声明

此文档是基于 **SparklingWind** 的《高中生学 FFT 算法》。本文可以当做对其的注解，并额外添加了雷德算法及其证明。

Something About FFT

Author 光和尘

复数的一些性质

1. $(\cos\theta + i\sin\theta)^n = \cos n\theta + i\sin n\theta$.

   证

   $$(\cos\theta + i\sin\theta)^n = e^{(i\theta)n} = e^{i(n\theta)} = \cos n\theta + i\sin n\theta.$$

   证毕

2. 令 $\mathbb{W}_n = e^{\frac{2\pi i}{n}}$，则 $\mathbb{W}_n^k (k = 0, 1, 2, \ldots, n-1)$ 为 1 的 n 个 n 次方根。

   证

   $$(\mathbb{W}_n^k)^n = (e^{\frac{2k\pi i}{n}})^n = e^{2k\pi i} = (e^{2\pi i})^k = 1^k = 1.$$

   证毕

3. 当 x 不被 n 整除时，$\sum\limits_{k=0}^{n-1} \mathbb{W}_n^{xk} = 0$.

   证

   $$\because\ n \nmid x \ \Rightarrow\ \mathbb{W}_n^x \neq 1, \quad n \mid n \ \Rightarrow\ \mathbb{W}_n^n = 1 \ \Rightarrow\ \mathbb{W}_n^{xn} = 1.$$

   $$\therefore\ \sum_{k=0}^{n-1} \mathbb{W}_n^{xk} = \frac{\mathbb{W}_n^0(1 - \mathbb{W}_n^{xn})}{1 - \mathbb{W}_n^x} = \frac{1(1-1)}{1 - \mathbb{W}_n^x} = 0.$$

   证毕

4. 消去律：$\mathbb{W}_{xn}^{xk} = \mathbb{W}_n^k$.

   证

   $$e^{\frac{2xk\pi i}{xn}} = e^{\frac{2k\pi i}{n}} = \mathbb{W}_n^k.$$

   证毕

5. $\mathbb{W}_n^{k+\frac{n}{2}} = \text{-}\mathbb{W}_n^k$.

   证

   $$\because\ \mathbb{W}_n^{\frac{n}{2}} = e^{\frac{\frac{n}{2}\times 2\pi i}{n}} = e^{\pi i} = \cos\pi + i\sin\pi = \text{-}1.$$

   $$\therefore\ \mathbb{W}_n^{k+\frac{n}{2}} = \mathbb{W}_n^k \mathbb{W}_n^{\frac{n}{2}} = \text{-}\mathbb{W}_n^k.$$

   证毕

正文

对于一个 $n-1$ 次多项式：$f(x)=a_0+a_1x+\ldots+a_{n-1}x^{n-1}$.
可以用 $\{a_0,a_1,a_2,\ldots,a_{n-1}\}$ 来表示这个多项式。这种表示法称为**系数表示法**。
如果 $a_{n-1}\neq 0$，则称 n 为该多项式的严格次数界。

如果代 $\{x_0,\ x_1,\ x_2,\ldots,\ x_{n-1}\}$ 入多项式，可以求出一些具体的值：

$$\Big\{\{x_0,f(x_0)\},\ \{x_1,f(x_1)\},\ldots,\ \{x_{n-1},f(x_{n-1})\}\Big\} \qquad I-1$$

可以用这些点对（$I-1$）来表示这个多项式，这种表示法称为**点值表示法**。

已知两个多项式：$\begin{cases}\mathbb{A}(x)=a_0+a_1x+\ldots+a_{n-1}x^{n-1}.\\ \mathbb{B}(x)=b_0+b_1x+\ldots+b_{n-1}x^{n-1}.\end{cases}$

我们想要得到这两个多项式的乘积 $\mathbb{C}(x)$，即：

$$\mathbb{C}(x)=\mathbb{A}(x)\times\mathbb{B}(x)=c_0+c_1x+\ldots+c_{n-1}x^{n-1}+\ldots+c_{2n-2}x^{2n-2}.$$

其中：$c_i=\sum_{k=0}^{i}a_k\times b_{i-k},\ \Big(0\leqslant i\leqslant 2n-2,a_t=b_t=0,\ (t\geqslant n)\Big)$。

显然，如果要用系数表示法求 $\mathbb{C}(x)$，则需要做 $\binom{2n}{2}$ 次加法和乘法。但是，如果我们用点值表示法来完成这个计算，显然仅需做 $2n-1$ 次乘法。这个过程称为 **DFT**。

约定： $\mathbb{W}_n=e^{\frac{2\pi i}{n}}$。

令 $x_r=\mathbb{W}_{2n}^r=e^{\frac{r\pi i}{n}}$，并将 $n-1$ 次多项式 $\mathbb{A}(x)$ 扩充到 $2n-1$ 次。即：

$$\mathbb{A}(x)=a_0+a_1x+\ldots+a_{n-1}x^{n-1}+a_nx^n+\ldots+a_{2n-1}x^{2n-1},\ \Big(a_t=0,\ (t\geqslant n)\Big)$$

并令：$\begin{cases}\mathbb{A}'(x)=a_0+a_2x+a_4x^2+\ldots+a_{2n-4}x^{n-2}+a_{2n-2}x^{n-1}\\ \mathbb{A}''(x)=a_1+a_3x+a_5x^2+\ldots+a_{2n-3}x^{n-2}+a_{2n-1}x^{n-1}\end{cases}$

那么：

$$\begin{aligned}\mathbb{A}(x_r)&=a_0+a_1x_r+\ldots+a_{n-1}x_r^{n-1}+a_nx_r^n+\ldots+a_{2n-1}x_r^{2n-1}\\&=\left[a_0x_r^0+a_2x_r^2+\ldots+a_{2n-2}(x_r^{2n-2}\right]+x_r\left[a_1x_r^0+a_3x_r^2+\ldots+a_{2n-1}x_r^{2n-2}\right]\\&=\left[a_0(x_r^2)^0+a_2(x_r^2)^1+\ldots+a_{2n-2}(x_r^2)^{n-1}\right]+x_r\left[a_1(x_r^2)^0+a_3(x_r^2)^1+\ldots+a_{2n-1}(x_r^2)^{n-1}\right]\\&=\mathbb{A}'(x_r^2)+x_r\mathbb{A}''(x_r^2)\\&=\mathbb{A}'(\mathbb{W}_{2n}^{2r})+\mathbb{W}_{2n}^r\mathbb{A}''(\mathbb{W}_{2n}^{2r})\\&=\mathbb{A}'(\mathbb{W}_n^r)+\mathbb{W}_{2n}^r\mathbb{A}''(\mathbb{W}_n^r)\qquad\textbf{[复数的性质 4]，消去律}\end{aligned}$$

$\because$ 当 $r\geqslant n$ 时，$\mathbb{W}_{2n}^r=\mathbb{W}_{2n}^{(r-n)+n}=-\mathbb{W}_{2n}^{r-n}$，$\mathbb{W}_n^r=\mathbb{W}_n^{r-n+n}=\mathbb{W}_n^{r-n}$ **【复数的性质 5】**

$\therefore$ 当 $r<n$ 时，令 $t=r$，　则 $\mathbb{A}(x_r)=\mathbb{A}(\mathbb{W}_{2n}^r)=\mathbb{A}'(\mathbb{W}_n^t)+\mathbb{W}_{2n}^t\mathbb{A}''(\mathbb{W}_n^t)$

当 $r\geqslant n$ 时，令 $t=r-n$，则 $\mathbb{A}(x_r)=\mathbb{A}(\mathbb{W}_{2n}^r)=\mathbb{A}'(\mathbb{W}_n^t)-\mathbb{W}_{2n}^t\mathbb{A}''(\mathbb{W}_n^t)$

也就是说：

$$\mathbb{A}(x_r)=\mathbb{A}(\mathbb{W}_{2n}^r)=\begin{cases}\mathbb{A}'(\mathbb{W}_n^r)+\mathbb{W}_{2n}^r\mathbb{A}''(\mathbb{W}_n^r) & (r<n)\\ \mathbb{A}'(\mathbb{W}_n^{r-n})-\mathbb{W}_{2n}^{r-n}\mathbb{A}''(\mathbb{W}_n^{r-n}) & (r\geqslant n)\end{cases}$$

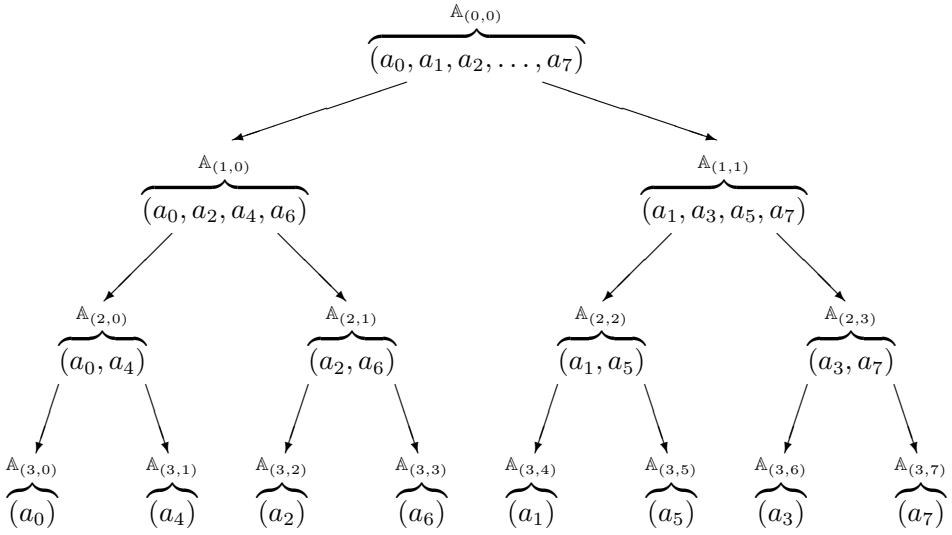
据此，我们就可以在 $O(n\log n)$ 的时间内将系数表示法转化为点值表示法，具体做法看如下伪代码：

- row_1 : *begin*
- row_2 : $DFT\big(2n,\{a_0,a_1,\ldots,a_{2n-1}\}\big)$
- row_3 : *if* $n=1$ *then return* a_0
- row_4 : $\{\mathbb{A}'(x_0),\mathbb{A}'(x_1),\ldots,\mathbb{A}'(x_{n-1})\}\ \leftarrow\ DFT\big(n,\{a_0,a_2,a_4,\ldots,a_{2n-2}\}\big)$
- row_5 : $\{\mathbb{A}''(x_0),\mathbb{A}''(x_1),\ldots,\mathbb{A}''(x_{n-1})\}\ \leftarrow\ DFT\big(n,\{a_1,a_3,a_5,\ldots,a_{2n-1}\}\big)$
- row_6 : *for* $t=0$ *to* $n-1$
- row_7 : $\mathbb{A}(x_t)=\mathbb{A}'(x_t)+\mathbb{W}_{2n}^t\mathbb{A}''(x_t)$
- row_8 : $\mathbb{A}(x_t+n)=\mathbb{A}'(x_t)-\mathbb{W}_{2n}^t\mathbb{A}''(x_t)$
- row_9 : *return* $\{\mathbb{A}(x_0),\mathbb{A}(x_1),\mathbb{A}(x_2),\ldots,\mathbb{A}(x_{2n-1})\}$
- row_{10} : *end*

有了前面的分析，伪码一目了然。

虽然它的复杂度是 $O(n\log n)$ 的，但是不难发现，在递归前需要先将奇数项和偶数项剥离开来，而这需要 $O(n)$ 的辅助空间。要砍掉辅助空间的开销，我们不得不在做变换前就按照整个算法的实际计算顺序去调整好元素的位置，为此，下面引进**雷德算法**。

为了有一个更直观的印象，我画了一个 $N=8$ 时的简图。



因此我们的实际计算顺序是：

- $Step_1$: $(A_{(3,0)},\ A_{(3,1)}) \Rightarrow A_{(2,0)},\quad (A_{(3,2)},\ A_{(3,3)}) \Rightarrow A_{(2,1)},$
  $(A_{(3,4)},\ A_{(3,5)}) \Rightarrow A_{(2,2)},\quad (A_{(3,6)},\ A_{(3,7)}) \Rightarrow A_{(2,3)}.$
- $Step_2 : (A_{(2,0)},\ A_{(2,1)}) \Rightarrow A_{(1,0)},\quad (A_{(2,2)},\ A_{(2,3)}) \Rightarrow A_{(1,1)}.$
- $Step_3 : (A_{(1,0)},\ A_{(1,1)}) \Rightarrow A_{(0,0)}.$

容易发现，如果我们能将原来的系数序列变成递归到最底层时的系数序列（上图中，则需将 $\{a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7\}$ 变成 $\{a_0,a_4,a_2,a_6,a_1,a_5,a_3,a_7\}$ ），那么，我们就可以将 **DFT** 过程写成迭代的方式，从而免去辅助空间的开销。

不妨假设 $a_r = 2^p \times b_p + 2^{p-1} \times b_{p-1} + \cdots + 2^0 \times b_0$ 在递归最底层的
序列中第 $a'_r = 2^p \times b'_p + 2^{p-1} \times b'_{p-1} + \cdots + 2^0 \times b'_0$ 个位置（从左往右，从 0 计起）。
其中，$2^{p+1} = n$; $b_i, b'_i \in \{0,1\}$。
我们来模拟下递归过程。

由上图，容易知 a'_r 必然是根的
左子树中的叶子节点，故 $b'_p = 0$

$$b_0 = 1? \xrightarrow[(\mathrm{Y})\ b'_p = 1]{(\mathrm{N})\ b'_p = 0} b_1 = 1? \xrightarrow[(\mathrm{Y})\ b'_{p-1} = 1]{(\mathrm{N})\ b'_{p-1} = 0} \cdots \quad b_p = 1? \xrightarrow[(\mathrm{Y})\ b'_0 = 1]{(\mathrm{N})\ b'_0 = 0}$$

类似地，容易知 a'_r 必然是根的
右子树中的叶子节点，故 $b'_p = 1$

不难发现，$b'_i = b_{p-i}$。

下面考虑该算法的实现。
先考虑这样的一个事实：当一个二进制数 x 自增 2^0 时，会不断试图从右往左进位。
注意到，b'_i 与 b_{p-i} 存在一一对应关系（具体地，$b'_i = b_{p-i}$ ）。不难想到，当 a_r 自增 1 时，让 a'_r 自增 2^p 并试图从左往右进位，得到的数 $a''_r = a'_{r+1}$。证明略。
为了更深刻理解这一算法，仅以 $n = 8$ （ $2^{p+1} = n$ ）为例模仿该过程：
为方便描述，记 $a'_r = a_{s(r)}$.

$0 + 2^0$ 往左进位 $\Rightarrow 1$, $s(0+1) + 2^p$ 往右进位 $\Rightarrow 4$, $\Rightarrow s(1) = 4$;
$1 + 2^0$ 往左进位 $\Rightarrow 2$, $s(1+1) + 2^p$ 往右进位 $\Rightarrow 2$, $\Rightarrow s(2) = 2$;
$2 + 2^0$ 往左进位 $\Rightarrow 3$, $s(2+1) + 2^p$ 往右进位 $\Rightarrow 6$, $\Rightarrow s(3) = 6$;
$3 + 2^0$ 往左进位 $\Rightarrow 4$, $s(3+1) + 2^p$ 往右进位 $\Rightarrow 1$, $\Rightarrow s(4) = 1$;
$4 + 2^0$ 往左进位 $\Rightarrow 5$, $s(4+1) + 2^p$ 往右进位 $\Rightarrow 5$, $\Rightarrow s(5) = 5$;
$5 + 2^0$ 往左进位 $\Rightarrow 6$, $s(5+1) + 2^p$ 往右进位 $\Rightarrow 3$, $\Rightarrow s(6) = 3$;
$6 + 2^0$ 往左进位 $\Rightarrow 7$, $s(6+1) + 2^p$ 往右进位 $\Rightarrow 7$, $\Rightarrow s(7) = 7$.

代码如下：

```
template <class T>
static void rader(complex<T>* F, int N) {
  int n = N >> 1, bit = 1, tib = n, arg;
  for (; bit < N; ++bit, tib ^= arg) {
```

```
    if (bit < tib) swap(F[bit], F[tib]);
    for (arg = n; arg & tib; arg >>= 1) tib ^= arg;
  }
}
```

之后，迭代版的 **DFT** 代码也就不难写出了。

```
template <class T>
static void transform(complex<T>* F, int N, int rev) {
  rader(F, N);
  for (int h = 2; h <= N; h <<= 1) {
    const complex<T> wn(cos(rev * 2 * PI / h), sin(rev * 2 * PI / h));
    int H = h >> 1;
    for (int i = 0; i < N; i += h) {
      complex<T> w(1, 0);
      int j = i, k = i + H;
      for (; j < k; ++j) {
        complex<T> u = F[j];
        complex<T> v = w * F[j + H];
        F[j] = u + v;
        F[j + H] = u - v;
        w = w * wn;
      }
    }
  }
}
```

那么，我们接下来的工作就是要将点值表示法，转化成系数表示法。

注意到，前文提到的 **DFT** 过程可以看做矩阵乘法：

$$\begin{bmatrix} 1 & (\mathbb{W}_n^0)^1 & (\mathbb{W}_n^0)^2 & \dots & (\mathbb{W}_n^0)^{n-1} \\ 1 & (\mathbb{W}_n^1)^1 & (\mathbb{W}_n^1)^2 & \dots & (\mathbb{W}_n^1)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\mathbb{W}_n^{n-1})^1 & (\mathbb{W}_n^{n-1})^2 & \dots & (\mathbb{W}_n^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbb{A}(\mathbb{W}_n^0) \\ \mathbb{A}(\mathbb{W}_n^1) \\ \vdots \\ \mathbb{A}(\mathbb{W}_n^{n-1}) \end{bmatrix}$$

如果我们令系数矩阵 $\mathbb{P} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$，令值矩阵 $\mathbb{R} = \begin{bmatrix} \mathbb{A}(\mathbb{W}_n^0) \\ \mathbb{A}(\mathbb{W}_n^1) \\ \vdots \\ \mathbb{A}(\mathbb{W}_n^{n-1}) \end{bmatrix}$

再令变量矩阵 $\mathbb{Q} = \begin{bmatrix} 1 & (\mathbb{W}_n^0)^1 & (\mathbb{W}_n^0)^2 & \dots & (\mathbb{W}_n^0)^{n-1} \\ 1 & (\mathbb{W}_n^1)^1 & (\mathbb{W}_n^1)^2 & \dots & (\mathbb{W}_n^1)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\mathbb{W}_n^{n-1})^1 & (\mathbb{W}_n^{n-1})^2 & \dots & (\mathbb{W}_n^{n-1})^{n-1} \end{bmatrix}$

则有：$\mathbb{QP} = \mathbb{R}$。如果能找到 $\mathbb{Q}$ 的逆矩阵 $\mathbb{Q}^{-1}$，那么 $\mathbb{Q}^{-1}\mathbb{R} = \mathbb{Q}^{-1}\mathbb{QP} = \mathbb{IP} = \mathbb{P}$（$\mathbb{I}$ 为单位矩阵），即完成了点值表示法向系数表示法的转化。这个过程称为 **IDFT**。

如何求 $\mathbb{Q}^{-1}$ 呢？虽然看上去很吓人，但 $\mathbb{Q}^{-1}$ 其实很特殊：$\mathbb{Q}^{-1}_{i,j} = \dfrac{1}{n\mathbb{Q}_{i,j}}$。其中，$(i,j)$ 为矩阵中第 i 行，第 j 列的元素。

证

$$\begin{aligned}\sum_{k=0}^{n-1}\mathbb{Q}_{i,k}\mathbb{Q}^{-1}_{k,j} &= \sum_{k=0}^{n-1}\mathbb{W}_n^{ik}\times\frac{1}{n}\times\mathbb{W}_n^{-jk}\\ &=\frac{1}{n}\sum_{k=0}^{n-1}\mathbb{W}_n^{(i-j)k}\\ &=\begin{cases}\dfrac{1}{n}\displaystyle\sum_{k=0}^{n-1}\mathbb{W}_n^0=\frac{1}{n}\times n=1 & (i=j)\\ 0 & (i\neq j)\end{cases}\text{【复数的性质 4】}\end{aligned}$$

因此，$\mathbb{Q}\mathbb{Q}^{-1}=\mathbb{I}$

证毕

因此，**IDFT** 过程可以看做作下图中的矩阵乘法：

$$\frac{1}{n}\times\begin{bmatrix}1 & (\mathbb{W}_n^0)^{-1} & (\mathbb{W}_n^0)^{-2} & \dots & (\mathbb{W}_n^0)^{-(n-1)}\\ 1 & (\mathbb{W}_n^1)^{-1} & (\mathbb{W}_n^1)^{-2} & \dots & (\mathbb{W}_n^1)^{-(n-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & (\mathbb{W}_n^{n-1})^{-1} & (\mathbb{W}_n^{n-1})^{-2} & \dots & (\mathbb{W}_n^{n-1})^{-(n-1)}\end{bmatrix}\begin{bmatrix}\mathbb{A}(\mathbb{W}_n^0)\\ \mathbb{A}(\mathbb{W}_n^1)\\ \vdots\\ \mathbb{A}(\mathbb{W}_n^{n-1})\end{bmatrix}=\begin{bmatrix}a_0\\ a_1\\ \vdots\\ a_{n-1}\end{bmatrix}$$

对比 **DFT** 过程，不难发现，我们直接套用 **DFT** 的算法框架，即可做 **IDFT**，因此，**IDFT** 过程的复杂度也是 $O(n\log n)$ 的。

另外，值得一提的是，对于两个数的乘法，我们没必要非得要按十进制来运算，实际应用时可以采用一百进制、一万进制来运算，会计算的更快（在精度允许的前提下）。

另外一个需要**注意**的地方是，要做 **FFT** 变换，要求 N 必须是 2 的幂次，若不是，则需要先补前导零补足长度；在输出时，注意一下前导零。

就此，问题完美解决!!

下面附上 **hoj10004** 的 AC 代码。。。采用的是一万进制。。。。。

温馨提示，复制代码会有奇怪的字符以致编译不过，手打吧亲。。。。

```
#include <algorithm>
#include <cmath>
#include <complex>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

```

```
10 struct FFT {
11   public:
12   static const int BIT = 4;
13   static const int BASE = (int)1e4;
14   static const long double PI;
15
16   private:
17   template <class T>
18   static void rader(complex<T>* F, int N) {
19     int n = N >> 1, bit = 1, tib = n, arg;
20     for (; bit < N; ++bit, tib ^= arg) {
21       if (bit < tib) swap(F[bit], F[tib]);
22       for (arg = n; arg & tib; arg >>= 1) tib ^= arg;
23     }
24   }
25
26   template <class T>
27   static void transform(complex<T>* F, int N, int rev) {
28     rader(F, N);
29     for (int h = 2; h <= N; h <<= 1) {
30       const complex<T> wn(cos(rev * 2 * PI / h), sin(rev * 2 * PI / h));
31       int H = h >> 1;
32       for (int i = 0; i < N; i += h) {
33         complex<T> w(1, 0);
34         int j = i, k = i + H;
35         for (; j < k; ++j) {
36           complex<T> u = F[j];
37           complex<T> v = w * F[j + H];
38           F[j] = u + v;
39           F[j + H] = u - v;
40           w = w * wn;
41         }
42       }
43     }
44   }
45
46   public:
47   template <class T>
48   static void dft(complex<T>* F, int N) {
49     transform(F, N, 1);
50   }
51   template <class T>
52   static void idft(complex<T>* F, int N) {
53     transform(F, N, -1);
54   }
55   template <class T>
56   static int convert(complex<T>* F, char* s) {
57     int len = strlen(s), rlen = len;
58     for (; len >= BIT; len -= BIT) {
59       int arg = 0;
```

```
60        for (int i = len - BIT; i < len; ++i) arg = arg * 10 + s[i] - '0';
61        *F++ = complex<T>(arg, 0);
62      }
63      if (len) {
64        int arg = 0;
65        for (int i = 0; i < len; ++i) arg = arg * 10 + s[i] - '0';
66        *F++ = complex<T>(arg, 0);
67      }
68      return rlen;
69    }
70  };
71
72  const long double FFT::PI = acos(-1.0);
73
74  typedef unsigned __int64 LL;
75  const int MAXN = 1 << 20;
76  const int MAXS = (int)1e6 + 10;
77  int T_T, N, ans[MAXN];
78  char in[MAXS];
79  complex<long double> F[2][MAXN];
80
81  void work() {
82    scanf("%d", &T_T);
83    while (T_T--) {
84      memset(F, 0, sizeof F);
85      scanf("%s", in);
86      N = FFT::convert(F[0], in);
87      scanf("%s", in);
88      N = max(N, FFT::convert(F[1], in));
89      int len = (N - 1) / FFT::BIT + 1;
90      for (N = 1; N < len; N <<= 1)
91        ;
92      N <<= 1;
93
94      FFT::dft(F[0], N);
95      FFT::dft(F[1], N);
96      for (int i = 0; i < N; ++i) F[0][i] *= F[1][i];
97      FFT::idft(F[0], N);
98      for (int i = 0; i < N; ++i) F[0][i] /= complex<long double>(N, 0);
99      for (int i = 0; i < N; ++i) {
100       LL arg = (LL)round(F[0][i].real());
101       ans[i] = arg % FFT::BASE;
102       F[0][i + 1] += complex<long double>(arg / FFT::BASE, 0);
103     }
104
105     while (N > 1 && !ans[N - 1]) --N;
106     printf("%d", ans[--N]);
107     while (N >= 1) printf("%04d", ans[--N]);
108     putchar('\n');
109   }
```

```
110 }
111
112 int main() {
113   work();
114   return 0;
115 }
```