快速傅里叶变换

前置知识：复数。

本文将介绍一种算法，它支持在 $O(n\log n)$ 的时间内计算两个 n 次多项式的乘法，比朴素的 $O(n^2)$ 算法更高效。由于两个整数的乘法也可以被当作多项式乘法，因此这个算法也可以用来加速大整数的乘法计算。

引入

我们现在引入两个多项式 A 和 B：

$$\begin{aligned}A &= 5x^2+3x+7\\ B &= 7x^2+2x+1\end{aligned}$$

两个多项式相乘的积 $C=A\times B$，我们可以在 $O(n^2)$ 的时间复杂度中解得（这里 n 为 A 或者 B 多项式的次数）：

$$\begin{aligned}C &= A\times B\\ &= 35x^4+31x^3+60x^2+17x+7\end{aligned}$$

很明显，多项式 C 的系数 c_i 满足 $c_i=\sum_{j=0}^{i} a_j b_{i-j}$。而对于这种朴素算法而言，计算每一项的时间复杂度都为 $O(n)$，一共有 $O(n)$ 项，那么时间复杂度为 $O(n^2)$。

能否加速使得它的时间复杂度降低呢？如果使用快速傅里叶变换的话，那么我们可以使得其复杂度降低到 $O(n\log n)$。

傅里叶变换

傅里叶变换（Fourier Transform）是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，傅里叶变换用正弦波作为信号的成分。

设 $f(t)$ 是关于时间 t 的函数，则傅里叶变换可以检测频率 ω 的周期在 $f(t)$ 出现的程度：

$$F(\omega)=\mathbb{F}[f(t)]=\int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}dt$$

它的逆变换是

$$f(t)=\mathbb{F}^{-1}[F(\omega)]=\frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega t}d\omega$$

逆变换的形式与正变换非常类似，分母 2π 恰好是指数函数的周期。

傅里叶变换相当于将时域的函数与周期为 2π 的复指数函数进行连续的内积。逆变换仍旧为一个内积。

傅里叶变换有相应的卷积定理，可以将时域的卷积转化为频域的乘积，也可以将频域的卷积转化为时域的乘积。

离散傅里叶变换

离散傅里叶变换（Discrete Fourier transform，DFT）是傅里叶变换在时域和频域上都呈离散的形式，将信号的时域采样变换为其 DTFT（discrete-time Fourier transform）的频域采样。

傅里叶变换是积分形式的连续的函数内积，离散傅里叶变换是求和形式的内积。

设 $\{x_n\}_{n=0}^{N-1}$ 是某一满足有限性条件的序列，它的离散傅里叶变换（DFT）为：

$$X_k = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}kn}$$

其中 e 是自然对数的底数，i 是虚数单位。通常以符号 $\mathcal{F}$ 表示这一变换，即

$$\hat{x} = \mathcal{F}x$$

类似于积分形式，它的 **逆离散傅里叶变换**（IDFT）为：

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \mathrm{e}^{\mathrm{i}\frac{2\pi}{N}kn}$$

可以记为：

$$x = \mathcal{F}^{-1}\hat{x}$$

实际上，DFT 和 IDFT 变换式中和式前面的归一化系数并不重要。在上面的定义中，DFT 和 IDFT 前的系数分别为 1 和 $\frac{1}{N}$。有时我们会将这两个系数都改 $\frac{1}{\sqrt{N}}$。

离散傅里叶变换仍旧是时域到频域的变换。由于求和形式的特殊性，可以有其他的解释方法。

如果把序列 x_n 看作多项式 $f(x)$ 的 x^n 项系数，则计算得到的 X_k 恰好是多项式 $f(x)$ 代入单位根 $\mathrm{e}^{\frac{-2\pi\mathrm{i}k}{N}}$ 的点值 $f(\mathrm{e}^{\frac{-2\pi\mathrm{i}k}{N}})$。

这便构成了卷积定理的另一种解释办法，即对多项式进行特殊的求值操作。离散傅里叶变换恰好是多项式在单位根处进行求值。

例如计算：

$$\binom{n}{3} + \binom{n}{7} + \binom{n}{11} + \binom{n}{15} + \dots$$

定义函数 $f(x)$ 为：

$$f(x) = (1+x)^n = \binom{n}{0}x^0 + \binom{n}{1}x^1 + \binom{n}{2}x^2 + \binom{n}{3}x^3 + \dots$$

然后可以发现，代入四次单位根 $f(\mathrm{i})$ 得到这样的序列：

$$f(\mathrm{i})=(1+\mathrm{i})^n=\binom{n}{0}+\binom{n}{1}\mathrm{i}-\binom{n}{2}-\binom{n}{3}\mathrm{i}+\dots$$

于是下面的求和恰好可以把其余各项消掉：

$$f(1)+\mathrm{i}f(\mathrm{i})-f(-1)-\mathrm{i}f(-\mathrm{i})=4\binom{n}{3}+4\binom{n}{7}+4\binom{n}{11}+4\binom{n}{15}+\dots$$

因此这道数学题的答案为：

$$\binom{n}{3}+\binom{n}{7}+\binom{n}{11}+\binom{n}{15}+\dots=\frac{2^n+\mathrm{i}(1+\mathrm{i})^n-\mathrm{i}(1-\mathrm{i})^n}{4}$$

这道数学题在单位根处求值，恰好构成离散傅里叶变换。

矩阵公式

由于离散傅立叶变换是一个 **线性** 算子，所以它可以用矩阵乘法来描述。在矩阵表示法中，离散傅立叶变换表示如下：

$$\begin{bmatrix}X_0\\X_1\\X_2\\\vdots\\X_{N-1}\end{bmatrix}=\begin{bmatrix}1&1&1&\cdots&1\\1&\alpha&\alpha^2&\cdots&\alpha^{N-1}\\1&\alpha^2&\alpha^4&\cdots&\alpha^{2(N-1)}\\\vdots&\vdots&\vdots&\ddots&\vdots\\1&\alpha^{N-1}&\alpha^{2(N-1)}&\cdots&\alpha^{(N-1)(N-1)}\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\\\vdots\\x_{N-1}\end{bmatrix}$$

其中 $\alpha=\mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}}$。

快速傅里叶变换

FFT 是一种高效实现 DFT 的算法，称为快速傅立叶变换（Fast Fourier Transform，FFT）。它对傅里叶变换的理论并没有新的发现，但是对于在计算机系统或者说数字系统中应用离散傅立叶变换，可以说是进了一大步。快速数论变换（NTT）是快速傅里叶变换（FFT）在数论基础上的实现。

在 1965 年，Cooley 和 Tukey 发表了快速傅里叶变换算法。事实上 FFT 早在这之前就被发现过了，但是在当时现代计算机并未问世，人们没有意识到 FFT 的重要性。一些调查者认为 FFT 是由 Runge 和 König 在 1924 年发现的。但事实上高斯早在 1805 年就发明了这个算法，但一直没有发表。

分治法实现

FFT 算法的基本思想是分治。就 DFT 来说，它分治地来求当 $x=\omega_n^k$ 的时候 $f(x)$ 的值。基 - 2 FFT 的分治思想体现在将多项式分为奇次项和偶次项处理。

举个例子，对于一共 8 项的多项式：

$$f(x)=a_0+a_1x+a_2x^2+a_3x^3+a_4x^4+a_5x^5+a_6x^6+a_7x^7$$

按照次数的奇偶来分成两组，然后右边提出来一个 x：

$$\begin{aligned}f(x)&=(a_0+a_2x^2+a_4x^4+a_6x^6)+(a_1x+a_3x^3+a_5x^5+a_7x^7)\\&=(a_0+a_2x^2+a_4x^4+a_6x^6)+x(a_1+a_3x^2+a_5x^4+a_7x^6)\end{aligned}$$

分别用奇偶次次项数建立新的函数：

$$\begin{aligned}G(x)&=a_0+a_2x+a_4x^2+a_6x^3\\H(x)&=a_1+a_3x+a_5x^2+a_7x^3\end{aligned}$$

那么原来的 $f(x)$ 用新函数表示为：

$$f(x)=G\left(x^2\right)+x\times H\left(x^2\right)$$

利用偶数次单位根的性质 $\omega^i_n=-\omega^{i+n/2}_n$，和 $G\left(x^2\right)$ 和 $H\left(x^2\right)$ 是偶函数，我们知道在复平面上 ω^i_n 和 $\omega^{i+n/2}_n$ 的 $G(x^2)$ 的 $H(x^2)$ 对应的值相同。得到：

$$\begin{aligned}f(\omega_n^k)&=G((\omega_n^k)^2)+\omega_n^k\times H((\omega_n^k)^2)\\&=G(\omega_n^{2k})+\omega_n^k\times H(\omega_n^{2k})\\&=G(\omega_{n/2}^k)+\omega_n^k\times H(\omega_{n/2}^k)\end{aligned}$$

和：

$$\begin{aligned}f(\omega_n^{k+n/2})&=G(\omega_n^{2k+n})+\omega_n^{k+n/2}\times H(\omega_n^{2k+n})\\&=G(\omega_n^{2k})-\omega_n^k\times H(\omega_n^{2k})\\&=G(\omega_{n/2}^k)-\omega_n^k\times H(\omega_{n/2}^k)\end{aligned}$$

因此我们求出了 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，就可以同时求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$。于是对 G 和 H 分别递归 DFT 即可。

考虑到分治 DFT 能处理的多项式长度只能是 $2^m(m\in\mathbf{N}^*)$，否则在分治的时候左右不一样长，右边就取不到系数了。所以要在第一次 DFT 之前就把序列向上补成长度为 $2^m(m\in\mathbf{N}^*)$（高次系数补 0）、最高项次数为 2^m-1 的多项式。

在代入值的时候，因为要代入 n 个不同值，所以我们代入 $\omega_n^0,\omega_n^1,\omega_n^2,\cdots,\omega_n^{n-1}(n=2^m(m\in\mathbf{N}^*))$ 一共 2^m 个不同值。

代码实现方面，STL 提供了复数的模板，当然也可以手动实现。两者区别在于，使用 STL 的 `complex` 可以调用 `exp` 函数求出 ω_n。但事实上使用欧拉公式得到的虚数来求 ω_n 也是等价的。

以上就是 FFT 算法中 DFT 的介绍，它将一个多项式从系数表示法变成了点值表示法。

值的注意的是，因为是单位复根，所以说我们需要令 n 项式的高位补为零，使得 $n=2^k,k\in\mathbf{N}^*$ 。

递归版 FFT

```cpp
#include <cmath>
#include <complex>

using Comp = std::complex<double>;  // STL complex

constexpr Comp I(0, 1);  // i
constexpr int MAX_N = 1 << 20;

Comp tmp[MAX_N];

// rev=1,DFT; rev=-1,IDFT
void DFT(Comp* f, int n, int rev) {
  if (n == 1) return;
  for (int i = 0; i < n; ++i) tmp[i] = f[i];
  // 偶数放左边，奇数放右边
  for (int i = 0; i < n; ++i) {
    if (i & 1)
      f[n / 2 + i / 2] = tmp[i];
    else
      f[i / 2] = tmp[i];
  }
  Comp *g = f, *h = f + n / 2;
  // 递归 DFT
  DFT(g, n / 2, rev), DFT(h, n / 2, rev);
  // cur 是当前单位复根，对于 k = 0 而言，它对应的单位复根 omega^0_n = 1。
  // step 是两个单位复根的差，即满足 omega^k_n = step*omega^{k-1}*n,
  // 定义等价于 exp(I*(2*M_PI/n*rev))
  Comp cur(1, 0), step(cos(2 * M_PI / n), sin(2 * M_PI * rev / n));
  for (int k = 0; k < n / 2;
       ++k) {  // F(omega^k_n) = G(omega^k*{n/2}) + omega^k*n\*H(omega^k*{n/2})
    tmp[k] = g[k] + cur * h[k];
    // F(omega^{k+n/2}*n) = G(omega^k*{n/2}) - omega^k_n*H(omega^k\_{n/2})
    tmp[k + n / 2] = g[k] - cur * h[k];
    cur *= step;
  }
  for (int i = 0; i < n; ++i) f[i] = tmp[i];
}
```

时间复杂度 $O(n \log n)$。

倍增法实现

这个算法还可以从「分治」的角度继续优化。对于基 - 2 FFT，我们每一次都会把整个多项式的奇数次项和偶数次项系数分开，一直分到只剩下一个系数。但是，这个递归的过程需要更多的内存。因此，我们可以先「模仿递归」把这些系数在原数组中「拆分」，然后再「倍增」地去合并这些算出来的值。

对于「拆分」，可以使用位逆序置换实现。

对于「合并」，使用蝶形运算优化可以做到只用 $O(1)$ 的额外空间来完成。

位逆序置换

以 8 项多项式为例，模拟拆分的过程：

- 初始序列为 $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$
- 一次二分之后 $\{x_0, x_2, x_4, x_6\}, \{x_1, x_3, x_5, x_7\}$
- 两次二分之后 $\{x_0, x_4\}\{x_2, x_6\}, \{x_1, x_5\}, \{x_3, x_7\}$
- 三次二分之后 $\{x_0\}\{x_4\}\{x_2\}\{x_6\}\{x_1\}\{x_5\}\{x_3\}\{x_7\}$

规律：其实就是原来的那个序列，每个数用二进制表示，然后把二进制翻转对称一下，就是最终那个位置的下标。比如 x_1 是 001，翻转是 100，也就是 4，而且最后那个位置确实是 4。我们称这个变换为位逆序置换（bit-reversal permutation），证明留给读者自证。

根据它的定义，我们可以在 $O(n \log n)$ 的时间内求出每个数变换后的结果：

位逆序置换实现（$O(n\log n)$）

```
/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 * len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  // 一开始 i 是 0...01，而 j 是 10...0，在二进制下相反对称。
  // 之后 i 逐渐加一，而 j 依然维持着和 i 相反对称，一直到 i = 1...11。
  for (int i = 1, j = len / 2, k; i < len - 1; i++) {
    // 交换互为小标反转的元素，i < j 保证交换一次
    if (i < j) swap(y[i], y[j]);
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的。
    // 这里 k 代表了 0 出现的最高位。j 先减去高位的全为 1 的数字，直到遇到了
    // 0，之后再加上即可。
    k = len / 2;
    while (j >= k) {
      j = j - k;
      k = k / 2;
    }
    if (j < k) j += k;
  }
}
```

实际上，位逆序置换可以 $O(n)$ 从小到大递推实现，设 $len=2^k$，其中 k 表示二进制数的长度，设 $R(x)$ 表示长度为 k 的二进制数 x 翻转后的数（高位补 0）。我们要求的是 $R(0),R(1),\cdots,R(n-1)$。

首先 $R(0)=0$。

我们从小到大求 $R(x)$。因此在求 $R(x)$ 时，$R\left(\left\lfloor \frac{x}{2} \right\rfloor\right)$ 的值是已知的。因此我们把 x 右移一位（除以 2），然后翻转，再右移一位，就得到了 x **除了（二进制）个位** 之外其它位的翻转结果。

考虑个位的翻转结果：如果个位是 0，翻转之后最高位就是 0。如果个位是 1，则翻转后最高位是 1，因此还要加上 $\frac{len}{2}=2^{k-1}$。综上

$$R(x)=\left\lfloor \frac{R\left(\left\lfloor \frac{x}{2} \right\rfloor\right)}{2} \right\rfloor+(x\bmod 2)\times \frac{len}{2}$$

举个例子：设 $k=5$，$len=(100000)_2$。为了翻转 $(11001)_2$：

1. 考虑 $(1100)_2$，我们知道 $R((1100)_2)=R((01100)_2)=(00110)_2$，再右移一位就得到了 $(00011)_2$。
2. 考虑个位，如果是 1，它就要翻转到数的最高位，即翻转数加上 $(10000)_2=2^{k-1}$，如果是 0 则不用更改。

位逆序置换实现（$O(n)$）

```
// 同样需要保证 len 是 2 的幂
// 记 rev[i] 为 i 翻转后的值
void change(Complex y[], int len) {
  for (int i = 0; i < len; ++i) {
    rev[i] = rev[i >> 1] >> 1;
    if (i & 1) {  // 如果最后一位是 1，则翻转成 len/2
      rev[i] |= len >> 1;
    }
  }
  for (int i = 0; i < len; ++i) {
    if (i < rev[i]) {  // 保证每对数只翻转一次
      swap(y[i], y[rev[i]]);
    }
  }
  return;
}
```

蝶形运算优化

已知 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，需要使用下面两个式子求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$：

$$
\begin{aligned}
f(\omega_n^k) &= G(\omega_{n/2}^k) + \omega_n^k \times H(\omega_{n/2}^k) \\
f(\omega_n^{k+n/2}) &= G(\omega_{n/2}^k) - \omega_n^k \times H(\omega_{n/2}^k)
\end{aligned}
$$

使用位逆序置换后，对于给定的 n, k：

- $G(\omega_{n/2}^k)$ 的值存储在数组下标为 k 的位置，$H(\omega_{n/2}^k)$ 的值存储在数组下标为 $k + \dfrac{n}{2}$ 的位置。
- $f(\omega_n^k)$ 的值将存储在数组下标为 k 的位置，$f(\omega_n^{k+n/2})$ 的值将存储在数组下标为 $k + \dfrac{n}{2}$ 的位置。

因此可以直接在数组下标为 k 和 $k + \frac{n}{2}$ 的位置进行覆写，而不用开额外的数组保存值。此方法即称为 **蝶形运算**，或更准确的，基 - 2 蝶形运算。

再详细说明一下如何借助蝶形运算完成所有段长度为 $\frac{n}{2}$ 的合并操作：

1. 令段长度为 $s = \frac{n}{2}$；
2. 同时枚举序列 $\{G(\omega_{n/2}^k)\}$ 的左端点 $l_g = 0, 2s, 4s, \cdots, N - 2s$ 和序列 $\{H(\omega_{n/2}^k)\}$ 的左端点 $l_h = s, 3s, 5s, \cdots, N - s$；
3. 合并两个段时，枚举 $k = 0, 1, 2, \cdots, s - 1$，此时 $G(\omega_{n/2}^k)$ 存储在数组下标为 $l_g + k$ 的位置，$H(\omega_{n/2}^k)$ 存储在数组下标为 $l_h + k$ 的位置；
4. 使用蝶形运算求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$，然后直接在原位置覆写。

快速傅里叶逆变换

傅里叶逆变换可以用傅里叶变换表示。对此我们有两种理解方式。

线性代数角度

IDFT（傅里叶反变换）的作用，是把目标多项式的点值形式转换成系数形式。而 DFT 本身是个线性变换，可以理解为将目标多项式当作向量，左乘一个矩阵得到变换后的向量，以模拟把单位复根代入多项式的过程：

$$\begin{bmatrix}y_0\\y_1\\y_2\\y_3\\\vdots\\y_{n-1}\end{bmatrix}=\begin{bmatrix}1&1&1&1&\cdots&1\\1&\omega_n^1&\omega_n^2&\omega_n^3&\cdots&\omega_n^{n-1}\\1&\omega_n^2&\omega_n^4&\omega_n^6&\cdots&\omega_n^{2(n-1)}\\1&\omega_n^3&\omega_n^6&\omega_n^9&\cdots&\omega_n^{3(n-1)}\\\vdots&\vdots&\vdots&\vdots&\ddots&\vdots\\1&\omega_n^{n-1}&\omega_n^{2(n-1)}&\omega_n^{3(n-1)}&\cdots&\omega_n^{(n-1)^2}\end{bmatrix}\begin{bmatrix}a_0\\a_1\\a_2\\a_3\\\vdots\\a_{n-1}\end{bmatrix}$$

现在我们已经得到最左边的结果了，中间的 x 值在目标多项式的点值表示中也是一一对应的，所以，根据矩阵的基础知识，我们只要在式子两边左乘中间那个大矩阵的逆矩阵就行了。

由于这个矩阵的元素非常特殊，它的逆矩阵也有特殊的性质，就是每一项 **取倒数**，再 **除以变换的长度** n，就能得到它的逆矩阵。

注意：傅里叶变换的长度，并不是多项式的长度，变换的长度应比乘积多项式的长度长。待相乘的多项式不够长，需要在高次项处补 0。

为了使计算的结果为原来的倒数，根据欧拉公式，可以得到

$$\frac{1}{\omega_k}=\omega_k^{-1}=\mathrm{e}^{-\frac{2\pi\mathrm{i}}{k}}=\cos\left(\frac{2\pi}{k}\right)+\mathrm{i}\sin\left(-\frac{2\pi}{k}\right)$$

因此我们可以尝试着把单位根 ω_k 取成 $\mathrm{e}^{-\frac{2\pi\mathrm{i}}{k}}$，这样我们的计算结果就会变成原来的倒数，之后唯一多的操作就只有再 **除以它的长度** n，而其它的操作过程与 DFT 是完全相同的。我们可以定义一个函数，在里面加一个参数 1 或者是 -1，然后把它乘到 π 上。传入 1 就是 DFT，传入 -1 就是 IDFT。

单位复根周期性

利用单位复根的周期性同样可以理解 IDFT 与 DFT 之间的关系。

考虑原本的多项式是 $f(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}=\sum_{i=0}^{n-1}a_ix^i$。而 IDFT 就是把你的点值表示还原为系数表示。

考虑 **构造法**。我们已知 $y_i=f\left(\omega_n^i\right),i\in\{0,1,\cdots,n-1\}$，求 $\{a_0,a_1,\cdots,a_{n-1}\}$。构造多项式如下

$$A(x)=\sum_{i=0}^{n-1} y_i x^i$$

相当于把 $\{y_0, y_1, y_2, \cdots, y_{n-1}\}$ 当做多项式 A 的系数表示法。

这时我们有两种推导方式，这对应了两种实现方法。

方法一

设 $b_i=\omega_n^{-i}$，则多项式 A 在 $x=b_0, b_1, \cdots, b_{n-1}$ 处的点值表示法为 $\{A(b_0), A(b_1), \cdots, A(b_{n-1})\}$。

对 $A(x)$ 的定义式做一下变换，可以将 $A(b_k)$ 表示为

$$\begin{aligned} A(b_k) &= \sum_{i=0}^{n-1} f(\omega_n^i)\omega_n^{-ik} = \sum_{i=0}^{n-1} \omega_n^{-ik} \sum_{j=0}^{n-1} a_j (\omega_n^i)^j \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_j \omega_n^{i(j-k)} = \sum_{j=0}^{n-1} a_j \sum_{i=0}^{n-1} \left(\omega_n^{j-k}\right)^i \end{aligned}$$

记 $S\left(\omega_n^a\right)=\sum_{i=0}^{n-1}\left(\omega_n^a\right)^i$。

当 $a=0 \pmod n$ 时，$S\left(\omega_n^a\right)=n$。

当 $a\neq 0 \pmod n$ 时，我们错位相减

$$\begin{aligned} S\left(\omega_n^a\right) &= \sum_{i=0}^{n-1}\left(\omega_n^a\right)^i \\ \omega_n^a S\left(\omega_n^a\right) &= \sum_{i=1}^{n}\left(\omega_n^a\right)^i \\ S\left(\omega_n^a\right) &= \frac{\left(\omega_n^a\right)^n-\left(\omega_n^a\right)^0}{\omega_n^a-1}=0 \end{aligned}$$

也就是说

$$S\left(\omega_n^a\right)=\begin{cases} n, & a=0 \\ 0, & a\neq 0 \end{cases}$$

那么代回原式

$$A(b_k)=\sum_{j=0}^{n-1} a_j S\left(\omega_n^{j-k}\right)=a_k\cdot n$$

也就是说给定点 $b_i=\omega_n^{-i}$，则 A 的点值表示法为

$$\begin{aligned} &\{(b_0, A(b_0)), (b_1, A(b_1)), \cdots, (b_{n-1}, A(b_{n-1}))\} \\ =&\{(b_0, a_0\cdot n), (b_1, a_1\cdot n), \cdots, (b_{n-1}, a_{n-1}\cdot n)\} \end{aligned}$$

综上所述，我们取单位根为其倒数，对 $\{y_0, y_1, y_2, \cdots, y_{n-1}\}$ 跑一遍 FFT，然后除以 n 即可得到 $f(x)$ 的系数表示。

方法二

我们直接将 ω_n^i 代入 $A(x)$。

推导的过程与方法一大同小异，最终我们得到 $A(\omega_n^k)=\sum_{j=0}^{n-1}a_jS\left(\omega_n^{j+k}\right)$。

当且仅当 $j+k=0\pmod n$ 时有 $S\left(\omega_n^{j+k}\right)=n$，否则为 0。因此 $A(\omega_n^k)=a_{n-k}\cdot n$。

这意味着我们将 $\{y_0,y_1,y_2,\cdots,y_{n-1}\}$ 做 DFT 变换后除以 n，再反转后 $n-1$ 个元素，同样可以还原 $f(x)$ 的系数表示。

代码实现

所以我们 FFT 函数可以集 DFT 和 IDFT 于一身。代码实现如下：

非递归版 FFT（对应方法一）

```
/*
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  // 位逆序置换
  change(y, len);
  // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长度为 len。
  for (int h = 2; h <= len; h <<= 1) {
    // wn：当前单位复根的间隔：w^1_h
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));
    // 合并，共 len / h 次。
    for (int j = 0; j < len; j += h) {
      // 计算当前单位复根，一开始是 1 = w^0_n，之后是以 wn 为间隔递增：w^1_n
      // ...
      Complex w(1, 0);
      for (int k = j; k < j + h / 2; k++) {
        // 左侧部分和右侧是子问题的解
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        // 这就是把两部分分治的结果加起来
        y[k] = u + t;
        y[k + h / 2] = u - t;
        // 后半个 「step」 中的ω一定和 「前半个」 中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  // 如果是 IDFT，它的逆矩阵的每一个元素不只是原元素取倒数，还要除以长度 len。
  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}
```

非递归版 FFT（对应方法二）

```cpp
/*
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);
  for (int h = 2; h <= len; h <<= 1) {              // 模拟合并过程
    Complex wn(cos(2 * PI / h), sin(2 * PI / h));  // 计算当前单位复根
    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);  // 计算当前单位复根
      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;  // 这就是把两部分分治的结果加起来
        y[k + h / 2] = u - t;
        // 后半个 「step」 中的ω一定和 「前半个」 中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  if (on == -1) {
    reverse(y + 1, y + len);
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}
```

FFT 模板（HDU 1402 - A * B Problem Plus）

```
#include <cmath>
#include <cstring>
#include <iostream>

const double PI = acos(-1.0);

struct Complex {
  double x, y;

  Complex(double _x = 0.0, double _y = 0.0) {
    x = _x;
    y = _y;
  }

  Complex operator-(const Complex &b) const {
    return Complex(x - b.x, y - b.y);
  }

  Complex operator+(const Complex &b) const {
    return Complex(x + b.x, y + b.y);
  }

  Complex operator*(const Complex &b) const {
    return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
  }
};

/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 *len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  int i, j, k;

  for (int i = 1, j = len / 2; i < len - 1; i++) {
    if (i < j) std::swap(y[i], y[j]);

    // 交换互为小标反转的元素，i<j 保证交换一次
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的
    k = len / 2;

    while (j >= k) {
      j = j - k;
      k = k / 2;
    }

    if (j < k) j += k;
  }
```

```cpp
}

/*
 * 做 FFT
 *len 必须是 2^k 形式
 *on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);

  for (int h = 2; h <= len; h <<= 1) {
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));

    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);

      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;
        y[k + h / 2] = u - t;
        w = w * wn;
      }
    }
  }

  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}

constexpr int MAXN = 200020;
Complex x1[MAXN], x2[MAXN];
char str1[MAXN / 2], str2[MAXN / 2];
int sum[MAXN];
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> str1 >> str2) {
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int len = 1;

    while (len < len1 * 2 || len < len2 * 2) len <<= 1;

    for (int i = 0; i < len1; i++) x1[i] = Complex(str1[len1 - 1
- i] - '0', 0);

```

```
    for (int i = len1; i < len; i++) x1[i] = Complex(0, 0);

    for (int i = 0; i < len2; i++) x2[i] = Complex(str2[len2 - 1 - i] - '0', 0);

    for (int i = len2; i < len; i++) x2[i] = Complex(0, 0);

    fft(x1, len, 1);
    fft(x2, len, 1);

    for (int i = 0; i < len; i++) x1[i] = x1[i] * x2[i];

    fft(x1, len, -1);

    for (int i = 0; i < len; i++) sum[i] = int(x1[i].x + 0.5);

    for (int i = 0; i < len; i++) {
      sum[i + 1] += sum[i] / 10;
      sum[i] %= 10;
    }

    len = len1 + len2 - 1;

    while (sum[len] == 0 && len > 0) len--;

    for (int i = len; i >= 0; i--) cout << char(sum[i] + '0');

    cout << '\n';
  }

  return 0;
}
```

参考文献

1. 桃酱的算法笔记.

本页面最近更新：2024/10/30 21:50:12，更新历史

发现错误？想一起完善？ 在 GitHub 上编辑此页！

本页面贡献者：H-J-Granger, ranwen, abc1763613206, Ahacad, Allenyou1126, AndrewWayne, AngelKitty, AtomAlpaca, Backl1ght, billchenchina, c-forrest, CCXXXI, Chrogeek, ChungZH, countercurrent-time, DepletedPrism, Early0v0, EarthMessenger, Enter-tainer, F1shAndCat, GavinZhengOI, Gesrua, Great-designer, greyqz, Haohu Shen, henryrabbit, heroming, hly1204, Ir1d, isdanni, jiang1997, kenlig, Lewy Zeng, lucifer1004, Menci, muoshuosha, NachtgeistW, needtocalmdown, opsiff, ouuan, ouuan, partychicken, schtonn, Sshwy, sshwy, StudyingFather, SukkaW, Tiphereth-A, TrisolarisHD, untitledunrevised, Xeonacid, YouXam, Yukimaikoriya