MATHEMATICS

3.数学知识

3.1数论

3.1.1扩展欧几里德算法

首先我们有欧几里德算法：

$$gcd(a,b)=gcd(a\ mod\ b,b)$$

扩展欧几里德算法解决了这样的问题：

$$ax+by=gcd(a,b)$$

我们先考察一种特殊的情况：

当$b=0$时，我们直接可以有解：

$$\begin{cases}x=1\\y=0\end{cases}$$

一般地，我们令$c=a\ mod\ b$，递归地解下面的方程：

$$bx^{'}+cy^{'}=gcd(b,c)$$

根据欧几里德算法，有

$$bx^{'}+cy^{'}=gcd(a,b)$$

根据mod的定义我们可以有

$$c=a-b\lfloor\frac{a}{b}\rfloor$$

带入原式

$$bx^{'}+(a-b\lfloor\frac{a}{b}\rfloor)y^{'}=gcd(a,b)$$

为了体现与a,b的关系

$$ay^{'}+b(x^{'}-\lfloor\frac{a}{b}\rfloor y^{'})=gcd(a,b)$$

所以这样就完成了回溯。

这个算法的思想体现在了下面的程序里。

```
void gcd(int a, int b, int &d, int &x, int &y) {
  if(!b) {d = a; x = 1; y = 0; }
  else { gcd(b, a%b, d, y, x); y -= x * (a/b); }
}
```

3.1.2线性筛与积性函数

3.1.2.1线性筛素数

首先给出线性筛素数的程序。

```
void get_su(int n) {
  tot = 0;
  for(int i = 2; i <= n; i++) {
    if(!check[i]) prime[tot++] = i;
    for(int j = 0; j < tot; j++) {
      if(i * prime[j] > n) break;
      check[i * prime[j]] = true;
      if(i % prime[j] == 0) break;
    }
  }
}
```

可以证明的是，每个合数都仅仅会被他的最小质因数筛去，这段代码的时间复杂度是$\Theta(n)$的，也就是所谓线性筛。

> 证明：设合数n最小的质因数为p，它的另一个大于p的质因数为$p^{'}$，另$n=pm=p^{'}m^{'}$。观察上面的程序片段，可以发现j循环到质因数p时合数n第一次被标记（若循环到p之前已经跳出循环，说明n有更小的质因数），若也被$p^{'}$标记，则是在这之前（因为$m^{'}<m$），考虑i循环到$m^{'}$，注意到$n=pm=p^{'}m^{'}$且$p,p^{'}$为不同的质数，因此$p|m^{'}$，所以当j循环到质数p后结束，不会循环到$p^{'}$，这就说明n不会被$p^{'}$筛去。

3.1.2.2积性函数

- 考虑一个定义域为$\mathbb{N}^{+}$的函数f（数论函数），对于任意两个互质的正整数a,b，均满足

$f(ab)=f(a)f(b)$，则函数f被称为积性函数。

- 如果对于任意两个正整数a,b，都有$f(ab)=f(a)f(b)$，那么就被称作完全积性函数。

容易看出，对于任意积性函数，$f(1)=1$。

- 考虑一个大于1的正整数N，设$N=\prod p_i^{a_i}$，那么

$f(N)=f(\prod p_i^{a_i})=\prod f(p_i^{a_i})$，如果$f$还满足完全积性，那么

$$f(N)=\prod f(p_i)^{a_i}$$

- 如果f是一个任意的函数，它使得和式$g(m)=\sum_{d|m}f(d)$为积性函数，那么f也一定是积性函数。
- 积性函数的Dirichlet前缀和也是积性函数。这个定理是上面定理的反命题。
- 两个积性函数的Dirichlet卷积也是积性函数。

3.1.2.3欧拉函数φ

- $\varphi(n)$表示$1..n$中与n互质的整数个数。
- 我们有欧拉定理：

$$n^{\varphi(m)}\equiv 1 \pmod{m} \quad n\perp m$$

可以使用这个定理计算逆元。

- 如果m是一个素数幂，则容易计算$\varphi(m)$，因为有$n\perp p^k\Leftrightarrow p\nmid n$。在$\{0,1,\dots,p^k-1\}$中的$p$的倍数是$\{0,p,2p,\dots,p^k-p\}$，从而有$p^{k-1}$个，剩下的计入$\varphi(p^k)$

$$\varphi(p^k)=p^k-p^{k-1}=(p-1)p^{k-1}$$

- 由上面的推导我们不难得出欧拉函数一般的表示：

$$\varphi(m)=\prod_{p|m}(p^{m_p}-p^{m_p-1})=m\prod_{p|m}(1-\frac{1}{p})=\prod(p-1)p^{m_p-1}$$

- 运用Mobius反演，不难得出$\sum_{d|n}\varphi(d)=n$。
- 当$n>1$时，$1..n$中与n互质的整数和为$\frac{n\varphi(n)}{2}$
- 降幂大法$A^B\ mod\ C=A^{B\ mod\ \varphi(C)+\varphi(C)}\ mod\ C$

3.1.2.4线性筛法求解积性函数

- 积性函数的关键是如何求$f(p^k)$。
- 观察线性筛法中的步骤，筛掉n的同时还得到了他的最小的质因数p，我们希望能够知道p在n中的次数，这样就能够利用$f(n)=f(p^k)f(\frac{n}{p^k})$求出$f(n)$。
- 令$n=pm$，由于p是n的最小质因数，若$p^2|n$，则$p|m$，并且p也是m的最小质因数。这样在筛法的同时，记录每个合数最小质因数的次数，就能算出新筛去合数最小质因数的次数。
- 但是这样还不够，我们还要能够快速求解$f(p^k)$，这时一般就要结合f函数的性质来考虑。
- 例如欧拉函数φ，$\varphi(p^k)=(p-1)p^{k-1}$，因此进行筛法求$\varphi(p*m)$时，如果$p|m$，那么$p*m$中$p$的次数不为1,所以我们可以从$m$中分解出$p$，那么$\varphi(p*m)=\varphi(m)*p$，否则$\varphi(p*m)=\varphi(m)*(p-1)$。
- 再例如默比乌斯函数μ，只有当$k=1$时$\mu(p^k)=-1$，否则$\mu(p^k)=0$，和欧拉函数一样根据m是否被p整除判断。

```
void Linear_Shaker(int N) {
  phi[1] = mu[1] = 1;
```

```
    if (!phi[i]) {
      phi[i] = i - 1;
      mu[i] = -1;
      prime[cnt++] = i;
    }
    for (int j = 0; j < cnt; j++) {
      if (prime[j] * i > N)
        break;
      if (i % prime[j] == 0) {
        phi[i * prime[j]] = phi[i] * prime[j];
        mu[i * prime[j]] = 0;
        break;
      } else {
        phi[i * prime[j]] = phi[i] * (prime[j] - 1);
        mu[i * prime[j]] = -mu[i];
      }
    }
  }
  for (int i = 2; i <= N; i++) {
    phi[i] += phi[i - 1];
    mu[i] += mu[i - 1];
  }
}
```

3.1.2.5线性筛逆元

令$f(i)$为i在$mod\ p$意义下的逆元。显然这个函数是积性函数，我们可以使用线性筛求。但是其实没有那么麻烦。

我们设$p=ki+r$，那么$ki+r\equiv 0(mod\ p)$，两边同时乘$i^{-1}r^{-1}$，有$kr^{-1}+i^{-1}\equiv 0$，那么$i^{-1}\equiv -kr^{-1}=-\lfloor\frac{p}{i}\rfloor*(p\ mod\ i)^{-1}$，这样就可以递推了。

```
void getinv(int n) {
  inv[1] = 1;
  for(int i = 2; i <= x; i++)
    inv[i] = (long long)(p - p/i)*inv[p % i] % p;
}
```

有了逆元，我们就可以预处理阶乘的逆元

$$n!^{-1}\equiv\prod_{k=1}^{n}k^{-1}\ mod\ p$$

3.1.3默比乌斯反演与狄利克雷卷积

3.1.3.1初等积性函数μ

μ就是容斥系数。

$$\mu(n)=\begin{cases}0, & \text{if } \exists x^2|n\\ (-1)^k & n=\prod_{i=1}^{k}p_i\end{cases}$$

μ函数也是一个积性函数。

下面的公式可以从容斥的角度理解。

$$\sum_{d|n}\mu(d)=[n=1]$$

3.1.3.2默比乌斯反演

首先给出Mobius反演的公式：

$$F(n)=\sum_{d|n}f(d)\Rightarrow f(n)=\sum_{d|n}\mu(\frac{n}{d})F(d)$$

有两种常见的证明，一种是运用Dirichlet卷积，一种是使用初等方法。

证明：

$$\begin{aligned}\sum_{d|n}\mu(d)F(\frac{n}{d})&=\sum_{d|n}\mu(\frac{n}{d})F(d)=\sum_{d|n}\mu(\frac{n}{d})\sum_{k|d}f(k)\\&=\sum_{d|n}\sum_{k|d}\mu(\frac{n}{d})f(k)=\sum_{k|n}\sum_{d|\frac{n}{k}}\mu(\frac{n}{kd})f(k)\\&=\sum_{k|n}\sum_{d|\frac{n}{k}}\mu(d)f(k)=\sum_{k|n}[\frac{n}{k}=1]f(k)=f(n)\end{aligned}$$

默比乌斯反演的另一种形式：

$$F(n)=\sum_{n|d}f(d)\Rightarrow f(n)=\sum_{n|d}\mu(\frac{d}{n})F(d)$$

这个式子的证明与上式大同小异，我在这里写一下关键步骤

$$\sum_{n|d}\sum_{d|k}\mu(\frac{d}{n})f(k)=\sum_{n|k}\sum_{d|\frac{k}{n}}\mu(d)f(k)=f(n)$$

对于一些函数$f(n)$，如果我们很难直接求出他的值，而容易求出倍数和或者因数和$F(n)$，那么我们可以通过默比乌斯反演来求得$f(n)$的值

3.1.3.3狄利克雷卷积

定义两个数论函数$f(x)$,$g(x)$的$Dirichlet$卷积

$$(f*g)(n)=\sum_{d|n}f(d)g(\frac{n}{d})$$

Dirichlet卷积满足交换律，结合律，分配律，单位元。

运用狄利克雷卷积不难证明默比乌斯反演。

3.1.4积性函数求和与杜教筛

3.1.4.1概述

如果能通过狄利克雷卷积构造一个更好计算前缀和的函数，且用于卷积的另一个函数也易计算，则可以简化计算过程。

设$f(n)$为一个数论函数，需要计算$S(n)=\sum_{i=1}^{n}f(i)$。

根据函数$f(n)$的性质，构造一个$S(n)$关于$S(\lfloor\frac{n}{i}\rfloor)$的递推式，如下例：

找到一個合适的数论函数$g(x)$

$$\sum_{i=1}^{n}\sum_{d|i}f(d)g(\frac{i}{d})=\sum_{i=1}^{n}g(i)S(\lfloor\frac{n}{i}\rfloor)$$

可以得到递推式

$$g(1)S(n)=\sum_{i=1}^{n}(f*g)(i)-\sum_{i=2}^{n}g(i)S(\lfloor\frac{n}{i}\rfloor)$$

在递推计算$S(n)$的过程中，需要被计算的$S(\lfloor\frac{n}{i}\rfloor)$只有$O(\sqrt{n})$种。

3.1.4.1欧拉函数求前缀和

利用$\varphi*I=id$的性质，可以有：

$$S(n)=\sum_{i=1}^{n}i-\sum_{i=2}^{n}S(\lfloor\frac{n}{i}\rfloor)$$

3.1.4.2默比乌斯函数前缀和

利用$\mu*I=e$的性质，可以有：

$$S(n)=1-\sum_{i=2}^{n}S(\lfloor\frac{n}{i}\rfloor)$$

3.1.4.3模板

```
ll get_p(int x) { return (x <= m) ? phi[x] : p[n / x]; };
ll get_q(int x) { return (x <= m) ? mu[x] : q[n / x]; };
void solve(ll x) {
  if (x <= m)
    return;
  int i, last = 1, t = n / x;
  if (vis[t])
```

```
  vis[t] = 1;
  p[t] = ((x + 1) * x) >> 1;
  q[t] = 1;
  while (last < x) {
    i = last + 1;
    last = x / (x / i);
    solve(x / i);
    p[t] -= get_p(x / i) * (last - i + 1);
    q[t] -= get_q(x / i) * (last - i + 1);
  }
}
//注：本代码为了避免数组过大，p[]和q[]记录的是分母的值。
```

3.1.5中国剩余定理

中国剩余定理给出了以下的一元线性同余方程组有解的判定条件：

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

中国剩余定理指出，如果模数两两互质，那么方程组有解，并且通解可以构造得到：

1. 设$M = \prod_{i=1}^{n} m_i$，并设$M_i = \frac{M}{m_i}$。
2. 设$t_i = M_i^{-1} \pmod{m_i}$。
3. 那么通解$x = \sum_{i=1}^{n} a_i t_i M_i$。

3.1.6高斯消元

Gauss消元法就是使用初等行列式变换把原矩阵转化为上三角矩阵然后回套求解。给定一个矩阵以后，我们考察每一个变量，找到它的系数最大的一行，然后根据这一行去消除其他的行。

```
double a[N][N]
void Gauss(){
    for(int i=1;i<=n;i++){
        int r=i;
        for(int j=i+1;j<=n;j++)
            if(abs(a[j][i])>abs(a[r][i])) r=j;
        if(r!=i) for(int j=1;j<=n+1;j++) swap(a[i][j],a[r][j]);

        for(int j=i+1;j<=n;j++){
            double t=a[j][i]/a[i][i];
            for(int k=i;k<=n+1;k++) a[j][k]-=a[i][k]*t;
        }
    }
    for(int i=n;i>=1;i--){
        for(int j=n;j>i;j--) a[i][n+1]-=a[j][n+1]*a[i][j];
        a[i][n+1]/=a[i][i];
```

```
}
```

对于xor运算，我们可以使用同样的方法消元。

另外，xor的话可以使用bitset压位以加速求解。

3.1.7大步小步BSGS算法

大步小步算法用于解决：

已知A,B,C，求x使得

$$A^x \equiv B \pmod C$$

我们令$x = i \times m - \frac{j}{m} = \lceil \sqrt{C} \rceil, i \in [1,m], j \in [0,m]$

那么原式就变成了$A^{im} = A^j \times B$.我们先枚举j，把$A^j \times B$加入哈希表，然后枚举i，在表中查找A^{im}，如果找到了，就找到了一个解。时间复杂度为$\Theta(\sqrt{n})$。

```
ll BSGS(ll A, ll B, ll C) {
    mp.clear();
    if(A % C == 0) return -2;
    ll m = ceil(sqrt(C));
    ll ans;
    for(int i = 0; i <= m; i++) {
        if(i == 0) {
            ans = B % C;
            mp[ans] = i;
            continue;
        }
        ans = (ans * A) % C;
        mp[ans] = i;
    }
    ll t = pow(A, m, C);
    ans = t;
    for(int i = 1; i <= m; i++) {
        if(i != 1)ans = ans * t % C;
        if(mp.count(ans)) {
            int ret = i * m % C - mp[ans] % C;
            return (ret % C + C)%C;
        }
    }
    return -2;
}
```

3.2组合数学

3.2.1组合计数与二项式定理

3.2.1.1二项式定理

$$(a+b)^n = \sum_{i=0}^{n} C_n^i a^i b^{n-i}$$

其中C_n^m为二项式系数，满足几个结论：

$$C_n^i = C_n^{n-i}$$

$$C_{n+1}^m = C_n^m + C_n^{m-1}$$

$$\sum_{i=0}^{n} C_n^i = 2^n$$

$$C_n^k = \frac{k}{n} C_{n-1}^{k-1}$$

3.2.1.2排列组合

- 隔板法与插空法
- n元素集合的循环r排列的数目是$\frac{P_n^r}{r}$
- 多重集合全排列$\frac{n!}{\prod_i n_i!}$
- 多重集合的组合，无限重复数，设S是有k种类型对象的多重集合，r组合的个数为$C_{r+k-1}^r = C_{r+k-1}^{k-1}$。
- $Lucas$定理(p为素数)：

$$C_n^m \equiv C_{n/p}^{m/p} \times C_{n \bmod p}^{m \bmod p} \pmod p$$

```
int C(int n, int m, int P) {
    if (n < m) return 0;
    return (ll)fac[n] * inv(fac[n-m], P) % P * inv(fac[m], P)%P;
}
int lucas(int n, int m, int P) {
    if(!n && !m) return 1;
    return (ll)lucas(n/P, m/P, P) * C(n%P, m%P, P) % P;
}
```

3.2.2常见数列

- 错位排列 $D_n = (n-1) * (D_{n-1} + D_{n-2})$
- Catanlan数
  $C(n) = \sum (C(n-I) * C(I))$
  计算公式：
  $C(n) = \frac{C(2n, n)}{n+1}$

应用：
满足递推关系的均可表示成catalan数，比如：
出栈顺序，二叉树种类个数，门票问题，格子问题（不穿过对角线），括号配对问题等等。

3.2.3置换群与$Polya$定理

- $Burnside$引理(Z_k:k不动置换类,$c_1(a)$:一阶循环的个数:
$l=\frac{1}{|G|}\sum_{k=1}^{n}|Z_k|=\frac{1}{|G|}\sum_{j=1}^{g}c_1(a_j)$
- 设置换群G作用于有限集合χ上，用k种颜色涂染集合χ中的元素，则χ在G作用下等价类的数目为($m(f)$为置换f的循环节)：$N=\frac{1}{|G|}\sum_{f\in G}k^{m(f)}$

3.2.4容斥原理

$$|\cup_{i=1}^{n}A_i|=\sum_{i=1}^{n}A_i-\sum_{i,j:i\neq j}|A_i\cap A_j|+\sum_{i,j,k:i\neq j\neq k}|A_i\cap A_j\cap A_k|-\cdots\pm|A_1\cap\cdots\cap A_n|$$

```
void calc(){
    for(int i = 1; i < (1 << ct); i++) {
        //do sth
        for(int j = 0; j < ct; j++) {
            if(i & (1 << j)) {
                cnt++;
                //do sth
            }
        }
        if(cnt & 1) ans += tmp;
        else ans -= tmp;
    }
}
```

3.3常见结论与技巧

3.3.1裴蜀定理

若a,b是整数,且（a,b)=d，那么对于任意的整数x,y,ax+by都一定是d的倍数，特别地，一定存在整数x,y，使ax+by=d成立。

它的一个重要推论是：a,b互质的充要条件是存在整数x,y使ax+by=1.

3.3.2底和顶

- 若连续且单调增的函数$f(x)$满足当$f(x)$为整数时可推出x为整数，则$\lfloor f(x)\rfloor=\lfloor f(\lfloor x\rfloor)\rfloor$和$\lceil f(x)\rceil=\lceil f(\lceil x\rceil)\rceil$
- $\lfloor\frac{\lfloor\frac{x}{a}\rfloor}{b}\rfloor=\lfloor\frac{x}{ab}\rfloor$

- 对于i，$\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \rfloor$是与$i$被$n$除并下取整取值相同的一段区间的右端点

3.3.3和式

3.3.3.1三大定律

- 分配律 $\sum_{k \in K} ca_k = c \sum_{k \in K} a_k$
- 结合律$\sum_{k \in K} (a_k + b_k) = \sum_{k \in K} a_k + \sum_{k \in K} b_k$
- 交换律$\sum_{k \in K} a_k = \sum_{p(k) \in K} a_{p(k)}$其中p(k)是n的一个排列
- 松弛的交换律:若对于每一个整数n，都恰好存在一个整数k使得$p(k) = n$，那么交换律同样成立。

3.3.3.2求解技巧

- 扰动法，用于计算一个和式，其思想是从一个未知的和式开始，并记他为S_n：$S_n = \sum_{0 \leqslant k \leqslant n} a_k$，然后，通过将他的最后一项和第一项分离出来，用两种方法重写S_{n+1}，这样我们就得到了一个关于S_n的方程，就可以得到其封闭形式了。
- 一个常见的交换
  $\sum_{d|n} f(d) = \sum_{d|n} f(\frac{n}{d})$

3.3.3.3多重和式

- 交换求和次序：

$$\sum_j \sum_k a_{j,k}[P(j,k)] = \sum_{P(j,k)} a_{j,k} = \sum_k \sum_j a_{j,k}[P(j,k)]$$

- 一般分配律：$\sum_{j \in J, k \in K} a_j b_k = (\sum_{j \in J} a_j)(\sum_{k \in K} b_k)$
- $Rocky\ Road$

$$\sum_{j \in J} \sum_{k \in K(j)} a_{j,k} = \sum_{k \in K'} \sum_{j \in J'} a_{j,k}$$

$$[j \in J][k \in K(j)] = [k \in K'][j \in J'(k)]$$

事实上，这样的因子分解总是可能的：我们可以设$J = K'$是所有整数的集合，而$K(j)$和$J'(K)$是与操控二重和式的性质$P(j,k)$相对应的集合。下面是一个特别有用的分解。

$$[1 \leqslant j \leqslant n][j \leqslant k \leqslant n] = [1 \leqslant j \leqslant k \leqslant n] = [1 \leqslant k \leqslant n][1 \leqslant j \leqslant k]$$

- 一个常见的分解

$$\sum_{d|n}\sum_{k|d}=\sum_{k|m}\sum_{d|\frac{m}{k}}$$

- 一个技巧

  如果我们有一个包含$k+f(j)$的二重和式，用$k-f(j)$替换k并对j求和比较好。

3.3.4数论问题的求解技巧

- $\{\lfloor\frac{n}{i}\rfloor|i\in[1,n]\}$只有$O(\sqrt{n})$种取值。所以可以使用这个结论降低复杂度。

例如，在bzoj2301中，我们最终解出了$f(n,m)=\sum_{1\leqslant d\leqslant min(n,m)}\mu(d)\lfloor\frac{n}{d}\rfloor\lfloor\frac{m}{d}\rfloor$我们就可以使用杜教筛计算出默比乌斯函数的前缀和，计算出商与除以i相同的最多延伸到哪里，下一次直接跳过这一段就好了。下面是这个题的一段程序。

```
int calc(int n, int m) {
    int ret = 0, last;
    if(n > m) std::swap(n, m);
    for(int i = 1; i <= n; i = last + 1) { //i就相当于原式中的d
        last = min(n / (n/i), m / (m/i));  //last计算了商与除以i相同的最多延伸到哪里，不难证明这样计算的正确性
        ret += (n / i) * (m / i) * (sum[last] - sum[i-1]);
    }
    return ret;
}
```

3.3.5一些可能会用到的定理

3.3.5.1费马小定理

$$a^{p-1}\equiv 1\pmod p$$

条件：$p\ is\ prime\ and\ (a,p)=1$

3.3.5.2欧拉定理

$$a^{\varphi(p)}\equiv 1\pmod p$$

条件：$a,p\in\mathbb{Z}^+,(a,p)=1$

3.3.5.3威尔逊定理

$$(p-1)!\equiv -1\pmod p\Leftrightarrow p\ is\ prime$$

3.3.5.4皮克定理

$$S = n + \frac{s}{2} - 1$$

(其中n表示多边形内部的点数,s表示多边形边界上的点数,S表示多边形的面积)

3.4其他数学工具

3.4.1快速乘

```
inline ll mul(ll a, ll b) {
  ll x = 0;
  while (b) {
    if (b & 1)
      x = (x + a) % p;
    a = (a << 1) % p;
    b >>= 1;
  }
  return x;
}
```

3.4.2快速幂

```
inline ll pow(ll a, ll b, ll p) {
  ll x = 1;
  while (b) {
    if (b & 1)
      x = mul(x, a);
    a = mul(a, a);
    b >>= 1;
  }
  return x;
}
```

3.4.3更相减损术

第一步：任意给定两个正整数；判断它们是否都是偶数。若是，则用2约简；若不是则执行第二步。

第二步：以较大的数减较小的数，接着把所得的差与较小的数比较，并以大数减小数。继续这个操作，直到所得的减数和差相等为止。

则第一步中约掉的若干个2与第二步中等数的乘积就是所求的最大公约数。

3.4.4逆元

根据费马小定理，

```
int inv(int x, int P){return pow(x, P-2, P);}
```

3.4.5博弈论

- Nim游戏
- SG函数

定义$SG(x)=mex(S)$，其中S是x的后继状态的SG函数集合，$mex(S)$表示不在S内的最小非负整数。

- SG定理

组合游戏和的SG函数等于各子游戏SG函数的Nim和。

3.4.6概率与数学期望

- 全概率公式$P(A)=P(A|B_1)*P(B_1)+P(A|B_2)*P(B_2)+\cdots+P(A|B_n)*P(B_n)$
- 数学期望

3.4.7快速傅立叶变换

3.4.7.1基本定义

快速傅立叶变换(FFT)用于求解多项式的卷积.

- 单位根:单位圆的n等分点为终点,作n个向量,所得的幅角为正且最小的向量对应的复数为ω_n,称为n次单位根.
  $\omega_n^k=cosk\frac{2\pi}{n}+isin\ k\frac{2\pi}{n}$
- 单位根的性质:$\omega_{2n}^{2k}=\omega_n^k$
- 单位根的性质:$\omega_n^{k+\frac{n}{k}}=-\omega_n^k$

3.4.7.2快速傅立叶变换

考虑将多项式$A_1(x)=a_0+a_2x^2+a_4x^4+\cdots+a_{n-2}x^{\frac{n}{2}-1}$

$A_2(x)=a_1+a_3x+a_5x^2+\cdots+a_{n-1}x^{\frac{n}{2}-1}$

则有$A(x)=A_1(x)+xA_2(x)$

有$A(\omega_n^k)=A_1(\omega_n^{2k})+\omega_n^kA_2(\omega_n^{2k})$

有$A(\omega_n^{k+\frac{n}{2}})=A_1\omega_{\frac{n}{2}}^k-\omega_n^kA_2(\omega_{\frac{n}{2}}^k)$.

注意到,当k取遍$[0,\frac{n}{2}-1]$时,k和$k+\frac{n}{2}$取遍了$[0,n-1]$,也就是说，如果已知$A_1(x)$和$A_2(x)$在$\omega_{n/2}^0,\omega_{n/2}^1,\cdots,\omega_{n/2}^{n/2-1}$处的点值，就可以在$O(n)$的时间内求得$A(x)$在$\omega_n^0,\omega_n^1,\cdots,\omega_n^{n-1}$处的取值。而关于 $A_1(x)$ 和 $A_2(x)$ 的问题都是相对于原问题规模缩小了一半的子问题，分治的边界为一个常数项a_0.

该算法的复杂度为$O(nlogn)$.

3.4.7.3傅立叶逆变换

设$(y_0,y_1,\cdots,y_{n-1})$为$(a_0,\cdots,a_{n-1})$的快速傅立叶变换. 考虑另一个向量$(c_0,\cdots,c_{n-1})$满足

$$c_k = \sum_{i=0}^{n-1} y_i (\omega_n^{-k})^i.$$

即多项式$B(x) = y_0 + y_1 x + \cdots + y_{n-1} x^{n-1}$在$\omega_n^0, \cdots, \omega_n^{-(n-1)}$处的点值表示.

可以得到(证明略)

$$a_i = \frac{1}{n} c_i.$$

所以，使用单位根的倒数代替单位根，做一次类似快速傅里叶变换的过程，再将结果每个数除以n，即为傅里叶逆变换的结果。

3.7.4.4代码实现

FFT有两种常见的代码实现: 递归版本和迭代版本,一般来讲递归效率很差,但由于我非常菜,一轮省选就先使用递归版本骗分.迭代版本以后会更新.

```
const double PI = acos(-1);
bool inversed = false;
inline std::complex<double> omega(const int n, const int k) {
  if (!inversed)
    return std::complex<double>(cos(2 * PI / n * k), sin(2 * PI / n * k));
  else
    return std::conj(
        std::complex<double>(cos(2 * PI / n * k), sin(2 * PI / n * k)));
}
void fft(std::complex<double> *a, std::complex<double> *ans, int n) {
  if (n == 1) {
    ans[0] = a[0];
    return;
  }
  std::complex<double> buf[maxn];
  int m = n >> 1;
  for (int i = 0; i < m; i++) {
    buf[i] = a[i * 2];
    buf[i + m] = a[i * 2 + 1];
  }
  std::complex<double> tmp[maxn];
  fft(buf, tmp, m);
  fft(buf + m, tmp + m, m);
  for (int i = 0; i < m; i++) {
    std::complex<double> x = omega(n, i);
    ans[i] = tmp[i] + x * tmp[i + m];
    ans[i + m] = tmp[i] - x * tmp[i + m];
  }
}
void solve() {
  //sth
```

```
    N <<= 1;
  N <<= 1;
  fft(a, ans1, N);
  fft(b, ans2, N);
  std::complex<double> ans3[maxn];
  for (int i = 0; i < N; i++)
    ans3[i] = ans1[i] * ans2[i];
  std::complex<double> tmp[maxn];
  inversed = true;
  fft(ans3, tmp, N);
  for (int i = 0; i < N; i++)
    c[i] = tmp[i].real() / N;
  //sth
}
```