# Bsg算法

用于求解a^x≡b(mod p)，a与p互质的最小正数解

方法：由欧拉定理可知：

a^x≡a^(x mod f（p）)（mod p）

可知a^x模p的最小循环节为f（p）

由于f（p）＜p

所以在x∈[0，p]的范围内一定能找到最小正数x

如果暴力枚举，则时间复杂度是o（p）

故可以考虑优化：

假设x=im-j，其中m=sqrt（p）（向上取整）

则i∈[1，m]，j∈[0，m-1]

所求即为a^（im-j）≡b（mod p）

即（a^m）^i≡ba^j（mod p）

先枚举j，把（ba^j，j）存入哈希表，如果存在相同的key值，则用更大的j替代（因为答案是im-j，所以j要尽可能的大答案才能尽可能地小）

然后枚举i，计算（a^m）^i，到哈希表中查找是否有相等的key值，找到第一个即结束，则最小的x=im-j为答案

优化后时间复杂度为o（sqrt（p））

LL bsgs(LL a, LL b, LL p){

a %= p; b %= p;

if(b == 1) return 0; //x=0

LL m = ceil(sqrt(p));

LL t = b;

unordered\_map<int,int> hash;

hash[b] = 0;

for(int j = 1; j < m; j++){

t = t\*a%p; //求b\*a^j

hash[t] = j;

}

LL mi = 1;

for(int i = 1; i <= m; i++)

mi = mi\*a%p; //求a^m

t = 1;

for(int i=1; i <= m; i++){

t = t\*mi%p; //求(a^m)^i

if(hash.count(t))

return i\*m-hash[t];

}

return -1; //无解

}

# 扩展bsg算法

用于求解a^x≡b（mod p）的最小整数解x，a和p不一定互质

思路：当a和p不互质时，想方法让他们互质：

原方程变形为a\*a^（x-1）+py=b

假设d1=gcd（a，p）若d1不是b的因子，则无解

否则方程两边同时除以d1，得a/d1\*a^（x-1）≡b/d1（mod p/d1）

如果a和p/d1仍然不互质就再除

最终原方程变为a^k/D\*a^（x-k）≡b/D（mod p/D）

D为历次di的乘积，k为除的次数

最后由于a与p/D互质，a^k/D≡p/D互质，所以a^k/D就存在逆元了，将它移到方程的右边就变成了普通的bsgs问题了，求解x-k再加上k即为最终答案

LL gcd(LL a, LL b){

return b==0?a:gcd(b,a%b);

}

LL exbsgs(LL a, LL b, LL p){

a %= p; b %= p;

if(b==1||p==1)return 0;//x=0

LL d, k=0, A=1;

while(true){

d = gcd(a,p);

if(d==1) break;

if(b%d) return -1; //无解

k++; b/=d; p/=d;

A = A\*(a/d)%p; //求a^k/D

if(A==b) return k;

}

LL m=ceil(sqrt(p));

LL t = b;

unordered\_map<int,int> hash;

hash[b] = 0;

for(int j = 1; j < m; j++){

t = t\*a%p; //求b\*a^j

hash[t] = j;

}

LL mi = 1;

for(int i = 1; i <= m; i++)

mi = mi\*a%p; //求a^m

t = A;

for(int i=1; i <= m; i++){

t = t\*mi%p; //求(a^m)^i

if(hash.count(t))

return i\*m-hash[t]+k;

}

return -1; //无解

}

# Fft多项式乘法

// 迭代版 1.5s

#include <iostream>

#include <cstring>

#include <algorithm>

#include <cmath>

using namespace std;

const int N=4e6;

const double PI=acos(-1);

struct complex{

double x, y;

complex operator+(const complex& t)const{

return {x+t.x, y+t.y};}

complex operator-(const complex& t)const{

return {x-t.x, y-t.y};}

complex operator\*(const complex& t)const{

return {x\*t.x-y\*t.y, x\*t.y+y\*t.x};}

}A[N], B[N];

int R[N];

void FFT(complex A[],int n,int op){

for(int i=0; i<n; ++i)

R[i] = R[i/2]/2 + ((i&1)?n/2:0);

for(int i=0; i<n; ++i)

if(i<R[i]) swap(A[i],A[R[i]]);

for(int i=2; i<=n; i<<=1){

complex w1({cos(2\*PI/i),sin(2\*PI/i)\*op});

for(int j=0; j<n; j+=i){

complex wk({1,0});

for(int k=j; k<j+i/2; ++k){

complex x=A[k], y=A[k+i/2]\*wk;

A[k]=x+y; A[k+i/2]=x-y;

wk=wk\*w1;

}

}

}

}

int main(){

int n,m;

scanf("%d%d", &n, &m);

for(int i=0; i<=n; i++)scanf("%lf", &A[i].x);

for(int i=0; i<=m; i++)scanf("%lf", &B[i].x);

for(m=n+m,n=1;n<=m;n<<=1);

FFT(A,n,1); FFT(B,n,1);//系数转值

for(int i=0;i<n;++i)A[i]=A[i]\*B[i];

FFT(A,n,-1);//值转系数

for(int i=0;i<=m;++i)

printf("%d ",(int)(A[i].x/n+0.5));

}

# Gcd的性质

gcd(a,b,c,...,z)=gcd(a,b−a,c−b,...,z−y)

d(ij)=Σ(x|i)Σ(y|j)[gcd(x,y)=1]，d(x)表示x的因子的个数

贝祖定理

a1x1+a2x2+……+anxn=gcd（a1，a2，……，an）一定有解

# 常见积性函数

积性函数：欧拉函数、莫比乌斯函数、d函数（约数和函数）（d(n)=Σ(i|n)i）

完全积性函数：ε(n)=[n=1]、1(n)=1、id(n)=n

常见卷积关系：

Σ(d|n)μ(d)=[n=1]⇔μ\*1=ε

Σ(d|n)f(d)=n⇔f\*1=id

Σ(d|n)μ(d)n/d=f(n)⇔μ\*id=f

f\*ε=f

f\*1≠f

μ和f分别为莫比乌斯函数和欧拉函数

狄利克雷卷积

卷积：ΣaiΣbj

定义f、g为两个积性函数卷积记为\*

则（f\*g）（n）=Σ（d|n）f（d）g（n/d）=Σ（d|n）f（n/d）g（d）

\*运算满足交换律、结合律和分配律

常用函数：

ε(n)=[n=1]

1(n)=1

id(n)=n

# 常见卷积关系：

Σ(d|n)μ(d)=[n=1]⇔μ\*1=ε

Σ(d|n)f(d)=n⇔f\*1=id

Σ(d|n)μ(d)n/d=f(n)⇔μ\*id=f

f\*ε=f

f\*1≠f

μ和f分别为莫比乌斯函数和欧拉函数

# 杜教筛

用于解决

s(n)=Σ(i=1~n)f(i)，f为积性函数的问题

解法：构造积性函数g，使得f\*g的前缀和容易计算\*为卷积

得出：

g(1)s(n)=Σ(i=1~n)(f\*g)(i)-Σ(i=2~n)g(i)s(n/i)

证明：

构造积性函数g，则：

Σ(i=1~n)(f\*g)(i)=Σ(i=1~n)Σ(d|i)f(i/d)g(d)=Σ(i=1~n)Σ(d=1~n)(d|i)f(i/d)g(d)=Σ(d=1~n)Σ(i=1~n)(d|i)f(i/d)g(d)=Σ(d=1~n)Σ(id=1~n)(d|id)f(id/d)g(d)=Σ(d=1~n)Σ(id=1~n)f(i)g(d)=Σ(d=1~n)Σ(i=1~n/d)f(i)g(d)

=Σ(d=1~n)g(d)Σ(i=1~n/d)f(i)=Σ(d=1~n)g(d)s(n/d)=Σ(d=2~n)g(d)s(n/d)+g(1)s(n)

故Σ(i=1~n)(f\*g)(i)=Σ(d=2~n)g(d)s(n/d)+g(1)s(n)

故Σ(i=1~n)(f\*g)(i)-Σ(d=2~n)g(d)s(n/d)=g(1)s(n)

即g(1)s(n)=Σ(i=1~n)(f\*g)(i)-Σ(d=2~n)g(d)s(n/d)

即g(1)s(n)=Σ(i=1~n)(f\*g)(i)-Σ(i=2~n)g(i)s(n/i)

当s函数代表欧拉函数f前缀和时：

构造g函数为1函数（即1(n)=1）

由f\*1=id（即id(n)=n）可知

g(1)s(n)=s(n)

Σ(i=1~n)(f\*g)(i)-Σ(i=2~n)g(i)s(n/i)=Σ(i=1~n)id(i)-Σ(i=2~n)s(n/i)

故s(n)=Σ(i=1~n)id(i)-Σ(i=2~n)s(n/i)=(1+n)\*n/2-Σ(i=2~n)s(n/i)

Σ(i=2~n)s(n/i)可以使用记忆化搜索+分块处理

当s函数代表莫比乌斯函数μ前缀和时：

构造g函数为1函数（即1(n)=1）

由μ\*1=ε（即ε(n)=[n=1]）可知

g(1)s(n)=s(n)

Σ(i=1~n)(f\*g)(i)-Σ(i=2~n)g(i)s(n/i)=Σ(i=1~n)ε(i)-Σ(i=2~n)s(n/i)

故s(n)=Σ(i=1~n)ε(i)-Σ(i=2~n)s(n/i)=1-Σ(i=2~n)s(n/i)

Σ(i=2~n)s(n/i)可以使用记忆化搜索+分块处理

杜教筛求欧拉函数前缀和：（使用euler\_sum函数前先调用work\_prime函数）

#define ll long long

#define NUMBER 5000000

#include<vector>

#include<unordered\_map>

using namespace std;

bool prime\_check[NUMBER + 1]{};

ll euler[NUMBER + 1]{};

vector<ll>prime;

unordered\_map<ll, ll>act\_euler;

void work\_prime() {

act\_euler[1] = 1;

for (ll q = 2; q <= NUMBER; ++q) {

if (!prime\_check[q]) {

prime.push\_back(q);

euler[q] = q - 1;

}

for (ll i = 0; i < prime.size() && prime[i] \* q <= NUMBER; ++i) {

prime\_check[prime[i] \* q] = true;

if (q % prime[i]) {

euler[prime[i] \* q] = euler[prime[i]] \* euler[q];

}

else {

euler[prime[i] \* q] = euler[q] \* prime[i];

break;

}

}

}

euler[1] = 1;

for (ll q = 1; q <= NUMBER; ++q) {

euler[q] += euler[q - 1];

}

}

ll euler\_sum(ll n) {

if (act\_euler.find(n) != act\_euler.end()) {

return act\_euler[n];

}

if (n <= NUMBER) {

return euler[n];

}

ll answer = (1 + n) \* n / 2;

ll done = 1;

while (true) {

done++;

if (done > n) {

break;

}

ll val = n / done;

ll en = n / val;

answer -= euler\_sum(val) \* (en - done + 1);

done = en;

}

return act\_euler[n] = answer;

}

杜教筛求莫比乌斯函数前缀和：（使用mu\_sum函数前先调用get\_mu函数）

ll p[NUMBER], vis[NUMBER], cnt;

ll mu[NUMBER];

void get\_mu(ll n) {//筛法求莫比乌斯函数

act\_mu[1] = 1;

mu[1] = 1;

for (int i = 2; i <= n; i++) {

if (!vis[i]) {

p[++cnt] = i;

mu[i] = -1;

}

for (int j = 1; i \* p[j] <= n; j++) {

int m = i \* p[j];

vis[m] = 1;

if (i % p[j] == 0) {

mu[m] = 0;

break;

}

else

mu[m] = -mu[i];

}

}

for (ll q = 1; q <= NUMBER; ++q) {

mu[q] += mu[q - 1];

}

}

ll mu\_sum(ll n) {

if (act\_mu.find(n) != act\_mu.end()) {

return act\_mu[n];

}

if (n <= NUMBER) {

return mu[n];

}

ll answer = 1;

ll done = 1;

while (true) {

done++;

if (done > n) {

break;

}

ll val = n / done;

ll en = n / val;

answer -= mu\_sum(val) \* (en - done + 1);

done = en;

}

return act\_mu[n] = answer;

}

# 费马小定理

若p为质数且gcd（a，p）=1，则a^（p-1）≡1（mod p）

常用于求逆元求模：

若p为质数且gcd（a，p）=1，则a^（-1）mod p=a^（p-2）mod p

# 扩展欧几里得算法

该算法用于解决以下问题：

ax+by=gcd（a，b）的整数解

由欧几里得算法可得：

gcd（a，b）=gcd（b，a%b）

则存在该方程的解：

bx1+（a%b）y1=gcd（b，a%b）=gcd（a，b）=ax+by

化简后可得a%b=a-b[a/b]（[]为向下取整）

bx1+（a-b[a/b]）y1=ax+by

化简可得：

a（y1-x）+b（x1-y-y1[a/b]）=0

故存在等式

x=y1

y=x1-y1[a/b]

同时，递归结果：当b=0时x=1，y=0即为解，此时通过上述等式可以回溯出最初ax+by=gcd（a，b）的解

long long exgcd(long long a, long long b, long long& x, long long& y) {

if (!b) {

x = 1;

y = 0;

return a;

}

long long gcd = exgcd(b, a % b, y, x);

y -= a / b \* x;

return gcd;

}

其返回值是a、b的最大公因数，x，y最后储存该方程的整数解

归结：

若ax+by=c有解x=x0，y=y0，则其通解可表示为：

x=x0-bt/gcd（a，b），y=y0+at/gcd（a，b）

t=（0，1，-1，2，-2，3，-3，……）

故可用ax+by=gcd（a，b）找到解

然后gcd（a，b）\*p=c

则原方程：

ax+by=gcd（a，b）\*p

ax/p+by/p=gcd（a，b）

卢卡斯定理

对于质数p有c（n，m）mod p=c（n/p，m/p）c（n mod p，m mod p）mod p

实现

long long Lucas(long long n, long long m, long long p) {

if (m == 0) return 1;

return (C(n % p, m % p, p) \* Lucas(n / p, m / p, p)) % p;

}

# 莫比乌斯

记作μ（n）：

若n=1：μ（n）=1

若n的某个质因子指数大于等于2：μ（n）=0

否则记s为n分解之后本质不同的质因子的数量，μ（n）=（-1）^s

性质：

Σ（n的所有因子d，即d|n）μ（d）=[n=1]

Σ（同上）μ（d）/d=f（n）/n（f为欧拉函数）

Σ（同上）μ（n/d）/d=f（n）（f为欧拉函数）

筛法求莫比乌斯函数：

const int N = 1000010;

int p[N], vis[N], cnt;

int mu[N];

void get\_mu(int n){//筛法求莫比乌斯函数

mu[1] = 1;

for(int i=2; i<=n; i++){

if(!vis[i]){

p[++cnt] = i;

mu[i] = -1;

}

for(int j=1; i\*p[j]<=n; j++){

int m = i\*p[j];

vis[m] = 1;

if(i%p[j] == 0){

mu[m] = 0;

break;

}

else

mu[m] = -mu[i];

}

}

}

f(n)=Σ(d|n)g(d)等价于g(n)=Σ(d|n)μ(d)f(n/d)

f(n),g(n)均为积性函数

μ(n)为莫比乌斯函数

# 逆元

若ax≡1（mod b），则x为a mod b的逆元

扩展欧几里得求逆元：

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 -= a / b \* x;

}

快速幂求逆元：

ll pi = 1e9 + 7;

ll qpow(ll a, ll b) {

ll res = 1;

while (b) {

if (b & 1) {

res = res \* a % pi;

}

a = a \* a % pi;

b >>= 1;

}

return res;

}

ll f(ll x) {

return qpow(x, pi - 2);

}

线性求逆元：（求1~n的每一个数在模p条件下的逆元）

inv[1] = 1;

for (int i = 2; i <= n; ++i) {

inv[i] = (long long)(p - p / i) \* inv[p % i] % p;

}

线性求任意n个数的逆元：（略）

# 欧拉定理

性质：欧拉函数：f（x）：小于等于x并且与x互质的数的个数（f（1）=1）

性质：如果gcd（a，b）=1，那么f（a）\*f（b）=f（a\*b）

若x为奇数，那么f（2n）=f（n）

以下f（x）表示欧拉函数

欧拉定理：

若gcd（a，b）=1，则a^f（b）≡1（mod b）

扩展欧拉定理：

若gcd（a，p）=1，则a^b≡a^[b mod f（p）]（mod p）

若gcd（a，p）≠1，则

若b<f（p），则a^b≡a^b（mod p）

若b≥f（p），则a^b≡a^[b mod f（p）+f（p）]（mod p）

# 欧拉函数

f（n）：小于等于n的与n互质的正整数的个数

性质：

如果n为质数，f（n）=n-1

如果p、q均为质数，则f（p\*q）=f（p）\*f（q）=（p-1）\*（q-1）

如果p是质数，则f（p^k）=p^k-p^（k-1）

f（n）=n（1-1/p1）（1-1/p2）……（1-1/pn），其中p1、p2、……、pn为n的质因子

若a为质数，b是a的倍数，则f（a\*b）=f（b）\*a

若a、b互质，则f（a\*b）=f（a）\*f（b）

若a为奇数，则f（2a）=f（a）

n=Σ（d|n）f（d）（n的所有因子的欧拉函数和数值上等于n）

若a，b不全为0，则f（ab）f（gcd（a，b））=f（a）f（b）gcd（a，b）

模板：欧拉筛求欧拉函数

#include<vector>

#define ll long long

#define NUMBER 100000

using namespace std;

bool prime\_check[NUMBER + 1]{};

ll euler[NUMBER + 1]{};

vector<ll>prime;

void work\_prime() {

for (ll q = 2; q <= NUMBER; ++q) {

if (!prime\_check[q]) {

prime.push\_back(q);

euler[q] = q - 1;

}

for (ll i = 0; i < prime.size() && prime[i] \* q <= NUMBER; ++i) {

prime\_check[prime[i] \* q] = true;

if (q % prime[i]) {

euler[prime[i] \* q] = euler[prime[i]] \* euler[q];

}

else {

euler[prime[i] \* q] = euler[q] \* prime[i];

break;

}

}

}

euler[1] = 1;

}

# 威尔逊定理

对于素数p有（p-1）！≡-1（mod p）

它是p为质数的充要条件

推论：若p为大于4的合数，则（p-1）！≡0（mod p）

# 中国剩余定理

求解一元线性同余方程组：

x≡a1（mod n1）

x≡a2（mod n2）

……

x≡ak（mod nk）

（n1、n2、……、nk两两互质）

步骤：

1.计算n=n1n2……nk

2.对于第i个方程：

a.计算mi=n/ni

b.计算mi在模ni意义下的逆元ti=（mi）^（-1）

c.计算ci=miti（不对ni求模）

3.该方程在模n意义下的唯一解为x=Σaici

实现：

LL CRT(int k, LL\* a, LL\* r) {

LL n = 1, ans = 0;

for (int i = 1; i <= k; i++) n = n \* r[i];

for (int i = 1; i <= k; i++) {

LL m = n / r[i], b, y;

exgcd(m, r[i], b, y); // b \* m mod r[i] = 1

ans = (ans + a[i] \* m \* b % n) % n;

}

return (ans % n + n) % n;

}

# 扩展中国剩余定理：

假设两个方程分别为

x≡a1（mod n1）

x≡a2（mod n2）

将它们转换为不定方程

x=n1p+a1

x=n2q+a2

则n1p-n2q=a2-a1

用扩展欧几里得定理解出一组可行解（p，q）

则原来的方程组转换为

x≡b（mod M）

其中b=n1p+a1

M=lcm（m1，m2）

用上述方法合并所有的同余方程组即可

# 堆优化Dijkstra

#include <cstring>

#include <iostream>

#include <algorithm>

#include <queue>

using namespace std;

const int N=100010;

int n,m,s,a,b,c;

struct edge{int v,w;};

vector<edge> e[N];

int d[N],vis[N];

void dijkstra(int s){

memset(d,0x3f,sizeof d); d[s]=0;

priority\_queue<pair<int,int>> q;

q.push({0,s});

while(q.size()){

auto t=q.top(); q.pop();

int u=t.second;

if(vis[u])continue; //再出队跳过

vis[u]=1; //标记u已出队

for(auto ed : e[u]){

int v=ed.v, w=ed.w;

if(d[v]>d[u]+w){

d[v]=d[u]+w;

q.push({-d[v],v}); //大根堆

}

}

}

}

int main(){

cin>>n>>m>>s;

for(int i=0; i<m; i++)

cin>>a>>b>>c, e[a].push\_back({b,c});

dijkstra(s);

for(int i=1;i<=n;i++) printf("%d ",d[i]);

}

# Tarjan

割点：无向图中去除该点以及相关边后连通块数量增加的点

利用tarjan算法

若某个点不是搜索树的根节点，则若存在子节点的low值大于等于该点的dfn值则该点为割点

若该点为搜索树根节点，则要求至少存在上述子节点两个成立

强连通分量：有向图中可以互相到达的点对或点集

可以用tarjan求最大环：

示例：（scc值相同的即可互相到达）

#include<iostream>

#include<vector>

#include<stack>

#include<map>

#define ll long long

using namespace std;

vector<vector<ll>>link;

vector<ll>low;

vector<ll>dfn;

vector<ll>scc;

vector<bool>cut;

vector<bool>check;

stack<ll>in;

ll num = 0;

ll n, m;

ll root;

void tarjan(ll pos) {

ll child = 0;

low[pos] = dfn[pos] = ++num;

in.push(pos);

for (auto point : link[pos]) {

if (dfn[point]) {

if (!check[point]) {

low[pos] = min(low[pos], dfn[point]);

}

}

else {

tarjan(point);

if(low[point] >= dfn[pos]){

child++;

}

low[pos] = min(low[pos], low[point]);

}

}

if(root == pos){

cut[root] = bool(child >= 2);

}

else{

cut[root] = bool(child >= 1);

}

if (low[pos] == dfn[pos]) {

scc[pos] = pos;

while (in.top() != pos) {

scc[in.top()] = pos;

check[in.top()] = true;

in.pop();

}

check[pos] = true;

in.pop();

}

}

int main() {

cin >> n >> m;

link.resize(n + 1);

low.resize(n + 1);

dfn.resize(n + 1);

scc.resize(n + 1);

check.resize(n + 1);

for (ll q = 1; q <= m; ++q) {

ll a, b;

cin >> a >> b;

link[a].push\_back(b);

}

for (ll q = 1; q <= n; ++q) {

if (!dfn[q]) {

tarjan(q);

}

}

}

强连通分量：有向图中可以互相到达的点对或点集

可以用tarjan求最大环：

示例：（scc值相同的即可互相到达）

#include<iostream>

#include<vector>

#include<stack>

#include<map>

#define ll long long

using namespace std;

vector<vector<ll>>link;

vector<ll>low;

vector<ll>dfn;

vector<ll>scc;

vector<bool>check;

stack<ll>in;

ll num = 0;

ll n, m;

void tarjan(ll pos) {

low[pos] = dfn[pos] = ++num;

in.push(pos);

for (auto point : link[pos]) {

if (dfn[point]) {

if (!check[point]) {

low[pos] = min(low[pos], dfn[point]);

}

}

else {

tarjan(point);

low[pos] = min(low[pos], low[point]);

}

}

if (low[pos] == dfn[pos]) {

scc[pos] = pos;

while (in.top() != pos) {

scc[in.top()] = pos;

check[in.top()] = true;

in.pop();

}

check[pos] = true;

in.pop();

}

}

int main() {

cin >> n >> m;

link.resize(n + 1);

low.resize(n + 1);

dfn.resize(n + 1);

scc.resize(n + 1);

check.resize(n + 1);

for (ll q = 1; q <= m; ++q) {

ll a, b;

cin >> a >> b;

link[a].push\_back(b);

}

for (ll q = 1; q <= n; ++q) {

if (!dfn[q]) {

tarjan(q);

}

}

}

void floyd(){

for(int k=1; k<=n; k++)

for(int i=1; i<=n; i++)

for(int j=1; j<=n; j++)

d[i][j]=min(d[i][j],d[i][k]+d[k][j]);

}

# Floyd

floyd可用于求解最小环问题：

for(int k=1; k<=n; k++){

for(int i=1; i<k; i++)

for(int j=i+1; j<k; j++)

ans=min(ans,d[i][j]+w[j][k]+w[k][i]);

for(int i=1; i<=n; i++)

for(int j=1; j<=n; j++)

d[i][j]=min(d[i][j],d[i][k]+d[k][j]);

在更新前先记录

用floyd算法更新的时间复杂度是n^3可以使用运行n次单源dijkstra来代替，复杂度为n^2logn

# 错排

一个长度为n的排列所有下标均不等于对应值的个数称为错排数，记为dn

dn=（n-1）[d（n-1）+d（n-2）]（先暂定下标为n时值为n，前n-1个是错排，任选一个交换：（n-1）d（n-1），有一个下标与值相同，选定它进行交换：（n-1）d（n-2））

# 第二类斯特林数

与第一类斯特林数区分，记作{n,m}

用n个元素构成m个非空集合的方案数（m个集合之间无区分）成为第二类斯特林数，记为s（n，m）

递推式：s（n，m）=s（n-1，m-1）+s（n-1，m）m

s（n，n）=1

s（n，0）=0

# 第一类斯特林数

与第二类斯特林数区分，记为[n,m]

用n个元素构成m个非空圆排列的方案数（m个非空圆排列之间无区分）成为第一类斯特林数，记为s（n，m）

递推式：s（n，m）=s（n-1，m-1）+（n-1）s（n-1，m）

递推式含义：将它视为已经放入了n-1人，接下来插入第n个人：情况一：已经有m-1个非空圆排列，第n个人自己形成一个新的圆排列：s（n-1，m-1），情况二：已经有m个圆排列，第n个人选择前n-1个人中的一人的左边插入：（n-1）s（n-1，m），求和即为递推式

特殊：

s（0，0）=1

s（n，0）=0

s（n，1）=q（n，n）=（n-1）！

s（n，n）=1

1/（1-x）^n=Σ（i=0~∞）c（n+i-1，i）x^i

扩展指数为负数

c（i，-n）=（-n）\*（-n-1）……（-n-i+1）/i！

下降幂：x^(k|)=x(x-1)……(x-k+1)=c(x,k)k!

上升幂：x^(|k)=x(x+1)……(x+k-1)=c(x+k-1,k)k!

x^n=Σ(k=0~n){n,k}x^(k|)

x^(n|)=Σ(k=0~n)[n,k](-1)^(n-k)x^k

x^n=Σ(k=0~n){n,k}(-1)^(n-k)x^(|k)

x^(|n)=Σ(k=0~n)[n,k]x^k

c(n,k)c(k,i)=c(n,i)c(n-i,k-i)（让某多项式可以移出和式，形成某个多项式）

# 圆排列

从n个元素中选出m个元素排列成一个环形称为圆排列，记为q（n，m）

q（n，m）=c（n，m）q（m，m）

显然q（m，m）=a（m，m）/m=（m-1）！（线排列在某个地方断开连接成环形成的圆排列均等价）

故q（n，m）=c（n，m）\*a（m，m）/m=n！/[m！（n-m）！]\*（m-1）！=n！/[m（n-m）！]

# 逆拓扑排序

给定最小字典拓扑序a生成可行有向图：对与每个j，找到最大的i满足i＜j且ai＞aj，连接一条ai->aj的边（即找对于每一个aj，找到下标差值最小的逆序对）

给定最大字典拓扑序a生成可行有向图：对与每个j，找到最大的i满足i＜j且ai＜aj，连接一条ai->aj的边（即找对于每一个aj，找到下标差值最小的顺序对）

\_\_int128

#define lll \_\_int128

\_\_int128 read()

{

\_\_int128 x = 0; int f = 1; char ch = getchar();

while (!isdigit(ch)) { if (ch == '-')f = -1; ch = getchar(); }

while (isdigit(ch)) { x = x \* 10 + ch - '0'; ch = getchar(); }

return x \* f;

}

void print(\_\_int128 x)

{

if (x < 0) { putchar('-'); x = -x; }

if (x > 9)print(x / 10);

putchar(x % 10 + '0');

}

Kmp

给定两个字符串a、b找出a中是否有b出现。若有则输出匹配的第一个字符的位置（从0开始计数），否则输出-1

分析：生成next数组（可跳过匹配位数），然后计算匹配（线性时间）

#include<iostream>

using namespace std;

void create(string b,int next[]){

next[0]=0;

int left=0,right=1;

while(right<b.length()){

if(b[left]==b[right]){

next[right++]=next[left++]+1;

}

else{

if(left){

left=next[left-1];

}

else{

next[right++]=0;

}

}

}

}

int kmp(string a,string b){

int\*next=new int[b.length()];

create(b,next);

int i=0,j=0;

while(i<a.length()){

if(a[i]==b[j]){

if(j==b.length()-1){

return i-b.length()+1;

}

else{

i++,j++;

}

}

else{

if(j){

j=next[j-1];

}

else{

++i;

}

}

}

return -1;

}

int main(){

string a,b;

cin>>a>>b;

cout<<kmp(a,b);

return 0;

}

树的重心

定义：

如果在树中选择某个节点并删除，这棵树将分为若干棵子树，统计子树节点数并记录最大值。取遍树上所有节点，使此最大值取到最小的节点被称为整个树的重心。

（这里以及下文中的「子树」若无特殊说明都是指无根树的子树，即包括「向上」的那棵子树，并且不包括整棵树自身。）

1、树上所有的点到树的重心的距离之和是最短的，如果有多个重心，那么总距离相等。

2、插入或删除一个点，树的重心的位置最多移动一个单位。

3、若添加一条边连接2棵树，那么新树的重心一定在原来两棵树的重心的路径上。

4、树的重心如果不唯一，则至多有两个，且这两个重心相邻。

5、以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。

6、树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。

7、把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。

8、在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

所有偶数位数的回文数都是11的倍数

（回文素数必定是奇数位）

不进准则：

1、2、3平方不进位，故可以构造平方数：

144、169、961、1004004、10609、9000006000001……

鸽巢原理

应用：对于任意正整数n必能找到一个数，它前部分由1组成，后部分由0组成，且一定为n的倍数：

证明：对于n，取出1、11、111、1111、……n+1个数，其中必有数在模n的情况下同余，取出这两个数做差，得到的数必定为n的倍数。

字符串哈希

先确定每个字符的哈希对应，以26个小写字符为例：

a~z分别对应1~26

再确定进制base

假设存在字符串ans

可知若F[pos]表示字符串ans的前pos位构成的字符串的哈希值，f(pos)表示第pos位的字符的哈希值，则存在以下转化：

F[pos+1]=F[pos]\*base+f(pos+1)

若g[l,r]表示字符串ans第l位到第r位所组成的子串的哈希值，则存在如下转化：

g[l,r]=F[r]-F[l-1]\*base^(r-l+1)

当字符串位数很大时极有可能溢出，故需要引入模运算，故上述公式可变形：

F[pos+1]=(F[pos]\*base+f(pos+1))%pi

g[l,r]=((F[r]-F[l-1]\*base^(r-l+1))%pi+pi)%pi

pi为一个极大的素数

当引入模运算后，不同的字符串的哈希值也有可能存在相同的情况，为了降低这种可能性，可以使用双哈希。

即使用两组base、pi值进行哈希运算。

但引入该模运算后哈希值不能直接用来比较大小，只能判断是否相等，可以结合二分进行字符串判断

字符串哈希回文

判断某个字符串的子是否为回文串时可以利用哈希表来解决：

对字符串进行哈希处理

将字符串反转，再次进行哈希值处理

若某子串为回文串，则其正反哈希值必定相等

注意：此处哈希值的求模运算不能使用质数，使用质数求模会导致同一回文子串的正反哈希值不同，要使用整数进行处理（如1e10、1e12）

进制采用十进制即可

重链剖分：

概念（结合线段树解决树上链维护问题）

重儿子：某个节点下拥有最多节点的子树的根节点

轻儿子：某个节点除重儿子以外的所有儿子节点

重链：从某个轻儿子出发，一直往重儿子延伸的链

轻链：除重链以外的所有链

引理：所有节点的父节点一定在一条重链上（利用两次dfs记录，然后建立树链剖分）

长链剖分：

与重链剖分相似，但其重儿子的定义为某个节点的子节点中拥有最长链的节点