快速数论变换 FNTT

Orange

2018 年 3 月 13 日

1 摘要

离散傅里叶变换（DFT）是解决卷积问题的有力途径，但是它涉及到了复数和三角函数等浮点数运算，这必然会产生浮点数运算常见的问题：计算速度慢，浮点误差大。

回忆使用 DFT 解决多项式乘法的过程，我们实际上是利用 n 次单位复数根的特殊性质使得整个过程可分治，最终得到了快速傅里叶变换（FFT）算法。这个特殊性质到底是什么呢？有没有代替品？这都是我们要弄清楚的问题。

本文从 FFT 的推导过程出发，在模意义下的整数域中讨论快速数论变换，并给出有效的实现。

关键词：数论变换（Number Theoretic Transform，NTT）

2 快速复习

2.1 离散傅里叶变换

简单地说，离散傅里叶变换是将在时域上长度有限且离散的信息转换到频域上，得到的频域信息仍然是离散的。

选择 DFT 来解决卷积运算问题的主要理由是 DFT 满足时域卷积定理和频域卷积定理。这样，实质为卷积的多项式乘法便能通过 DFT 将问题转换成时间复杂度为 $O(n)$ 的点积运算。

DFT 及离散傅里叶逆变换（IDFT）是整个算法的时间瓶颈，因为点积运算的时间复杂度为 $O(n)$，而朴素的 DFT 的时间复杂度为 $O(n^2)$；对应的，通过逆矩阵的方法，也可以得到朴素的 IDFT 算法，时间复杂度为 $O(n^2)$。

DFT 以及 IDFT 的公式为：

$$X_k = \sum_{j=0}^{n-1} x_j \cdot w_n^{jk} \qquad (0 \le k < n)$$

$$x_k = \frac{1}{n} \sum_{j=0}^{n-1} X_j \cdot w_n^{-jk} \qquad (0 \le k < n)$$

其中 x 代表时域序列，X 代表频域序列。

利用 n 次单位复数根的特殊性质，可以将 DFT 的时间复杂度优化为 $O(n \log n)$，IDFT 情况类似。就这样，我们得到了一个时间复杂度为 $O(n \log n)$ 的多项式乘法算法。[1]

2.2 模 m 意义下的 n 次单位根

2.2.1 定义

在模 m 意义下，对于非零元素 a，如果存在正整数 n，使得 $a^n \equiv 1 \pmod m$，则称 a 是模 m 意义下的 n 次单位根。

2.2.2 阶

对于两个互质的数 a,m，称最小的满足 $a^x \equiv 1 \pmod m$ 的 x 为 a 在模 m 意义下的阶，记作 $x = \delta_m(a)$ [2]；而称 a 是对模 m 的 n 阶本原单位根。

2.2.3 原根

由欧拉定理，不难得到 $a^{\varphi(m)} \equiv 1 \pmod m$，但是 $\varphi(m)$ 是 a 在模 m 意义下的阶吗？不一定。

如果对于 x，满足 $x^{\varphi(m)} \equiv 1 \pmod m$，那么我们称 x 为 m 的一个原根。

2.3 多项式

2.3.1 次数界

定义一个多项式的次数界为其最高项的次数加一，记为 N_A。

[1] 具体的内容可以参看离散傅里叶变换。

[2] δ 读作 delta。

2.3.2 点值运算与插值运算

点值运算指对一个次数界为 N_A 的多项式 $A(x)$ 代入至少 N_A 个不同参数 x，得到至少 N_A 个形如 $(x_j, A(x_j))$ 的点值对。这个操作得到的结果称为多项式的点值表示。

插值运算指通过至少 N_A 个点值对计算出一个次数界为 N_A 的多项式的各项系数。当点值对不足 N_A 个时，对应的多项式将会不唯一。

DFT 实质上就是一个点值运算；IDFT 实质上就是一个插值运算。

2.3.3 系数表示的多项式乘法与点值表示的多项式乘法

对两个使用系数表示的多项式做乘法，实际上是做一次两个向量的卷积，朴素计算的时间复杂度为 $O(n^2)$。

对两个使用点值表示的多项式做乘法，首先要保证点值对个数相同（设为 (x, y)），其次要保证各点值对的 x 相同，最后只需要将各个 y 对应相乘即可。时间复杂度为 $O(n)$。

3 数论变换

3.1 代数系统

为什么我们要选择运算量巨大的复数域？因为我们要进行 DFT，要使用 n 次单位复数根的几个特殊性质：[3]

$$w_n^0 = w_n^n = 1 \tag{1}$$

$$(w_n^k)^2 = w_{\frac{n}{2}}^k \tag{2}$$

$$\sum_{j=0}^{n-1} w_n^{jk} = 0 \qquad (k \neq 0) \tag{3}$$

(1) 代表的性质使得 DFT 满足循环卷积特性。

(2) 代表消去引理。结合 (1) 和 (2)，我们可以得到折半引理。

(3) 代表求和引理，可以由它推导出 DFT 的逆运算 IDFT。

已经证明，在复数域中，DFT 是唯一满足循环卷积特性的变换。

能否在别的代数系统中找到类似的变换呢？

[3]具体的内容可以参看快速傅里叶变换的相关文献。

3.2 数论变换

定义长度为 n 的序列 x，有以下变换：

$$X_k = \sum_{j=0}^{n-1} x_j a^{jk} \pmod m \qquad (0 \le k < n)$$

我们称上面的变换为数论变换（**Number-Theoretic Transform，NTT**）。

应当注意到，正如数论变换的名字，它涉及到的所有数都是在模 m 意义下的整数。

3.2.1 数论逆变换

仿照 DFT 转化为代表线性组方程的矩阵乘法的过程，我们对 NTT 也进行相同的转化：

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & a & a^2 & \cdots & a^{n-1} \\ 1 & a^2 & a^4 & \cdots & a^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a^{n-1} & a^{2(n-1)} & \cdots & a^{(n-1)^2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \equiv \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{n-1} \end{bmatrix} \pmod m$$

记作：

$$V_n x \equiv X \pmod m$$

我们要找到满足以下条件的 V_n^{-1}：

$$V_n V_n^{-1} \equiv I_n \pmod m$$

其中 I_n 为 n 阶单位矩阵。

仿照 DFT 构造逆矩阵，我们能否构造出下面这个矩阵作为 V_n^{-1}？

$$\begin{bmatrix} n^{-1} & n^{-1} & n^{-1} & \cdots & n^{-1} \\ n^{-1} & a^{-1}n^{-1} & a^{-2}n^{-1} & \cdots & a^{-(n-1)}n^{-1} \\ n^{-1} & a^{-2}n^{-1} & a^{-4}n^{-1} & \cdots & a^{-2(n-1)}n^{-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n^{-1} & a^{-(n-1)}n^{-1} & a^{-2(n-1)}n^{-1} & \cdots & a^{-(n-1)^2}n^{-1} \end{bmatrix}$$

其中 n^{-1} 表示 n 的逆元，a^{-k} 表示 a 的逆元的 k 次方。

我们只在这里证明 V_n^{-1} 为其逆矩阵的必要性。

光是 n^{-1} 就已经有难度了：对于模数 m，如果 m 不是质数，那么 n 和 a 不一定存在逆元。到现在，我们并没有对 m 和 a 作出任何限制，所以我们可以任选一个 m。为了保证存在逆元，我们强行规定 m 为一个质数，并且重新定义符号 p 代表新的模数。

现在只需要证明：

$$\sum_{k=0}^{n-1} V_{n\ (i,k)} \cdot V^{-1}_{n\ (k,j)} \equiv [i=j] \pmod p$$

即：

$$\sum_{k=0}^{n-1} a^{ki} \cdot a^{-kj} \cdot n^{-1} \equiv [i=j] \pmod p$$

左式化简得：

$$n^{-1} \cdot \sum_{k=0}^{n-1} a^{k(i-j)}$$

当 $i=j$ 时，上式显然同余 1。当 $i \neq j$ 时，如果我们有：

$$\sum_{k=0}^{n-1} a^{k(i-j)} \equiv 0 \pmod p$$

即可得证。

可以将左式看作等比数列求和，化简左式，易得：

$$\frac{1-a^{n(i-j)}}{1-a^{(i-j)}}$$

现在的问题转化为了求证：

$$a^n \equiv 1 \pmod p$$

要使上式成立，必须有：

$$a^k \not\equiv 1 \pmod p \qquad (1 \le k < n)$$

否则会有 $a^n \equiv 0 \pmod p$。

注意，a 的取值也是任选的。要使 $a^n \equiv 1 \pmod p$ 成立，等价于 a 是模 p 意义下的 n 阶本原单位根。 我们强行规定 a 为模 p 意义下的 n 阶本原单位根，并且重新定义符号 g 代表新的底数，则原命题成立。

3.2.2 p 和 g 的取值

先明确目前我们对 p 和 g 的要求。目前，p 是一个质数，而 g 是 p 的 n 阶本原单位根。

我们有：

$$g^n \equiv 1 \pmod p$$

由 n 阶本原单位根的定义和二次探测定理，有：

$$g^{\frac{n}{2}} \equiv -1 \pmod p$$

如果给定一个 n，事实上直接求出 p 的 n 阶本原单位根并不是一个容易的事，但我们可以考虑使用原根：

$$a^{p-1} \equiv 1 \pmod p$$
$$a^i \not\equiv a^j \pmod p \qquad (0 \leq i < j < p-1)$$

如果我们有：

$$p - 1 = k \cdot n$$

令 $g_n \equiv a^k \pmod p$，则 g_n 就是 p 的一个 n 阶本原单位根。所以我们可以先计算出 p 的一个原根 g，再令 $g_n = g^{\frac{p-1}{n}}$ 作为底数。

注意到，n 可能是一定数据范围内的任意取值，所以要使 $p - 1 = k \cdot n$ 对于任意的 n 都满足是不可能的。

对比快速傅里叶变换，我们要求 $n = 2^t$ 是为了让问题可以不断分治。这里我们也需要用到这个思路。我们强制规定 $n = 2^t$，如果原长度不足 2^t，则补 0 至 n 位。

现在 $n = 2^t$，如果 $p-1 = k \cdot 2^t$，问题就变得很容易了。所以我们强制规定 $p = k \cdot 2^t+1$，这能够解决所有 $n \leq 2^t$ 规模的问题。

3.2.3 小结数论逆变换

明确目前 p 和 g 的要求。目前，$p = k \cdot 2^t + 1$ 且为质数，$g = k \cdot 2^{t'}$，能够解决 $n = \frac{2^t}{2^{t'}}$ 问题。

再次明确为什么要这么规定 p 和 g 的取值：令 p 为质数的原因是保证存在 n^{-1} 且保证存在原根。令 g 为这个值的原因是保证 g 是 p 的一个 n 阶本原单位根，使得 NTT 的逆运算 INTT 存在。

称下列变换为数论变换和数论逆变换（Inverse Number Theoretic Transform，INTT）：

$$X_k = \sum_{i=0}^{n-1} x_i \cdot g^{ik} \pmod p$$
$$x_k = n^{-1} \sum_{i=0}^{n-1} X_i \cdot g^{-ik} \pmod p$$

其中 $p = k \cdot 2^t + 1$ 且为质数，$n = 2^{t'}$，$g = k \cdot 2^{t-t'}$。

3.3 快速数论变换

容易得到，直接进行 NTT 和 INTT 的时间复杂度为 $O(n^2)$。我们可以用类似快速傅里叶变换的方法，进行快速数论变换（Fast Number Theoretic Transform，FNTT）。

3.3.1 循环卷积特性和三个引理

在快速傅里叶变换中，我们用到了 n 次单位复数根带来的循环卷积特性和三个引理：

1. 消去引理
2. 折半引理
3. 求和引理

我们实际上已经证明了求和引理。不难证明，n 阶本原单位根也会给 NTT 带来循环卷积特性：

$$g_n^0 \equiv g_n^n \equiv 1 \pmod p$$

其中 g_n 为 n 对应的 n 阶本原单位根。

下面让我们来证明消去引理和折半引理。($p = k \cdot 2^t + 1$，$n = 2^{t-1}$，$g_n = k \cdot 2^{t-t'}$，设 g 为 p 的一个原根)

对于 g_n^d $(d \bmod 2 = 0)$，有：

$$g_n^d \equiv g^{d(k \cdot 2^{t-t'})} \equiv g^{\frac{d}{2}(k \cdot 2^{t-t'+1})} \equiv g_{\frac{n}{2}}^{\frac{d}{2}} \pmod p$$

即证明了消去引理。

我们有：

$$g_n^k \equiv g_n^{k+n} \pmod p$$

对于 $g_{\frac{n}{2}}^{k}\ (0 \le k < n)$，由上式，易得在模 p 意义下只有 $\frac{n}{2}$ 个不同的数。即证明了折半引理。

3.3.2 快速数论变换

能够实现快速傅里叶变换，得益于循环卷积特性和三个引理。现在在模 p 意义下，循环卷积特性和三个引理也得证了。能否用相同的方法推导出快速数论变换呢？答案是肯定的。

下面我们用 $=$ 代替 $\equiv$，并且省略模数。设原多项式：

$$X_k = x_0 + x_1 \cdot g_n^k + x_2 \cdot g_n^{2k} + \cdots + x_{n-1} \cdot g_n^{(n-1)k}$$

可以分解成两个多项式的线性变换：

$$\begin{gathered} X_k^{[0]} = x_0 + x_2 \cdot g_n^{2k} + \cdots + x_{n-2} \cdot g_n^{(n-2)k} \\ X_k^{[1]} = x_1 + x_3 \cdot g_n^{2k} + \cdots + x_{n-1} \cdot g_n^{(n-2)k} \\ X_k = X_k^{[0]} + g_n^k \cdot X_k^{[1]} \end{gathered}$$

由消去引理，两个多项式分别为：

$$\begin{gathered} X_k^{[0]} = x_0 + x_2 \cdot g_{\frac{n}{2}}^{k} + \cdots + x_{n-2} \cdot g_{\frac{n}{2}}^{\frac{n-2}{2}k} \\ X_k^{[1]} = x_1 + x_3 \cdot g_{\frac{n}{2}}^{k} + \cdots + x_{n-1} \cdot g_{\frac{n}{2}}^{\frac{n-2}{2}k} \end{gathered}$$

当 k 取值 $0 \le k < n$ 时，可以发现，有：

$$\begin{gathered} X_{k'}^{[0]} = X_{k'+\frac{n}{2}}^{[0]} \\ X_{k'}^{[1]} = X_{k'+\frac{n}{2}}^{[1]} \end{gathered} \qquad (0 \le k' < \frac{n}{2})$$

代入原多项式，可以发现：

$$\begin{gathered} X_{k'} = X_{k'}^{[0]} + g_n^{k'} \cdot X_{k'}^{[1]} \\ X_{k'+\frac{n}{2}} = X_{k'+\frac{n}{2}}^{[0]} + g_n^{k'+\frac{n}{2}} \cdot X_{k'+\frac{n}{2}}^{[1]} = X_{k'}^{[0]} - g_n^{k'} \cdot X_{k'}^{[1]} \end{gathered} \qquad (0 \le k' < \frac{n}{2})$$

至此，可由主定理得时间复杂度为 $O(n \log n)$。

相应的，INTT 也使用类似的方式进行转化，具体内容不在阐述。

3.3.3 实现

由于我们已经知道了 FFT 的实现，所以完全可以照搬 FFT 的过程：**FNTT 与 FFT** 在实现上差距并不大。这里直接给出代码：

```
struct FNTT
{
    static const int mod = 998244353;
    static const int root = 3;
    static const int base = 119;
    static const int limit = 23;
    static inline int power(int x, int y)
    {
        int ret = 1;
        while (y)
        {
            if (y & 1) ret = (long long)ret * x % mod;
            x = (long long)x * x % mod;
            y >>= 1;
        }
        return ret;
    }
    int n, logn;
    inline int revbit(int x)
    {
        int ret = 0;
        for (int i = 0; i < logn; i++)
            ret = (ret << 1) | (bool)(x & (1 << i));
        return ret;
    }
    FNTT(int* a, int logn, int sig) : n(1 << logn), logn(logn)
    {
        for (int i = 0; i < n; i++)
        {
            int t = revbit(i);
            if (i < t) std::swap(a[i], a[t]);
        }
        for (int i = 1; i <= logn; i++)
        {
            int S = 1 << i;
            int half = S >> 1;
            int g1 = power(root, base * (1 << (23 - i)));
```

```
            if (sig == -1) g1 = power(g1, mod - 2);
            for (int j = 0; j < n; j += S)
            {
                int* A = a + j;
                int g = 1;
                for (int k = 0; k < half; k++)
                {
                    int t = (long long)A[k + half] * g % mod;
                    A[k + half] = ((A[k] - t) % mod + mod) % mod;
                    A[k] = (A[k] + t) % mod;
                    g = (long long)g * g1 % mod;
                }
            }
        }
    }
};
```

4 小结

NTT 是在模意义下进行的运算，它的主要作用还是计算多项式乘法。若多项式乘法的结果保证为小于模数的非负整数，则可以使用 NTT 代替 FFT；如果题目要求对一个适用于 NTT 的模数取模，则只能使用 NTT。

有时候，受代码常数影响，NTT 比 FFT 运行得要慢一点，但是整体上 NTT 运行得要快一点，且内存占用比 FFT 少得多。

NTT 不受浮点误差影响。

5 附表：部分质数与其对应的原根

$$p = k \cdot 2^t + 1$$

k 和 g[4] 均对实现没有影响。t 决定了能够解决的问题的最大规模。

p	k	t	g
998244353	119	23	3
1004535809	479	21	3

如果题目给定的模数不是一个满足要求的质数，就需要做这么一个思维转化：既然题目要求求模，那就可以保证所有的数小于 m，两个数的乘积一定小于 $(m-1)^2$，而卷积的结果就一定小于 $n(m-1)^2$。如果 NTT 的模数是一个符合条件的大于 $n(m-1)^2$ 的质数，问题也就迎刃而解了。

但是可能很难找到这样的质数，找到了在计算机中也存不下。所以需要找 b 个符合条件的质数，使得它们的乘积大于等于 $n(m-1)^2$，最后使用中国剩余定理合并。

6 参考文献

Miskcoo，从多项式乘法到快速傅里叶变换，2015.

[4] g 为 p 的原根。