快速傅里叶变换实现多项式乘法

原创 梁宇航 LOA算法学习笔记 2021-02-03 16:15

01 问题描述

输入两个多项式的系数表示(低幂到高幂) $A=[a_0,a_1,\ldots,a_{n-1}]$ ， $B=[b_0,b_1,\ldots,b_{m-1}]$ ，输出这两个多项式乘积的系数表示。
两个多项式的乘积

$$
\begin{aligned}
&A(x)=a_0+a_1x+\ldots+a_{n-1}x^{n-1}\\
&B(x)=b_0+b_1x+\ldots+b_{m-1}x^{m-1}\\
&C(x)=A(x)B(x)=c_0+c_1x+c_2x^2+\ldots+c_{m+n-1}x^{m+n-1}
\end{aligned}
$$

其中 c_k 由卷积表达式给出

$$c_k=\sum_{i=0}^{k}a_ib_{k-i},k=0,1,\ldots,m+n-1$$

如果直接遍历 $A(x),B(x)$ 时间复杂度是 $O(mn)$ ，这在 m,n 数值比较接近且比较大时非常消耗时间，下面考虑一种新的高效的方法——快速傅里叶变换(FFT)。
离散傅里叶变换（DFT）是来计算多项式在 n 个特殊点(单位根)的值。而**快速傅里叶变换（FFT）**是一种快速有效率的对DFT的实现。FFT 加速多项式乘法，其基本思想是将两个多项式的系数表示通过 FFT 转化为特殊点处的点值表示，然后计算两个多项式点值表示的乘积得到原多项式卷积的点值表示，再将多项式卷积的点值表示进行**逆离散傅里叶变换（IDFT）**就得到了乘积多项式的系数表示。

02 多项式的表示

一个多项式一般有有两种表示方法，系数表示法和点值表示法。系数表示法：

$$f(x)=a_0+a_1x+a_2x^2+\ldots+a_{n-1}x^{n-1},a_{n-1}\neq 0$$

点值表示法：

$$f(x)=\{(x_0,y_0),(x_1,y_1),\ldots,(x_{n-1},y_{n-1})\},x_i=(x_i^0,x_i^1,\ldots,x_i^{n-1})$$

并且当 $i\neq j$ 时， $x_i\neq x_j$ ，这样一个 $n-1$ 次多项式就由 n 个不同点处的取值唯一确定。证明如下，在一个点处我们有：

$$\begin{pmatrix} x_i^0 & x_i^1 & \dots & \dots & x_i^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ \dots \\ a_{n-1} \end{pmatrix} = y_i$$

在n个点处：

$$\begin{pmatrix} x_0^0 & x_0^1 & x_0^2 & \dots & x_0^{n-1} \\ x_1^0 & x_1^1 & x_1^2 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \dots \\ \dots \\ y_{n-1} \end{pmatrix}$$

当点互不相同时，对于未知数 a_i 而言，其系数矩阵为范德蒙矩阵，必可逆，故有唯一解，这唯一确定了多项式系数也即唯一确定了多项式。

03 FFT

在复数域内考虑方程 $x^n = 1$ ，由代数学知识我们知道，它有n个根，分别为 $w_n^k = e^{2\pi \cdot \frac{k}{n}}, k = 0, 1, 2, \dots, n-1$ ，其中由Euler公式， $w_n^k = e^{2\pi \cdot \frac{k}{n}} = cos(2\pi \cdot \frac{k}{n}) + i \cdot sin(2\pi \cdot \frac{k}{n})$ 。下面来看一下关于 w_n^k 的重要性质，这些性质可以利用Euler公式经过简单计算轻易得到：

$$w_n^k = w_n^{k-1} \cdot w_n^1 \tag{1}$$
$$w_n^0 = w_n^n = 1 \tag{2}$$
$$w_{2n}^{2k} = w_n^k = w_{mn}^{mk} \tag{3}$$
$$w_n^{k+\frac{n}{2}} = -w_n^k \tag{4}$$
$$w_n^{2k} = w_{\frac{n}{2}}^k \tag{5}$$

$$\sum_{j=0}^{n-1} w_n^{jk} = \begin{cases} 0, & k \, mod \, n \neq 0 \\ n, & k \, mod \, n = 0 \end{cases}$$

这是因为：

$$\sum_{j=0}^{n-1} \omega_n^{jk} = \frac{\omega_n^0 \left(1 - \left(\omega_n^k\right)^n\right)}{1 - \omega_n^k} = \frac{1 - \left(\omega_n^n\right)^k}{1 - \omega_n^k}, k \, mod \, n \neq 0$$

DFT 就是要求 $f(x)=a_0+a_1x+a_2x^2+\ldots+a_{n-1}x^{n-1}$ 在上述n个单位根处的取值。如果是朴素的DFT，使用Horner方法：

$$f(x)=a_0+x(a_1+x(a_2+\ldots+xa_{n-1})$$

时间复杂度将仍是 $O(n^2)$ 。但是现在是在一些特殊的点（ $x^n=1$ 的n个单位根），利用特殊性可以将时间复杂度降低到 $O(nlogn)$ ，这就是 FFT。其思想如下，考虑在单位根处的点值表示：

$$\begin{cases} f(\omega_n^0)=a_0+a_1(\omega_n^0)^1+a_2(\omega_n^0)^2+\ldots+a_{n-1}(\omega_n^0)^{n-1} \\ f(\omega_n^1)=a_0+a_1(\omega_n^1)^2+a_2(\omega_n^1)^2+\ldots+a_{n-1}(\omega_n^1)^{n-1} \\ \ldots \\ f(\omega_n^{n-1})=a_0+a_1(\omega_n^{n-1})^1+a_2(\omega_n^{n-1})^2+\ldots+a_{n-1}(\omega_n^{n-1})^{n-1} \end{cases}$$

现在考察 $f(x)$ ，将奇数项与偶数项分开：

$$\begin{aligned} f(x)&=a_0+a_1x+a_2x^2+\ldots+a_{n-1}x^{n-1} \\ &=(a_0+a_2x^2+\ldots+a_{n-2}x^{n-2})+ \\ x&(a_1+a_3x^2+\ldots+a_{n-1}x^{n-2}) \end{aligned}$$

令

$$\begin{aligned} f_1(x)&=(a_0+a_2x+\ldots+a_{n-2}x^{n-2/2}) \\ f_2(x)&=(a_1+a_3x+\ldots+a_{n-1}x^{n-2/2}) \end{aligned}$$

则可得

$$f(x)=f_1(x^2)+xf_2(x^2)$$

于是

$$\begin{aligned} f(\omega_n^k)&=f_1(\omega_n^{2k})+\omega_n^k\cdot f_2(\omega_n^{2k})=f1\left(\omega_{\frac{n}{2}}^k\right)+\omega_n^k\cdot f_2\left(\omega_{\frac{n}{2}}^k\right) \\ f\left(\omega_n^{k+\frac{n}{2}}\right)&=f_1(\omega_n^{2k+n})+\omega_n^{k+\frac{n}{2}}\cdot f_2(\omega_n^{2k+n}) \\ &=f\left(\omega_n^{k+\frac{n}{2}}\right)=f_1\left(\omega_{\frac{n}{2}}^k\right)-\omega_n^k\cdot f_2\left(\omega_{\frac{n}{2}}^k\right) \end{aligned}$$

可以看到计算 f_1 和 f_2 各自只需要 f 规模的一半计算量即可得到。这就厉害了，利用 Divide and Conquer思想，如果我们已经知道 f_1 和 f_2 分别在 $w_{n/2}^0, w_{n/2}^1, w_{n/2}^2, \ldots, w_{n/2}^{n/2-1}$ 处的值，可以在常数时间求得 $f(x)$ 的值。所以总的时间复杂度为 $T(n) = 2T(n/2) + O(n) = O(nlogn)$ ，其中 $O(n)$ 是每次分治求对应单位根花费的时间。

04 IDFT

将多项式从系数表示转化为点值表示后，如何将其变回来？由

$$\begin{pmatrix} x_0^0 & x_0^1 & x_0^2 & \ldots & x_0^{n-1} \\ x_1^0 & x_1^1 & x_1^2 & \ldots & x_1^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \ldots \\ \ldots \\ y_{n-1} \end{pmatrix}$$

只需要两边同称范德蒙矩阵 X 的逆。对一个普通的范德蒙矩阵求逆，时间复杂度是 $O(n^3)$ ，但是现在我们有一个特殊的范德蒙矩阵：

$$W = \begin{pmatrix} (w_n^0)^0 & (w_n^0)^1 & (w_n^0)^2 & \ldots & (w_n^0)^{n-1} \\ (w_n^1)^0 & (w_n^1)^1 & (w_n^1)^2 & \ldots & (w_n^1)^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ (w_n^{n-1})^0 & (w_n^{n-1})^1 & (w_n^{n-1})^2 & \ldots & (w_n^{n-1})^{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & w_n^{1\times1} & w_n^{1\times2} & \ldots & w_n^{1\times(n-1)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & w_n^{(n-1)\times1} & w_n^{(n-1)\times2} & \ldots & w_n^{(n-1)\times(n-1)} \end{pmatrix}$$

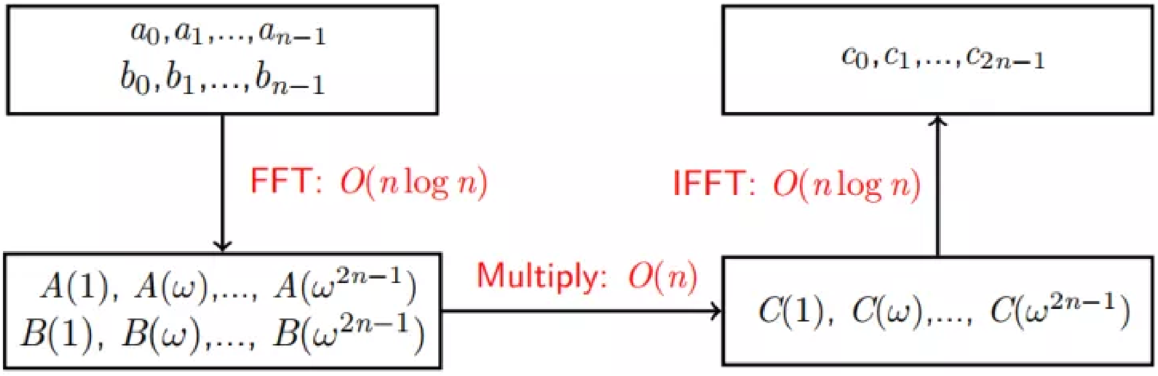
则

$$W^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & w_n^{-1\times1} & w_n^{-1\times2} & \ldots & w_n^{-1\times(n-1)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & w_n^{-(n-1)\times1} & w_n^{-(n-1)\times2} & \ldots & w_n^{-(n-1)\times(n-1)} \end{pmatrix}$$

即原矩阵每个元素的共轭再除以n，下面给一个计算性证明，设上述两个矩阵的乘积矩阵为 M ，则

$$M_{ij} = \sum_{k=0}^{n-1} \omega_n^{i\cdot k} \cdot \frac{\omega_n^{-k\cdot j}}{n} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{k(i-j)} = \begin{cases} 0, i \neq j \\ 1, i = j \end{cases}$$

可见两矩阵的乘积确为单位阵。这样我们就可以反求出系数表示。整个过程如下图所示



注意：实际为了处理方便会将多项式项数统一为 2 的幂次个。

05 代码

```
import numpy as np
from decimal import Decimal
# 单点处离散傅里叶变换
def singleDFT(xn):
    N = xn.shape[0]
    j = np.arange(N)
    k = j.reshape((N, 1))
    w = np.exp(-2j * np.pi * k * j / N)
    return np.dot(w, xn)
# 单点处逆离散傅里叶变换
def singleIDFT(xk):
    N = xk.shape[0]
    j = np.arange(N)
    k = j.reshape((N, 1))
    w = np.exp(2j * np.pi * k * j / N)
    return 1 / N * np.dot(w, xk)
# FFT 过程
def FFT(xn):
    N = xn.shape[0]
    AX = singleDFT(xn=xn.reshape((N, -1)))
    while AX.shape[0] < N:
        AX_even = AX[:, :int(AX.shape[1] / 2)]  # 偶数项多项式
        AX_odd = AX[:, int(AX.shape[1] / 2):]  # 奇数项多项式
        w = np.exp(-2j * np.pi * np.arange(AX.shape[0]) / AX.shape[0] / 2)
        w = w.reshape((w.shape[0], 1))
        w_AX_odd = w * AX_odd
        AX = np.vstack([AX_even + w_AX_odd, AX_even - w_AX_odd])
    return AX.ravel()
# IFFT 过程
def IFFT(xk):
    N = xk.shape[0]
    AX = N * singleIDFT(xk=xk.reshape((N, -1)))
    while AX.shape[0] < N:
```

```
            AX_even = AX[:, :int(AX.shape[1] / 2)]
            AX_odd = AX[:, int(AX.shape[1] / 2):]
            w = np.exp(2j * np.pi * np.arange(AX.shape[0]) / AX.shape[0] / 2)
            w = w.reshape((w.shape[0], 1))
            w_AX_odd = w * AX_odd
            AX = np.vstack([AX_even + w_AX_odd, AX_even - w_AX_odd])
        return 1 / N * AX.ravel()
# 快速傅里叶变换实现多项式乘法
def FFT_polymul(x1, x2):
    '''
    传入列表x1, x2: 多项式的系数表示，低幂到高幂(a_0+a_1x+a_2x^2+a_n-1x^n-1)
    '''
    # 将两个多项式的项数补全为2的幂次
    poly = 1
    while 2 ** poly < len(x1) + len(x2):
        poly += 1
    N = 2 ** poly


    A = FFT(xn=np.array(x1 + [0] * (N - len(x1))))
    B = FFT(xn=np.array(x2 + [0] * (N - len(x2))))
    C = A * B
    D = IFFT(xk=C)
    res = []
    for i in range(D.shape[0]):
        res.append(round(Decimal(D[i].real)))
    return res # 返回系数表示(低幂到高幂)


x1 = [0, 1, 2, 3, 4, 6, 9]
x2 = [5, 6, 7, 8]
r = FFT_polymul(x1=x1, x2=x2)
print(r)
```