数论

逆元

- 求逆元的方式有很多中，根据不同的情况可以利用不同的方法来进行求解。
- 如果模数为素数，可以使用费马小定理进行求解。
- 利用扩展欧几里得算法进行求解。

```
void exgcd(int a,int b,int &x,int &y){
    if(b==0){
        x=1,y=0;
        return;
    }
    exgcd(b,a%b,y,x);
    y=y-a/b*x;
}
```

- 线性求1-n逆元。线性求逆元可以是我们在O(n）的时间复杂度内实现求1-n中每个数关于p的逆元。如果我们用上述的方法求解，很可能会超时。
  - 首先，我们要知道1的逆元是1，。
  - 然后，对于每个i^{-1},我们很显然知道了i之前的每个数j的逆元了。
  - 模板如下：

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

  - 这里使用p-p/i来防止负数的产生。同时要注意的是，这里的i应该与p互质，逆元的定义就是两个互质的数之间存在逆元，如果不互质，这个逆元是不存在的。
- 线性求n个数之间的逆元
  - 首先，我们求出n个数的前缀积，然后利用快速幂或者扩展欧几里得算法计算S_n的逆元，记为SV_n
  - 然后，我们将SV_n乘上n个数中的每一个的a_i,我们就得到了SV_i,这个数就是n个数中不包含a_i的乘积的逆元了。
  - 最后a_i^{-1}就可以用$S_{i-1}SV_i$来进行表示出来。

    ```
    s[0]=1;
    for(int i=1;i<=n;i++){
        s[i]=s[i-1]*a[i]%p;
    }
    s[n]=qpow(s[n],p-2);
    for(int i=n;i>=1;--i){
        sv[i-1]=sv[i]*a[i]%p;
    }
    for(int i=1;i<=n;i++){
        inv[i]=sv[i]*s[i-1]%p;
    }
    ```

  -

蜚蜀定理

- 蜚蜀定理时关于最大公因数的定理，定义

  若gcd(a,b)=d,那么对于任意的整数x,y,ax+by都一定时d的倍数，特别的，一定存在整数x,y，使得ax+by=d,也就是ax+by=gcd(a,b)成立。

- n个整数间的蜚蜀定理

  设a1,a2,a3...an为n个整数，**d时他们的最大公约数**，那么存在整数x1,x2,x3..xn使得x1a1+x2a2+..+xnan=d,特殊的，如果这n个数字都是互质的，那么就存在x1a1+x2a2+..+xnan=1.

```
ll gcd(ll a,ll b){
    if(b==0) return a;
    return gcd(b,a%b);
}
```

概率论

- 给出n个物体，让我们每次从里面取出一个，问刚好取出n个不同的物品的期望。

$$E(x)=n\sum_{i=1}^{n}\frac{1}{i}=n(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n})$$

线性代数

- 矩阵的乘法
  - 快速求非波纳西数列的第n项

  给出一个非波纳西数列的最初始的矩阵为| 1 1 | ，然后，对于非波纳西数列的第n项

$$(f(n),f(n-1))=(1,1)\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-2}$$

  模板

```
#include<iostream>
using namespace std;
double a[20][20],b[20][20],c[20];
int main(){
    int n;
cin>>n;
for(int i=1;i<=n;i++){    //输入矩阵的系数行列式
    for(int j=1;j<=n;j++){
        cin>>a[i][i];
    }
}
for(int i=1;i<=n;i++){  //输入方程右边
    cin>>b[i];
}
//将行列式化为上三角行列式
for(int i=1;i<=n;i++){  //按照列的顺序进行处理
    for(int j=i;j<=n;j++){
        if(fabs(a[i][j])>1e-8){
            for(int k=1;k<=n;k++){
                swap(a[i][k],a[j][k]);
            }
            swap(b[i],b[j]);
        }
```

```
        }
        for(int j=1;j<=n;j++){  //对行进行遍历
            if(i==j) continue;  //对角线自动忽略
            double rate=a[j][i]/a[i][i];
            for(int k=i;k<=n;k++){
                a[j][k]-=a[i][k]*rate;
            }
            b[j]-=b[i]*rate;
        }
    }
    //最后化为只有对角线的系数不为0的行列式
    for(int i=1;i<=n;i++){
        cout<<b[i]/a[i][i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

组合数

- 卢卡斯定理

  卢卡斯定理的使用范围是对于n，m很大，然后p比较小的情况来的。但是要注意的是这个p要为素数

$$lucas(n,m,p) = C(n\%p, m\%p, p) * lucas(n/p, m/p, p)$$

```
ll jc[N]  //表示从i-N模p的阶乘
ll qk(ll a,ll b){
    if(b==0) return 1;
    if(b&1) return a*qk(a,b-1)%p;
    else{
        ll mul=qk(a,b/2);
        return mul*mul%p;
    }
}
ll C(ll n,ll m,ll p){
    if(n<m) return 0;
    return jc[n]*qk(jc[n-m],p-2)%p*qk(jc[m],p-2)%p;
}
ll lucas(ll n,ll m,ll p){
    if(n<m) return 0;
    if(m==0) return 1;
    if(n==0) return 1;
    return lucas(n/p,m/p,p)*C(n%p,m%p,p)%p
}
```

- 扩展卢卡斯

  对于p不为素数的情况，我们可以对p进行质因数分解，然后利用中国剩余定理来求解，这里目前对原理还是没有深入地理解，所以先给出模板

```
#include<iostream>
#include<stack>
#include<cmath>
#include<vector>
#include<cstring>
#include<map>
```

```cpp
#include<algorithm>
using namespace std;
typedef long long ll;
ll n,m,p;
ll exgcd(ll a,ll b,ll &x,ll &y)
{
    if(!b){x=1;y=0;return a;}
    ll res=exgcd(b,a%b,x,y),t;
    t=x;x=y;y=t-a/b*y;
    return res;
}
ll power(ll a,ll b,ll mod)
{
    ll sm;
    for(sm=1;b;b>>=1,a=a*a%mod)if(b&1)
        sm=sm*a%mod;
    return sm;
}

ll fac(ll n,ll pi,ll pk)
{
    if(!n)return 1;
    ll res=1;
    for(register ll i=2;i<=pk;++i)
        if(i%pi)(res*=i)%=pk;
    res=power(res,n/pk,pk);
    for(register ll i=2;i<=n%pk;++i)
        if(i%pi)(res*=i)%=pk;
    return res*fac(n/pi,pi,pk)%pk;
}
ll inv(ll n,ll mod)
{
    ll x,y;
    exgcd(n,mod,x,y);
    return (x+=mod)>mod?x-mod:x;
}

ll CRT(ll b,ll mod){return b*inv(p/mod,mod)%p*(p/mod)%p;}

const int MAXN=11;

ll w[MAXN];

ll C(ll n,ll m,ll pi,ll pk)
{
    ll up=fac(n,pi,pk),d1=fac(m,pi,pk),d2=fac(n-m,pi,pk);
    ll k=0;
    for(register ll i=n;i;i/=pi)k+=i/pi;
    for(register ll i=m;i;i/=pi)k-=i/pi;
    for(register ll i=n-m;i;i/=pi)k-=i/pi;
    return up*inv(d1,pk)%pk*inv(d2,pk)%pk*power(pi,k,pk)%pk;
}

ll exlucus(ll n,ll m){
    ll res=0,tmp=p,pk;
```

```
    int lim=sqrt(p)+5;
    for(int i=2;i<=lim;i++){
        if(tmp%i==0){
            pk=1;
            while(tmp%i==0){
                pk*=i;
                tmp/=i;
            }
            (res+=CRT(C(n,m,i,pk),pk))%=p;
        }
    }
    if(tmp>1){
        (res+=CRT(C(n,m,tmp,tmp),tmp))%=p;
    }
    return res;
}
int main(){
   // freopen("test.in","r",stdin);
    cin>>n>>m>>p;
    cout<<exlucus(n,m)<<endl;
    fclose(stdout);
    return 0;
}
```

- 对于一个1-n的排列，求集合的排列中没有一个数在它的指定的位置的数的排列的个数
  $D_n = (n-1)(D_{n-1} + D_{n-2})$

博弈论

- min游戏，若干堆石头，每次可以从任意一堆取出若干个，问先手是否可以比赢。

  这个要考虑到画博弈图，我们可以发现

  $$ans = a_1 \bigoplus a_2 \bigoplus a_3 \ldots \bigoplus a_n$$

  若ans=0,则先手比输，否则先手比赢。

- 巴什博弈，有一堆石头，总个数为n，两名玩家轮流在石头堆里面取石头，每次至少取一次，至多取m个，取走最后一个的玩家为胜者，判定先后手谁赢。

  如果(m+1)|n,则先手必败，否则先手必胜。

- 威左夫博弈，有两堆石头，两人轮流取石头，每次，可以从任意一堆取，或者从两队同时取若干个，取到最后一个石头的赢。

  $$a = int((b-a) * \frac{\sqrt{5}+1}{2})$$

  若上面等式成立，那么就是后者胜，否则就是先者胜。

- 非波纳西博弈

  有一堆个数为n的石头，游戏双方轮流取石头，满足：

  先手不能在第一次就把所有的石头取完;

  子后的每次取的石头数介于1到对手刚取的石头数的两倍之间。

  结论：先手胜当且仅当n不是非波纳西数。

- 尼姆博弈

  有三堆物品若干个，两个人轮流从某一堆中取任意多的物品，规定每次至少取一个，多的不限，最后取光者得胜。

  将n堆的物品数量进行异或，结果为0则必败，否则必胜。

矩阵

- 矩阵快速幂是我们平时用于矩阵之间的乘法的一种很常见的方法，他的核心就是利用模拟矩阵的乘法的运算来进行求解，平时我们常见的可能是快速幂的写法，对于矩阵的快速幂其实也是需要掌握的

```
struct node {
    ll a[120][120];
}Node,I;
node operator *(const node&x,const node&y){
    node z;
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
           z.a[i][j]=0;
        }
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            for(int k=1;k<=n;k++){
                z.a[i][j]=(z.a[i][j]+x.a[i][k]*y.a[k][j]%mod)%mod;
            }
        }
    }
    return z;
}
```

- 另外，这里要说明的一个是，加入我们的指数很大的话，用递归的写法很容易爆内存，所以，建议使用非递归的写法。
- 对于利用矩阵进行加速的一些题目，我们首要就是通过推导出原始的矩阵，最后的答案一般是某一个矩阵的某一个元素，如果我们不确定是哪一个矩阵的哪个元素，我们可以先进行打表观察。

高斯消元

求逆矩阵

- 这个板子是利用先构造一个单位矩阵，然后将原矩阵化为单位矩阵，对原矩阵的操作同时施加到对单位矩阵的操作上面，最后得到的那个单位矩阵的情况就是我们要求的矩阵的逆了。

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<vector>
#include<map>
#include<stack>
#include<queue>
#include<fstream>
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
map<int,int> mp;
const int mod = 1e9+7;
const int N=200010;
ll a[450][450<<1];
ll qk(ll a,ll b){
    ll c=1;
    while(b){
        if(b&1) c=c*a%mod;
```

```
        a=a*a%mod;
        b>>=1;
    }
    return c;
}
int main(){
    //freopen("test.in","r",stdin);
    int n;
    cin>>n;
    int m=n+n;
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            cin>>a[i][j];
        }
        a[i][i+n]=1;
    }
    for(int i=1;i<=n;i++){
        for(int j=i;j<=n;j++){
    if(a[j][i]){
        for(int k=1;k<=m;k++){
            swap(a[i][k],a[j][k]);
        }
        break;
    }
}

//无解的情况
if(!a[i][i]){
    puts("No Solution");
    return 0;
}


//将对角线上的元素化为1
ll r=qk(a[i][i],mod-2);
for(int j=i;j<=m;j++){
    a[i][j]=a[i][j]*r%mod;
}



    for(int j=1;j<=n;j++){  //对行进行遍历
        if(j!=i){
            r=a[j][i];
            for(int k=i;k<=m;k++){
                a[j][k]=(a[j][k]-r*a[i][k]%mod+mod)%mod;
            }
        }
    }
}
for(int i=1;i<=n;i++){
    for(int j=n+1;j<=m;j++){
        cout<<a[i][j]<<" ";
    }
    cout<<endl;
```

```
    }
    fclose(stdout);
    return 0;
}
```

求线性方程的解

- 求解线性方程组的解，它的原理是将矩阵化为只有对角线不为0,其他地方全为0的矩阵，我们通过矩阵的行与行之间的消元公式求得。

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<vector>
#include<map>
#include<stack>
#include<queue>
#include<fstream>
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
map<int,int> mp;
const int mod = 1e9+7;
const int N=200010;
double a[120][120];
int main(){
    //freopen("test.in","r",stdin);
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n+1;j++){
            scanf("%lf",&a[i][j]);
        }
    }
    for(int i=1;i<=n;i++){    //枚举列
        int mx=i;
        for(int j=i+1;j<=n;j++){  //找出对应列的行的最大值
            if(fabs(a[j][i])>fabs(a[mx][i])){
                mx=j;
            }
        }
    for(int j=1;j<=n+1;j++){
        swap(a[i][j],a[mx][j]);
    }


    //判断是否无解
    if(!a[i][i]){
        puts("No Solution");
        return 0;
    }

    //同一列的每一行的数都减去这个数
    for(int j=1;j<=n;j++){
        if(j!=i){
            double tmp=a[j][i]/a[i][i];
```

```
                for(int k=i+1;k<=n+1;k++){
                    a[j][k]-=a[i][k]*tmp;
                }
            }
        }
    }
    for(int i=1;i<=n;i++){
        printf("%.2lf\n",a[i][n+1]/a[i][i]);
    }
    fclose(stdout);
    return 0;
}
```

卡特兰数

- 卡特兰数是组合数学里面的一个知识点，它的使用场景为有两种情况，两种情况都是等价的，问最后某一个事物的种类数.它的前几项通常是1,1,2,5,14,42,132...

  以下是关于卡特兰数的一些公式：

- $$\frac{C_n^{2n}}{n+1}(n >= 2)$$
- $$\sum_{i=1}^{n} H_{i-1}H_{n-i}$$

线性基

- 线性基的元素相互异或得到原籍和的元素的所有相互异或的得到的值。
- 线性基满足性质1的最小的集合。
- 线性基没有异或和为0的子集。
- 线性基中每个元素的异或方案唯一。
- 线性基中每个元素的二进制最高为互不相同。
- 构造线性基的方法：对原集合的每个数p转为二进制，从高位向低位进行扫描，如果第i为为1,并且ai不存在，那么令ai=p并结束扫描，如果存在，令p=p^ai.
- 线性基可以用来求一堆数之间相互异或，查询最值和第k大值。

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int mod =1e9+7;
const int MN=60;
ll a[62],tmp[62];
bool flag;
void ins(ll x){
    for(int i=MN;i;i--){
        if(x&(1ll<<i)){
            if(!a[i]){
                a[i]=x;
                return;
            }
            x^=a[i];
        }
    }
    flag=true;
}
```

```cpp
bool check(ll x){
    for(int i=MN;i;i--){
        if(x&(1ll<<i)){
            if(!a[i]){
                return false;
            }
            x^=a[i];
        }
    }
    return true;
}
//查询最大值
ll qmx(ll res=0){
    for(int i=MN;i;i--){
        res=max(res,res^a[i]);
    }
    return res;
}
//查询最小值
ll qmi(){
    if(flag) return 0;
    for(int i=0;i<=MN;i++){
        if(a[i]) return a[i];
    }
    return 0;
}
// 查询一堆数异或出来的第k大值
ll query(ll k){
    ll res=0;
    int cnt=0;
    k-=flag;
    if(k){
        return 0;
    }
    for(int i=0;i<=MN;i++){
        for(int j=i-1;j;j--){
            if(a[i]&(1ll<<j)) a[i]^=a[j];
        }
        if(a[i]) tmp[cnt++]=a[i];
    }
    if(k>=(1ll<<cnt)) return -1;
    for(int i=0;i<cnt;i++){
        if(k&(1ll<<i)) res^=tmp[i];
    }
    return res;
}
int  main(){
    //freopen("test.in","r",stdin);
    ll n,x;
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>x;
        ins(x);
    }
    cout<<qmx()<<endl;
```

```
    fclose(stdout);
    return 0;
}
```

字符串

字符串哈希

- 构造哈希函数
  - 哈希函数的一般构造方法：

$$\sum_{i=1}^{l} s[i]b^{l-i}(modM)$$

    上述构造方法是通过给每个位置的字符增加一个权值，权值的大小根据字符串的位置，这样可以很大程度上避免冲突的产生。
  - 上述的M通常为一个素数，一般可以去233，2333，23333....，为了防止哈希冲突，可以使用多个不同的M；b可以使用任意的值，但是一般都是选择10的倍数。
- 使用场景
  - 可以用于求某一个字符串的子串是否存在？这一场景一般可以利用KMP算法来进行实现，但是当我们的字符串的长度很大的时候，KMP算法容易超时。这样，我们就可以使用字符串哈希+二分的方法来进行求解。
  - 给出n个字符串，要求我们求这n个字符串的最长公共的子字符串。我们可以利用二分来求字符串的长度，然后对于每个长度，我们求每个字符该长度的子串，然后将其存入哈希表里面，然后求这n个哈希表的交集，这里取交集的方法我们可以用一个map来进行存储每个哈希值，同时用一个ans时刻更新map的最大值。
  - 求字符串的最长回文子串。我们首先对字符串进行预处理，然后进行枚举该最长回文子串的中心的坐标。

KMP算法

- next数组

  对于一个字符串，它的每个前缀的next数组就是这个前缀的最长前缀==最长后缀，例如：`abcdeabcd`，它的next数组的值就是4，因为最长相等的前后缀为`abcd`
- 因此，当我们进行模式串匹配的时候，给出的一个文本串t，和一个模式串s，我们可以构造出一个字符串为s#t，这样我们可以通过KMP求出每个位置的next的值，然后对于每一个值，等于模式串长度的就存在它的子串。
- next数组的长度不能为这个字符串的自身。

```
string s;
cin>>s;
int n=s.size();
for(int i=1;i<n;i++){
     int j=kmp[i-1];
     while(j>0&&s[i]!=s[j]) j=kmp[j-1];    //kmp[i]表示的是最长相等的前后缀的长度
     if(s[i]==s[j]) j++;
     kmp[i]=j;
}
```

扩展KMP算法

- 扩展kmp算法是z[i]表示的是字符串s[i,i+1,i+2...n-1]与字符串s的最长公共前缀的长度。
- 利用这个算法可以在线性时间内求出一个字符串在另一个字符串中出现的所有的位置。
- 算法过程

- 最与每个z[i]，我们确保了i之前的所有的z都已经求出来了。那么我们用[l,r]表示我们求到的目前的最长的公共前缀。如果i<r,那么z[i]=min(z[i-l],r-i+1)
- 如果i>r,那么我们直接按照朴素的算法进行求解。
- 最后，如果z[i]>r，我们可以更新l,r

```
vector<int> z_function(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r && z[i - l] < r - i + 1) {
      z[i] = z[i - l];
    } else {
      z[i] = max(0, r - i + 1);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    }
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

字典树

- ```
  //以0为根节点
  struct trie{
      int nxt[100010][26],cnt;
      bool exist[100010];  //该结点结尾的字符串是否存在
      void insert(char *s,int l){  //插入字符串, l为字符串的长度
          int p=0;
          for(int i=0;i<l;i++){
              int c=s[i]-'a';
              if(!nxt[p][c]) nxt[p][c]=++cnt;
              p=nxt[p][c];
          }
          exist[p]=1;
      }

      bool find(char *s,int l){  //l为字符串的长度
          int p=0;
          for(int i=0;i<l;i++){
              int c=s[i]-'a';
              if(!nxt[p][c]) return 0;
              p=nxt[p][c];
          }
          return exist[p];
      }
  }
  ```

- 模板

  ```
  #include<iostream>
  #include<algorithm>
  #include<cstring>
  #include<vector>
  #include<map>
  #include<stack>
  #include<queue>
  ```
  ```

```cpp
#include<fstream>
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int mod = 1e9+7;
struct node{
    int nxt[2];
    int end,sum;
}Node[500010];
int a[100010];
int main(){
  //  freopen("test.in","r",stdin);
    int n,m,js=1;
    memset(Node,0,sizeof(Node));
    cin>>m>>n;
    for(int i=1;i<=m;i++){
        int len;
        cin>>len;
        for(int j=1;j<=len;j++){
            cin>>a[j];
        }
        int now=1;
        for(int j=1;j<=len;j++){
            if(Node[now].nxt[a[j]]!=0) now=Node[now].nxt[a[j]];
            else{
                js++;
                Node[now].nxt[a[j]]=js;
                Node[js].nxt[1]=Node[js].nxt[0]=0;
                now=js;
            }
            Node[now].sum++;
        }
        Node[now].end++;
    }
    for(int i=1;i<=n;i++){
        int len;
        cin>>len;
        for(int j=1;j<=len;j++){
            cin>>a[j];
        }
        int now=1,ans=0;
        bool flag=true;
        for(int j=1;j<=len;j++){
            if(Node[now].nxt[a[j]]!=0) now=Node[now].nxt[a[j]];
            else{
                flag=false;
                break;
            }
            ans+=Node[now].end;
        }
        if(!flag) cout<<ans<<endl;
        else cout<<ans+Node[now].sum-Node[now].end<<endl;
    }
    fclose(stdout);
    return 0;
```

```
}
```

- 字典树就是我们把一个字符串构建成一颗树的形式，我们可以对每一个节点进行标号，然后我们可以从根节点进行便利过去，如果遇到某一个节点的下一个不存在的话，那么就构建一个新的节点出来。最后我们所有的字符串就构建成了一棵树的形式，我们同时可以给每个节点加上一些其他的属性，比如表示经过这个节点的字符串的个数或者是以这个节点结束的字符的个数等等。

Manacher算法

- 马拉车算法是用来求最大的回文字符串的长度的一个算法，它的时间复杂度为O(n).这种算法是采用一边优化，一边暴力的算法进行求解的。核心思想是每次记录下当前的最长的回文串的最左边和最右边的长度，然后，我们枚举每个中心，通过与左右边界的位置来进行判断它的边界。这里奇数和偶数是分开来计算的。我们可以在现在原字符串的左右两边插入一个肯定不会出现的字符，然后在每个字符中间插入一个'#',最后我们只需要按奇数长度的字符来进行处理就行了。
- 计算最长回文半径的模板

```
scanf("%s",c);
int len=strlen(c);
int cnt=0;
s[++cnt]='^',s[++cnt]='#';
for(int i=0;i<len;i++){
    s[++cnt]=c[i];
    s[++cnt]='#';
}
s[++cnt]='!';
int mx=0;
int id=0;
for(int i=1;i<=cnt;i++){
    if(i<mx) p[i]=min(p[id*2-i],mx-i);
    else p[i]=1;
    while(s[i-p[i]]==s[i+p[i]]) p[i]++;
    if(mx<i+p[i]) id=i,mx=i+p[i];
}
```

- 计算最长回文长度的模板

```
#include<bits/stdc++.h>
using namespace std;
int d[23000000];
int main(){
    string s;
    cin>>s;
    int len=s.size();
    string c="";
    for(int i=0;i<len;i++){
        c+="#";
        c+=s[i];
    }
    c+="#";
    len=c.size();
    int ans=0;
    for(int i=0,l=0,r=-1;i<len;i++){
        int k=(i>r)?1:min(d[l+r-i],r-i);
        while(0<=i-k&&i+k<len&&c[i-k]==c[i+k]){
            k++;
        }
```

```
        d[i]=k--;
        ans=max(ans,d[i]-1);
        if(i+k>r){
            l=i-k;
            r=i+k;
        }
    }
    cout<<ans;
    return 0;
}
```

ac自动机

- ac自动机是KMP+Tire两种算法的综合，它解决的问题是如何查询一个文本串可以与多少个模式串的匹配。
- ac自动机是将所有的模式串构造成一棵Trie，我们称之为Trie树。
- 同时，我们要引入一个失配指针fail的概念，失配指针是指一个字符串中匹配当前状态的最长后缀。可以将它和KMP算法的next数组比较起来理解。fail指针就是一个模式串的后缀集合
- ac自动机主要有两个方面，一个是构建fail指针，另一个就是构建自动机。

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e6+6;  //字符串的长度
int n;
int tr[N][26],tot;
int e[N],fail[N];
void insert(char *s){
    int u=0;
    for(int i=1;s[i];i++){
        if(!tr[u][s[i]-'a']) tr[u][s[i]-'a']=++tot;
        u=tr[u][s[i]-'a'];
    }
    e[u]++;
}
queue<int> q;
void build(){
    for(int i=0;i<26;i++){
        if(tr[0][i]){
            q.push(tr[0][i]);
        }
    }
    while(q.size()){
        int u=q.front();
        q.pop();
        for(int i=0;i<26;i++){
            if(tr[u][i]){
                fail[tr[u][i]]=tr[fail[u]][i];
                q.push(tr[u][i]);
            }else{
                tr[u][i]=tr[fail[u]][i];
            }
        }
    }
}
//进行询问的文本串
int query(char *t){
```

```
    int u=0,res=0;
    for(int i=1;t[i];i++){
        u=tr[u][t[i]-'a'];
        for(int j=u;j&&e[j]!=-1;j=fail[j]){
            res+=e[j];
            e[j]=-1;
        }
    }
    return res;
}
char s[N];
int main(){
    //freopen("test.in","r",stdin);
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%s",s+1);
        insert(s);
    }
    scanf("%s",s+1);
    build();
    printf("%d\n",query(s));
    fclose(stdout);
    return 0;
}
```

高精度

- 下面主要是用C++模拟的一些大数加法，减法，乘法和除法

```
vector<int> add(vector<int> &A,vector<int> &B){
    vector<int> C;
    int t=0;
    for(int i=0;i<A.size()||i<B.size();i++){
        if(i<A.size())  t+=A[i];
        if(i<B.size())  t+=B[i];
        C.push_back(t%10);
        t/=10;
    }
    if(t)   C.push_back(t);
    return C;
}

vector<int> mul(vector<int> &A,int b){
    vector<int> C;
    int t=0;
    for(int i=0;i<A.size()||t;i++){
        if(i<A.size())  t+=A[i]*b;//模拟
        C.push_back(t%10);
        t/=10;
    }
    while(C.size()>1&&C.back()==0)  C.pop_back();//排除前导0,b为0时会出现多余前导0
    return C;
}

vector<int> div(vector<int> &A,int b,int &r){
```

```
    vector<int> C;
    for(int i=A.size()-1;i>=0;i--){
        r=r*10+A[i];
        C.push_back(r/b);
        r%=b;
    }
    reverse(C.begin(),C.end());//翻转是为了方便去除前导0
    while(C.size()>1&&C.back()==0)  C.pop_back();
    return C;
}
void print(vector<int> a){
    for(int i=a.size()-1;i>=0;i--){
        cout<<a[i];
    }
    cout<<endl;
}
```

计算几何

距离

- 欧式距离
  - 欧式距离指的是n维的两个点之间的距离。
- 曼哈顿距离
  - 曼哈顿距离是n维坐标中的每一维之间的绝对值的和。
- 切比雪夫距离
  - 切比雪夫距离指的是两个点的坐标绝对值的差的最大值。
- 曼哈顿距离与切比雪夫距离之间可以相互转化。
  - $$d(A,B)=|x_1-x_2|+|y_1-y_2|=max(|(x_1+y_1)-(x_2+y_2)|,|(x_1-y_1)-(x_2-y_2)|)$$
  - $$d(A,B)=max(|x1-x2|,|y1-y2|)=max(|\frac{x1+y1}{2}-\frac{x2+y2}{2}|+|\frac{x1-y1}{2}-\frac{x2-y2}{2}|)$$

- L_m距离
  - $$d(L_m)=(|x_1-x_2|^m+|y_1-y_2|^m)^{\frac{1}{m}}$$
- 汉明距离
  - 汉明距离针对的是两个长度相等的字符串对应位字符不一样的数量。
  - 解决的办法是通过将两个串进行异或运算，如果结果对应位的值为1的话说明就不是相等的。

凸包

- 在平面上能包含所有给定点的最小的凸多边形叫做凸包。
- 对于给定的集合，所有包含x的凸集的交集称为x的凸包。
- 凸包用一个最小的周长围住了给定的所有的点，如果一个凹多边形围住了所有的点，它的周长一定不是最小。
- Andrew算法。时间复杂度为O($nlogn$)。
  - 我们把x坐标作为第一关键字，把y坐标作为第二关键字。
  - 显然，排序最小的元素和最大的元素一定是在凸包上面的，我们从某一个点逆时针出发，我们发现我们一定是往左拐的，一定出现右拐的情况，就说明我们目前走的是凹包，那么显然不会是最短的周长解了。
  - 我们用一个单调栈来维护上下凸壳。先用升序枚举出下凸壳，然后降序枚举出上凸壳。

- 求凸壳的时候，一旦发现即将进栈的点和栈顶的两个点是向右旋转的，也就是叉积小于等于0,那么就弹出栈顶，回退到上一步，知道遇到叉积大于0的情况。

```
// stk[] 是整型，存的是下标
// p[] 存储向量或点
tp = 0;                          // 初始化栈
std::sort(p + 1, p + 1 + n);  // 对点进行排序
stk[++tp] = 1;
//栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
for (int i = 2; i <= n; ++i) {
  while (tp >= 2  // 下一行 * 操作符被重载为叉积
         && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
    used[stk[tp--]] = 0;
  used[i] = 1;  // used 表示在凸壳上
  stk[++tp] = i;
}
int tmp = tp;  // tmp 表示下凸壳大小
for (int i = n - 1; i > 0; --i)
  if (!used[i]) {
    //       ↓求上凸壳时不影响下凸壳
    while (tp > tmp && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
      used[stk[tp--]] = 0;
    used[i] = 1;
    stk[++tp] = i;
  }
for (int i = 1; i <= tp; ++i)  // 复制到新数组中去
  h[i] = p[stk[i]];
int ans = tp - 1;
```

- Graham算法
  - 该算法是通过找出y坐标最小，x坐标最小的那个点。然后利用单调栈的思想来进行求出凸包上面的点。重点是在于排序函数

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
struct node{
    double x,y;
}Node[100010],s[100010];
//逆时针方向走，检查两个向量(a,b)之间的叉乘，以此来判断是否产生凸包
double check(node a1,node a2,node b1,node b2){
    return (a2.x-a1.x)*(b2.y-b1.y)-(b2.x-b1.x)*(a2.y-a1.y);  //x1y2-x2y1
}

//计算两个点之间的距离
double d(node a,node b){
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}


bool cmp(node a,node b){
    double tmp=check(Node[1],a,Node[1],b);
    if(tmp>0){
        return true;
    }
    if(tmp==0&&d(Node[1],a)<d(Node[1],b)) {
```

```
        return true;
    }
    return false;
}
int main(){
    //freopen("test.in","r",stdin);
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%lf%lf",&Node[i].x,&Node[i].y);
        double mid;

        //找出y坐标最小的那个点，如果有多个y最小的点，那么就找x最小的点
        if(i!=1){
            if(Node[i].y<Node[1].y){
                mid=Node[i].y,Node[i].y=Node[1].y,Node[1].y=mid;
                mid=Node[i].x,Node[i].x=Node[1].x,Node[1].x=mid;
            }else if(Node[i].y==Node[1].y){
                if(Node[i].x<Node[1].x){
                    mid=Node[i].y,Node[i].y=Node[1].y,Node[1].y=mid;
                    mid=Node[i].x,Node[i].x=Node[1].x,Node[1].x=mid;
                }
            }

        }
    }
    sort(Node+2,Node+1+n,cmp);


    //s来模拟栈
    s[1]=Node[1];
    int cnt=1;
    for(int i=2;i<=n;i++){

        //栈顶两个点的向量和栈顶与当前点之间形成的向量，这是一个单调栈的思想
        while(cnt>1&&check(s[cnt-1],s[cnt],s[cnt],Node[i])<=0){
            cnt--;
        }
        cnt++;
        s[cnt]=Node[i];
    }
    s[cnt+1]=Node[1]; //合成一个封闭的凸包
    double ans=0;
    for(int i=1;i<=cnt;i++){
        ans+=d(s[i],s[i+1]);
    }
    printf("%.2lf\n",ans);
    fclose(stdout);
    return 0;
}
```

平面最近点对

- 背景：请平面内n个点之间最近的点对。使用的是nlogn的分治算法。
- 首先，我们对于每一个坐标的x和y，我们以x为第一关键字，以y为第二关键字进行排序。我们以中点为m=n/2。中心点的坐标为xm,ym
- 然后，我们将两个点集拆分成两个大小相同的集合S1和S2，然后，我们假设两个集合的最近的点对的距离分别为h1和h2，我们取最小值为h。
- 当我们合并的时候，我们试图找到这样的一个点对，一个在集合A1，另一个在集合A2,两者的距离都是小于h，然后我们将所有横坐标到xm的差小于h的放入一个集合B。得到了之后我们将b中的点按照y进行排序。
- 第一次排序是在分治之前进行一次，第二次排序是将排序过的点集进行归并。
- 非递归的写法

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <set>
const int N = 200005;
int n;
double ans = 1e20;
struct point {
    double x, y;
    point(double x = 0, double y = 0) : x(x), y(y) {}
};

struct cmp_x {
    bool operator()(const point &a, const point &b) const {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
};

struct cmp_y {
    bool operator()(const point &a, const point &b) const { return a.y < b.y; }
};

inline void upd_ans(const point &a, const point &b) {
    double dist = sqrt(pow((a.x - b.x), 2) + pow((a.y - b.y), 2));
    if (ans > dist) ans = dist;
}

point a[N];
std::multiset<point, cmp_y> s;

int main() {
    //freopen("/home/lyp/code/algorithm/test.in","r",stdin);
    scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%lf%lf", &a[i].x, &a[i].y);
    std::sort(a, a + n, cmp_x());
    for (int i = 0, l = 0; i < n; i++) {
        while (l < i && a[i].x - a[l].x >= ans) s.erase(s.find(a[l++]));
        for (auto it = s.lower_bound(point(a[i].x, a[i].y - ans));
             it != s.end() && it->y - a[i].y < ans; it++)
            upd_ans(*it, a[i]);
        s.insert(a[i]);
    }
    printf("%.4lf", ans);
    fclose(stdout);
```

```
    return 0;
}
```

- 对于求平面最进或者最远的点对的问题，我们可以先对点进行x的从小到大的排序，然后我们可以发现，如果求最近的点，那么他们之间肯定是靠得比较近的，所以，我们可以指定一个常数s，当我们遍历到某一个点的时候，我们找一下这个点与他后面的s个点的距离，同时不断更新最小值对于找最大值的情况，我们需要做的是遍历到每一个点i的时候，我们计算最后s个点与这个点的距离，同时不断更新最大值即可。

```
#include<iostream>
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;
const int s=13;

struct node{
    double x,y;
    friend bool operator <(const node &x1,const node &x2){
        return x1.x<x2.x;
    }
}a[200010];

int main() {
    //freopen("/home/lyp/code/algorithm/test.in","r",stdin);
    double mi=1e20,mx=0;
    int n;
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%lf %lf",&a[i].x,&a[i].y);
    }
    sort(a,a+n);
    for(int i=0;i<n;i++){

        //对后面的s个进行比较
        for(int j=i+1;j<n&&j<i+s;j++){
            mi=min(mi,(a[i].x-a[j].x)*(a[i].x-a[j].x)+(a[i].y-a[j].y)*(a[i].y-a[j].y));
        }

        //从最后面找出s个去最大值
        for(int j=n-1;j>=i&&j>=n-s;j--){
            mx=max(mx,(a[i].x-a[j].x)*(a[i].x-a[j].x)+(a[i].y-a[j].y)*(a[i].y-a[j].y));
        }
    }

    printf("%.4lf %.4lf\n",sqrt(mi), sqrt(mx));
    fclose(stdout);
    return 0;
}
```

最小圆覆盖问题

- 假设找到了圆的前i-1个点的最小覆盖圆，当我们加入第i个点的时候，如果这个点在圆上，我们什么都不做，如果不在圆上，我们就找到新的最小覆盖圆，这个圆肯定要经过第i个点。我们以第i个点为基础，重复以上过程加入第j个点，如果第j个点在圆外，那么最小覆盖圆必经过第j个点。
- 模板求包含n个点的最小的圆的半径和圆心坐标

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

using namespace std;

int n;
double r;

struct point {
    double x, y;
} p[100005], o;

inline double sqr(double x) { return x * x; }

inline double dis(point a, point b) {
    return sqrt(sqr(a.x - b.x) + sqr(a.y - b.y));
}

inline bool cmp(double a, double b) { return fabs(a - b) < 1e-8; }

point geto(point a, point b, point c) {
    double a1, a2, b1, b2, c1, c2;
    point ans;
    a1 = 2 * (b.x - a.x), b1 = 2 * (b.y - a.y),
    c1 = sqr(b.x) - sqr(a.x) + sqr(b.y) - sqr(a.y);
    a2 = 2 * (c.x - a.x), b2 = 2 * (c.y - a.y),
    c2 = sqr(c.x) - sqr(a.x) + sqr(c.y) - sqr(a.y);
    if (cmp(a1, 0)) {
        ans.y = c1 / b1;
        ans.x = (c2 - ans.y * b2) / a2;
    } else if (cmp(b1, 0)) {
        ans.x = c1 / a1;
        ans.y = (c2 - ans.x * a2) / b2;
    } else {
        ans.x = (c2 * b1 - c1 * b2) / (a2 * b1 - a1 * b2);
        ans.y = (c2 * a1 - c1 * a2) / (b2 * a1 - b1 * a2);
    }
    return ans;
}

int main() {
    //freopen("/home/lyp/code/algorithm/test.in","r",stdin);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%lf%lf", &p[i].x, &p[i].y);
    for (int i = 1; i <= n; i++) swap(p[rand() % n + 1], p[rand() % n + 1]);
    o = p[1];
```

```
    for (int i = 1; i <= n; i++) {
        if (dis(o, p[i]) < r || cmp(dis(o, p[i]), r)) continue;
        o.x = (p[i].x + p[1].x) / 2;
        o.y = (p[i].y + p[1].y) / 2;
        r = dis(p[i], p[1]) / 2;
        for (int j = 2; j < i; j++) {
            if (dis(o, p[j]) < r || cmp(dis(o, p[j]), r)) continue;
            o.x = (p[i].x + p[j].x) / 2;
            o.y = (p[i].y + p[j].y) / 2;
            r = dis(p[i], p[j]) / 2;
            for (int k = 1; k < j; k++) {
                if (dis(o, p[k]) < r || cmp(dis(o, p[k]), r)) continue;
                o = geto(p[i], p[j], p[k]);
                r = dis(o, p[i]);
            }
        }
    }
    printf("%.10lf\n%.10lf %.10lf", r, o.x, o.y);
    fclose(stdout);
    return 0;
}
```

数据结构

单调栈

- 单调栈的使用是插入一个数的时候，保持栈的单调性的情况下弹出栈里面最少的元素。
- 当我们往单调栈插入数据的时候，我们可以保持这个栈按照某一单调性，方法就是将这个数和栈顶的元素之间进行比较。

```
typedef long long ll;
struct node{
    int val;
    int index;
}Node[3000010];
stack<node> st;
ll id[3000010];
int main() {
    //freopen("test.in","r",stdin);
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%d",&Node[i].val);
        Node[i].index=i;
        id[i]=0;
    }
```

```
    for(int i=1;i<=n;i++){
        if(st.empty()){  //判断栈是否为空
            st.push(Node[i]);
        }else{
            if(st.top().val>Node[i].val){
                st.push(Node[i]);
            }else{
                while(st.size()&&st.top().val<Node[i].val){
                    id[st.top().index]=i;
                    st.pop();
                }
                st.push(Node[i]);
            }
        }
    }
    for(int i=1;i<=n;i++){
        printf("%d ",id[i]);
    }
    puts("");
    fclose(stdout);
    return 0;
}
```

单调队列

- 单调队列是维护区间内连续的k个值的最值的问题。
- 比如，我们要维护一个区间内的最大值的时候，我们维护的其实就是一个单调递减的队列，所以，每次我们只需要从这个序列里面拿出队首的那个元素就是当前的最值了。而且，对于每次符合队列的元素，我们将这些元素都放到队尾进行一个维护，因为这些元素后面也可能成为对手元素。
- 维护最小值的也是同样的想法，我们维护一个单调递增的队列，每次我们只需要找出队首的那个元素就行了。

```
const int maxn=1000010;
int a[maxn],q[maxn];
int n,k;
void getmx(){
    int head=0,tail=0;
    for(int i=1;i<=k;i++){
        while(head<=tail&&a[q[tail]]<=a[i]) tail--;
        q[++tail]=i;
    }
    for(int i=k;i<=n;i++){
        while(head<=tail&&a[q[tail]]<=a[i]) tail--;
        q[++tail]=i;
        while(q[head]+k<=i) head++;
        printf("%d ",a[q[head]]);
    }
    printf("\n");
}

void getmi(){
    int head=0,tail=0;
    for(int i=1;i<k;i++){
        while(head<=tail&&a[q[tail]]>=a[i]) tail--;
        q[++tail]=i;
    }
```

```
    for(int i=k;i<=n;i++){
        while(head<=tail&&a[q[tail]]>=a[i]) tail--;
        q[++tail]=i;
        while(q[head]+k<=i) head++;
        printf("%d ",a[q[head]]);
    }
    printf("\n");
}
int main() {
    //freopen("test.in","r",stdin);
    scanf("%d%d",&n,&k);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
    }
    getmi();
    getmx();
    fclose(stdout);
    return 0;
}
```

ST表(倍增表)

- ST表可以解决RMD算法。也就是维护区间内某一个属性的值，如gcd，最值等，缺点是不能进行区间的修改。
- ST表是将一个int范围内的数进行二进制划分，我们可以发现，最多也就20或者30个区间,f[i] [j]表示的是区间[i,i+2^j-1]内维护的某一个值。那么，我们发现，如果要求查询某一个区间的时候，我们可以将区间进行划分，我们利用两个区间重叠之后覆盖整个区间的特点尽可能的去覆盖两个区间。
- 首先，我们要进行的是预处理出每个点倍增一定的长度的值，然后就可以很容易地进行状态方程的转移了。

```
inline int read() {
  char c = getchar();
  int x = 0, f = 1;
  while (c < '0' || c > '9') {
    if (c == '-') f = -1;
    c = getchar();
  }
  while (c >= '0' && c <= '9') {
    x = x * 10 + c - '0';
    c = getchar();
  }
  return x * f;
}

const int logn=21;
const int maxn=2000010;
int f[maxn][30],lg[maxn+1];
void pre(){
    lg[1]=0;
    lg[2]=1;
    for(int i=3;i<=maxn;i++){
        lg[i]=lg[i/2]+1;
    }
}

int main() {
    //freopen("test.in","r",stdin);
```

```
    int n=read(),m=read();
    for(int i=1;i<=n;i++){
        f[i][0]=read();
    }
    pre();
    for(int j=1;j<=logn;j++){
        for(int i=1;i+(1<<j)-1<=n;i++){
            f[i][j]=max(f[i][j-1],f[i+(1<<(j-1))][j-1]);
        }
    }
    while(m--){
        int x=read(),y=read();
        int s=lg[y-x+1];
        printf("%d\n", max(f[x][s],f[y-(1<<s)+1][s]));
    }
```

树状数组

- 树状数组的代码想对于线段树来说更加精简，但是树状数组一般支持的是单点修改，在我看来，树状数组其实和倍增的思想很一致，就是将在一个范围内的区间进行二进制划分，然后我们每次对某一个点的修改，我们只需要更新涉及这个位置的相关的区间就行了，这样我们就可以将时间复杂度降低为log级别的了。
- 首先，我们介绍一个lowbit函数，这个函数是找出一个数的二进制表示下的最低位的1。

```
int lowbit(int x){
    return x*(-x);
}
```

- 然后，如果想要对某一个位置的数进行修改的话，那么我们就只需要后进行二进制区间的更新;如果想要对某一个区间范围进行查询的话，我们只需要利用前缀和的思想进行一个处理。往前面进行查找。