再探快速傅里叶变换

雅礼中学 毛啸

摘 要

本文主要介绍了快速傅里叶变换的实现和一些在算法竞赛中有用的技巧和优化。快速傅里叶变换作为一个早已经在算法竞赛中流行起来的算法，对大家来说并不陌生，因此本文题为“再探快速傅里叶变换”。“再探”，顾名思义，本文肯定会介绍许多已经普及了的方法和知识，但仍有相当一部分篇幅将会用在介绍一些在现在算法竞赛中还不是很普及，但是却非常简单而有效的方法。例如，在模10^9+7意义下做两个次数不超过10^5的实多项式的乘法，目前传统的方法是在三个可以用于数论变换的模数下求出结果，最后利用中国剩余定理来求得最终的答案，需要做9次模意义下的离散傅里叶变换，而利用本文3.3节中介绍的方法，只需要做4次复数意义下的离散傅里叶变换即可。

前言

近几年来，各种基于FFT的算法如雨后春笋般流行起来。在2014年的NOI冬令营营员交流上，出现了关于多项式除法的内容[1]，之后在Codeforces上出现了使用多项式除法的题目[2]，之后基于牛顿迭代的各种算法纷纷流行起来，例如在Universal Online Judge(UOJ) 上出现了使用牛顿迭代解微分方程的题目[3]，在2016年NOI冬令营第一课堂的《多项式导论》中，这些算法被再次普及，UOJ上最近还出现了多点求值的题目[4]。基于FFT的各种算法的出现，给解题带来了极大的便利。

[1]余行江、彭雨翔《多项式除法及其应用》

[2]http://codeforces.com/contest/438/problem/E

[3]http://uoj.ac/problem/50

[4]http://uoj.ac/problem/182

然而，所有这些算法都离不开一个东西——FFT，如果没有掌握FFT这个最基本的算法，所有这些基于FFT的算法肯定更加难以掌握。好比盖一栋大楼，如果我们把楼盖得很高，却因为基底不牢固而坍塌，这栋楼再怎么高也没有用。一棵大树，枝叶繁茂，根却因为种种原因坏死，这棵树也只会盖着一头枯枝烂叶。只有夯实了基础，才能走的更远。本人写此文正是基于这样一个思路。希望能与各位读者回到原点，重新研究一下FFT这个最基础而又十分神奇的算法。而什么叫掌握一个算法呢，本人认为，掌握一个算法不仅仅限于写过、甚至背过这个算法，而是既了解这个算法的基本原理，也知道这个算法的各种应用、各种优化和技巧。

1 基本介绍

快速傅里叶变换(Fast Fourier Transform(FFT))是现在比较流行的算法之一，该算法在1965年被普及，但是有一些方法在1805年就有出现[5]。该算法通过计算一个序列的离散傅里叶变换(Discrete Fourier Transform(DFT))来实现将信号从原始域(通常是时间或空间)到到频域的互相转化。该算法在当今算法竞赛中已经十分常见，主要应用于计算两个序列的卷积。

1.1 引入

我们来看一下这样一个问题：

给定两个有限长度的序列a_i，b_i，求序列c使得$c_i = \sum_{k=0}^{i} a_k b_{i-k}$，显然$c$的长度为$a, b$的长度之和减一。本文中的序列下标均从0开始。

1.2 离散傅里叶变换

我们找一个大于c的长度且为2的整数次幂的数n，至于为什么是2的整数次幂后面会有提到。**给定序列s，假设它的长度为$|s|$，那么对于$k \notin [0, |s|)$我们将s_k视为0。**

[5]Heideman, M. T.; Johnson, D. H.; Burrus, C. S. (1984). "Gauss and the history of the fast Fourier transform". IEEE ASSP Magazine 1 (4): 14 – 21

那么我们有：

$$c_r = \sum_{p,q} [(p+q) \bmod n = r] a_p b_q \tag{1.1}$$

设ω满足$\omega^n = 1$，即ω是n次单位根，**本文中提到的单位根的意思是说，n是最小的使得$\omega^n = 1$ 的数**。易知：

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{vk} = [v \bmod n = 0] \tag{1.2}$$

故：

$$\begin{aligned} & [(p+q) \bmod n = r] \\ = \ & [(p+q-r) \bmod n = 0] \\ = \ & \frac{1}{n} \sum_{k=0}^{n-1} \omega^{(p+q-r)k} \\ = \ & \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-rk} \omega^{pk} \omega^{qk} \end{aligned}$$

我们有：

$$\begin{aligned} c_r &= \sum_{p,q} [(p+q) \bmod n = r] a_p b_q \\ &= \sum_{p,q} \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-rk} \omega^{pk} \omega^{qk} a_p b_q \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-rk} \sum_{p,q} \omega^{pk} a_p \omega^{qk} b_q \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-rk} \sum_{p} \omega^{pk} a_p \sum_{q} \omega^{qk} b_q \end{aligned}$$

我们发现了这样两种关系式：

$$a_m = \sum_{k=0}^{n-1} \omega^{mk} b_k \tag{1.3}$$

$$c_m = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-mk} d_k \tag{1.4}$$

现在，我们发现我们只要对于每一种关系式，知道右边快速的求得左边。进一步我们发现，我们设多项式$A(x)=\sum_{k=0}^{n-1}b_kx^k$，那么$a_m=A(\omega^m)$，相当于给定一个多项式的系数表示，对于所有$k$求出其在点$\omega^k$的值，即在系数表示和点值表示之间转化，这就是**离散傅里叶变换(DFT)**的过程。

我们后面提到多项式与系数序列的对应关系都是指类似这里$A(x)$与b_k的关系，我们说对$A(x)$进行DFT，实际上指的就是对b_k进行DFT，说对一个东西DFT得到了$A(x)$，指的就是得到了序列b_k。

1.3 快速傅里叶变换

前一节中已经提到了利用DFT的公式，但是按照公式，显然还是太慢了。这时候需要利用单位复根的性质来加速DFT，也就是**快速傅里叶变换(Fast Fourier Transform(FFT))**。

下面仅考虑(1.3)式，我们知道b要求a。

注意到n是2的整数次幂。我们用ω_n^k来表示n次单位复根，那么我们有：

$$\omega_{2n}^{2m} = \omega_n^m \tag{1.5}$$

$$\omega_{2n}^{m} = -\omega_{2n}^{m+n} \tag{1.6}$$

设$A_0(x)$是$A(x)$的偶次项的和，$A_1(x)$是奇次项的和。那么：

$$A(\omega_n^m) = A_0((\omega_n^m)^2)+\omega_n^m A_1((\omega_n^m)^2) \tag{1.7}$$

$$= A_0(\omega_{\frac{n}{2}}^m)+\omega_n^m A_1(\omega_{\frac{n}{2}}^m) \tag{1.8}$$

$$A(\omega_n^{m+\frac{n}{2}}) = A_0((\omega_n^m)^2)+\omega_n^{m+\frac{n}{2}} A_1((\omega_n^m)^2) \tag{1.9}$$

$$= A_0(\omega_{\frac{n}{2}}^m)-\omega_n^m A_1(\omega_{\frac{n}{2}}^m) \tag{1.10}$$

即我们只要对$A_0(x)$和$A_1(x)$求出DFT，就可以在线性时间内求出$A(x)$的DFT，(1.8)(1.10)式常被称为“蝴蝶操作”。

设给长度为n的序列做DFT的复杂度为$T(n)$，则有：

$$T(n)=2T(\frac{n}{2})+O(n) \tag{1.11}$$

根据主定理$T(n)=O(n\log n)$。

我们发现，如果要完成上述算法，必须要做到能够把一个多项式的奇数项和偶数项分开，虽然在递归中是可以实现的，但是常数过大，作为一个经常要使用的算法，如果我们的实现方式常数过大是很不好的。

我们观察程序执行的时候每个函数的系数，以长度为8为例：

$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$

$(a_0, a_2, a_4, a_6)(a_1, a_3, a_5, a_7)$

$(a_0, a_4)(a_2, a_6)(a_1, a_5)(a_3, a_7)$

$(a_0)(a_4)(a_2)(a_6)(a_1)(a_5)(a_3)(a_7)$

观察$(0, 4, 2, 6, 1, 5, 3, 7)$的二进制表示为$(000, 100, 010, 110, 001, 101, 011, 111)$，我们发现就是把$(0, 1, 2, 3, 4, 5, 6, 7)$的二进制表示$(000, 001, 010, 011, 100, 101, 110, 111)$中的每一位的二进制位翻转，称为“位逆序置换”，那么如果能够较快的求出位逆序置换之后的序列，那么在求解的时候就可以每次处理的都是一段连续的区间，很容易实现非递归的FFT，并且速度比递归版本更快[6]。

此外，为了进一步优化常数，我们应该预处理单位根。

预处理单位根可以利用$\omega^{k+1} = \omega^k \times \omega$递推求，但是这样精度误差稍微有点大，虽然一般情况下可以接受，但是在第3节中介绍的FFT方法中要求精度达到10^{14}，这样预处理单位根会有严重的精度问题。

另外一种方法是设n次单位根为ω_n，那么$\omega_n^k = cos(\frac{2\pi k}{n}) + isin(\frac{2\pi k}{n})$，这里$i = \sqrt{-1}$，这样虽然速度比上述方法慢，但是精度误差更小，可以满足10^{14}的精度要求。

做卷积的过程无非就是将a, b数组均做1次DFT，然后乘起来再做1次逆DFT(Inverse Discrete Fourier Transform(IDFT))，即实现(1.4)式，方法类似不再赘述[7]。

能不能做到更优呢，当然能！通过本文第3节介绍的方法，我们可以在2次的DFT甚至更优的时间内实现实序列[8]的卷积。

1.4 数论变换

现在算法竞赛中常常要求对某一个数取模，如果取模的数是一个质数P，且

[6]这里本人推荐非常好的模板题：http://uoj.ac/problem/34，较好的代码可以从提交记录统计中看到。

[7]其实这种做2次变换再乘起来再逆变换的思想在反演中应用广泛，有兴趣的同学可以去看参考文献1。

[8]即每一项都是实数。

这个质数减一是一个比DFT的长度要大的2的整数次幂的倍数，那么我们可以求出$\mathbb{F}_{\mathbb{P}}$下的原根g，并在$\mathbb{F}_{\mathbb{P}}$下用$g^{\frac{P-1}{n}}$来代替单位复根ω，这就是**数论变换(Number Theorem Transform(NTT))**。

如果取模的数不是满足这种条件的质数，方法可以参见3.3节。

对于其他域上的多项式乘法，有$O(n\log n\log\log n)$的做法，限于篇幅不再赘述，有兴趣的同学可以参考2016年的NOI冬令营第一课堂的《多项式导论》的相关内容。

1.5 Bluestein's Algorithm

我们前面提到1.1式都是忽略掉了式子中的mod，如果考虑这个mod的话，c实际上是长度为2^k下的**循环卷积**。有时候题目中要求的操作就是做循环卷积，比如如果我们要做循环卷积意义下的快速幂，注意到如果序列长度是2的整数次幂，那么完全可以直接DFT完之后对每一项快速幂，再IDFT回来，然是如果题目要求做循环卷积且长度n不是2的整数次幂，怎么办呢？

一种方法是直接做一遍普通卷积得到答案c，最后再将c_{k+n}加到c_k上。

注意这种方法仅仅只能做1次循环卷积，如果我们要求做多次循环卷积，就必须一遍一遍的做，如果要求在循环卷积意义下做快速幂还要多一个$O(\log n)$的复杂度，我们当然希望能避免。

如果我们考虑的域较为特殊的存在n次单位复根的域，(如$\mathbb{R}$、$\mathbb{C}$、某些$\mathbb{F}_{\mathbb{P}}$等)，那么我们可以考虑能否直接做长度为n的序列的DFT。

这里我们仅介绍较为好理解的Bluestein's Algorithm，这个算法在目前算法竞赛中还不是很普及。

我们继续考虑(1.3)式：

$$\begin{aligned} a_m &= \sum_{k=0}^{n-1}\omega^{mk}b_k \\ &= \sum_{k=0}^{n-1}\omega^{-\frac{(m-k)^2+m^2+k^2}{2}}b_k \\ &= \omega^{-\frac{m^2}{2}}\sum_{k=0}^{n-1}\omega^{-\frac{(m-k)^2}{2}}\omega^{-\frac{k^2}{2}}b_k \end{aligned}$$

于是我们化成了卷积的形式。换句话说我们平时用DFT实现卷积，这里我们用卷积实现DFT!

具体怎么做呢，注意(1.12)式的上下界并不是卷积的条件，但是我们注意到b_k仅在k在0到$n-1$的时候才有可能非0，这是一种可以很好利用的性质。

令$B_i=\omega^{-\frac{(i-n)^2}{2}}$，我们有：

$$\begin{aligned} A_{m+n} &= \omega^{-\frac{2}{m^2}a_m} && (1.12)\\ &= \sum_{k=0}^{n-1}\omega^{-\frac{(m-k)^2}{2}}\omega^{-\frac{k^2}{2}}b_k && (1.13)\\ &= \sum_{k=0}^{n-1}B_{m+n-k}\omega^{-\frac{k^2}{2}}b_k && (1.14)\\ &= \sum_{k=0}^{m+n}B_{m+n-k}\omega^{-\frac{k^2}{2}}b_k && (1.15) \end{aligned}$$

(1.14)式到(1.15)式就是利用了b_k仅在k在0到$n-1$的时候才有可能非0的性质。

这样，我们用一遍普通FFT，就可以做任意长度的DFT了，这就是Bluestein's Algorithm的过程。

Bluestein's Algorithm计算的是Chirp Z-Transform(CZT)，CZT是DFT的广义形式。简单来说这个算法不仅可以实现任意长度的DFT，还可以将(1.3)式中的ω换为任意其他的数。

2 基本应用

FFT最大的作用就在于做卷积，卷积不仅仅用于多项式乘法，还可以用于其他各内题目当中，接下来一节中将介绍几种常见的应用。

2.1 最基本的应用

例题1. *力*[9]

给出n个数q_i，定义F_j为$F_j=\sum_{i<j}\frac{q_iq_j}{(i-j)^2}-\sum_{i>j}\frac{q_iq_j}{(i-j)^2}$，求所有的$E_i=\frac{F_i}{q_i}$。$n$不超过100000。

题目中要求的式子与普通卷积还有细微区别，但是我们有一个技巧就是设$A_{i+n}=E_i$使之完全符合卷积的形式。

[9]来源：ZJOI2014

根据A序列的定义我们有：

$$\begin{aligned} A_{j+n} &= E_j \\ &= \frac{F_j}{q_j} \\ &= \frac{\sum\limits_{i<j} \frac{q_i}{(i-j)^2} - \sum\limits_{i>j} \frac{q_i}{(i-j)^2}}{q_j} \\ &= \sum_{i<j} \frac{q_i}{(i-j)^2} - \sum_{i>j} \frac{q_i}{(i-j)^2} \\ &= \sum_{k=0}^{j+n} q_k P_{j+n-k} \end{aligned}$$

其中，P_i的定义如下：

1. 当$i > n$时$P_i = (i-n)^{-2}$。
2. 当$i < n$时$P_i = -(i-n)^{-2}$。
3. 当$i = n$时$P_i = 0$。

直接对q和p做卷积之后得到A，再根据$E_i = A_{i+n}$求得E即可。

例题2. *Fuzzy Search*[10]

给定母串和模式串，字符集大小为4，给定k，模式串在某个位置匹配当且仅当对于任意位置模式串的这个字符所对应的母串的位置的左右k个字符之内有一个与它相等的，求匹配次数。

串长不超过200000。

很容易求出母串的每个位置每种字符是否能匹配。

对于每种字符，我们将母串中能匹配这种字符的和模式串中所有这个位置是这种字符的地方设为1，其他地方设为0，若每个位置母串和模式串如果同时为1则贡献1，否则没有贡献，也就是两个值乘起来，那么匹配当且仅当四种字符的贡献和是串长，考虑怎么计算每种字符的贡献。

接下来将用的是常用的一种技巧，我们将模式串或者母串翻转。我们发现翻转之后，贡献的计算成了卷积的计算，直接用FFT计算卷积即可。

[10]来源：http://codeforces.com/problemset/problem/528/D

这题体现到了将匹配问题中的模式串或者母串翻转从而转换为卷积的常用技巧。

2.2 更高层次的应用

接下来将介绍FFT更高层次的应用。

例题3. *Kyoya and Train*[11]

n个点m条边的图，给定起点和终点，如果不能在T时间内到达终点则需要付出给定的代价，走每条边需要付出给定的代价，每条边经过时间的分布列给定，最小化所需要的代价的期望。保证有$n \leq 50$，$m \leq 100$，$T \leq 20000$。

考虑暴力DP怎么做，令$\mathrm{DP}_{x,t}$表示到了第x个点，当前经过的时间是t，到终点需要的最小代价，$P_{e,k}$ 表示这条边经过时间为k的概率。那么转移方程就是枚举每一条边e，假设这条边到的点是y，那么用$\sum_k \mathrm{DP}_{y,t+k}P_{e,k}$ 更新答案即可。

如果$t > T$，那么t的具体值是不会有影响的，所以对于所有$t > T$我们都不妨都用$t = T + 1$表示。

考虑对于一条边$e : x \to y$，我们设$S_{e,t} = \sum_k \mathrm{DP}_{y,t+k}P_{e,k}$，那么对于每个$t$，$\mathrm{DP}_{x,t}$的值就是所有$x$连出的边的$e$的$S_{e,t}$的最大值。

现在考虑优化，考虑分治，如果我们要求出所有$l \leq t \leq r$的$\mathrm{DP}_{x,t}$与$S_{e,t}$，那么我们设$mid = \lceil \frac{l+r}{2} \rceil$，先求出所有$mid \leq t \leq r$的答案，然后我们注意到$S_{e,t} = \sum_k \mathrm{DP}_{y,t+k}P_{e,k}$很容易变为卷积的形式，我们可以用FFT求出$mid$到$r$这一段的DP值对前面所有$S$的贡献，然后递归求$l \leq t < mid$的答案。如果$l = r$，那么直接用$S$更新DP即可。

根据主定理，时间复杂度是$O(mT \log^2 T)$。

这道题巧妙的利用了分治FFT来解决问题，实际上分治FFT是非常常见的解决问题的方法。

分治FFT在多点插值，多点求值等问题上也有应用，但其与本文研究FFT基本应用的出发点相悖，因此具体做法不再赘述。

例题4. *Transforming Sequence*[12]

[11]来源：http://codeforces.com/problemset/problem/553/E
[12]来源：http://codeforces.com/problemset/problem/623/E

求长度为n的满足前缀按位或单调递增的k位序列。要求每个位置为$[0, 2^k - 1]$之间的整数。保证$k \leq 30000$，答案要求模$10^9 + 7$后输出。

继续从暴力算法出发，考虑用$\mathrm{DP}_{i,j}$表示长度为i，且当前按位或已经有j位的合法的序列个数，那么对$\mathrm{DP}_{i+1,j+l}$有$\mathrm{DP}_{i,j} \times \binom{k-j}{l}$的贡献。

考虑优化，我们如果对所有x求出了$\mathrm{DP}_{i,x}$和$\mathrm{DP}_{j,x}$，考虑对所有x快速求出$\mathrm{DP}_{i+j,x}$。

对于任意l，$\mathrm{DP}_{i,x}$对$\mathrm{DP}_{i+j,x+y}$有$\mathrm{DP}_{i,x} \times 2^{jx} \times \mathrm{DP}_{j,y} \times \binom{k-x}{y}$ 的贡献，2^{jx} 是这样来的，考虑前i位的或值的x位1，后j位这些地方的值已经无所谓了所以可以任意填。这是一个卷积的形式，可以用FFT快速计算。

这样我们就用简单的快速幂在$O(k \log^2 k)$的时间内解决这个问题。

FFT的应用还远远不止这些，限于篇幅本人暂且只介绍这么多。

2.3 一些优化技巧

2.3.1 减少DFT次数

考虑计算$A(x) = B(x)C(x) + D(x)E(x)$。

最简单的方法就是用两个DFT计算出$B(x)C(x)$和$D(x)E(x)$再加起来，但是我们容易发现，在DFT之后给两个序列的每一项进行乘除的组合再IDFT回去，得到的就是这个多项式的对应乘除组合，前提是长度不超过限制，所以我们先把$B(x), C(x), D(x), E(x)$统统DFT得到序列b, c, d, e，再对每一个k令$a_k = b_k c_k + d_k e_k$，最后给a进行IDFT即可得到$A(x)$。

这个方法在复杂的多项式运算中十分有用，缺点是难以与第3节中介绍的合并DFT方法相容。

还有一个简单技巧，如果一个序列多次被用于DFT，那么显然可以预处理好它的DFT，然后每次直接使用预处理的值。

2.3.2 利用循环卷积

考虑对于两个长度为n的序列a, b，计算它们的卷积c的第$\lceil 0.5n \rceil$项到第$\lfloor 1.5n \rfloor$项，传统的做法是全部往后补零扩充为长度为$2n$的序列然后用FFT算卷积，但是因为FFT求的实际上是在1.5节中我们已经提到的循环卷积，我们如果只补零

扩充到长度为$\lceil 1.5n \rceil$以上的序列然后继续调用FFT求卷积，那么第$\lceil 0.5n \rceil$项到第$\lfloor 1.5n \rfloor$ 项的值不受影响。

上述两种优化在基于牛顿迭代的算法中经常出现，能起到比较大的常数优化作用。

2.3.3 分治FFT中的小范围暴力

在分治FFT算法中，由于FFT自带较大的常数，因此在序列长度较小的时候换用暴力能起到比较大的优化效果。其实这种优化不仅用在分治FFT 中，在所有分治算法中都能起到一定的作用。

上述优化在多点插值，多点求值等算法中有很大的作用。

2.3.4 快速幂乘法次数的优化

如果我们要计算一个多项式的n次幂，在条件允许的情况可以通过取多项式ln，再乘上n，再求多项式exp，复杂度减少了一个$O(\log n)$，多项式ln和多项式exp的方法与本文研究FFT基本应用的出发点相悖，因此不再赘述。

假如我们要快速幂的东西无法用上述方法优化的话，有一个现在还不是很普及的技巧：利用addition chain。

我们注意到快速幂最坏需要$2\log_2 n + C$次乘法，但是这并不是下界。我们可以做到更少的乘法次数。我们这里定义addition chain 为一条链，最开始是一个1，后一个数减前一个数的差是这条链上这个数前面的某一个数。我们称这条addition chain能得到这条链上的所有数。

例如$1 \to 2 \to 4 \to 6$，$6-4=2$在6 之前的数中出现过, $4-2=2$在4之前也出现过。那么这条链能得到$\{1,2,4,6\}$，那么根据这条链我们能设计一种方法用乘法从1次幂得到1、2、4、6次幂，比如我们要求6次幂，那么根据这条addition chain ，我们可以从1次幂出发，用1次幂乘1次幂得到2次幂，再乘2次幂得到4次幂，再乘2次幂得到6次幂。

求出能得到一个数的最优的addition chain是NPH的，但是我们有很好的近似算法，例如我们直接BFS，记录每个数对应的addition chain。首先从1出发，1对应的addition chain就是一个数1，每次我们对于当前数x，枚举它对应的addition chain中的数y，如果$x+y$还没有访问过那么将其入队，并且置它对

应的addition chain为x对应的addition chain后面接上$x+y$。这种方法最坏情况下比快速幂优秀得多，但是它有一个缺点是必须线性预处理。

如果我们要对很大的n(例如10^{10000})求出一个东西的n次幂，我们可以通过Method of Four Russians将乘法次数由$2\log_2 n+C$减少至$\log_2 n+O(\frac{\log n}{\log\log n})$。具体如下，设$2^k$为最接近$\frac{\log n}{\log\log n}$的二的整数次幂，预处理要求的东西的0到$2^k-1$次方，复杂度显然是$O(\frac{\log n}{\log\log n})$，然后每次倍增，每次倍增完$k$次之后再考虑当前模$2^k$的额外乘一个余数次幂，这样倍增中的乘法次数是$\log_2 n+C$，而额外乘的次数是$O(\frac{\log n}{k})=O(\frac{\log n}{\log\log n})$。

此外，如果询问次数很多的话，可以将k设大。

上述方法虽然只是常数优化，但是实现起来十分简单，效果也非常不错。

3 一个新的技巧

接下来我们将研究一种能显著减少DFT次数从而大幅度优化常数的技巧。

事实上这个技巧在现在算法竞赛中并没有完全普及，但是也有一部分人可能听说过这个方法，这里本人做一些介绍希望能进一步普及这个技巧。

这个技巧是在Codechef MGCH3D一题中使用到的优化方法。事实上Codeforces上俄罗斯的enot1.10(Vladimir Smykalov)在与本人talk中告诉本人了一个看似不同的优化方法。但经过一些推敲即可发现事实上该优化方法与Codechef MGCH3D一题中使用到的优化方法大同小异，故这里只描述Codechef MGCH3D的题解当中的描述，题解原文可见参考文献2。

讨论DFT的次数时我们有时会不区分DFT和IDFT，比如3次IDFT加3次DFT被称为6次DFT。

3.1 基本介绍

我们考虑对长度为n的实多项式$A(x),B(x)$进行DFT，假设n已经调整为2的整数次幂。

我们定义：

$$P(x) = A(x)+iB(x) \tag{3.1}$$

$$Q(x) = A(x)-iB(x) \tag{3.2}$$

这里i指$\sqrt{-1}$，设$F_p[k], F_q[k]$分别表示对P和Q进行DFT 之后得到序列的第k项，即$F_p[k] = P(\omega^k), F_q[k] = Q(\omega^k)$，$\omega$是$n$次单位根。

接下来我们进行一系列推导：

由于排版问题我们用X代替$\frac{2\pi jk}{2L}$这个式子，每个X所对应的j, k的含义在上下文中可以看出。$\mathrm{conj}(x)$表示x的共轭复数。

$$\begin{aligned}
F_p[k] &= A(\omega_{2L}^k) + iB(\omega_{2L}^k) \\
&= \sum_{j=0}^{2L-1} A_j\omega_{2L}^{jk} + iB_j\omega_{2L}^{jk} \\
&= \sum_{j=0}^{2L-1} (A_j + iB_j)(\cos X + i\sin X)
\end{aligned}$$

$$\begin{aligned}
F_q[k] &= A(\omega_{2L}^k) - iB(\omega_{2L}^k) \\
&= \sum_{j=0}^{2L-1} A_j\omega_{2L}^{jk} - iB_j\omega_{2L}^{jk} \\
&= \sum_{j=0}^{2L-1} (A_j - iB_j)(\cos X + i\sin X) \\
&= \sum_{j=0}^{2L-1} (A_j\cos X + B_j\sin X) + i(A_j\sin X - B_j\cos X) \\
&= \mathrm{conj}\left(\sum_{j=0}^{2L-1} (A_j\cos X + B_j\sin X) - i(A_j\sin X - B_j\cos X)\right) \\
&= \mathrm{conj}\left(\sum_{j=0}^{2L-1} (A_j\cos(-X) - B_j\sin(-X)) + i(A_j\sin(-X) + B_j\cos(-X))\right) \\
&= \mathrm{conj}\left(\sum_{j=0}^{2L-1} (A_j + iB_j)(\cos(-X) + i\sin(-X))\right) \\
&= \mathrm{conj}\left(\sum_{j=0}^{2L-1} (A_j + iB_j)\omega_{2L}^{-jk}\right) \\
&= \mathrm{conj}\left(\sum_{j=0}^{2L-1} (A_j + iB_j)\omega_{2L}^{(2L-k)j}\right) \\
&= \mathrm{conj}(F_p[2L-k])
\end{aligned}$$

于是我们仅用1次DFT就可以算出F_p和F_q。

令DFT($P[k]$)表示对$P(x)$进行DFT之后得到的序列的第k项，我们发现：

$$\mathrm{DFT}(A[k]) = \frac{F_p[k]+F_q[k]}{2} \tag{3.3}$$

$$\mathrm{DFT}(B[k]) = i\frac{F_p[k]-F_q[k]}{2} \tag{3.4}$$

于是我们将2次DFT合并为了1次，可以减少1次DFT。

3.2 更快速的卷积

我们能不能将2次DFT继续往下优化呢？当然能!

前面有提到过我们将多项式$A(x)$分为奇次项和偶次项的技巧，我们继续考虑用这个技巧优化。

同前面的定义，我们设$A_0(x)$是$A(x)$的偶次项的和，$A_1(x)$是奇次项的和，那么$A(x)=A_0(x^2)+xA_1(x^2)$。

假设我们要求$A(x)$与$B(x)$的乘积，且两个多项式的次数均为$n-1$（假设n为2的整数次幂），则有：

$$\begin{aligned}A(x)B(x) &= (A_0(x^2)+xA_0(x^2))(B_0(x^2)+xB_0(x^2))\\ &= A_0(x^2)B_0(x^2)+xA_0(x^2)B_1(x^2)+xA_1(x^2)B_0(x^2)+x^2A_1(x^2)B_1(x^2)\\ &= A_0(x^2)B_0(x^2)+x(A_0(x^2)B_1(x^2)+A_1(x^2)B_0(x^2))+x^2A_1(x^2)B_1(x^2)\end{aligned}$$

所以我们要求的式子是：

$$A_0(x^2)B_0(x^2)+x(A_0(x^2)B_1(x^2)+A_1(x^2)B_0(x^2))+x^2A_1(x^2)B_1(x^2) \tag{3.5}$$

我们需要做4次多项式乘法，但是如果把譬如$A(x^2)$这种式子看作关于x^2的多项式，那么结果多项式的次数均不超过$n-1$。

所以，我们一开始需要做4次长度为n的DFT，两两合并可以优化为2次。

考虑IDFT，看似需要3次IDFT，其实不然，考虑(3.5)式右边的第一项和第三项，它们依然只有偶数项有系数，而第二项只有奇数项有系数。所以我们考虑合并第一项和第三项。

合并第一项和第三项我们显然也可以看成关于x^2的多项式，于是我们要做的就是根据$A(x)$的DFT求$xA(x)$的DFT，容易发现，$xA(x)$的DFT就是将$A(x)$的DFT的第k项乘上ω^k。

于是我们只需要2次长度为n的IDFT，如果我们能两两合并可以优化为1次。

我们考虑所有需要IDFT的序列，它们并不一定所有项都是实数，初看是难以合并的。

但是我们注意到一点，它们IDFT之后的结果是实数！注意到无论序列是否是纯实数，IDFT都是DFT的逆，我们考虑(3.3)(3.4)二式，我们知道右边能求左边，知道左边我们也能恢复右边，我们只需要对所有k恢复出所有的$F_p[k]$(或者$F_q[k]$)，然后复原$P(x)$(或者$Q(x)$)，再根据IDFT的结果是实数，我们可以直接取实部和虚部来得到IDFT的结果。这样，IDFT的次数被减少到了1次，大功告成。

所以我们只需要做3次长度为n即单倍长度的DFT，而前面的方法需要2次两倍长度的DFT，如果我们仍然以两倍长度来衡量的话我们甚至可以用“1.5次DFT”来形容刚才介绍的优化方法。

3.3 任意模数卷积

接下来我们来介绍如何在任意模数下进行卷积，这里假设模数M在10^9级别，要卷积的序列长度在10^5级别。

我们考虑两个长度为10^5级别的序列的卷积，考虑它们在实数域中卷积之后的结果，每个数的大小在10^{23}级别，一般的浮点类型显然会有误差。我们显然可以使用更高精度的浮点数，如python中的decimal等，但是python效率本身较低且目前不能在国内的NOI赛事中使用，而C++等其他很多语言中却没有类似decimal一样的高精度浮点类型，如果手写的话会非常麻烦且效率并不可观。

3.3.1 三模数NTT

考虑找三个大小为10^9且满足NTT性质的模数，分别求出在这三个模数意义下的卷积的结果，因为每个数的大小在10^{23}级别，所以根据中国剩余定理，我们可以唯一确定每个数。中国剩余定理的具体内容这里不再赘述。

具体实现怎么办呢，一种方法是利用128位整型或者高精度，128位整型在很多地方都不能用，而高精度太麻烦也比较慢。

另一种方法是用一种更巧妙的方式来使用中国剩余定理，我们假设模数分别是mod_0, mod_1, mod_2，先合并前两个模数，也就是求出答案在模$mod_0 \times mod_1$意

义下的值，然后用逆元将模$mod_0 \times mod_1 \times mod_2$意义下的数也就是答案表示成$k \times mod_0 \times mod_1 + b$的形式，这个东西我们不必真正求出，我们只需要在模$M$意义下求即可，这样只需要使用64位整型，而64位整型在基本上所有时候都是可以用的，且速度相对前面的方法非常快。

但是由于上述方法需要使用9次模意义下的DFT，效率仅仅只能用一般来形容。

3.3.2 拆系数FFT

我们设$M_0 = \lceil \sqrt{M} \rceil$，根据带余除法我们可以将所有整数$x$表示为$x = k[x]M_0 + b[x]$，其中$k[x]$和$b[x]$都是整数。

我们假设多项式$A(x)$的系数序列为a_i，多项式$B(x)$的系数序列为b_i，那么我们把$k[a_i], b[a_i], k[b_i], b[b_i]$形成的四个序列两两做1次卷积，卷积的结果中每个数的大小在10^{14}级别，使用精度较高的浮点数以及注意用2中介绍的精度较高的预处理单位复根方法即可接受，做完卷积之后，对每个卷积的结果乘上相应的系数并贡献到答案中去，如2.3.1节中提到的那样，我们可以预处理这四个序列的DFT，然后对于对应的系数相同的两个卷积我们直接合并，于是我们需要做7次DFT。

我们能不能把做DFT的次数继续减少呢，当然能!

我们考虑前面所述的过程，第一步是对4个数列分别做DFT，我们发现用前面所述的合并DFT技巧DFT是可以两两合并的，这样我们将DFT次数减少到了5次。

IDFT我们需要做3次，两两合并我们可以减少到2次。

于是DFT次数被成功的减少到了4次，大功告成。

本人的C++代码实现可以参见`http://uoj.ac/submission/49836`。

同样的，前面介绍的将普通卷积优化到3次单倍长度DFT的方法，显然也可以应用到任意模数卷积下，即实现所谓的“3.5次DFT”做任意模数卷积，但是式子太复杂且优化效果并不显著，这里不再赘述。

3.4 总结

本节介绍了一种十分有力而暂时并不是很普及的减少DFT次数的方法，最后我们发现**两个DFT只要它们都是实数序列是可以合并的，两个IDFT只要它**

们的结果都是实数序列都是可以合并的。这种优化基本不需要增加任何代码量，效率却有了很可观的提升。

然而这种优化也有一定的局限性：只适用于实数域的FFT，而不适用于$\mathbb{F}_{\mathbb{P}}$下的NTT。如果有人能想办法让这种做法能适用于NTT，欢迎与本人交流。

致谢

1. 感谢中国计算机学会提供学习和交流的平台。
2. 感谢雅礼中学的汪星明老师多年来给予的关心和指导。
3. 感谢杜瑜皓同学为本人写本文提供的巨大帮助。
4. 感谢金策同学的验稿。
5. 感谢机房里的其他同学为本文检漏挑错。
6. 感谢Vladimir Smykalov、MGCH3D 的出题人等最早提出本文所述的合并DFT 技巧的人。
7. 感谢其他所有本文所参考过的资料的提供者。
8. 感谢其他对我有过帮助和启发的老师和同学。
9. 感谢父母对我的关心和照顾。
10. 感谢各位百忙之中抽出宝贵的时间阅读本文。

参考文献

[1] 《炫酷反演魔术》，吕凯风

`http://vfleaking.blog.uoj.ac/blog/87`

[2] Fast Fourier Transform，彭雨翔

`http://picks.logdown.com/posts/177631-fast-fourier-transform`

[3] Codechef MGCH3D题解

https://discuss.codechef.com/questions/74772/mgch3d-editorial

[4] CZT的英文维基百科

https://en.wikipedia.org/wiki/Chirp_Z-transform