x 无关, 因此可以在转化成对数后预先用低精度实数运算得出, 因此达到所需的相对误差需要计算的项数估计就可以得到了.

当$x > 1$ 时, 可以通过实数的存储形式得到$x = x^* \times 10^E$, 其中$x^* < 1$. 这样

$$e^x = e^{x^* \times 10^E} = \left(e^{x^*}\right)^{10^E}$$

于是转化为求e^{x^*} 后求一次快速幂.

泰勒展开: 应用

这样e^x 的计算就被解决了. 但是有个问题, 假如要求相对误差不超过10^{-N}, 那么项数的级别仅仅稍低于$O(N)$, 因此就算用FFT 一项一项递推计算, 复杂度也会达到至少$O(N^2)$.

一个办法是, 刚才是对$x \le 1$ 时的误差进行估计, 但是这样并不是最优的. 实际上, 假如要求相对误差是10^{-N}, 那么可以把$x \le 1$ 变成$x \le 10^{-\lceil\sqrt{N}\rceil}$. 这样仅仅再作出这样的误差估计

$$\Delta \le \left|\frac{e^x x^{n+1}}{(n+1)!}\right| < \left|e^x x^{n+1}\right| \le \left|e^x 10^{-(n+1)\sqrt{N}}\right|$$

就可以知道项数n 的级别不高于$O(\sqrt{N})$. 这样假如x 是$O(1)$ 级别的, 之后求快速幂时乘法的次数也是$O(\sqrt{N})$, 最终复杂度至多为$O(N^{1.5}\log N)$.

需要注意的是, 由于多项相加的累积误差较大, 在中间计算时应当再加$O(\log N)$ 位.

泰勒展开: 应用

这样e^x 的计算就被解决了. 但是有个问题, 假如要求相对误差不超过10^{-N}, 那么项数的级别仅仅稍低于$O(N)$, 因此就算用FFT 一项一项递推计算, 复杂度也会达到至少$O(N^2)$.

一个办法是, 刚才是对$x \le 1$ 时的误差进行估计, 但是这样并不是最优的. 实际上, 假如要求相对误差是10^{-N}, 那么可以把$x \le 1$ 变成$x \le 10^{-\lceil\sqrt{N}\rceil}$. 这样仅仅再作出这样的误差估计

$$\Delta \le \left|\frac{e^x x^{n+1}}{(n+1)!}\right| < \left|e^x x^{n+1}\right| \le \left|e^x 10^{-(n+1)\sqrt{N}}\right|$$

就可以知道项数n 的级别不高于$O(\sqrt{N})$. 这样假如x 是$O(1)$ 级别的, 之后求快速幂时乘法的次数也是$O(\sqrt{N})$, 最终复杂度至多为$O(N^{1.5}\log N)$.

需要注意的是, 由于多项相加的累积误差较大, 在中间计算时应当再加$O(\log N)$ 位.

泰勒展开：应用

下面用类似的方法计算$\sin x$，$\cos x$.

首先需要算出π (这可以用之后$\arctan x$ 来计算), 然后用诱导公式将x 诱导到$[0, \frac{\pi}{4}]$ 的范围内.

在$\sin x$ 的计算中，

$$\Delta = \left|\frac{\sin(\theta x)x^{4n}}{(4n)!}\right| < \left|\frac{\sin x x^{4n}}{(4n)!}\right|;$$

在$\cos x$ 的计算中，

$$\Delta = \left|\frac{\cos(\theta x)x^{4n}}{(4n)!}\right| < \sqrt{2}\left|\frac{\sqrt{2}/2x^{4n}}{(4n)!}\right| < \sqrt{2}\left|\frac{\cos x x^{4n}}{(4n)!}\right|.$$

因此都可以像e^x 那样做.

泰勒展开：应用

但在应用刚刚的优化的时候，和$e^{nx} = (e^x)^n$ 不同，需要用到

$$\sin(3x) = 3\cos^2 x \sin x - \sin^3 x = 3\sin x - 4\sin^3 x,$$

$$\cos(2x) = \cos^2 x - \sin^2 x = 2\cos^2 x - 1.$$

注意到这并不是10 的倍数，因此需要先用2,3 暴力除，直到达到想要的范围内为止.

这样，$\sin x$ 和$\cos x$ 也能在和e^x 复杂度相同的时间内算出.

泰勒展开：应用

下面计算$\ln x$. 假如$x = 1$, 那么$\ln x = 0$; 假如$x < 1$, 那么$\ln x = -\ln(1/x)$. 因此下面解决$x > 1$ 的情况. 用之前给的$\ln[1 + (x-1)]$ 计算, 误差估计如下:

$$\Delta = \left| \frac{(-1)^n (x-1)^{n+1}}{(n+1)[1 + \theta x - \theta]^{n+1}} \right| < \left|(x-1)^{n+1}\right| \approx |\ln x (x-1)^n|$$

上式对于较小的$x - 1$ 成立. 这时必须要想办法使x 与1 非常接近. 因此假如要求相对误差10^{-N}, 需要仿照e^x 中的方法, 将$x - 1$ 缩小到$10^{-\sqrt{N}}$. 但是如果只用$\ln x = 2\ln\sqrt{x}$ 在将x 减1 时会造成精度损失. 更高明的方法是直接存储$x - 1$, 然后令$\mathrm{ln1p}(x) = \ln(1 + x)$, 利用以下公式缩小$x - 1$

$$\mathrm{ln1p}(x-1) = 2\mathrm{ln1p}\left(\sqrt{x} - 1\right) = 2\mathrm{ln1p}\left[\frac{x-1}{\sqrt{1 + (x-1)} + 1}\right]$$

这样$\ln x$ 也可以在$O(N^{1.5}\log N)$ 的时间内算出.

泰勒展开：应用

对于$\arctan x$，假如$x < 0$，那么$\arctan x = -\arctan(-x)$；假如$x = 0$，那么$\arctan x = 0$。因此下面解决$x > 0$ 的情况.

误差估计如下:

$$\Delta = \left|\frac{\cos^{4n} y_{\theta x} \sin(4ny_{\theta x})x^{4n}}{4n}\right| < \left|x^{4n}\right| \approx \left|\arctan x x^{4n-1}\right|$$

上式对于较小的x 成立. 同$\ln(1+x)$ 一样缩小x 后可以在$O(N^{1.5}\log N)$ 的时间内算出. 不同的是缩小x 需要用到以下公式(注意分子有理化是为了减少精度损失):

$$\arctan x = 2\arctan\left(\frac{x}{\sqrt{1+x^2}+1}\right)$$

用$\arctan x$ 可以计算之前的遗留问题π. 最简单的方法是用$\pi = 4\arctan 1$，另外还有梅钦公式$\pi = 16\arctan(1/5) - 4\arctan(1/239)$ 等等.

泰勒展开：应用

对于$\arctan x$，假如$x<0$，那么$\arctan x = -\arctan(-x)$；假如$x=0$，那么$\arctan x = 0$．因此下面解决$x>0$ 的情况．

误差估计如下：

$$\Delta = \left|\frac{\cos^{4n} y_{\theta x} \sin(4ny_{\theta x}) x^{4n}}{4n}\right| < \left|x^{4n}\right| \approx \left|\arctan x x^{4n-1}\right|$$

上式对于较小的x 成立．同$\ln(1+x)$ 一样缩小x 后可以在$O(N^{1.5}\log N)$ 的时间内算出．不同的是缩小x 需要用到以下公式(注意分子有理化是为了减少精度损失)：

$$\arctan x = 2\arctan\left(\frac{x}{\sqrt{1+x^2}+1}\right)$$

用$\arctan x$ 可以计算之前的遗留问题π．最简单的方法是用$\pi = 4\arctan 1$，另外还有梅钦公式$\pi = 16\arctan(1/5) - 4\arctan(1/239)$ 等等．

AGM 方法: 理论基础

接下来讲一下用AGM 方法在$O(N\log^2 N)$ 时间内计算各基本初等函数至N 位精度的方法. 这是迄今为止计算这些基本初等函数理论复杂度最低的方法. 由于我并没有这方面的数学基础, 因此只能大概讲一下思路.

定义两个正数a, b 的算术几何平均数(AGM) 如下: 构造数列$\{a_n\}$ 与$\{b_n\}$, 其中$a_0 = a, b_0 = b$,

$$a_n = (a_{n-1} + b_{n-1})/2, b_n = \sqrt{a_{n-1}b_{n-1}} \ \forall n \in \mathbf{N}_+$$

那么$\{a_n\}$ 与$\{b_n\}$ 收敛于同一极限, 称该极限为a, b 的AGM, 可记为$AG(a, b)$. 并且和牛顿迭代一样是二阶收敛的.

不过稍令人遗憾的是, $AG(a, b)$ 不能用初等函数表示. 它的表达式和椭圆积分有关

$$AG(a, b) = \frac{\pi}{2}\left(\int_0^{\pi/2} \frac{\mathrm{d}x}{\sqrt{a^2\cos^2 x + b^2\sin^2 x}}\right)^{-1}$$

AGM 方法: 理论基础

接下来讲一下用AGM 方法在$O(N\log^2 N)$ 时间内计算各基本初等函数至N 位精度的方法. 这是迄今为止计算这些基本初等函数理论复杂度最低的方法. 由于我并没有这方面的数学基础, 因此只能大概讲一下思路.

定义两个正数a, b 的算术几何平均数(AGM) 如下：构造数列$\{a_n\}$ 与$\{b_n\}$, 其中$a_0 = a, b_0 = b$,

$$a_n = (a_{n-1} + b_{n-1})/2, b_n = \sqrt{a_{n-1}b_{n-1}} \ \forall n \in \mathbf{N}_+$$

那么$\{a_n\}$ 与$\{b_n\}$ 收敛于同一极限, 称该极限为a, b 的AGM, 可记为$AG(a, b)$. 并且和牛顿迭代一样是二阶收敛的.

不过稍令人遗憾的是, $AG(a, b)$ 不能用初等函数表示. 它的表达式和椭圆积分有关

$$AG(a, b) = \frac{\pi}{2}\left(\int_0^{\pi/2} \frac{\mathrm{d}x}{\sqrt{a^2\cos^2 x + b^2\sin^2 x}}\right)^{-1}$$

AGM 方法: 应用

虽然上述AGM 并不能用初等函数表示, 但是它具有如下性质:

$$\left|\frac{\pi/2}{AG(1,4/x)} - \ln x\right| \le \frac{64}{x^2}[8+\ln(x/4)]$$

对所有的$x \ge 4$ 成立. 因此, 只要取$x > 10^{N/2}$, 所得精度就可以接近N 位(实际还要再取大一些). 这样对于比较大的x, 通过

$$\ln x \approx \frac{\pi/2}{AG(1,4/x)}$$

计算$\ln x$ 就可以达到N 位精度.

对于较小的x, 首先将问题归结于$x > 1$, 然后用$\ln x = 1/2 \ln x^2$ 逐步放大x; 或者可用$\ln x = \ln(x \cdot 10^M) - M \ln 10$ 直接放大x, 但需预先计算$\ln 10$, 且有少许精度损失.

最终$\ln x$ 即可通过$O(\log N)$ 次迭代达到N 位精度, 时间复杂度为$O(N \log^2 N)$.

AGM 方法: 应用

且慢, 这里有一个问题: π. 用泰勒展开计算它显然太慢了, 与AGM 算法的时间复杂度不相称.

Salamin-Brent 算法可以用来解决这个问题.
取$a = 1, b = 1/\sqrt{2}$ 构造$\{a_n\}$ 与$\{b_n\}$, 那么下式

$$\frac{4a_n^2}{1 - \sum_{j=0}^{n} 2^j (a_j - b_j)^2}$$

将逐渐趋于π.

这个算法可在与AGM 方法相同复杂度的时间内算出π.

AGM 方法: 应用

有了$\ln x$, e^x 也可以解决了.

取一个比e^x 小的数, 用牛顿迭代法即可解得e^x. 不过不需要每次迭代都用N 位精度的数计算, 这样最后时间复杂度会多出一个log 来. 可以只取有效位数部分, 然后每次计算后保留的有效位数加倍.

这样, e^x 也可以在与AGM 方法相同复杂度的时间内算出了.

AGM 方法: 应用

计算三角函数需要用到复对数. 计算复对数仍然可以用刚才的公式

$$\ln x \approx \frac{\pi/2}{AG(1, 4/x)}$$

但这里$4/x$ 是复数. 不过还好, 如果x 不是负数, 那么此时AGM 是可以进行的, 只要在开平方时取实部为正的那个即可.

这样特判x 是负数的情况后, 再想办法使得$\ln x$ 符合误差要求. 但是我并没有找到相关的论文详细说明这点.

最后, 利用$\arctan x = \mathrm{Im}[\log(1+ix)]$, $\sin x = \mathrm{Im}(e^{ix})$, $\cos x = \mathrm{Re}(e^{ix})$ 可在与AGM 方法相同复杂度的时间内算出这三个三角函数. 其中e^{ix} 需要用复数域上的牛顿迭代求解.