FFT · IFFT

Ⅰ.DFT

以四次多项式$F(x)=a_0x^0+a_1x^1+a_2x^2+a_3x^3$为例，如果带入$w_4^1\sim w_4^4$作为自变量，那么可使用矩阵形式表达为一下形式：

$$(a_0,a_1,a_2,a_3)\begin{pmatrix} w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \\ w_4^0 & w_4^4 & w_4^8 & w_4^{12} \end{pmatrix}=\begin{pmatrix} F(w_4^1) & F(w_4^2) & F(w_4^3) & F(w_4^4) \end{pmatrix}$$

$$\because w_n^i=w_n^{i\%n}$$

$$\therefore \begin{pmatrix} w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \\ w_4^0 & w_4^4 & w_4^8 & w_4^{12} \end{pmatrix}=\begin{pmatrix} w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^0 & w_4^2 \\ w_4^0 & w_4^3 & w_4^2 & w_4^1 \\ w_4^0 & w_4^0 & w_4^0 & w_4^0 \end{pmatrix}$$

发现有很多重复的值，所以可以优化。

Ⅱ.FFT

· 算法推导

对于$F(x)=a_0x^0+a_1x^1+\cdots+a_nx^n$，将x按照次数的奇偶分开可得(n为偶数)：

$$F_0(x)=a_0x^0+a_2x^1+\cdots+a_nx^{\frac{n}{2}}$$

$$F_1(x)=a_1x^0+a_3x^1+\cdots+a_{n\text{-}1}x^{\frac{n}{2}}$$

所以可得$F(x)=F_0(x^2)+x\cdot F_1(x^2)$,然后带入复数的单位根可得：

$$F(w_n^j)=F_0(w_n^{2i})+w_n^j\cdot F_1(w_n^{2i}),i<=\frac{n}{2}$$

$$F(w_n^{i+\frac{n}{2}})=F_0(w_n^{2i+n})-w_n^j\cdot F_1(w_n^{2i+n}),i<=\frac{n}{2}$$

设$\begin{matrix} M=F_0(w_n^{2i}) \\ N=w_n^i\cdot F_1(w_n^{2i}) \end{matrix}$，有$\begin{matrix} F(w_n^i)=M+N \\ F(w_n^{i+\frac{n}{2}})=M-N \end{matrix}$，所以对于每一层，假设这一层的是n次多项式，那么我们只需要代入$w_n^i\sim w_n^{i+\frac{n}{2}}$计算对应的M和N的值即可，他们只有$\frac{n}{2}$个。相当于我们把原来的函数拆成两个函数来计算，这样可以让我们代值得

时候少代很多。

不难发现，拆分第 i 次的函数需要代入 $n/2^i$ 个值，总共有 2^i 个函数，所以拆分到第 i 次需要代入的函数值仍然是 n 个。另外，很显然可以得到我们拆分 $\log_2 n$ 次后每个函数就只需要带入 1 个值了，这样就可以直接带入了。所以最后一层会直接带入 n 个函数值，每一个都可以在 $O(1)$ 的时间内算出来，时间复杂度是 $O(n)$ 的，对于其他层，每一个需要带入的值仍可以使用下层已经算出的值（上文提到的 M，N）在 $O(1)$ 的时间内，使用 $F(x)=M\pm N$ 算出，所以每一层的时间复杂度仍然是 $O(n)$ 的，又因为有 $\log_2 n$ 层，所以总体的时间复杂度为 $O(n\log_2 n)$

· 代码实现

对于第一层，有 n 个项，原本需要带 $w_n^1 \sim w_n^n$。

实际只需要带入 $w_n^1 \sim w_n^{\frac{n}{2}}$ 这些函数值。因为我们要拆成 $F(x)=F_0(x^2)+x\cdot F_1(x^2)$

所以对于 $F_0(x)$ 和 $F_1(x)$，实际上我们应该带入 $w_n^2 \sim w_n^n = w_{n/2}^1 \sim w_{n/2}^{n/2}$。

即对于第二层，有 n/2 个项，需要带入 $w_{n/2}^1 \sim w_{n/2}^{n/2}$。

不难发现，对于第 i 层的函数，有 m 个项，只需要带入 $w_m^1 \sim w_m^m$ 即可

这样我们就把每一层给统一了起来，所以代码实现就非常方便了

参考代码：http://paste.ubuntu.com/16975558/

Ⅲ.IFFT

我们回到之前提到的矩阵形式，不难发现

$$(a_0,a_1,a_2,a_3)\begin{pmatrix} w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \\ w_4^0 & w_4^4 & w_4^8 & w_4^{12} \end{pmatrix}\begin{pmatrix} \textbf{inverse} \\ \textbf{matrix} \end{pmatrix} = \begin{pmatrix} F(w_4^1) & F(w_4^2) & F(w_4^3) & F(w_4^4) \end{pmatrix}\begin{pmatrix} \textbf{inverse} \\ \textbf{matrix} \end{pmatrix}$$

$$\text{即}(a_0,a_1,a_2,a_3) = \begin{pmatrix} F(w_4^1) & F(w_4^2) & F(w_4^3) & F(w_4^4) \end{pmatrix}\begin{pmatrix} \textbf{inverse} \\ \textbf{matrix} \end{pmatrix}$$

所以我们只需要知道值矩阵对应的逆矩阵是什么即可。由复平面的性质可得：

$$w_n^i = \cos\theta + i\cdot\sin\theta$$

$$\therefore w_n^j \cdot (\cos\theta - i\cdot\sin\theta) = \cos^2\theta + \sin^2\theta = 1$$

$$又\because \sum_{j=0}^{n-1} (w_n^k)^j = 0$$

所以逆矩阵不好求，但是我们可以搞出下面这个等式：

设 $w_n^j = \cos\theta + i\cdot\sin\theta, k_n^j = \cos\theta - i\cdot\sin\theta$

$$(a_0,a_1,a_2,a_3)\begin{pmatrix} w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \\ w_4^0 & w_4^4 & w_4^8 & w_4^{12} \end{pmatrix}\begin{pmatrix} k_4^0 & k_4^1 & k_4^2 & k_4^3 \\ k_4^0 & k_4^2 & k_4^4 & k_4^6 \\ k_4^0 & k_4^3 & k_4^6 & k_4^9 \\ k_4^0 & k_4^4 & k_4^8 & k_4^{12} \end{pmatrix} = \begin{pmatrix} F(w_4^1) & F(w_4^2) & F(w_4^3) & F(w_4^4) \end{pmatrix}\begin{pmatrix} k_4^0 & k_4^1 & k_4^2 & k_4^3 \\ k_4^0 & k_4^2 & k_4^4 & k_4^6 \\ k_4^0 & k_4^3 & k_4^6 & k_4^9 \\ k_4^0 & k_4^4 & k_4^8 & k_4^{12} \end{pmatrix}$$

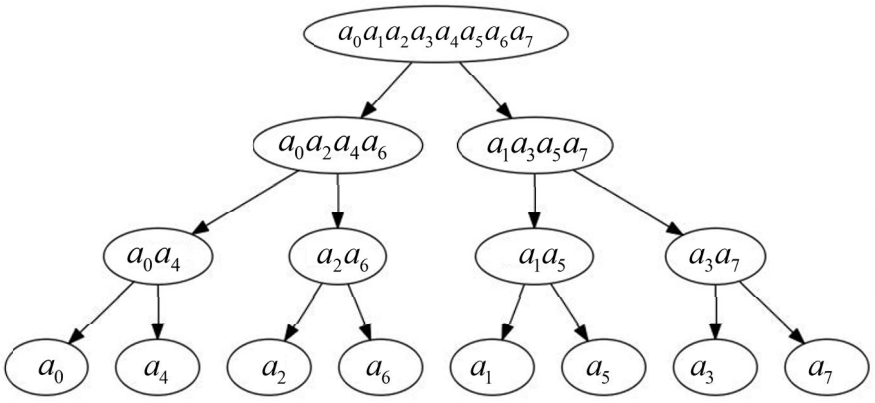
$$\therefore (a_0,a_1,a_2,a_3)\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 0 & n \end{pmatrix} = \begin{pmatrix} F(w_4^1) & F(w_4^2) & F(w_4^3) & F(w_4^4) \end{pmatrix}\begin{pmatrix} k_4^0 & k_4^1 & k_4^2 & k_4^3 \\ k_4^0 & k_4^2 & k_4^4 & k_4^6 \\ k_4^0 & k_4^3 & k_4^6 & k_4^9 \\ k_4^0 & k_4^4 & k_4^8 & k_4^{12} \end{pmatrix}$$

$$\therefore (na_0, n a_1, n a_2, n a_3) = \begin{pmatrix} F(w_4^1) & F(w_4^2) & F(w_4^3) & F(w_4^4) \end{pmatrix}\begin{pmatrix} k_4^0 & k_4^1 & k_4^2 & k_4^3 \\ k_4^0 & k_4^2 & k_4^4 & k_4^6 \\ k_4^0 & k_4^3 & k_4^6 & k_4^9 \\ k_4^0 & k_4^4 & k_4^8 & k_4^{12} \end{pmatrix}$$

我们不难发现，最后一个等式的形式已经和我们 FFT 的形式一模一样了，FFT 和 IFFT 实际上是高度统一的。在代码实现方面我们也完全可以使用 FFT 的代码，只需要把 w_n^j 的虚数部分取反即可。

Ⅳ.迭代实现

让我们先来观察一下划分完之后的数组的顺序



转换后	0	4	2	6	1	5	3	7
	000	100	010	110	001	101	011	111
原数	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111

不难发现，把原数组的每个数按照二进制反转之后就是转换后的数组[1]。那么这个是巧合吗？其实不然，FFT 按照奇偶性分组的的过程不就是按照二进制从低位到高位进行反转的过程吗？

所以我们只要将原数组进行二进制反转，这样就可以得到转换后的数组，然后一层一层迭代回去。这样就省掉了递归的过程。

但我们怎么进行二进制反转呢？直接来的话，时间复杂度是 O(n*32)的，虽然可以承受，但不优雅。观察二进制的特点，不难发现，对于 i 和 i>>1,他们有很位都是相同的。实际上，把 i>>1 反转之后的数再>>1 之后，就只有最高位的数字不一定和 i 反转后的数相同了。但我们发现，最高位是否为 1 只与 i 的最低位是否为一相关，所以我们特殊处理一下就好了。这样我们就可以使用递推的模式很优雅地在 O(n)的时间复杂度内完成转换。并且舍弃掉了递归。实测，迭代版使用的时间只有递归版的一半左右。[2]

代码参考：http://paste.ubuntu.com/17021532/

Ⅴ.算法竞赛中的应用要点

根据本人为数不多的做题经验来讲，FFT 主要用来计算卷积[3]相关的数学式子。所以对于能使用 FFT 的题目来讲，一般步骤如下：

① 将数学式子转化为函数相乘的形式

② 将函数变形，式之成为卷积形式

③ FFT 进行计算并得出结果

比较典型的例子可以参考下题[4]：http://www.lydsy.com/JudgeOnline/problem.php?id=3527

Ⅵ.复数、复平面相关知识补充[5]

① $w_n^j = w_{n/c}^{j/c}, c \in N^*$

② $(w_n^j)^2 = w_n^{2\cdot i}$

③ $w_n^j = w_n^{j\%n}$

④ $\sum_{j=0}^{n-1} (w_n^k)^j = 0$

Ⅶ.Refrence

① 退役的狗的博客：http://blog.csdn.net/iamzky/article/details/22712347

② OI 技术宅的博客：http://techotaku.lofter.com/post/4856f0_70629c6

③ 算法导论·第 30 章

[1] 只有下标从 0 开始的时候才满足该性质，否则不满足

[2] 速度测试结果不一定准确，不过从大家的经验和题目的提交情况来看，在数据较大的情况下迭代版比递归版至少快一倍，手写复数运算又比使用库函数快一倍。

[3] 算法竞赛中，卷积一般可写作如下形式：$c_i = \sum_{i=0}^{i} a_j b_{i-j}$

[4] 题面请参考：http://www.cnblogs.com/iwtwiioi/p/4126284.html

[5] 以下仅给出结论，证明请参考《算法导论》