快速傅里叶变换初步

SGColin

目录

1 多项式 **2**

1.1 表示法 2

1.2 系数表示法下的基本运算 2

1.3 点值表示法下的基本运算 2

2 Fast Fourier Transform **3**

2.1 复数 3

2.1.1 代数意义下复数的运算 4

2.1.2 几何意义下复数乘法的性质 4

2.2 单位根 4

2.2.1 表示法 4

2.2.2 单位根的性质 4

2.3 计算过程 5

2.3.1 Discrete Fourier Transform 5

2.3.2 Inverse Discrete Fourier Transform 5

2.3.3 Butterfly Diagram 6

2.4 代码实现 6

3 Fast Number-Theoretic Transform **7**

3.1 原根 7

3.2 代码实现 7

1 多项式

1.1 表示法

次数界：我们称最高次项为 n 次的多项式的次数界为 $n+1$ 。

系数表示 (coefficient representation)：将多项式写作 $A(x)=\sum_{i=0}^{n-1}a_ix^i$，以系数形式表示的，将 n 次多项式 $A(x)$ 的系数 $a_0,a_1,\dots,a_n$ 看作 $n+1$ 维向量 $(a_0,a_1,\dots,a_n)$ 的表示方法。

点值表示 (point-value representation)：选取 $n+1$ 个不同的数 $x_0,x_1,\dots,x_n$ 对多项式进行求值，用集合 $\{(x_i,A(x_i)):0\le i\le n,i\in Z\}$ 来表示多项式的表示方法。

多项式 $A(x)$ 的点值表示不止一种，选取不同的数就可以得到不同的点值表示。但是系数表示和任何一种点值表示都能**唯一确定一个多项式**。

从系数表示转换为点值表示的运算称为**求值**，从点值表示转换为系数表示的运算称为**插值**。

1.2 系数表示法下的基本运算

加法：系数直接相加，复杂度 $\mathrm{O}(n)$。

$$A(x)=\sum_{i=0}^{n-1}a_ix^i\ ,\ B(x)=\sum_{i=0}^{n-1}b_ix^i\ ,\ A(x)+B(x)=\sum_{i=0}^{n-1}(a_i+b_i)x^i$$

乘法：也叫**卷积 (convolution)**，两个次数界为 n 的多项式相乘得到次数界为 $2n-1$ 的多项式。暴力复杂度 $\mathrm{O}(n^2)$，使用 FFT 优化到 $\mathrm{O}(n\log n)$。

$$C(x)=A(x)\times B(x)\ ,\ C(x)=\sum_{i=0}^{n-1}c_ix^i$$

$$A(x)=\sum_{i=0}^{n-1}a_ix^i\ ,\ B(x)=\sum_{i=0}^{n-1}b_ix^i\ ,\ c_i=\sum_{k=0}^{i}a_k\times b_{i-k}$$

1.3 点值表示法下的基本运算

此时对于运算相关的多项式有特殊要求，即其点值表示应使用相同的 x 集去计算。

加法：所求值直接相加，复杂度 $\mathrm{O}(n)$。

$$A(x)=\{(x_1,y_{(a,1)}),(x_2,y_{(a,2)}),...,(x_{n+1},y_{(a,n+1)})\}\ ,\ B(x)=\{(x_1,y_{(b,1)}),(x_2,y_{(b,2)}),...,(x_{n+1},y_{(b,n+1)})\}$$

$$A(x)+B(x)=\{(x_1,y_{(a,1)}+y_{(b,1)}),(x_2,y_{(a,2)}+y_{(b,2)}),...,(x_{n+1},y_{(a,n+1)}+y_{(b,n+1)})\}$$

乘法：所求值直接相乘，复杂度 $\mathrm{O}(n)$，此处复杂度的优越性有着重要意义。

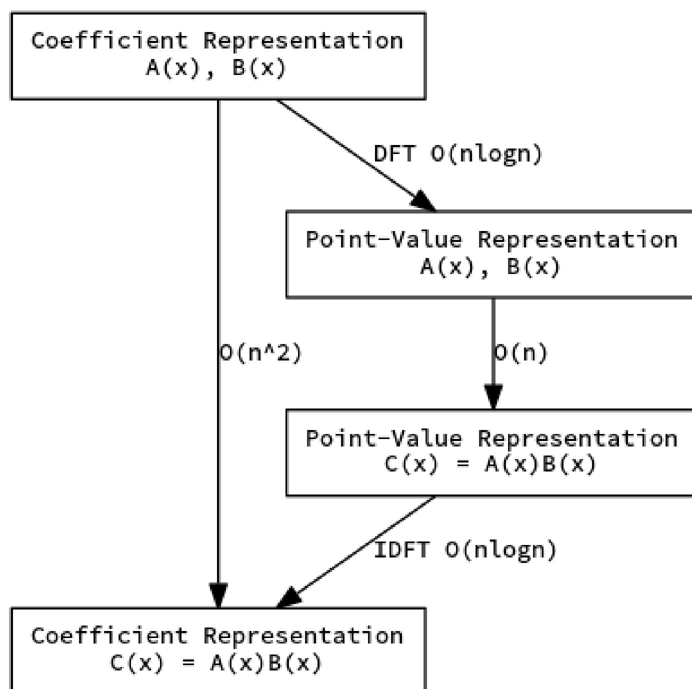
$$A(x)=\{(x_1,y_{(a,1)}),(x_2,y_{(a,2)}),...,(x_{n+1},y_{(a,n+1)})\}\ ,\ B(x)=\{(x_1,y_{(b,1)}),(x_2,y_{(b,2)}),...,(x_{n+1},y_{(b,n+1)})\}$$

$$A(x)+B(x)=\{(x_1,y_{(a,1)}\times y_{(b,1)}),(x_2,y_{(a,2)}\times y_{(b,2)}),...,(x_{n+1},y_{(a,n+1)}\times y_{(b,n+1)})\}$$

2 Fast Fourier Transform

观察多项式的乘法，可以发现系数表示法下乘法复杂度为 $O(n^2)$，而点值表示法下复杂度为 $O(n)$。如果我们把两个相乘的多项式都先求值出点值表示，然后再在点值表示法下进行乘法，再插值回去，复杂度可能会更优秀。其复杂度瓶颈显然是求值和差值的过程。

快速傅里叶变换 (Fast Fourier Transform) 利用了单位根的性质加速了求值和插值的过程，使得复杂度优化到 $\mathrm{O}(n\log n)$ 。求值的部分称为**离散傅里叶变换 (Discrete Fourier Transform)**，插值的部分称为**逆离散傅里叶变换 (Inverse Discrete Fourier Transform)**。



2.1 复数

复数域常用 $\mathbb{C}$ 表示。

定义 $i^2=-1$ ，复数写作 $z=a+ib$ 的形式。a 称作实部，b 称作虚部，i 称作虚部单位。

容易看出，复数实际上是一个二元组，因此我们可以把它映射到一个二维平面上，横坐标为实部，纵坐标为虚部。这个平面我们称作复平面。

因此复数的模长 l 即为 $\sqrt{a^2+b^2}$，记作 $|z|$。当点 z 不是原点，即复数 $z\neq 0$ 时，向量与 x 轴正向的夹角称为复数 z 的辐角，记作 $Argz$。

在复平面上复数 z 显然可以表示成 $z=|z|(\cos(Argz)+i\sin(Argz))$ ，称为复数的三角表示。

2.1.1 代数意义下复数的运算

设 $z_1 = a_1 + ib_1$ ，$z_2 = a_2 + ib_2$ ，有：

$z_1 + z_2 = (a_1 + a_2) + i(b_1 + b_2)$

$z_1 \times z_2 = (a_1a_2 - b_1b_2) + i(a_1b_2 + b_1a_2)$

2.1.2 几何意义下复数乘法的性质

现在考虑几何意义下复数的乘法，将两个三角表示的复数相乘。

$$\begin{aligned} z_1 &= l_1(\cos(\theta) + i\sin(\theta)) \text{ , } l_1 = |z_1| \\ z_2 &= l_2(\cos(\phi) + i\sin(\phi)) \text{ , } l_2 = |z_2| \\ z_1z_2 &= l_1l_2((\cos(\theta)\cos(\phi) - \sin(\theta)\sin(\phi)) + i(\cos(\theta)\sin(\phi) + \sin(\theta)\cos(\phi))) \\ &= l_1l_2(\cos(\theta+\phi) + i\sin(\theta+\phi)) \end{aligned}$$

其中用到了三角函数的两个恒等式。可以发现三角表示下，满足相乘时**模长相乘，幅角相加**。

2.2 单位根

考虑 $z^n = 1$ 的复数解，解有 n 个，第 k 个解可以表示为

$$z_k = \cos\left(\frac{2\pi k}{n}\right) + i\sin\left(\frac{2\pi k}{n}\right)$$

我们称这些复数解为 n 次单位根，显然解一共有 n 个，它们把单位圆等分成 n 个部分。

2.2.1 表示法

我们用 ω 来表示这些单位根，定义 $\omega_n^k = z_k$。

我们称 $\omega_n = \omega_n^1 = z_1 = \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right)$ 为**主 n 次单位根**。

显然 n 次单位根的模长都为 1，同时相邻两个单位根的夹角相同，我们不难得出单位根的倍数关系 $\omega_n^i = \omega_n * \omega_n^{i-1} = (\omega_n)^i$。因此 n 次单位根组成的数列可以看成公比为 ω_n 的等比数列。

2.2.2 单位根的性质

周期性 $\omega_n^k = \omega_n^{k \bmod n}$ ，因为每一圈都是 n 个单位根。

乘法关系 $\omega_n^j\omega_n^k = \omega_n^{j+k}$ ，根据模长都为 1 和复数乘法的性质可得。

消去引理 $\omega_{dn}^{dk} = \omega_n^k$ ，因为复平面上表示的向量相同。

折半引理 $(\omega_n^k)^2 = (\omega_n^{k+\frac{n}{2}})^2 = \omega_n^{2k}$ [$2 \mid n$]

折半引理正确性是因为当 n 为偶数时单位根是对称的，由此我们也可以得出 $\omega_n^k = -\omega_n^{k+\frac{n}{2}}$。

2.3 计算过程

2.3.1 Discrete Fourier Transform

离散傅里叶变换可以做到 $O(n\log n)$ 复杂度内求出 **所有 n 次单位根对应的点值**。

为了利用折半引理，我们把多项式补一些系数为 0 的高次项，使得多项式的次数为 2 的幂。

考虑对于一个单位根 ω_n^k ，我们需要求出

$$A(\omega_n^k)=\sum_{i=0}^{n-1}a_j(\omega_n^k)^i$$

我们将奇偶项分离，重新设一个包含偶数项的多项式为 A_0，包含奇数项的多项式为 A_1，有

$$A_0(\omega_n^k)=\sum_{i=0\ \mid\ i\%2=0}^{n-1}a_i(\omega_n^k)^{\frac{i}{2}}\ ,\ A_1(\omega_n^k)=\omega_n^k\sum_{i=0\ \mid\ i\%2=1}^{n-1}a_i(\omega_n^k)^{\frac{i}{2}}$$

由此可以看出多项式 A 可以写作次数界都减半的两个多项式 A_0, A_1 的和

$$A(\omega_n^k)=A_0(\omega_n^{2k})+\omega_n^kA_1(\omega_n^{2k})$$

考虑使用消去引理。我们把 ω_n^k 与 $\omega_n^{k+\frac{n}{2}}$ 对应的答案都写出来

$$A(\omega_n^k)=A_0(\omega_n^{2k})+\omega_n^kA_1(\omega_n^{2k})=A_0(\omega_{\frac{n}{2}}^k)+\omega_n^kA_1(\omega_{\frac{n}{2}}^k)$$
$$A(\omega_n^{k+\frac{n}{2}})=A_0(\omega_n^{2k})+\omega_n^{k+\frac{n}{2}}A_1(\omega_n^{2k})=A_0(\omega_n^{2k})-\omega_n^kA_1(\omega_n^{2k})=A_0(\omega_{\frac{n}{2}}^k)-\omega_n^kA_1(\omega_{\frac{n}{2}}^k)$$

注意到如果求出来了 $A_0(\omega_{\frac{n}{2}}^k)$ 和 $A_1(\omega_{\frac{n}{2}}^k)$ 的值，就可以 $O(1)$ 得到 $A(\omega_n^k)$ 与 $A(\omega_n^{k+\frac{n}{2}})$ 的值。

这是一个分治的形式。也就是说问题变成了两个子问题，问题的规模也缩小了一半都是求 $\frac{n}{2}$ 次多项式，在 $\frac{n}{2}$ 次单位根下的点值。由主定理知

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

2.3.2 Inverse Discrete Fourier Transform

设 $y_k=A(\omega_n^k)$ ，考虑另一个多项式 $B(x)=\sum_{k=0}^{n-1}y_kx^k$, 把上面的 n 个单位根的**倒数**代入，得到了另一个离散傅里叶变换 $z_k=B(\omega_n^{-k})$:

$$z_k=\sum_{i=0}^{n-1}y_i(\omega_n^{-k})^i=\sum_{i=0}^{n-1}(\sum_{j=0}^{n-1}a_j(\omega_n^i)^j)(\omega_n^{-k})^i=\sum_{j=0}^{n-1}a_j(\sum_{i=0}^{n-1}(\omega_n^{j-k})^i)$$

此处当 $j=k$ 时，$\sum_{i=0}^{n-1}(\omega_n^{j-k})^i$ 的值等于 n，否则等于 0，证明使用等比数列求和公式即可。

因此有 $z_k=na_k$，即 $a_k=\frac{z_k}{n}$ ，此时我们完成了插值的过程，也就是傅里叶变换的逆变换。

因此我们只需取单位根的倒数作为 x 代入 $B(x)$，用 DFT 优化计算，得到的每个数再除以 n，得到的是 $A(x)$ 的各项系数。

2.3.3 Butterfly Diagram

前人们找到了 DFT 迭代实现的方法，实际上分治最后每个系数的位置是有规律可循的。

考虑在每一次分治，我们都是把二进制末位为 0 的放到左侧，二进制末位为 1 的放到右侧，然后去掉二进制末位。那么最后最靠左的项一定是二进制全部为 0，其次是二进制最高位为 1。

假设 reverse(i) 是将二进制位反转的操作，DFT 分治到最后的系数数组是 B，原来的系数数组是 A，那么有 B[reverse(i)]=A[i]，反转过来之后就是我们刚才描述的分组过程。

蝴蝶变换的本意是优化常数，而逐个求二进制反转的复杂度也是 $O(n\log n)$ 的并不优秀。

这里介绍一种线性递推的方法：$rev[i] = ((rev[i >> 1] >> 1)|(i\&1) << (\log(N) - 1))$。含义是不考虑最高位，剩下的部分反转答案已经在之前求出，然后再单独考虑最高位的答案。

为了实现 swap(A[i],A[reverse(i)]) 并避免同一位置两次交换，我们可以当 $i < reverse(i)$ 时再交换。然后我们就可以模拟分治的过程以优化常数了。

```
  for (int i = 1; i < l; ++i)
    rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
  for (int i = 0; i < l; ++i) if (rev[i] > i) swap(a[i], a[rev[i]]);
```

2.4 代码实现

使用三重循环模拟，先枚举当前分治区间的长度（当前多项式的次数），然后是枚举每一个分治区间，最后枚举次数依次计算每一项的答案。

```
void FFT(complex *f, int len, int o) {
  for (int i = 0; i < len; ++i)
    if (rev[i] > i) swap(f[i], f[rev[i]]);  //反转序列
  for (int i = 1; i < len; i <<= 1) {  //枚举当前分治区间长度的一半
    complex wn = complex(cos(PI / i) , o * sin(PI / i));  //i次主单位根
    for (int j = 0; j < len; j += (i << 1)) {  //枚举每一个分治区间
      complex w = complex(1, 0), x, y;  //初始化第一个单位根的位置
      for (int k = 0; k < i; ++k, w = w * wn) {  //前一半贡献给后一半
        x = f[j + k]; y = w * f[i + j + k];
        f[i + j + k] = x - y; f[j + k] = x + y;  //折半引理
      }
    }
  }
}
```

3 Fast Number-Theoretic Transform

3.1 原根

快速数论变换提出，从数论角度找出一个具有单位根性质的东西。

定义**素数** p **的原根** g 为使得 $g^0, g^1, \dots, g^{p-2} \pmod p$ 互不相同的数。

主单位根的选取与循环性

如果我们取素数 $p = k \times 2^n + 1$，找到它的原根 g，那么主单位根就是 $g_n \equiv g^k \pmod p$，其幂就相当于更改 $k \times a$ 中 a 的值，显然是两两不同的，因此 $(g_n)^0, (g_n)^1, \dots, (g_n)^{p-2} \pmod p$ 也就满足了互不相同的性质。同时也就有了特殊值 $g_n^0 \equiv g_n^n = g^{nk} \equiv 1 \pmod p$ 的性质。

折半引理在数论意义下的体现

由于 p 是素数，并且 $g_n^n \equiv 1 \pmod p$，这样 $g_n^{\frac{n}{2}} \pmod p$ 必然是 -1 或 1，再根据 g_k 互不相同这个特点，所以 $g_n^{\frac{n}{2}} \equiv -1 \pmod p$，满足折半引理。

3.2 代码实现

因此我们可以用原根替代单位根，运算变为模意义下，代码与 FFT 基本相同。这里的模数为 $998244353 = 119 \times 2^{23} + 1$，其原根为 3。更多的模数与原根可以去 这里 看。

```
inline void NTT(ll *f, int len, int o) {
  for (int i = 0; i < len; ++i)
    if (rev[i] > i) swap(f[i], f[rev[i]]);
  for (int i = 1; i < len; i <<= 1) {
    ll wn = qpow(3, (mod - 1) / (i << 1));
    if (o == -1) wn = qpow(wn, mod - 2);
    for (int j = 0; j < len; j += (i << 1)) {
      ll w = 1, x, y;
      for (int k = 0; k < i; ++k, w = w * wn % mod) {
        x = f[j + k];
        y = w * f[i + j + k] % mod;  //不要忘了这里的取模
        f[i + j + k] = (x - y + mod) % mod;
        f[j + k] = (x + y) % mod;
      }
    }
  }
}
```