摘自：https://blog.csdn.net/enjoy_pascal/article/details/81478582

# 快速傅里叶变换 FFT

## FFT前言

> 快速傅里叶变换(fast Fourier transform，FFT)，即利用计算机计算离散傅里叶变换(DFT)的高效、快速计算方法的统称。
>
> 快速傅里叶变换根据离散傅氏变换的奇、偶、虚、实等特性，对离散傅立叶变换的算法进行改进获得的。
>
> 采用这种算法能使计算机计算离散傅里叶变换所需要的乘法次数大为减少，特别是被变换的抽样点数N越多，FFT算法计算量的节省就越显著。

- 朴素高精度乘法时间复杂度：$O(n^2)$
- FFT时间复杂度：$O(n \cdot logn)$

## 多项式的系数表示法和点值表示法

> FFT其实是个用 O(nlogn) 时间将多项式从系数表示转换为点值表示的算法

### 系数表示法

- n-1次n项的多项式 $f(x)$ 可以表示为：$f(x)=\sum_{i=0}^{n-1} a_i x^i$
- 也即用每一项的系数来表示 $f(x)$ 为：$f(x)=\{a_0,a_1,a_2,\cdots,a_{n-1}\}$

### 点值表示法

- 多项式 $f(x)$ 在二维笛卡尔坐标系中，看作函数，不同的 $x$ 对应着不同的 $y$ ，$(x,y)$ 即一个点
- 用 $n$ 个点**唯一确定**该多项式，**有且仅有**一个多项式满足：$\forall k, f(x_k)=y_k$
  - 因为 $n$ 条式子联立即 $n$ 条方程的 $n$ 元方程组，每一项系数都可解
- 点值表示法即 $f(x)=\{(x_0,f(x_0)),(x_1,f(x_1)),\cdots,(x_{n-1},f(x_{n-1})\}$

### 高精度乘法下两种表示法的区别

- 两个**系数表示**的多项式 $A(x)$、$B(x)$ 相乘，复杂度 $O(n^2)$ 但 若采用**点值表示**，复杂度 $O(n)$
- 设两个点值多项式分别为：$f(x)=\{(x_0,f(x_0)),(x_1,f(x_1)),\cdots,(x_{n-1},f(x_{n-1})\}$ 则乘积
  $$g(x)=\{(x_0,g(x_0)),(x_1,g(x_1)),\cdots,(x_{n-1},g(x_{n-1})\}$$
  $$h(x)=\{(x_0,f(x_0)\cdot g(x_0)),(x_1,f(x_1)\cdot g(x_1)),\cdots,(x_{n-1},f(x_{n-1})\cdot g(x_{n-1}))\}$$
- 离散傅里叶变换 **DFT**：朴素的系数转点值算法 离散傅里叶逆变换 **IDFT** ：点值转系数

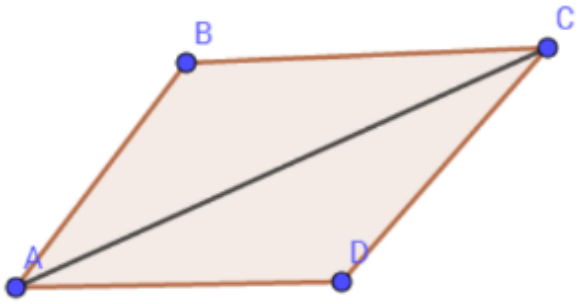
## FFT前置

知识回忆

### 复数

- $Z=a+bi$ 在复平面上可以是一个点 $(a,b)$
- $Z=a\pm bi$ 的共轭复数 $\overline{Z}=a\mp bi$
- 复数加法：$z_1=a+bi$，$z_2=c-di$，则 $z_1+z_2=(a+c)+(b+d)i$，减法类似。

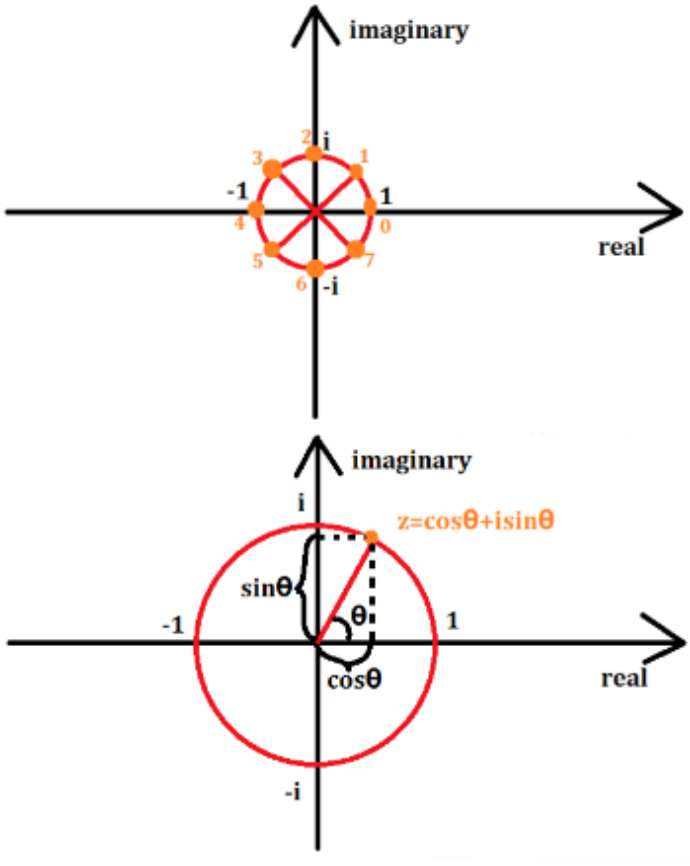
- 复数乘法：$z_1 = a + bi$，$z_2 = c - di$，则 $z_1 \cdot z_2 = (ac + bd) + (ad + bc)i$，除法为上下同乘分母的共轭复数再化简。
- 复数相乘的性质：模长相乘、极角相加，$(\alpha_1, \theta_1) \times (\alpha_2, \theta_2) = (\alpha_1\alpha_2, \theta_1 + \theta_2)$




## DFT

> 从这里开始所有的 n 为 $n = 2^{整数}$

- 对于系数转点值，**随意**取 $n$ 个 $x_i$ 值代入多项式 $f(x)$ 计算，每个 $x$ 都要暴力计算 $x_i^0, x_i^1, \cdots, x_i^{n-1}$，$i \in [0, n)$ 是 $O(n^2)$ 时间的。 傅里叶的思想则是取一组**神奇**的 $x$，使得代入后不需要做那么多运算。
- 考虑取一些 $x$ ，使每个 $x$ 的**若干次方**等于 $1$，这样我们就不用做全部的幂运算。 复平面单位圆上的点可满足要求，将单位元 n=8 等分且对标号如下，：




- 记 n 等分下编号为 k 的点代表的复数值 $z = cos\theta + isin\theta$ 为 $w_n^k$ 由 “模长相乘、极角相加” 得 $(w_n^1)^k = w_n^k$ 其中 $w_n^1$ 成为 $n$ 次单位根， $w_n^k = cos(\frac{k}{n}2\pi) + isin(\frac{k}{n}2\pi)$
- $w_n^0, w_n^1, \cdots, w_n^{n-1}$ 即为我们要代入的 $x_0, x_1, \cdots, x_{n-1}$

## 单位根的一些性质

- $w_n^k = w_{2n}^{2k}$
  - 即它们表示的点(向量、复数)是相同的
  - 证明： $w_n^k = cos(\frac{k}{n}2\pi) + isin(\frac{k}{n}2\pi) = cos(\frac{2k}{2n}2\pi) + isin(\frac{2k}{2n}2\pi) = w_{2n}^{2k}$

- $w_n^{k+\frac{n}{2}} = -w_n^k$
  - 即它们表示的点(向量)关于原点对称，所表示的复数实部相反
  - 证明：$w_n^{\frac{n}{2}} = cos(\frac{\frac{n}{2}}{n}2\pi) + isin(\frac{\frac{n}{2}}{n}2\pi) = cos\pi + isin\pi = -1$
- $w_n^0 = w_n^n$
  - 都等于 1 或 1+0i

# FFT

> DFT还是暴力 $O(n^2)$：虽然代入这类特殊 $x$ 值叫 DFT，但是还需代入单位根暴力计算。 FFT 即是将 DFT 用分治来做

## 朴素FFT

设一个多项式 $A(x)$：

- $A(x) = \sum_{i=0}^{n-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$

按 $A(x)$ 下标的奇偶性把 $A(x)$ 分成两半，右边再提一个 $x$ ：

- $$\begin{aligned} A(x) &= (a_0 + a_2 x^2 + \cdots + a_{n-2} x^{n-2}) + (a_1 x + a_3 x^3 + \cdots + a_{n-1} x^{n-1}) \\ &= (a_0 + a_2 x^2 + \cdots + a_{n-2} x^{n-2}) + x(a_1 + a_3 x^2 + \cdots + a_{n-1} x^{n-2}) \end{aligned}$$

用 $A_1(x)$、$A_2(x)$ 代替 $A(x)$ 中相似的两部分：

- $A_1(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1}$
  $A_2(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}$

则明显有：

- $A(x) = A_1(x^2) + xA_2(x^2)$

再设 $k < \frac{n}{2}$，把 $w_n^k$ 作为 $x$ 代入 $A(x)$：

- $$\begin{aligned} A(w_n^k) &= A_1((w_n^k)^2) + w_n^k A_2((w_n^k)^2) \\ &= A_1(w_n^{2k}) + w_n^k A_2(w_n^{2k}) \\ &= A_1(w_{\frac{n}{2}}^k) + w_n^k A_2(w_{\frac{n}{2}}^k) \end{aligned}$$

对于 $w_n^{k+\frac{n}{2}}$ 有：

- $$\begin{aligned} A(w_n^{k+\frac{n}{2}}) &= A_1(w_n^{2k+n}) + w_n^{k+\frac{n}{2}} A_2(w_n^{2k+n}) \\ &= A_1(w_n^{2k+n}) - w_n^k A_2(w_n^{2k+n}) \\ &= A_1(w_n^{2k} w_n^n) - w_n^k A_2(w_n^{2k} w_n^n) \\ &= A_1(w_n^{2k}) - w_n^k A_2(w_n^{2k}) \\ &= A_1(w_{\frac{n}{2}}^k) - w_n^k A_2(w_{\frac{n}{2}}^k) \end{aligned}$$

$A(w_n^k)$ 和 $A(w_n^{k+\frac{n}{2}})$ 仅第二项符号相异 即已知 $A_1(w_{\frac{n}{2}}^k)$、$A_2(w_{\frac{n}{2}}^k)$ 则知 $A(w_n^k)$ 和 $A(w_n^{k+\frac{n}{2}})$

每一次回溯时，只扫描当前前面一半的序列，即可得到后面一半的序列答案，$O(n \cdot logn)$

## IFFT

快速傅里叶逆变换。

我们把两个多项式相乘（也叫求卷积），做完两遍 FFT 求到这两个多项式的点值表示，也得到了积的多项式点值表示

由于常用系数表示多项式，现在用 $O(n \cdot logn)$ 将点值表示转回系数表示

记住结论：**一个多项式在分治的过程中乘上单位根的共轭复数，分治完的每一项除以 $n$ ，即为原多项式的每一项系数**

## 递归版代码

利用C++自带的complex库：

FFT 函数中传入的 `complex<double>` 数组应该是幂次从低到高的 $x$ 的系数

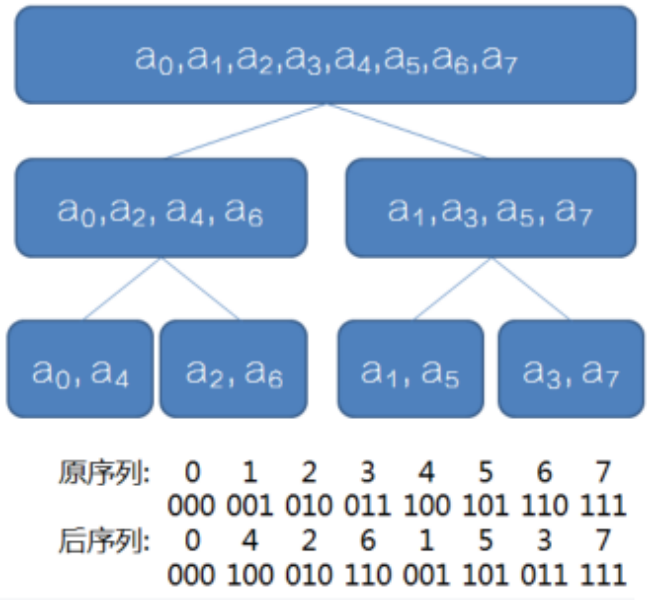
```
// luogu-P1919
#include <stdio.h>
#include <string.h>
#include <complex>
using namespace std;
typedef complex<double> cp;
#define rep(i,s,t) for(int i=s;i<=t;i++)
#define rrep(i,s,t) for(int i=s;i>=t;i--)
const int MAXN = 1e6+5;
const int MAXS = 6e4+5;
const double pi = acos(-1.0);
int rev[MAXN];
cp a[MAXN], b[MAXN];
int n, bit=2, output[MAXN];
char s1[MAXS], s2[MAXS];

void FFT(cp *a, int n, int inv) // inv 代表是否取共轭复数
{
    if(n == 1) return;
    int mid = n>>1;
    static cp tmp[MAXN];
    // 奇偶下标分离到 mid 左右
    rep(i, 0, mid-1) tmp[i] = a[i<<1], tmp[i+mid] = a[i<<1|1];
    memcpy(a, tmp, sizeof(cp)*n);
    // 分治
    FFT(a, mid, inv); FFT(a+mid, mid, inv);
    // 枚举x计算A(x)
    rep(i, 0, mid-1)
    {
        cp x(cos(2*pi*i/n), inv*sin(2*pi*i/n)); // w_n^i
        // a[i]是A_1, a[i+mid]是A_2
        tmp[i] = a[i] + x*a[i+mid];
        tmp[i+mid] = a[i] - x*a[i+mid];
    }
    memcpy(a, tmp, sizeof(cp)*n);
}

int main()
{
    scanf("%d%s%s", &n, s1, s2);
    rep(i, 0, n-1) a[i]=s1[n-1-i]-'0', b[i]=s2[n-1-i]-'0';
    for(int i=1;(1<<i)<=2*n;i++) bit<<=1;
    FFT(a, bit, 1); FFT(b, bit, 1);   // 分别转点值表示
    rep(i, 0, bit-1) a[i] = a[i]*b[i];// 乘积的点值表示
    FFT(a, bit, -1);                  // IFFT将乘积表示转系数表示
    rep(i, 0, bit-1)
    {
        output[i] += a[i].real()/bit+0.5; // 除bit因为递归的IFFT里面没有除
        output[i+1] = output[i]/10;
```

```
        output[i] %= 10;
    }
    bool ok = false;
    rrep(i, n*2, 0) if(ok || output[i]) printf("%d", output[i]), ok=true;
    if(!ok) printf("0");
}
```

# FFT优化

递归常数过大，考虑优化为迭代。

## 二进制位逆序




容易发现，每个位置分治后的最终位置为其二进制逆序后得到的位置

现在求出 rev 数组，rev[x] 表示 $a_x$ 最终所在位置：

- 以 $x$ 从小到大，即 $x = 00...0 \to 11...1$，去求 rev[x]
- 在求 rev[x] 时，由于我们是从小到大求过来的，所以 rev[x>>1] 一定在之前算好了
- 把 rev[x>>1] 右移一位，并把 rev[x] 的最低位放到其最高位即可。

```
for(int i=0;i<n;i++)
{
    rev[i] = (rev[i>>1]>>1) | ((i&1)<<(bit-1));
    if(i<rev[i]) swap(a[i], a[rev[i]]); // 交换1次即可，交换2次等于没交换
}
```

## 蝴蝶操作

考虑合并两个子问题的过程：

- 假设 $A_1(w_{\frac{n}{2}}^k)$ 和 $A_2(w_{\frac{n}{2}}^k)$ 分别存储在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中 $A(w_n^k)$ 和 $A(w_n^{k+\frac{n}{2}})$ 分别存储在 $tmp[k]$ 和 $tmp[k+\frac{n}{2}]$ 中
- 合并的单位操作可表示为：
$$tmp[k] \leftarrow a[k] + w_n^k \times a[k+\frac{n}{2}]$$
$$tmp[k+\frac{n}{2}] \leftarrow a[k] - w_n^k \times a[k+\frac{n}{2}]$$
- 考虑加入一个临时变量 $t$ ，使得这个过程可在原地完成，不需要另一个数组 $tmp$ ，也即是说，将 $A(w_n^k)$ 和 $A(w_n^{k+\frac{n}{2}})$ 存储在 $a[k]$ 和 $a[k+\frac{n}{2}]$ 中，覆盖原来的值

$$t \leftarrow w_n^k \times a[k+\frac{n}{2}]$$
$$a[k+\frac{n}{2}] \leftarrow a[k] - t$$
$$a[k] \leftarrow a[k] + t$$

## 优化版代码

```cpp
// luogu-P1919
#include <stdio.h>
#include <complex>
using namespace std;
typedef complex<double> cp;
#define rep(i,s,t) for(int i=s;i<=t;i++)
#define rrep(i,s,t) for(int i=s;i>=t;i--)
const int MAXN = 1e6+5;
const int MAXS = 6e4+5;
const double PI = acos(-1.0);
int rev[MAXN];
int n, output[MAXN];
cp  a[MAXN], b[MAXN];
char s1[MAXS], s2[MAXS];

void FFT(cp *a, int n, int inv)
{
    int bit = 0;
    while((1<<bit)<n) bit++;
    rep(i, 0, n-1)
    {
        rev[i] = (rev[i>>1]>>1) | ((i&1)<<(bit-1));
        if(i<rev[i]) swap(a[i], a[rev[i]]);
    }
    for(int len=2;len<=n;len<<=1)
    {
        int mid = len>>1;
        cp unit(cos(PI/mid), inv*sin(PI/mid)); // 单位根
        for(int i=0;i<n;i+=len)
        { // mid<<1是准备合并序列的长度, i是合并到了哪一位
            cp omega(1, 0);
            for(int j=0;j<mid;j++,omega*=unit)
            { // 只扫左半部分, 得到右半部分的答案
                cp x=a[i+j], y=omega*a[i+j+mid];
                a[i+j]=x+y;
                a[i+j+mid]=x-y;
            }
        }
    }
    if(inv == -1) rep(i, 0, n) a[i] /= n;
}

int main()
{
    scanf("%d%s%s", &n, s1, s2);
    rep(i, 0, n-1) a[i]=s1[n-1-i]-'0', b[i]=s2[n-1-i]-'0';

    int bitn = 1;
    while(bitn < 2*n) bitn<<=1;

    FFT(a, bitn, 1); FFT(b,bitn, 1);
```

```
    rep(i, 0, bitn-1) a[i]=a[i]*b[i];
    FFT(a, bitn, -1); // IFFT

    rep(i, 0, bitn-1)
    {
        output[i] += a[i].real()+0.5;
        output[i+1] = output[i]/10;
        output[i] %= 10;
    }
    bool ok = false;
    rrep(i, n*2, 0) if(ok | output[i]) printf("%d", output[i]),ok=true;
    if(!ok) printf("0");
    return 0;
}
```

## 模板代码

menci的模板：

```
struct FastFourierTransform {
    std::complex<double> omega[MAXN], omegaInverse[MAXN];

    void init(const int n) {
        for (int i = 0; i < n; i++) {
            omega[i] = std::complex<double>(cos(2 * PI / n * i), sin(2 * PI / n * i));
            omegaInverse[i] = std::conj(omega[i]);
        }
    }

    void transform(std::complex<double> *a, const int n, const std::complex<double> *omega) {
        int k = 0;
        while ((1 << k) < n) k++;
        for (int i = 0; i < n; i++) {
            int t = 0;
            for (int j = 0; j < k; j++) if (i & (1 << j)) t |= (1 << (k - j - 1));
            if (i < t) std::swap(a[i], a[t]);
        }

        for (int l = 2; l <= n; l *= 2) {
            int m = l / 2;
            for (std::complex<double> *p = a; p != a + n; p += l) {
                for (int i = 0; i < m; i++) {
                    std::complex<double> t = omega[n / l * i] * p[m + i];
                    p[m + i] = p[i] - t;
                    p[i] += t;
                }
            }
        }
    }

    void dft(std::complex<double> *a, const int n) {
        transform(a, n, omega);
    }

    void idft(std::complex<double> *a, const int n) {
        transform(a, n, omegaInverse);
```

```
        for (int i = 0; i < n; i++) a[i] /= n;
    }
} fft;

inline void multiply(const int *a1, const int n1, const int *a2, const int n2, int *res) {
    int n = 1;
    while (n < n1 + n2) n *= 2;
    static std::complex<double> c1[MAXN], c2[MAXN];
    for (int i = 0; i < n1; i++) c1[i].real(a1[i]);
    for (int i = 0; i < n2; i++) c2[i].real(a2[i]);
    fft.init(n);
    fft.dft(c1, n), fft.dft(c2, n);
    for (int i = 0; i < n; i++) c1[i] *= c2[i];
    fft.idft(c1, n);
    for (int i = 0; i < n1 + n2 - 1; i++) res[i] = static_cast<int>(floor(c1[i].real() + 0.5));
}
```

# 题集

## AxB Problem Plus(HDU-1402)

两个n位数a和b，输出a*b

将 a 和 b 看成多项式：$f(x) = a_0 + a_1 \times 10^1 + a_2 \times 10^2 + \cdots + a_{n-1} \times 10^{n-1}$ 进行 FFT 即可。

```
// luogu-P1919
#include <stdio.h>
#include <string.h>
#include <complex>
using namespace std;
typedef complex<double> cp;
#define rep(i,s,t) for(int i=s;i<=t;i++)
#define rrep(i,s,t) for(int i=s;i>=t;i--)
const int MAXN = 2e5+5;
const int MAXS = 5e4+5;
const double PI = acos(-1.0);
int rev[MAXN];
int output[MAXN];
cp a[MAXN], b[MAXN];
char s1[MAXS], s2[MAXS];

void FFT(cp *a, int n, int inv)
{
    int bit = 0;
    while((1<<bit)<n) bit++;
    rep(i, 0, n-1)
    {
        rev[i] = (rev[i>>1]>>1) | ((i&1)<<(bit-1));
        if(i<rev[i]) swap(a[i], a[rev[i]]);
    }
    for(int len=2;len<=n;len<<=1)
    {
        int mid = len>>1;
        cp unit(cos(PI/mid), inv*sin(PI/mid)); // 单位根
        for(int i=0;i<n;i+=len)
        { // mid<<1是准备合并序列的长度，i是合并到了哪一位
```

```
            cp omega(1, 0);
            for(int j=0;j<mid;j++,omega*=unit)
            { // 只扫左半部分，得到右半部分的答案
                cp x=a[i+j], y=omega*a[i+j+mid];
                a[i+j]=x+y;
                a[i+j+mid]=x-y;
            }
        }
    }
    if(inv == -1) rep(i, 0, n) a[i] /= n;
}

int main()
{
    while(~scanf("%s%s", s1, s2))
    {
        int n1 = strlen(s1), n2 = strlen(s2);

        int bitn = 1;
        while(bitn < n1+n2) bitn<<=1;

        memset(a, 0, sizeof(cp)*(bitn+1));
        memset(b, 0, sizeof(cp)*(bitn+1));
        memset(output, 0, sizeof(int)*(bitn+1));

        rep(i, 0, n1-1) a[i]=s1[n1-1-i]-'0';
        rep(i, 0, n2-1) b[i]=s2[n2-1-i]-'0';

        FFT(a, bitn, 1); FFT(b, bitn, 1);
        rep(i, 0, bitn-1) a[i]=a[i]*b[i];
        FFT(a, bitn, -1); // IFFT

        rep(i, 0, bitn-1)
        {
            output[i] += a[i].real()+0.5;
            output[i+1] = output[i]/10;
            output[i] %= 10;
        }
        bool ok = false;
        rrep(i, n1+n2, 0) if(ok | output[i]) printf("%d", output[i]),ok=true;
        if(!ok) printf("0");
        printf("\n");
    }
    return 0;
}
```

## 3-idiots (HDU-4609)

T组数据，每组数据n个树枝，给出n个树枝的长度，问随机拿3条树枝能组成三角形的概率

1. 构造多项式 $f(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1} \times x^{n-1}$ ，其中 $a_ix^i$ 的系数 $a_i$ 代表了树枝长度为 $i$ 的数目
2. 利用 FFT 计算 $f(x) \times f(x)$，可得到任取 2 条树枝组合的情况（注意去掉一条边用两次的情况、(边1,边2)和(边2,边1)的顺序重复的情况），然后累前缀和便于枚举第三边时得到两边之和大于第三边的情况总数
3. 枚举每一条边 a[i] **作为最长边**，得到任取两边之和大于 a[i]的方案数目，减去三种情况
    - 任取的两条边中含有了 a[i] 的情况数目：$n-1$

- 任取的两条边中一条比 a[i] 大，一条比 a[i] 小的情况数目：$i \times (n-1-i)$
- 任取的两条边中两条都比 a[i] 大的情况数目：$(n-1-i) \times (n-1-i-1)/2$

4. 最后除以 $C_n^3$ 的总数。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i,s,t) for(int i=s;i<=t;i++)
typedef long long LL;
typedef complex<double> cp;
const int MAXN = 1e5+5;
const double PI = acos(-1.0);

int rev[MAXN<<2];
void FFT(cp *a, int n, int inv)
{
    int bit = 0;
    while((1<<bit)<n) bit++;
    rep(i, 0, n-1)
    {
        rev[i] = (rev[i>>1]>>1) | ((i&1)<<(bit-1));
        if(i<rev[i]) swap(a[i], a[rev[i]]);
    }
    for(int len=2;len<=n;len<<=1)
    {
        int mid = len>>1;
        cp unit(cos(PI/mid), inv*sin(PI/mid)); // 单位根
        for(int i=0;i<n;i+=len)
        { // mid<<1是准备合并序列的长度，i是合并到了哪一位
            cp omega(1, 0);
            for(int j=0;j<mid;j++,omega*=unit)
            { // 只扫左半部分，得到右半部分的答案
                cp x=a[i+j], y=omega*a[i+j+mid];
                a[i+j]=x+y;
                a[i+j+mid]=x-y;
            }
        }
    }
    if(inv == -1) rep(i, 0, n) a[i] /= n;
}

int T, n, cnt[MAXN], a[MAXN], len;
LL ans[MAXN<<2], Ans;
cp x[MAXN<<2];

int main()
{
    scanf("%d", &T);
    while(T--)
    {
        // input
        scanf("%d", &n);
        memset(cnt, 0, sizeof(cnt));
        rep(i, 0, n-1) scanf("%d", &a[i]), cnt[a[i]]++;
        sort(a, a+n);
        len = a[n-1]+1;

        // solve
        int bitn=1;
```

```
        while(bitn < 2*len) bitn<<=1;

        rep(i, 0, len-1) x[i] = cp(cnt[i], 0);
        rep(i, len, bitn-1) x[i] = cp(0, 0);
        FFT(x, bitn, 1);
        rep(i, 0, bitn-1) x[i] = x[i]*x[i];
        FFT(x, bitn, -1);

        rep(i, 0, bitn-1) ans[i] = 1LL*(x[i].real()+0.5);
        bitn = 2*a[n-1];
        rep(i, 0, n-1) ans[2*a[i]]--;       // 减去取相同边的情况
        rep(i, 0, bitn) ans[i] >>= 1;       // 减去(边1,边2)与(边2,边1)顺序重复
        rep(i, 1, bitn) ans[i] += ans[i-1]; // 前缀和, 便于枚举两边之和大于第3边
        Ans = 0;
        rep(i, 0, n-1)
        { // 枚举长度为a[i]的边
            Ans += ans[bitn] - ans[a[i]];   // 另外两边之和大于a[i]的方案数
            Ans -= 1LL*i*(n-1-i);           // 减去一大一小情况
            Ans -= 1LL*(n-1-i)*(n-2-i)/2;   // 减去两大情况
            Ans -= 1LL*(n-1) ;              // 减去含自身的不合法情况
        }
        // output
        LL all = 1LL*n*(n-1)*(n-2)/6; // C_n^3的情况总数
        printf("%.7f\n", 1.0*Ans/all);
    }
    return 0;
}
```

## 力 (BZOJ-3527)

中文题面。给出n个数qi，给出Fj定义如下，令Ei=Fi/qi，求Ei

$E_i = \sum_{j<i} \frac{q_j}{(i-j)^2} - \sum_{j>i} \frac{q_j}{(i-j)^2}$ 可化为：

- $E_i = \sum\limits_{j=1}^{i-1} \frac{q_j}{(i-j)^2} - \sum\limits_{j=i+1}^{n} \frac{q_j}{(i-j)^2}$

转换下标从 0 开始：

- $E_i = \sum\limits_{j=0}^{i-1} \frac{q_j}{(i-j)^2} - \sum\limits_{j=i+1}^{n-1} \frac{q_j}{(i-j)^2}$

记 $f(x) = q_x$、$g(x) = \frac{1}{x^2}$，其中 $g(0) = 0$ 则 $E_i$ 化为：

- $E_i = \sum\limits_{j=0}^{i-1} f(j) \times g(j-i) - \sum\limits_{j=i+1}^{n-1} f(j) \times g(j-i)$

$E_i$ 的第一个式子可化为卷积的形式：

- $\sum\limits_{j=0}^{i-1} f(j) \times g(j-i) = \sum\limits_{j=0}^{i} f(j) \times g(j-i)$
- 又由于 $g(x) = \frac{1}{x^2}$，对 $\sum\limits_{j=0}^{i} f(j) \times g(i-j)$ 用 FFT 算卷积

$E_i$ 的第二个式子：

- $\sum\limits_{j=i+1}^{n-1} f(j) \times g(j-i) = \sum\limits_{j=i}^{n-1} f(j) \times g(j-i) = \sum\limits_{j=0}^{n-1-i} f(j+i) \times g(j)$

令 $f^T(x) = q_{n-1-x}$，即将原 $q_i$ 序列倒序，则上式记为：

- $X_i = \sum_{j=0}^{n-1-i} f^T(n-1-(j+i)) \times g(j)$
- 再化为 $X_{n-1-i} = \sum_{j=0}^{i} f^T(i-j) \times g(j)$
- 对 $\sum_{j=0}^{i} f^T(i-j) \times g(j)$ 用 FFT 算卷积

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef complex<double> cp;
#define rep(i,s,t) for(int i=s;i<=t;i++)
#define rrep(i,s,t) for(int i=s;i>=t;i--)
const int MAXN = 3e5+5;
const double PI = acos(-1.0);
int rev[MAXN];

void FFT(cp *a, int n, int inv)
{
    int bit = 0;
    while((1<<bit)<n) bit++;
    rep(i, 0, n-1)
    {
        rev[i] = (rev[i>>1]>>1) | ((i&1)<<(bit-1));
        if(i<rev[i]) swap(a[i], a[rev[i]]);
    }
    for(int len=2;len<=n;len<<=1)
    {
        int mid = len>>1;
        cp unit(cos(PI/mid), inv*sin(PI/mid)); // 单位根
        for(int i=0;i<n;i+=len)
        { // mid<<1是准备合并序列的长度, i是合并到了哪一位
            cp omega(1, 0);
            for(int j=0;j<mid;j++,omega*=unit)
            { // 只扫左半部分, 得到右半部分的答案
                cp x=a[i+j], y=omega*a[i+j+mid];
                a[i+j]=x+y;
                a[i+j+mid]=x-y;
            }
        }
    }
    if(inv == -1) rep(i, 0, n) a[i] /= n;
}

int n;
double a[MAXN], b[MAXN], c[MAXN];
cp x1[MAXN], x2[MAXN];

int main()
{
    scanf("%d", &n);
    rep(i, 0, n-1) scanf("%lf", &a[i]);
    rep(i, 1, n-1) b[i] = 1.0/i/i;
    int bitn=1; while(bitn < n*2) bitn<<=1;
    // first
    rep(i, 0, n-1) x1[i] = cp(a[i], 0), x2[i] = cp(b[i], 0);
    rep(i, n, bitn-1) x1[i] = x2[i] = cp(0, 0);
    FFT(x1, bitn, 1);  FFT(x2, bitn, 1);
```

```
    rep(i, 0, bitn-1) x1[i] = x1[i]*x2[i];
    FFT(x1, bitn, -1);
    rep(i, 0, n-1) c[i] = x1[i].real();
    // second
    rep(i, 0, n-1) x1[i] = cp(a[n-1-i], 0), x2[i] = cp(b[i], 0);
    rep(i, n, bitn-1) x1[i] = x2[i] = cp(0, 0);
    FFT(x1, bitn, 1); FFT(x2, bitn, 1);
    rep(i, 0, bitn-1) x1[i] = x1[i]*x2[i];
    FFT(x1, bitn, -1);
    rep(i, 0, n-1) c[i] -= x1[n-1-i].real();
    // output
    rep(i, 0, n-1) printf("%.3f\n", c[i]);
    return 0;
}
```

## Rock Paper Scissors (Gym-101667H)

给出n、m(m<n)，接下来给出一个长度为n的石头(R)剪刀(S)布(P)串，和一个长度为m的石头剪刀布串，串2移动到串1的哪个位置使得胜利最多

1. 首先将串 2 的字符转换为需要赢该手势所需字符，随后，问题即转化为串2移动到串1哪个位置匹配度最高的问题
2. 对石头、剪刀、布分别处理，每种情况中将当前串转换为 01 串，1 表示该位置是石头(剪刀/布)
3. 构造多项式 $f(x)=a_0+a_1x^1+a_2x^2+\cdots+a_{n-1}x^{n-1}$ ，其中 $a_ix^i$ 的系数 $a_i$ 代表了串1的 $i$ 位置是否是石头(剪刀/布)
4. 构造多项式 $g(x)=a_{n-1}x^{n-1}+\cdots+a_2x^2+a_1x^1+a_0$ ，其中 $a_ix^i$ 的系数 $a_i$ 代表了串2的 $n-1-i$ 位置是否是石头(剪刀/布)
5. FFT 算乘积 $f(x)\times g(x)$，得到的 $x^i$ 系数即计数情况。
    - 例如，01串 "11100" 和 "0111" 进行匹配，即 $(1+x+x^2)\times(x^2+x+1)$ ，得到的结果取从 $x^2$ 到 $x^4$ 的系数（注意题意中串2可以右移到移出串1，但不可左移）

```
#include <bits/stdc++.h>
using namespace std;
typedef complex<double> cp;
#define rep(i,s,t) for(int i=s;i<=t;i++)
#define rrep(i,s,t) for(int i=s;i>=t;i--)
const int MAXN = 4e5+5;
const double PI = acos(-1.0);
int rev[MAXN];
void FFT(cp *a, int n, int inv)
{
    int bit = 0;
    while((1<<bit)<n) bit++;
    rep(i, 0, n-1)
    {
        rev[i] = (rev[i>>1]>>1) | ((i&1)<<(bit-1));
        if(i<rev[i]) swap(a[i], a[rev[i]]);
    }
    for(int len=2;len<=n;len<<=1)
    {
        int mid = len>>1;
        cp unit(cos(PI/mid), inv*sin(PI/mid)); // 单位根
        for(int i=0;i<n;i+=len)
        { // mid<<1是准备合并序列的长度，i是合并到了哪一位
            cp omega(1, 0);
```

```
            for(int j=0;j<mid;j++,omega*=unit)
            { // 只扫左半部分，得到右半部分的答案
                cp x=a[i+j], y=omega*a[i+j+mid];
                a[i+j]=x+y;
                a[i+j+mid]=x-y;
            }
        }
    }
    if(inv == -1) rep(i, 0, n) a[i] /= n;
}

const map<char, char> MP({{'R', 'S'}, {'P', 'R'}, {'S', 'P'}});
const char HAND[] = "RPS";
char s1[MAXN], s2[MAXN];
int n, m, ans[MAXN];
cp a[MAXN], b[MAXN];
int main()
{
    scanf("%d%d%s%s", &n, &m, s1, s2);
    rep(i, 0, m-1) s2[i] = MP.at(s2[i]);
    int bitn = 1; while(bitn < n+m) bitn<<=1;
    rep(t, 0, 2)
    {
        rep(i, 0, n-1) a[i] = cp(s1[i]==HAND[t], 0);
        rep(i, n, bitn-1) a[i] = cp(0, 0);
        rep(i, 0, m-1) b[i] = cp(s2[m-1-i]==HAND[t], 0);
        rep(i, m, bitn-1) b[i] = cp(0, 0);
        FFT(a, bitn, 1);  FFT(b, bitn, 1);
        rep(i, 0, bitn-1) a[i] = a[i]*b[i];
        FFT(a, bitn, -1);
        rep(i, m-1, n+m-2) ans[i] += a[i].real()+0.5;
    }
    int Ans = 0;
    rep(i, m-1, n+m-2) if(Ans < ans[i]) Ans=ans[i];
    printf("%d", Ans);
    return 0;
}
```

### K-neighbor substrings (UVALive-4671)

输入n和两个ab串，串2小于串1，将串1的全部子串拿出来，去掉重复，求与串2长度相等并且对应位相异数目小于等于n的子串的数目

# 快速沃尔什变换 FWT

## FWT前言

## 作用

多项式卷积 $C_k = \sum\limits_{i+j=k} A_i \times B_j$ ，这样的可以用 FFT 做

甚至在一些特殊情况下， $C_k = \sum\limits_{i\times j=k} A_i \times B_j$ 也可做

但是对于以下：

- $C_k = \sum\limits_{i\vert j=k} A_i \times B_j$
- $C_k = \sum\limits_{i\&j=k} A_i \times B_j$
- $C_k = \sum\limits_{i\wedge j=k} A_i \times B_j$

就要用到 FWT 进行解决——多项式的位运算卷积

## 思路

FFT 的思路：求出多项式在若干个单位根的点值表示，乘起来，再复原为系数表示。

FWT 类似，但是与 FFT 最大区别在于，没有基于类似于单位根的优化方式，所有优化都由位运算符不同而不同。

## 符号约定

- $A, B$ —— 多项式，长度为 $n$，次数为 $n-1$
- $A_0, A_1$ —— 多项式 $A$ 的前 $\frac{n}{2}$ 项、后 $\frac{n}{2}$ 项
- $A + B$ —— 将多项式 $A, B$ 对应位加(减/乘) ，系数表达式为 $(A_0 + B_0, A_1 + B_1, \cdots, A_{n-1} + B_{n-1})$
- $A \oplus B$ —— 将多项式 $A, B$ 异或(与/或)卷积，系数表达式为 $(\sum\limits_{i\oplus j=0} (A_i \times B_j), \sum\limits_{i\oplus j=1} (A_i \times B_j), \cdots, \sum\limits_{i\oplus j=n-1} (A_i \times B_j))$
- $\mathrm{FWT}(A)$ —— 多项式 $A$ 的 FWT 变换
- $(A, B)$ —— 将多项式 $A, B$ 拼接起来

# 或(or)卷积

## 定义

定义或卷积的 FWT：

- $\mathrm{FWT}(A) = \begin{cases} (\mathrm{FWT}(A_0), \mathrm{FWT}(A_0 + A_1)) & n > 1 \\ A & n = 1 \end{cases}$

## 证明

- 两个多项式相加后的 FWT 变换等于分别 FWT 的和： $\mathrm{FWT}(A + B) = \mathrm{FWT}(A) + \mathrm{FWT}(B)$
  - 当 $n = 1$ 时，明显 $\mathrm{FWT}(A + B) = A + B = \mathrm{FWT}(A) + \mathrm{FWT}(B)$
  - 当 $n = 2$ 时，
    $$\begin{aligned} \mathrm{FWT}(A + B) &= (\mathrm{FWT}(A_0 + B_0), \mathrm{FWT}(A_0 + A_1 + B_0 + B_1)) \\ &= (\mathrm{FWT}(A_0) + \mathrm{FWT}(B_0), \mathrm{FWT}(A_0) + \mathrm{FWT}(A_1) + \mathrm{FWT}(B_0) + \mathrm{FWT}(B_1)) \\ &= (\mathrm{FWT}(A_0) + \mathrm{FWT}(A_1) + \mathrm{FWT}(A_2)) + (\mathrm{FWT}(B_0) + \mathrm{FWT}(B_1) + \mathrm{FWT}(B_2)) \\ &= \mathrm{FWT}(A) + \mathrm{FWT}(B) \end{aligned}$$
  - 当 $n = 4$、$8$、$16$、$\cdots$ 时，可同理类推。
- 两个多项式或卷积的 FWT 变换等于分别 FWT 的积： $\mathrm{FWT}(A\vert B) = \mathrm{FWT}(A) \times \mathrm{FWT}(B)$
  - 当 $n = 1$ 时，$\mathrm{FWT}(A\vert B) = A \times B = \mathrm{FWT}(A) \times \mathrm{FWT}(B)$

- 当 $n=2$ 时，

$$\begin{aligned}\mathrm{FWT}(A|B)&=\mathrm{FWT}((A|B)_0,(A|B)_1)\\&=\mathrm{FWT}(A_0|B_0,A_0|B_1+A_1|B_0+A_1|B_1)\\&=(\mathrm{FWT}(A_0|B_0),\mathrm{FWT}(A_0|B_0+A_0|B_1+A_1|B_0+A_1|B_1))\\&=(\mathrm{FWT}(A_0)\times\mathrm{FWT}(B_0),\mathrm{FWT}(A_0)\times\mathrm{FWT}(B_0)+\mathrm{FWT}(A_0)\times\mathrm{FWT}(B_1)\\&\qquad+\mathrm{FWT}(A_1)\times\mathrm{FWT}(B_0)+\mathrm{FWT}(A_1)\times\mathrm{FWT}(B_1))\\&=(\mathrm{FWT}(A_0)\times\mathrm{FWT}(B_0),(\mathrm{FWT}(A_0)+\mathrm{FWT}(A_1))\times(\mathrm{FWT}(B_0)+\mathrm{FWT}(B_1)))\\&=(\mathrm{FWT}(A_0)\times\mathrm{FWT}(B_0),\mathrm{FWT}(A_0+A_1)\times\mathrm{FWT}(B_0+B_1))\\&=(\mathrm{FWT}(A_0),\mathrm{FWT}(A_0+A_1))\times(\mathrm{FWT}(B_0),\mathrm{FWT}(B_0+B_1))\\&=\mathrm{FWT}(A)\times\mathrm{FWT}(B)\end{aligned}$$

- 当 $n=4$、$8$、$16$、$\cdots$ 时，可同理类推。

# 和(and)卷积

## 定义

定义和卷积的 FWT：

- $\mathrm{FWT}(A)=\begin{cases}(\mathrm{FWT}(A_0+A_1),FWT(A_1)) & n>0\\A & n=0\end{cases}$

# 异或(xor)卷积

## 定义

定义异或卷积的 FWT：

- $\mathrm{FWT}(A)=\begin{cases}(\mathrm{FWT}(A_0+A_1),FWT(A_0-A_1)) & n>0\\A & n=0\end{cases}$

# 同或(xnor)卷积

## 定义

定义同或卷积的 FWT：

- $\mathrm{FWT}(A)=\begin{cases}(\mathrm{FWT}(A_1-A_0),FWT(A_0+A_1)) & n>0\\A & n=0\end{cases}$

## FWT模板

无 IFWT

```
int add(int x,int y) { return (x+=y)>=MOD ? x-MOD : x; }
int sub(int x,int y) { return (x-=y)<0 ? x+MOD : x; }
void FWT(int a[],int n)
{ // n是2^k形式
    for(int d=1; d<n; d<<=1)
        for(int m=d<<1,i=0; i<n; i+=m)
            for(int j=0; j<d; j++)
            {
                int x=a[i+j], y=a[i+j+d];
                要用哪个激活哪个
                //   or: a[i+j+d]=add(y, x);                  // or
                //  and: a[i+j]=add(x, y);                    // and
                //  xor: a[i+j]=add(x,y), a[i+j+d]=sub(x,y);   // xor
```

```
                // xnor: a[i+j]=sub(y, x), a[i+j+d]=add(x, y); // xnor
            }
}
```

# 题集

## 快速沃尔什变换模板 (luogu-P4717)

板子题

就是求开篇说的：

- $C_k = \sum\limits_{i|j=k} A_i \times B_j$
- $C_k = \sum\limits_{i\&j=k} A_i \times B_j$
- $C_k = \sum\limits_{i\wedge j=k} A_i \times B_j$

基于数组的模板：

```cpp
#include <stdio.h>
#include <algorithm>
#include <vector>
using namespace std;

const int MOD = 998244353;
const int MAXN = 1<<18;

struct FWT
{
    static const int OR = 1, AND = 2, XOR = 3, XNOR = 4;
    int add(int x,int y) { return (x+=y)>=MOD ? x-MOD : x; }
    int sub(int x,int y) { return (x-=y)<0 ? x+MOD : x; }
    int extend(int n)
    {
        int N=1; while(N < n) N<<=1;
        return N;
    }
    void Transform(int op, int a[], int n, int rev)
    { // n是2^k形式
        int inv2 = (MOD+1)>>1;
        for(int d=1; d<n; d<<=1)
            for(int m=d<<1,i=0; i<n; i+=m)
                for(int j=0; j<d; j++)
                {
                    int x=a[i+j],y=a[i+j+d];
                    if (rev == 1)
                    {
                        if(op == OR) a[i+j+d]=add(y, x);
                        else if(op == AND) a[i+j]=add(x, y);
                        else if(op == XOR) a[i+j]=add(x,y), a[i+j+d]=sub(x,y);
                        else if(op == XNOR)a[i+j]=sub(y,x), a[i+j+d]=add(x,y);
                    }else
                    {
                        if(op == OR) a[i+j+d]=sub(y, x);
                        else if(op == AND) a[i+j]=sub(x, y);
```

```
                        else if(op == XOR) a[i+j]=1LL*add(x,y)*inv2%MOD,
a[i+j+d]=1LL*sub(x,y)*inv2%MOD;
                        else if(op == XNOR)a[i+j]=1LL*sub(y,x)*inv2%MOD,
a[i+j+d]=1LL*add(x,y)*inv2%MOD;
                    }
                }
    }
    void Mul(int op, int a1[], int n1, int a2[], int n2, int ans[])
    {
        int n = max(n1, n2), N = extend(n);
        static int tmp[MAXN];
        for(int i=0;i<N;i++) ans[i]=a1[i], tmp[i]=a2[i];
        Transform(op, ans, N, 1);
        Transform(op, tmp, N, 1);
        for(int i=0;i<N;i++) ans[i] = 1LL*ans[i]*tmp[i]%MOD;
        Transform(op, ans, N, -1);
    }
} fwt;

int a[MAXN], b[MAXN], c[MAXN];;
int main()
{
    int n; scanf("%d", &n); n=1<<n;
    for(int i=0;i<n;i++) scanf("%d", &a[i]);
    for(int i=0;i<n;i++) scanf("%d", &b[i]);
    fwt.Mul(fwt.OR, a, n, b, n, c);
    for(int i=0;i<n;i++) printf(i==n-1?"%d\n":"%d ", c[i]);
    fwt.Mul(fwt.AND, a, n, b, n, c);
    for(int i=0;i<n;i++) printf(i==n-1?"%d\n":"%d ", c[i]);
    fwt.Mul(fwt.XOR, a, n, b, n, c);
    for(int i=0;i<n;i++) printf(i==n-1?"%d\n":"%d ", c[i]);
    return 0;
}
```

基于vector的板子：

```
#include <stdio.h>
#include <algorithm>
#include <vector>
using namespace std;

const int MOD = 998244353;
void add(int &x, int y) { (x+=y)>=MOD && (x-=MOD); }
void sub(int &x, int y) { (x-=y)<0 && (x+=MOD); }
struct FWT
{
    int extend(int n)
    {
        int N=1; while(N < n) N<<=1;
        return N;
    }
    void FWTor(vector<int> &a, bool rev)
    {
        int n = a.size();
        for(int l=2, m=1; l<=n; l<<=1, m<<=1)
            for(int j=0; j<n; j+=l)
                for(int i=0;i<m;i++)
                    if(!rev) add(a[i+j+m], a[i+j]);
```

```
                    else sub(a[i+j+m], a[i+j]);
    }
    void FWTand(vector<int> &a, bool rev)
    {
        int n = a.size();
        for(int l=2, m=1; l<=n; l<<=1, m<<=1)
            for(int j=0; j<n; j+=l)
                for(int i=0;i<m;i++)
                    if(!rev) add(a[i+j], a[i+j+m]);
                    else sub(a[i+j], a[i+j+m]);
    }
    void FWTxor(vector<int> &a, bool rev)
    {
        int n = a.size(), inv2 = (MOD+1)>>1;
        for(int l=2, m=1; l<=n; l<<=1, m<<=1)
            for(int j=0; j<n; j+=l)
                for(int i=0;i<m;i++)
                {
                    int x=a[i+j], y=a[i+j+m];
                    if(!rev) a[i+j]=(x+y)%MOD, a[i+j+m]=(x-y+MOD)%MOD;
                    else a[i+j]=1LL*(x+y)*inv2%MOD, a[i+j+m]=1LL*(x-y+MOD)*inv2%MOD;
                }
    }
    vector<int> Or(vector<int> a1, vector<int> a2)
    {
        int n = max(a1.size(), a2.size()), N = extend(n);
        a1.resize(N); FWTor(a1, false);
        a2.resize(N); FWTor(a2, false);
        vector<int> A(N);
        for(int i=0;i<N;i++) A[i] = 1LL*a1[i]*a2[i]%MOD;
        FWTor(A, true);
        return A;
    }
    vector<int> And(vector<int> a1, vector<int> a2)
    {
        int n = max(a1.size(), a2.size()), N = extend(n);
        a1.resize(N); FWTand(a1, false);
        a2.resize(N); FWTand(a2, false);
        vector<int> A(N);
        for(int i=0;i<N;i++) A[i] = 1LL*a1[i]*a2[i]%MOD;
        FWTand(A, true);
        return A;
    }
    vector<int> Xor(vector<int> a1, vector<int> a2)
    {
        int n = max(a1.size(), a2.size()), N = extend(n);
        a1.resize(N); FWTxor(a1, false);
        a2.resize(N); FWTxor(a2, false);
        vector<int> A(N);
        for(int i=0;i<N;i++) A[i] = 1LL*a1[i]*a2[i]%MOD;
        FWTxor(A, true);
        return A;
    }
} fwt;

int main()
{
    int n; scanf("%d", &n); n=1<<n;
    vector<int> a1(n), a2(n);
```

```
    for(int i=0;i<n;i++) scanf("%d", &a1[i]);
    for(int i=0;i<n;i++) scanf("%d", &a2[i]);
    vector<int> A;
    A = fwt.Or(a1, a2);
    for(int i=0;i<n;i++) printf(i==n-1?"%d\n":"%d ", A[i]);
    A = fwt.And(a1, a2);
    for(int i=0;i<n;i++) printf(i==n-1?"%d\n":"%d ", A[i]);
    A = fwt.Xor(a1, a2);
    for(int i=0;i<n;i++) printf(i==n-1?"%d\n":"%d ", A[i]);
    return 0;
}
```

## Hard Nim (HYSBZ-4589)

多组数据。有n堆石子，每堆的数量是不超过m的质数，小C和小N两个人轮流从某一堆石子取若干（可取空，但不可不取），取走最后一个石子的人赢，问小N能赢的局面有多少种

引入 Nim 博弈的结论：异或和为 0，先手必输。

定义多项式：$f(x)=a_0+a_1x+a_2x^2+\cdots+a_{m-1}x^{m-1}$ ，其中 $a_ix^i$ 的系数 $a_i$ 表示 $i$ 是否为质数。

那么：$C_k^2=\sum\limits_{i\oplus j=k}a_i\times a_j$ 的意义是任取两个质数堆石子，堆数异或和为 k 的方案数

再乘一次：$C_k^3=\sum\limits_{i\oplus j=k}C_i^2\times a_j$ 的意义是任取三个质数堆石子，堆数异或和为 k 的方案数

再乘一次：$C_k^4=\sum\limits_{i\oplus j=k}C_i^3\times a_j$ 的意义是任取四个质数堆石子，堆数异或和为 k 的方案数

再乘一次：$C_k^5=\sum\limits_{i\oplus j=k}C_i^4\times a_j$ 的意义是任取五个质数堆石子，堆数异或和为 k 的方案数

以此类推，先得到 $\mathrm{FWT}(f(x))$ ，然后快速幂，然后 $\mathrm{IFWT}$ 后即可。

注意，不超过m的质数，即 $0,1,\cdots,m$，是一个 m+1 次的多项式，可以先 m++ 再处理。

```
#include <stdio.h>
#include <vector>
using namespace std;
typedef long long LL;

const LL MOD = 1e9+7;
void add(LL &x, LL y) { (x+=y)>=MOD && (x-=MOD); }
void sub(LL &x, LL y) { (x-=y)<0 && (x+=MOD); }
struct FWT
{
    int extend(int n)
    {
        int N=1; while(N < n) N<<=1;
        return N;
    }
    void FWTxor(vector<LL> &a, bool rev)
    {
        int n = a.size();
        LL inv2 = (MOD+1)>>1;
        for(int l=2, m=1; l<=n; l<<=1, m<<=1)
            for(int j=0; j<n; j+=l)
                for(int i=0;i<m;i++)
                {
                    LL x=a[i+j], y=a[i+j+m];
```

```
                    if(!rev) a[i+j]=(x+y)%MOD, a[i+j+m]=(x-y+MOD)%MOD;
                    else a[i+j]=1LL*(x+y)*inv2%MOD, a[i+j+m]=1LL*(x-y+MOD)*inv2%MOD;
                }
    }
} fwt;

const int MAXM = 3e5+5;
bool vis[MAXM];
void init()
{
    vis[0] = vis[1] = true;
    vis[2] = false;
    for(int i=2;i<MAXM;i++)
    {
        if(vis[i]) continue;
        for(int j=i+i;j<MAXM;j+=i) vis[j]=true;
    }
}

int main()
{
    init();
    int n, m, M;
    vector<LL> A(MAXM);
    while(~scanf("%d%d", &n, &m))
    {
        m++;
        M = fwt.extend(m);
        A.resize(M);
        for(int i=0;i<m;i++) A[i]=!vis[i];
        for(int i=m;i<M;i++) A[i]=0;
        fwt.FWTxor(A, false);
        for(int i=0;i<M;i++)
        {
            LL res=1;
            int _n = n;
            for(;_n;_n>>=1,A[i]=A[i]*A[i]%MOD) if(_n&1) res=res*A[i]%MOD;
            A[i]=res;
        }
        fwt.FWTxor(A, true);
        printf("%lld\n", A[0]);
    }
    return 0;
}
```

## Binary Table (CodeForces-662C)

给n*m的01阵，每次操作可以使得一行或一列求反，求一系列操作后使得数字1最少，输出数字1的数目

注意到 n<=20, m<=1e5，可以将矩阵压缩成一行，每个位置即一个 20 位的数 $a[j]$

设 $f(x)$ 为某一列状态为 $x$ 时，该列翻转或不翻转的最少 1 的个数，例如 $f(a[j])$ 就是考虑第 j 列翻转与否的最少 1 个数，即 $f(x) = min(\_\_builtin\_popcount(x), n - \_\_builtin\_popcount(x))$ ，其中 $\_\_builtin\_popcount(x)$ 为 gcc 计算二进制中 1 的个数的内建函数。

二进制异或操作中，与 1 异或即求反，可以设一个 20 位数 $k$ 枚举对行的翻转，当 $k$ 的对应位为 1 时即翻转第 i 行。

在确定行翻转情况后，答案为 $ans(k)=\sum_{j=1}^{m} f(a[j]\oplus k)$

因为 $a[j]\oplus k$ 相同的列对答案的贡献相同，上式可写为 $ans(k)=\sum f(a[j]\oplus k)\times num(a[j])$

令 $i=a[j]\oplus k$、$j=a[j]$，有$i\oplus=j=k$ ，原式化为 $ans(k)=\sum_{i\oplus j=k} f(i)\oplus num(j)$

```
#include <stdio.h>
#include <vector>
using namespace std;
typedef long long LL;

const LL MOD = 1e9+7;
void add(int &x, int y) { (x+=y)>=MOD && (x-=MOD); }
void sub(int &x, int y) { (x-=y)<0 && (x+=MOD); }
struct FWT
{
    int extend(int n)
    {
        int N=1; while(N < n) N<<=1;
        return N;
    }
    void FWTxor(vector<int> &a, bool rev)
    {
        int n = a.size(), inv2 = (MOD+1)>>1;
        for(int l=2, m=1; l<=n; l<<=1, m<<=1)
            for(int j=0; j<n; j+=l)
                for(int i=0;i<m;i++)
                {
                    int x=a[i+j], y=a[i+j+m];
                    if(!rev) a[i+j]=(x+y)%MOD, a[i+j+m]=(x-y+MOD)%MOD;
                    else a[i+j]=1LL*(x+y)*inv2%MOD, a[i+j+m]=1LL*(x-y+MOD)*inv2%MOD;
                }
    }
    vector<int> Xor(vector<int> a1, vector<int> a2)
    {
        int n = max(a1.size(), a2.size()), N = extend(n);
        a1.resize(N); FWTxor(a1, false);
        a2.resize(N); FWTxor(a2, false);
        vector<int> A(N);
        for(int i=0;i<N;i++) A[i] = 1LL*a1[i]*a2[i]%MOD;
        FWTxor(A, true);
        return A;
    }
} fwt;

const int MAXM = 1e5+5;
const int MAXS = 2e6+5;
int a[MAXM];
char s[MAXM];
vector<int> f, num, ans;

int main()
{
    int n, m, x;
    scanf("%d%d", &n, &m);
    for(int i=0;i<n;i++)
    {
```

```
        scanf("%s", s);
        for(int j=0;j<m;j++) a[j] |= ((s[j]-'0')<<i);
    }
    f.resize(1<<n); num.resize(1<<n);
    for(int i=0;i<1<<n;i++) f[i]=min(__builtin_popcount(i), n-
__builtin_popcount(i));
    for(int j=0;j<m;j++) num[a[j]]++;
    ans = fwt.Xor(f, num);
    int Ans = 1<<30;
    for(int i=0;i<1<<n;i++) Ans=min(Ans, ans[i]);
    printf("%d", Ans);
    return 0;
}
```

## Parity of Tuples (2019牛客第1场-D)

题意如下

输入 k，输入 n 个 m 元组 $v_1, v_2, \cdots, v_n$，其中 $v_i = (a_{i,1}, a_{i,2}, \cdots, a_{i,m})$。定义 $count(x)$ 为符合以下条件的 m 元组的个数：元组所有元素 $a_{i,1}, a_{i,2}, \cdots, a_{i,m}$ 与 $x$ 进行与运算结果的二进制表示都有奇数个 1。输出 $\oplus_{x=0}^{2^k-1}\left(count(x)\cdot 3^x \bmod (10^9+7)\right)$

官方题解，$O(n\cdot 2^m + k\cdot 2^k)$ ：

- 对于每个 $m$ 元组，它的某个子集 $S$ 异或和 $\oplus_{i\in S} a_i$ ，把所有子集 $F(\oplus_{i\in S} a_i) += (-1)^{|S|}$
- 对 $F(x)$ 进行 FWT，$\frac{FWT(x)}{2^m}$ 即是 $count(x)$

正确性证明（by sdcgvhgj）：

- 考虑元组 $(a_1, a_2, \ldots, a_m)$ 的子集 S 对 FWT(x) 贡献
- FWT 变换的定义： $FWT(x) = \sum_{i=0}^{n}(-1)^{|i \bigcap x|}F(i)$
- 所以贡献为

$$
\begin{aligned}
&(-1)^{|(\bigoplus\limits_{i\in S} a_i)\bigcap x|} \times (-1)^{|S|} \\
=&(-1)^{|\bigoplus\limits_{i\in S} a_i \bigcap x|} \times (-1)^{|S|}, \quad \bigcap \text{对} \bigoplus \text{有分配率} \\
=&(-1)^{\sum\limits_{i\in S}|a_i \bigcap x|} \times (-1)^{|S|}, \quad x\text{和}y\text{中}1\text{的个数和的奇偶性和}x\oplus y\text{中}1\text{的个数的奇偶性是一样的} \\
=&(-1)^{\sum\limits_{i\in S}1+|a_i \bigcap x|} \\
=&\prod_{i\in S}(-1)^{1+|a_i \bigcap x|}
\end{aligned}
$$

- 所有子集 S 对 FWT(x) 的贡献之和为 $\sum\limits_{S}\prod\limits_{i\in S}(-1)^{1+|a_i\cap x|} = \prod\limits_{i}(1+(-1)^{1+|a_i\cap x|})$
- 若 $|a_i \cap x|$ 全为奇数则值为 $2^m$ ，否则为 0

照着官方题解写的代码：

- `rep(s, 1, (1<<m)-1) s_xor[s] = s_xor[s^(s&-s)] ^ a[__builtin_ctz(s&-s)];`是求所有子集异或和
  - m 个元素的集合的子集数目为 $2^m$，定义数组 $\mathrm{s\_xor}[i]$ 为编号 $i$ 的子集的元素异或和
  - 赋予编号意义：编号为 i 的子集，i 的二进制各个位 j 代表了子集 i 包不包含 a[j]；编号为 0 的是空集，$\mathrm{s\_xor}[0] = 0$ 即空集的元素异或和为 0
  - `x & -x` ：若 x 是奇数， 结果必为 1；若 x 是偶数， 结果是能整除这个偶数的最大的 $2^k$。 相当于只取 x 的二进制最右边的 1
  - `x^(x&-x)`：相当于把 x 的二进制最右边的 1 置为0
  - `__builtin_ctz(s&-s)`：其中内建函数是求某个数二进制的末导零个数，`a[__builtin_ctz(s&-s)]` 可以看作 `a[j]` ，j 是 s 的末导零个数。那么 `s_xor[s二进制第j位置`

`0] ^ a[j]` 即是由前面结果s_xor推得当前结果的s_xor 。

- `rep(s, 0, (1<<m)-1) if(__builtin_parity(s)) f[s_xor[s]]--; else f[s_xor[s]]++;` 是将 $F(\oplus_{i\in S}a_i)+=(-1)^{|S|}$
  - |S| 为 奇数时，子集 S 的异或和对应的 f -= 1 ； |S| 为 偶数时， f += 1。
    `__builtin_parity(s)` 是求 s 的二进制中 1 的个数是否为奇数，即编号为 s 的子集 S 的大小是否为奇数。
- 上述两个过程用递归写也行。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
#define rep(i,s,t) for(int i=s;i<=t;i++)
const int maxn = (1<<20)+25;
const int MOD =  1e9+7;
int n,m,k,a[25],f[maxn];
LL s_xor[2020];

int add(int x,int y) { return (x+=y)>=MOD? x-MOD:x; }
int sub(int x,int y) { return (x-=y)<0 ? x+MOD:x; }
void FWTxor(int a[],int n)
{
    for(int d=1; d<n; d<<=1)
        for(int m=d<<1,i=0; i<n; i+=m)
            for(int j=0; j<d; j++)
            {
                int x=a[i+j],y=a[i+j+d];
                a[i+j]=add(x,y),a[i+j+d]=sub(x,y);
            }
}

LL qpow(LL x, LL n)
{
    LL res=1;
    for(;n;n>>=1,x=x*x%MOD) if(n&1) res=res*x%MOD;
    return res;
}

int main()
{
    while(~scanf("%d%d%d", &n, &m, &k))
    {
        memset(f, 0, sizeof(int)*((1<<k)+1));
        rep(i, 0, (1<<k)-1) f[i] = 0;
        rep(i, 1, n)
        {
            rep(j, 0, m-1) scanf("%d", &a[j]);
            rep(s, 1, (1<<m)-1) s_xor[s] = s_xor[s^(s&-s)] ^ a[__builtin_ctz(s&-s)];
            rep(s, 0, (1<<m)-1) if(__builtin_parity(s)) f[s_xor[s]]--; else
f[s_xor[s]]++;
        }
        FWTxor(f, 1<<k);
        LL y=1, ans=0, inv2=qpow(1<<m, MOD-2); // 由于算FWRxor带模，所以除2^m时要用逆
元
        rep(x, 0, (1<<k)-1) ans ^= add(f[x]*y%MOD*inv2%MOD, MOD), y=y*3%MOD;
        printf("%lld\n", ans);
    }
    return 0;
```

```
48 }
```

## Mysterious LCM (Gym 102141B)

给一组数，和一个数<=1e18，问这组数中最小集合使得集合lcm等于这个数，输出集合大小

首先将数组模一遍筛选一遍。

设 $X=\prod_{i=1}^{\mathrm{cnt}} p_i^{c_i}$ ，现在要从数组挑最少的数，使得对于 X 的每个质因子 $p_i$ ，都存在 $p_i^{c_i}$ 的倍数。

若能求出所有 $p_i^{c_i}$ ，那么可对数组中每个数运用 bitmark，用二进制中 1 的位置来表示它是哪些 $p_i^{c_i}$ 的倍数。

然后问题变成：选一些数，使得他们位或的结果是 $2^{\mathrm{cnt}-1}$。FWT 可做

可是实际情况略为特殊，注意到 X 的范围很大，没法快速进行质因数分解，不妨只用 $10^6$ 内的质数，分解后剩下的 X 无非有以下几种：

- X = 1
- X = 一个大质数 (>=1e6)
- X = 一个大质数 (>=1e6) 的平方
- X = 两个大质数 (>=1e6) 的乘积

同样对 $a_i$ 筛去 $10^6$ 以内的质数，剩余部分和 $X$ 情况相同，但肯定都是 $X$ 的因子

所以枚举每个数的剩余部分，和 $X$ 做比较，若剩余部分等于 1 或者 与 $X$ 的剩余部分相同，则忽略；否则就相当于找到了这两个质因子！！！

如果所有的 $a_i$ 的剩余部分都等于 1 或者 与 $X$ 剩余部分相同，那么就把 $X$ 的剩余部分当作一个质数。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i,s,t) for(int i=s;i<=t;i++)

typedef long long LL;
#define PT pair<LL, int>
const int maxn = 5e4+5;
const int maxm = 1e6+5;

LL gcd(LL x,LL y) { return y?gcd(y,x%y):x; }

vector<int> prime; bool vis[maxm];
void init()
{
    for(int i=2;i<maxm;i++)
    {
        if(vis[i]) continue;
        prime.push_back(i);
        for(int j=i+i;j<maxm;j+=i) vis[j]=true;
    }
}

void FWTor(vector<LL> &a,int n,int inv)
{
     for(int i=1;i<n;i<<=1)
         for(int p=i<<1,j=0;j<n;j+=p)
             for(int k=0;k<i;k++)
                 inv==-1?a[i+j+k]-=a[j+k]:a[i+j+k]+=a[j+k];
}
```

```cpp

int main()
{
    init(); // 素筛
    for(int T,t=1|!scanf("%d",&T);t<=T&&printf("Case %d: ",t);t++)
    {
        int n; LL x; scanf("%d%lld",&n,&x);
        vector<LL> a;
        for(LL v;n--;)
        {
            scanf("%lld",&v);
            if(x%v == 0) a.push_back(v);
        }
        if(!a.size()) printf("-1\n");
        else if(x==1) printf("1\n");
        else
        {
            vector<PT> p; // 存放 P_i^C
            for(int v : prime)
            {
                if(x<v) break;
                int num=0;
                while(x%v==0) x/=v, ++num;
                if(num) p.push_back(PT(v,num));
            }
            int cnt=p.size();
            sort(a.begin(),a.end());
            a.erase(unique(a.begin(),a.end()),a.end());
            vector<int> mask(a.size());
            rep(i,0,a.size()-1)
            {
                mask[i]=0;
                rep(j,0,cnt-1)
                {
                    int num=0;
                    while(a[i]%p[j].first==0)
                        a[i]/=p[j].first, ++num;
                    if(num==p[j].second) mask[i]|=1<<j;
                }
            }
            if(x>1)
            {
                vector<LL> w;
                for(LL v : a)
                    if(v!=x && v!=1) // 每个数剩余部分放进去
                        w.push_back(v),w.push_back(x/v);
                sort(w.begin(), w.end());
                w.erase(unique(w.begin(),w.end()),w.end());
                if(!w.size()) w.push_back(x);    // x=a
                else if(w.size()==1) w[0]=x;     // x=两个大质数平方
                rep(i,0,a.size()-1)
                    rep(j,0,w.size()-1)
                        if(a[i]%w[j]==0)
                            mask[i]|=1<<(cnt+j);
                cnt+=w.size();
            }
            int cur=0;
            for(int v : mask) cur|=v;
            if(cur+1 != (1<<cnt)) printf("-1\n"); // 全集lcm!=X
```

```
            else
            {
                vector<LL> f(1<<cnt);
                rep(i,0,(1<<cnt)-1) f[i]=0;
                for(int v:mask) f[v]=1;

                vector<LL> g=f; FWTor(g,1<<cnt,1);

                int ans=1;
                while(!f[(1<<cnt)-1])
                {
                    FWTor(f,1<<cnt,1);
                    rep(i,0,(1<<cnt)-1) f[i]*=g[i];
                    FWTor(f,1<<cnt,-1);
                    rep(i,0,(1<<cnt)-1) f[i]=(f[i]>0);
                    ++ans;
                }
                printf("%d\n",ans);
            }
        }
    }
    return 0;
}
```