Fast Fourier Transformation

2017-02-17

Preface

在计算生成函数等多项式的乘积或者进行大整数乘法时，计算答案的过程常常成为算法时间的瓶颈，然而接下来介绍的快速傅里叶变换 (Fast Fourier Transformation, FFT) 可以在 $O(nlogn)$ 的时间内处理出问题的答案。

Preparatory knowledge

1. 多项式的两种表示: 系数表示法和点值表示法，这个不多说…
2. 一些简单的复数姿势:

主要是单位复数根，定义复平面上满足 $w^n = 1$ 的复数 w 为 n 次单位复数根，根据复数乘法模长相乘，幅角相加的特点，不难发现一些性质：

a) n 次单位复数根在复平面上是对称分布的，且在单位圆上构成一个正 n 边形。

b) 次单位复数根构成一个乘法群，且生成元的个数为 $\phi(n)$。

c) 这样复平面上的单位复数根也可以用更一般的形式来表示了:

$$w_n^k = e^{\frac{2ki\pi}{n}} k = 0, 1, 2n - 1$$

欧拉公式 $e^{ki} = sin(k) + icos(k)$, 利用这个可以证明许多结论。

Algorithm

将待乘的两个多项式用 $O(nlogn)$ 时间转成点值形式，将点值相乘得到新多项式的点值，然后 $O(nlogn)$ 时间转换成系数表示即可。

考虑如何快速将系数表示转化成点值表示及其逆过程, 这时候刚才提到的复数知识就变得很有用了。

因为

$$w_n^{2k} = e^{\frac{2ki\pi}{n}} = e^{\frac{ki\pi}{\frac{n}{2}}} = w_{\frac{n}{2}}^k$$

$$w_n^{k+\frac{n}{2}} = w_n^k w_n^{\frac{n}{2}} = -w_n^k$$

所以对于 $(k < \frac{n}{2})$:

$$\begin{aligned} A(w_n^k) &= \sum_{i=0}^{n-1} a_i w_n^{ki} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} w_n^{2ki} + w_n^k \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w_n^{2ki} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} w_{\frac{n}{2}}^{ki} + w_n^k \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w_{\frac{n}{2}}^{ki} \end{aligned}$$

$$\begin{aligned} A(w_n^{k+\frac{n}{2}}) &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} w_{\frac{n}{2}}^{ki} + w_n^{k+\frac{n}{2}} \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w_{\frac{n}{2}}^{ki} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} w_{\frac{n}{2}}^{ki} - w_n^k \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w_{\frac{n}{2}}^{ki} \end{aligned}$$

这样原问题就成功地被划分为两个规模减半的子问题，即对于两部分分别做奇偶系数的 n/2 次单位根求点值，于是可以递归求解。

接下来考虑如何求解逆过程，如果我们把之前求出的点值表示当成一个线性方程组的形式，考虑如下矩阵:

$$\begin{bmatrix} (w_n^{-0})^0 & (w_n^{-0})^1 & \cdots & (w_n^{-0})^{n-1} \\ (w_n^{-1})^0 & (w_n^{-1})^1 & \cdots & (w_n^{-1})^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ (w_n^{-(n-1)})^0 & (w_n^{-(n-1)})^1 & \cdots & (w_n^{-(n-1)})^{n-1} \end{bmatrix}$$

发现这个矩阵与原矩阵的乘积恰好为 nI, 所以这个矩阵乘以 $\frac{1}{n}$ 的矩阵与原矩阵互逆

所以求逆的过程可以把所有单位复数根的指数取反，执行相同的操作然后将所得到的值除以 n 即可。

Code

```
void FFT(complex<double> x[], int n, int type) {
    if(n == 1) return;
    complex<double> l[n >> 1], r[n >> 1];
    for(int i = 0; i < n; i += 2) {
        l[i >> 1] = x[i];
        r[i >> 1] = x[i + 1];
    }
    FFT(l, n >> 1, type);
    FFT(r, n >> 1, type);
    complex<double> wn(cos(type*2*PI/n), sin(type*2*PI/n)), w(1, 0);
    for(int i = 0; i < (n >> 1); w *= wn, i++) {
        x[i] = l[i] + w * r[i];
        x[i + (n >> 1)] = l[i] - w * r[i];
    }
}
```

由于函数传参数组，常数很大，下面介绍一种迭代实现的版本，常数更小。

Some details

通过观察我们发现对于一个系数 a_i，它最后一层到达的位置和 i 的二进制为有关，我们从上往下观察，对于 i，如果在第一层它被选到左边，则说明它二进制的最末位是 0，否则是 1。然后考虑第二层，与第一层类似，考虑 i 的二进制倒数第二位即可。

不难发现假定 i 最终到达的位置为 Rev[i]，则 i 与 Rev[i] 的二进制是互逆的。

```
void init() {
    int dis = 0;
    for(base = 1; base <= n + m; base <<= 1) ++dis;
    for(int i=0; i<=base; i++)
        rev[i] = (rev[i>>1] >> 1) | ((i & 1) << (dis - 1));
}
```

那么这样有什么好处呢?

于是我们可以直接算出最后一层每一个系数所在的位置，然后往回迭代计算，之后的计算过程与递归版本类似，将系数序列分段计算即可。

```
void FFT(complex<double> x[], int n, int type) {
    for(int i=0; i<n; ++i)
        if(i < rev[i]) swap(x[i], x[rev[i]]);
    for(int s=1; 1 << s <= n; ++s) {
        int m = 1 << s;
        complex<double> wm(cos(type*2*PI/m), sin(type*2*PI/m));
        for(int k=0; k < n; k += m) {
            complex<double> w(1, 0);
            for(int j=0; j < m >> 1; ++j) {
                complex<double> u = x[k + j];
                complex<double> t = x[k + j + m / 2] * w;
                x[k + j] = u + t;
                x[k + j + m / 2] = u - t;
                w *= wm;
```

```
            }
        }
    }
}
```