FFT 学习笔记

看了好几篇 $blog$ ，结果发现看一句忘一句，所以自己写一下。

[注]：本文中很多概念不一定准确，有的纯粹是为了好理解，如果与其他 $blog$ 出现冲突，那一定是我错了。

突然发现几乎所有的 $blog$ 目录都是一个套路，于是我也这样写吧。

- 多项式基本知识
    - 多项式乘法
    - 多项式的点值表示和系数表示
- 复数
    - 复数基本知识
        - 复数相乘的几何意义
    - 单位根
        - 介绍
        - 求解
- FFT
    - 基本操作
        - DFT
        - $IDFT$
        - $DFT+IDFT=FFT$
    - 蝴蝶变换
- 最终的代码

多项式基本知识

多项式乘法：

~~就是把两个多项式乘起来~~

举个例子

$F(x)=\sum_{i=0}^{n-1}a_ix^i$

$G(x)=\sum_{i=0}^{m-1}b_ix^i$

$F\times G(x)=\sum_{i=0}^{n+m-2}\sum_{j=0}^{i}a_jb_{i-j}x^i$

```
scanf("%d%d",&n,&m);
for (R i=n-1;i>=0;--i) scanf("%d",&f[i]);
for (R i=m-1;i>=0;--i) scanf("%d",&g[i]);
for (R i=0;i<=n-1;++i)
    for (R j=0;j<=m-1;++j)
        h[i+j]+=f[i]*g[j];
for (R i=n+m-2;i>=0;--i)
    printf("%d ",h[i]);
```

这个做法的复杂度是$O(N^2)$的，有一点慢，还可以优化吗？FFT 就是用来做这个的。

多项式的点值表示和系数表示及其互相转换

在这一小节，统一以一个比较简单的多项式 $F(x)=x^2+x+1$ 为例。

对于一个 $n-1$ 多项式，把它视为一个 $n-1$ 次函数，可以用平面直角坐标系中的 n 个点唯一确定。所以有的时候也可以直接用 n 个点表示一个多项式，这就是多项式的点值表示：

$(-1,1)$，$(1,3)$，$(2,7)$

系数表示也很简单，就是把系数列出来，放进一个一维数组里：

$[1\text{，}1\text{，}1]$

接下来是互相转换，其实在小学/初中应该就学过。

点值转系数：高斯消元；

$$\begin{cases} a_2-a_1+a_0=1 \\ a_2+a_1+a_0=3 \\ 4a_2+2a_1+a_0=7 \end{cases}$$

系数转点值：

找一些数代入求值就好了。

这里有一个很好用的性质，如果已经知道了两个函数的点值表达，就可以 $O(n)$ 求出它们的积的点值表达。$f(x)\times g(x)=f\times g(x)$ 至于为什么，把多项式乘法那个式子展开来看一下就会明白了。

注意，这要求两个因子多项式求点值表示时选择的 x 相同，而且要代入 $n+m-1$ 个值，这样求出来的新点值表示才能唯一确定它们的积。

这样 FFT 的基本框架就有了，首先把给出的两个多项式的系数表示转成点值表示，再乘起来，最后转为答案的系数表示即可。

系数转点值$\rightarrow DFT$

点值转系数$\rightarrow IDFT$

这里还有最后一个问题，也是最重要的问题：系数转点值是 $O(N^2)$,点值转系数是 $O(N^3)...$

还需要优化。

复数

要不要说一下这里为什么要讲复数？~~因为复数很有趣啊~~

复数基本知识

复数和我们平常所说的数看起来差别很大，它由两部分组成：实部和虚部。看起来像是这样的：

$a+bi$，那么 i 是什么？ $i=\sqrt{-1}$

这看起来很...奇怪？先不管奇怪不奇怪，它有什么实际用处吗？答案是肯定的，有了 i ，我们可以解一切一元二次方程，因为...现在负数可以开方了...

考虑虚数的几何意义，数轴上显然是没有它的位置的，最多只能放下一个实部，所以给数轴加一维，把虚部也放进去。就像一个平面直角坐标系。

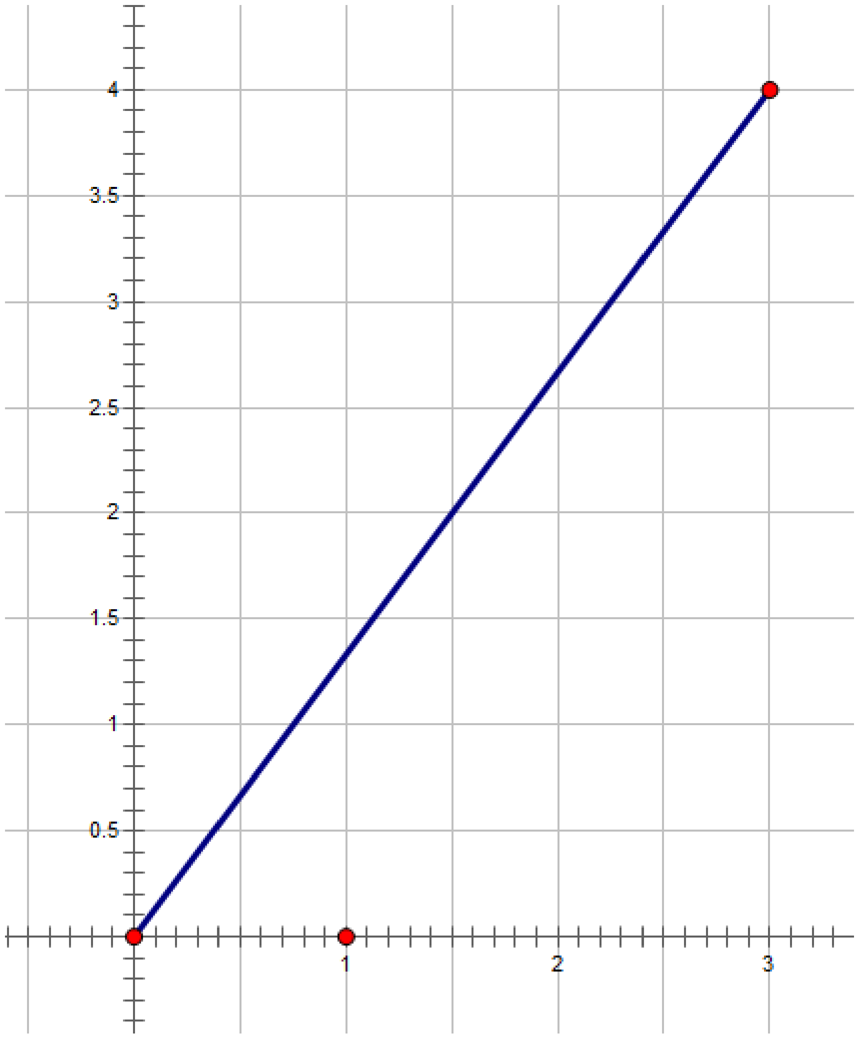
虚数的模长：就是把虚数表示到坐标系里后与原点的距离。$\sqrt{a^2+b^2}$

虚数的幅角：想象一下，把点和原点的连线延长成一条直线，这条线的倾斜角就是虚数的幅角，

计算方法：$cot\frac{b}{a}$

虚数完全可以仅用模长 l 和幅角 θ 表示，转换：

$lcos\theta+l\sin\theta i$



复数长得更像二维平面上的坐标，复数的运算更像向量的运算。

$(a+bi)+(c+di)=(a+c)+(b+d)i$

$(a+bi)-(c+di)=(a-c)+(b-d)i$

$(a+bi)(c+di)=ac+adi+bci+bdi^2=(ac-bd)+(ad+bc)i$

$\frac{(a+bi)}{(c+di)}=\frac{(a+bi)(c-di)}{(c+di)(c-di)}=\frac{(ac+bd)+(bc-ad)i}{c^2+d^2}=\frac{(ac+bd)}{c^2+d^2}+\frac{(bc-ad)}{c^2+d^2}i$

```
struct complex
{
    double r,c;
    complex (double r=0,double c=0):r(r),c(c) {}
    void read () { scanf("%lf%lf",&r,&c); }
    void write () { printf("%.5lf+%.5lfi",r,c); }
};
complex operator + (complex a,complex b) { return complex(a.r+b.r,a.c+b.c); }
complex operator - (complex a,complex b) { return complex(a.r-b.r,a.c-b.c); }
complex operator * (complex a,complex b) { return complex(a.r*b.r-a.c*b.c,a.r*b.c+a.c*b.r); }
complex operator / (complex a,complex b) { return complex((a.r*b.r+a.c*b.c)/(b.r*b.r+b.c*b.c),
(a.c*b.r-a.r*b.c)/(b.r*b.r+b.c*b.c)); }
```

复数相乘的几何意义

前置知识：三角函数的公式

$cos(\alpha+\beta)=cos\alpha\cos\beta-sin\alpha sin\beta$

$sin(\alpha+\beta)=sin\alpha\cos\beta+cos\alpha sin\beta$

两个复数 f_1,f_2

$f_1f_2=l_1(cos\theta_1+sin\theta_1 i)l_2(cos\theta_2+sin\theta_2 i)$

$=l_1l_2(cos(\theta_1+\theta_2)+sin(\theta_1+\theta_2)i)$

这时候的形式看起来和之前完全一样，所以复数相乘得到的结果就是：

模长相乘，幅角相加。

单位根

单位圆就是半径为一的圆，我们把它的圆心点到坐标原点。

看看这个方程 $x^n=1$

这个方程有几个解？看这篇文章之前我们知道，如果n是0，那么有无数个，如果n是偶数，有1，-1，否则就只有 1 了。

但是现在有了复数这种奇妙的东西，答案就不同了。

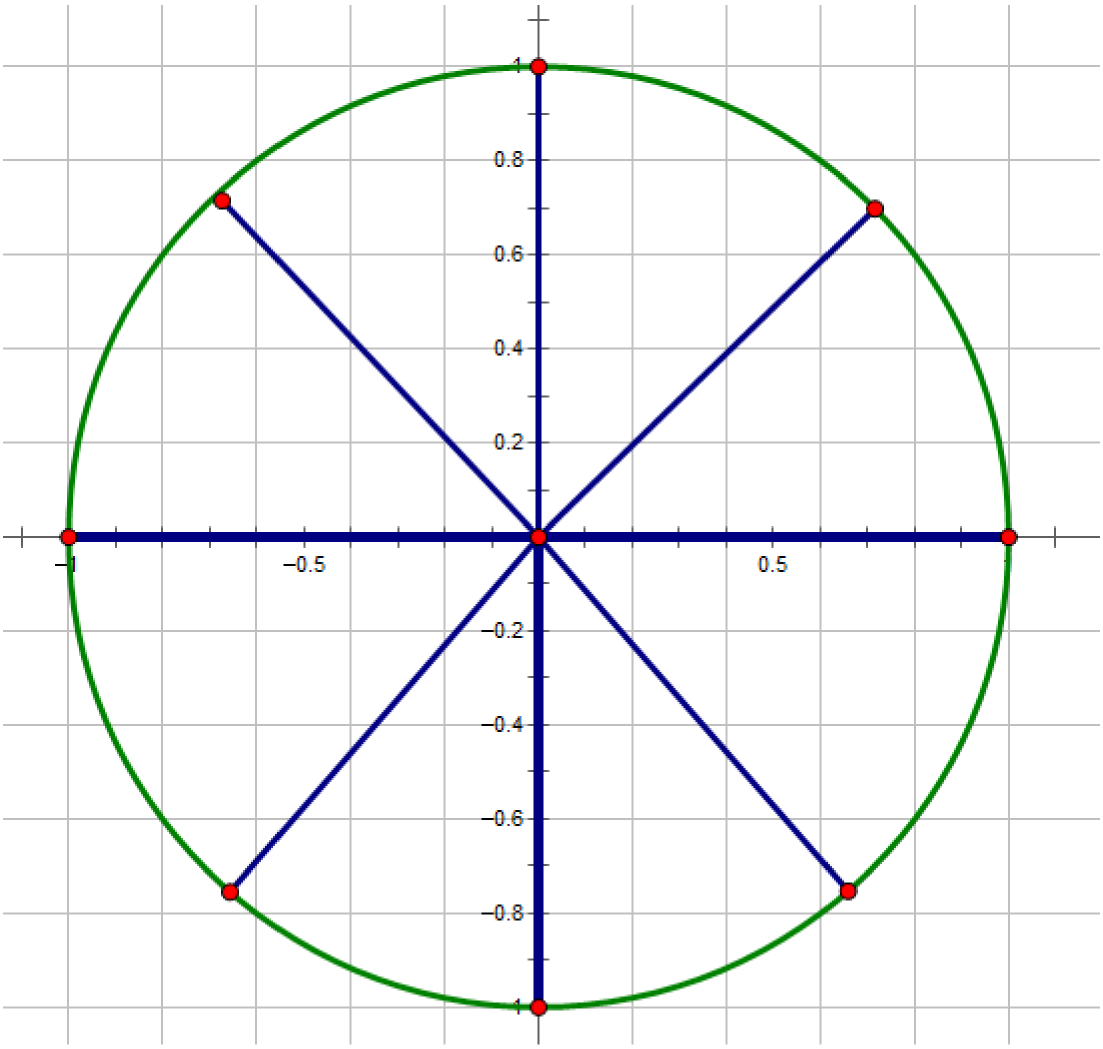
首先将一个普通的 1 看做复数，那么 1 的模长为 1，幅角为 0 ；

所以现在要求的就是一个复数，它的 n 次幂等于 $1+0i$ ，如果从几何意义上来看，就是 $l^n=1$,$n\theta|2\pi$ (弧度制)

所以 $\frac{2\pi}{n}|\theta$

感性理解一下，就是 n 个大小相同的角，叠加起来正好是一个圆的整数倍，举个例子吧：

$x^8=1$



显然 45^o 角的 8 倍满足要求，其实 90 度也是满足的。因为弧度制我用的还不是很好，所以接下来还是用角度制说。n 次单位根 n 等分单位圆，为什么？

$360^o|360^o\frac{x}{n}n$

这样就很显然了。

接下来是 n 次单位根。

实际上就是幅角为 $\frac{2\pi}{n}$ 的复数，把它完整的表示出来就是 $cos\frac{2\pi}{n}+isin\frac{2\pi}{n}$。

我们给它一个新的符号 ω_n ，虽然我的印象中 $omega$ 是电阻那个符号来着.....，但是这个符号的latex公式才叫 $omega$

对于幅角为 $\frac{2k\pi}{n}$ 的单位根，我们就叫它 ω_n^k

关于单位根，网上往往会写很多的性质，但是只要把它当做一条射线来看，这些性质就都不难理解。

$\omega_{2n}^{2k}=\omega_n^k$ ，$2n$ 份中取 $2k$ 份和 n 份中取 k 份当然是一样多的.

如果 n 是偶数，$\omega_n^{k+n/2} = -\omega_n^k$ ，这里当做射线来想！射线转上半圈，就等于原来的反方向了。

$(\omega_n^k)^j = (\omega_n^j)^k$，模长都是一，怎么乘也是一，角度相加，那么…$k \times j = j \times k$

$\omega_n^k = \omega_n^{k\%n}$ ，多转一圈等于没转

但是！单位根依旧是数，不是线！后面代入多项式的时候一定要把思维转过来。~~我就想了好久怎么把一条线带入多项式~~

那么怎么求单位根呢？非常简单~

首先可以知道，$(\omega_n^k)^j = \omega_n^{kj}$，那么只要求出 $\omega_n^x = (\omega_n^1)^x$

所以只要求出 ω_n^1 ，就可以求出所有需要的单位根了，真开心。接着回到几何意义上来，现在有这么一个直角三角形，斜边长是单位圆的半径 1 ，角度是 $\frac{2\pi}{n}$ ，它的两条直角边长度就都很好求了。

又到了愉快的代码时间！

```
scanf("%d",&n);
og[0]=complex(1,0);
og[1]=complex(cos(Pi2/n),sin(Pi2/n));
for (R i=2;i<n;++i) og[i]=og[i-1]*og[1];
for (R i=0;i<n;++i) og[i].write();
```

快速傅里叶变换

这好像才是这篇文章的主角吧...

它现在终于出现了。

基本操作

本节的例子是 $F(x) = x^3 + 2x^2 + 3x + 4$

因为 FFT 是要分治来做的，如果分的不均匀就会很麻烦，所以多项式的补成 2^x 项就会非常方便，不管怎么分都是平分。怎么补？多加一些高次项，系数为 0 不就好咯？

DFT

把多项式按照次数奇偶性拆开成两部分，定义为 $G(x), H(x)$

$G(x) = x + 3$

$H(x) = 2x + 4$

这样，$F(x) = xG(x^2) + H(x^2)$

假设 $k < \frac{n}{2}$，将 ω_n^k 代入,就有:

$F(\omega_n^k) = \omega_n^k G((\omega_n^k)^2) + H((\omega_n^k)^2)$

$\therefore F(\omega_n^k) = \omega_n^k G(\omega_{n/2}^k) + H(\omega_{n/2}^k)$

这样递归的分治下去，每一项最多被算 $logN$ 次，这样的效率还是挺不错的。

刚刚我们人为的定义了 $k <= n/2$ ，所以求值也只能求出一半，现在想一下怎么求另一半：

这一次把刚刚没有代入的项代入 $\to \omega_n^{k+n/2}$

$$F(\omega_n^{k+n/2}) = \omega_n^{k+n/2} G((\omega_n^{k+n/2})^2) + H((\omega_n^{k+n/2})^2)$$

$$\therefore F(\omega_n^{k+n/2}) = \omega_n^{k+n/2} G(\omega_n^{2k+n}) + H(\omega_n^{2k+n})$$

$$\therefore F(\omega_n^{k+n/2}) = \omega_n^{k+n/2} G(\omega_n^{2k}) + H(\omega_n^{2k})$$

$$\therefore F(\omega_n^{k+n/2}) = -\omega_n^{k} G(\omega_{n/2}^{k}) + H(\omega_{n/2}^{k})$$

诶，好像跟前半部分差不多？只是 G 的正负变了一变而已。

这也是 FFT 快的原因，逐层分下去后每层有一半是可以不用算的，左右两边只需要一共算一次，剩下的靠翻转，所以就会快了。

递归地求下去，直到只剩下一项，那肯定是一个 0 次项，返回即可。

```
void DFT (complex *f,int len)
{
    if(!len) return ;
    complex g[len+1],h[len+1];
    for (R i=0;i<=len;++i)
        g[i]=f[i<<1|1],h[i]=f[i<<1];
    DFT(g,len>>1);
    DFT(h,len>>1);
    complex og1,og;
    len<<=1;
    og=complex(1,0);
    og1=complex(cos(Pi*2/len),sin(Pi*2/len));
    for (R k=0;k<len/2;++k)
    {
        f[k]=og*g[k]+h[k];
        f[k+len/2]=h[k]-og*g[k];
        og=og1*og;
    }
}
```

复数除法在这里用不到，可以不写。

IDFT

刚才那个部分确实很快，但是求出来的是点值表达，怎么把它转换成系数表达呢？

把求点值表示的过程写成矩阵乘法。

$$\begin{pmatrix} (\omega_n^0)^0 & (\omega_n^0)^1 & \dots & (\omega_n^0)^{n-1} \\ (\omega_n^1)^0 & (\omega_n^1)^1 & \dots & (\omega_n^1)^{n-1} \\ \dots & \dots & \dots & \dots \\ (\omega_n^{n-1})^0 & (\omega_n^{n-1})^1 & \dots & (\omega_n^{n-1})^{n-1} \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} F(0) \\ F(1) \\ \dots \\ F(n-1) \end{pmatrix}$$

记得简单一点：$AB = C$

现在通过 DFT 求出了 C ，希望求出 B ，怎么求？

显然 $\frac{C}{A}=B$ .....

只要一个矩阵除法，就可以解决问题了！~~梦里什么都有~~

$CA^{-1}=B$

这真的很科学。

矩阵求逆：https://www.luogu.org/problemnew/show/P4783

可惜矩阵求逆的复杂度是 $O(n^3)$

不过对于这类特殊的矩阵，可以直接把逆矩阵找出来背过，先摆结论：

$$\frac{1}{n}\times\begin{pmatrix}(\omega_n^0)^0 & (\omega_n^0)^1 & \dots & (\omega_n^0)^{n-1}\\ (\omega_n^{-1})^0 & (\omega_n^{-1})^1 & \dots & (\omega_n^{-1})^{n-1}\\ \dots & \dots & \dots & \dots\\ (\omega_n^{-n+1})^0 & (\omega_n^{-n+1})^1 & \dots & (\omega_n^{-n+1})^{n-1}\end{pmatrix}$$

来证明一下这两个矩阵乘起来真的是单位矩阵，就说明这是逆矩阵了。为了写起来方便，给逆矩阵一个新的名字：$S=A^{-1}$ $SA=I$ ~~多写几遍SA可以顺便复习字符串~~

找规律时间，$A_{ij}=\omega_n^{ij}$, $S_{ij}=\frac{1}{n}\omega_n^{-ij}$

$C_{ij}=\sum_{k=0}^{n-1}A_{ik}S_{kj}$

$=\frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{ik}\omega_n^{-kj}$

$=\frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{ik-kj}$

$if(i=j)$

$C_{ij}=\frac{1}{n}\sum_{k=0}^{n}\omega_n^0=\frac{1}{n}\sum_{k=0}^{n-1}1=1$

$else$

$x=i-j$, $C_{ij}=\frac{1}{n}\sum_{k=0}^{n}\omega_n^{kx}=\frac{1}{n}\sum_{k=0}^{n}(\omega_n^x)^k$

出现了一个等比数列求和

$=\frac{1}{n}\frac{1-\omega^{xn}}{\omega_n^x-1}=\frac{1}{n}\frac{1-\omega_n^0}{\omega_n^x-1}=\frac{1}{n}\frac{1-1}{\omega_n^x-1}=0$

得证：$C=I\rightarrow SA=I$

所以可以得到：

$$\begin{pmatrix}\frac{1}{n}(\omega_n^0)^0 & \frac{1}{n}(\omega_n^0)^1 & \dots & \frac{1}{n}(\omega_n^0)^{n-1}\\ \frac{1}{n}(\omega_n^{-1})^0 & \frac{1}{n}(\omega_n^{-1})^1 & \dots & \frac{1}{n}(\omega_n^{-1})^{n-1}\\ \dots & \dots & \dots & \dots\\ \frac{1}{n}(\omega_n^{-n+1})^0 & \frac{1}{n}(\omega_n^{-n+1})^1 & \dots & \frac{1}{n}(\omega_n^{-n+1})^{n-1}\end{pmatrix}\times\begin{pmatrix}F(0)\\ F(1)\\ \dots\\ F(n-1)\end{pmatrix}=\begin{pmatrix}a_0\\ a_1\\ \dots\\ a_{n-1}\end{pmatrix}$$

```
void IDFT (complex *f,int len)
{
    if(!len) return ;

    complex g[len+1],h[len+1];
```

```
    for (R i=0;i<=len;++i)
        g[i]=f[i<<1|1],h[i]=f[i<<1];
    IDFT(g,len>>1);
    IDFT(h,len>>1);
    complex og1,og;
    len<<=1;
    og=complex(1,0);
    og1=complex(cos(Pi*2/len),-sin(Pi*2/len));
    for (R k=0;k<len/2;++k)
    {
        f[k]=og*g[k]+h[k];
        f[k+len/2]=h[k]-og*g[k];
        og=og1*og;
    }
}
```

$DFT+IDFT=FFT$

上述两个函数组合起来就是 FFT 啦！

注意到 DFT 和 $IDFT$ 非常的相似，只有初始单位根不同，所以改一改，传一个参数告诉函数这次是哪一种，就可以简化代码。

下面是一份普通的 FFT ：

```
# include <cstdio>
# include <iostream>
# include <cmath>
# define R register int

using namespace std;

const int maxn=2000006;
const double Pi=acos(-1);
int n,m,len;
struct complex
{
    double r,c;
    complex (double r=0,double c=0):r(r),c(c) {}
    void read () { scanf("%lf%lf",&r,&c); }
}f[maxn],g[maxn];
complex operator + (complex a,complex b) { return complex(a.r+b.r,a.c+b.c); }
complex operator - (complex a,complex b) { return complex(a.r-b.r,a.c-b.c); }
complex operator * (complex a,complex b) { return complex(a.r*b.r-a.c*b.c,a.r*b.c+a.c*b.r); }

void FFT (complex *f,int len,int v)
{
    if(!len) return ;
    complex g[len+1],h[len+1];
    for (R i=0;i<=len;++i)

        g[i]=f[i<<1|1],h[i]=f[i<<1];
```

```
    FFT(g,len>>1,v);
    FFT(h,len>>1,v);
    complex og1,og;
    len<<=1;
    og=complex(1,0);
    og1=complex(cos(Pi*2/len),v*sin(Pi*2/len));
    for (R k=0;k<len/2;++k)
    {
        f[k]=og*g[k]+h[k];
        f[k+len/2]=h[k]-og*g[k];
        og=og1*og;
    }
}

int main()
{
    scanf("%d%d",&n,&m);
    for (R i=0;i<=n;++i) scanf("%lf",&f[i].r),f[i].c=0;
    for (R i=0;i<=m;++i) scanf("%lf",&g[i].r),g[i].c=0;
    len=1; while(len<n+m+1) len<<=1;
    FFT(f,len>>1,1);
    FFT(g,len>>1,1);
    for (R i=0;i<=len;++i) f[i]=f[i]*g[i];
    FFT(f,len>>1,-1);
    for (R i=0;i<=m+n;++i)
    {
        f[i].r/=len;
        if(fabs(f[i].r)<=0.00001) f[i].r=0;
    }
    for (R i=0;i<=m+n;++i) printf("%.0lf ",f[i].r);
    return 0;
}
```

于是我把这份代码交到了 LOJ 上，全都 RE 了...

LOJ 全都是极限数据，没递归几层就已经爆栈了，所以要想一种不需要递归的方法。

蝴蝶变换

手动模拟一下这个递归的过程，看看在每一层中，每个位置保存的是原来哪个位置的编号；

$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7$

$0 \quad 2 \quad 4 \quad 6 \mid 1 \quad 3 \quad 5 \quad 7$

$0 \quad 4 \mid 2 \quad 6 \mid 1 \quad 5 \mid 3 \quad 7$

$0 \mid 4 \mid 2 \mid 6 \mid 1 \mid 5 \mid 3 \mid 7$

单独拿出第一行，第四行来看：

000 001 010 011 100 101 110 111

000 100 010 110 001 101 011 111

好像有一些神奇的性质？

第一行中的位置用二进制表示出来，并进行镜像翻转后，就成为了第四行。（似乎镜像翻转就叫蝴蝶变换 ）

这样，我们就可以一步到位，直接求出第四行来，再逐步向上合并，~~效率极高~~

问题是怎么求二进制数的蝴蝶变换？递推。

从小到大循环，保证处理某个数的时候，比它小的数都处理完了。

取出这个数的前几位（除了最后一位）$i >> 1$

这个数的镜像已经求出来了，但是它的末尾一位是多出来补位用的，把它去掉，再看一看原数的末位是不是一，如果是，就把新数的首位填上一。

```
for (R i=1;i<len;++i) rev[i]=(rev[i>>1]>>1)|((i&1)?len>>1:0);
```

最终代码

可以发现，对于任意一层蝴蝶变换，左区间的数变换后都比自己原先的编号要大，所以首先将左区间向上合并，把空间留下来用于右侧的合并，就可以完美的避开合并时互相覆盖，甚至不需要辅助空间，非常优秀。

```
# include <cstdio>
# include <iostream>
# include <cmath>
# define R register int

using namespace std;

const int maxn=3000006;
const double Pi=acos(-1);
int n,m,len,rev[maxn],ln;
struct complex
{
    double r,c;
    complex (double r=0,double c=0):r(r),c(c) {}
    void read () { scanf("%lf%lf",&r,&c); }
}f[maxn],g[maxn],og,og1,t;
complex operator + (complex a,complex b) { return complex(a.r+b.r,a.c+b.c); }
complex operator - (complex a,complex b) { return complex(a.r-b.r,a.c-b.c); }
complex operator * (complex a,complex b) { return complex(a.r*b.r-a.c*b.c,a.r*b.c+a.c*b.r); }

inline void FFT (complex *f,int v)
{
    for (R i=0;i<len;++i) if(i<rev[i]) swap(f[i],f[ rev[i] ]);
    for (R i=2;i<=len;i<<=1)
    {
        ln=i>>1; //子区间的长度
        og1=complex(cos(Pi/ln),v*sin(Pi/ln));
        for (R l=0;l<len;l+=i) //左端点
        {
            og=complex(1,0);
            for (R x=l;x<l+ln;++x)
```

```
                {
                    t=og*f[ln+x];
                    f[ln+x]=f[x]-t;
                    f[x]=f[x]+t;
                    og=og1*og;
                }
            }
        }
}

int main()
{
    scanf("%d%d",&n,&m);
    for (R i=0;i<=n;++i) scanf("%lf",&f[i].r),f[i].c=0;
    for (R i=0;i<=m;++i) scanf("%lf",&g[i].r),g[i].c=0;
    len=1; while(len<n+m+1) len<<=1;
    for (R i=1;i<len;++i) rev[i]=(rev[i>>1]>>1)|((i&1)?len>>1:0);
    FFT(f,1);
    FFT(g,1);
    for (R i=0;i<=len;++i) f[i]=f[i]*g[i];
    FFT(f,-1);
    for (R i=0;i<=m+n;++i)
    {
        f[i].r/=len;
        if(fabs(f[i].r)<=0.00001) f[i].r=0;
    }
    for (R i=0;i<=m+n;++i) printf("%.0lf ",f[i].r);
    return 0;
}
```

仅仅只有60行，还是挺短的。

刚刚看了一篇国集的论文，发现那上面把分奇偶性的部分叫做蝴蝶变换，所以到底蝴蝶变换指的是分治的部分还是下标翻转呢...？

---shzr