# 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

# 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

# 莫比乌斯函数

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];

}

}

}

# 莫比乌斯反演

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

# 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）

# 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;

}

# 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

# 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）

# 异或性质

a-b≤a^b≤a+b

应用：给定p和m，问多少个非负整数g满足g^(p-1)≡1(mod p)

解法：设g^(p-1)=kp+1，则g的数量等于k的数量，k为非负整数

即求(kp+1)^(p-1)≤m，k≥0关于k的解

应用上述性质可知：

当k≤⌊m/p⌋-1时上式恒成立

当k≥⌈m/p⌉+1时上式恒不成立

故只需要查看[⌊m/p⌋,⌈m/p⌉]区间内成立的数量即可，将所得加上⌊m/p⌋即为答案

# 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

# Spfa判负环

//BFS\_spfa 判负环 530ms

#include <iostream>

#include <cstring>

#include <algorithm>

#include <queue>

using namespace std;

const int inf=0x3f3f3f3f;

const int N=2010,M=6010;

int n,m;

int to[M],ne[M],w[M],h[N],tot;

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

void add(int a,int b,int c){

to[++tot]=b;w[tot]=c;

ne[tot]=h[a];h[a]=tot;

}

bool spfa(){ //判负环

memset(d,0x3f,sizeof d);

memset(vis,0,sizeof vis);

memset(cnt,0,sizeof cnt);

queue<int>q;

q.push(1); vis[1]=1; d[1]=0;

while(q.size()){

int u=q.front();q.pop();vis[u]=0;

for(int i=h[u];i;i=ne[i]){

int v=to[i];

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

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

cnt[v]=cnt[u]+1;

if(cnt[v]>=n)return 1;//判边数

if(!vis[v])q.push(v),vis[v]=1;

}

}

}

return 0;

}

int main(){

int T; scanf("%d",&T);

while(T--){

tot=0; memset(h,0,sizeof(h));

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

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

int u,v,w;

scanf("%d%d%d",&u,&v,&w);

add(u,v,w);

if(w>=0)add(v,u,w);;

}

puts(spfa()?"YES":"NO");

}

return 0;

}

# Tarjan边双连通分量

无向图中极大的不包含割边的连通块被称为边双连通分量

# Tarjan割边

对于无向图

当搜索树上存在x的一个子节点y，满足low[y]>dfn[x]，则(x，y)这条边就是割边

# Tarjan强连通分量

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);

}

}

}

# Tarjan求割点

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++;

if (pos != root || child > 1) {

cut[pos] = true;

}

}

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 (root = 1; root <= n; ++root) {

if (!dfn[root]) {

tarjan(root);

}

}

}

# 差分约束

求解问题1：

对于形如xa-xb≤ck的不等式组求解是否存在

解法：将所有的节点b向节点a连接一条长度为ck的单向边，跑spfa判负环，如果存在负环则无解

求解问题2：

在满足有解的情况下判断某个变量的最值

解法：建立超级源点x0，向所有点连边，边权均为0，然后跑最长路（求解最小值）或最短路（求解最大值）

# 单源dijkstra

//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]);

}

# 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）

# 欧拉路

欧拉路：即一笔画路径（不重复走完所有边）。

连通有向图：出度+1，入度-1，所有点的度数均为0或者有一个为+1（起点），一个为-1（终点）其它均为0时存在。若度数均为0，则以任意点作为起点或终点都能找到欧拉路。

连通无向图：图中仅有0个或2个奇数度数节点，若存在奇数节点，则这两个节点一个是起点，一个是终点，若不存在，以任意点作为起点或终点都能找到欧拉路。

# 上下界网络流

每一条边都有容量上限和容量下限，问是否存在分配方案：

无源汇：即源点和汇点均未定：

将原来的网络图拆分为两个网络：低保网络和增量网络

低保网络：连边情况和原网络相同，每条边的容量上限改为原来的容量下限

增量网络：连边情况和原网络相同，每条边的容量上限改为原来的容量上限减去容量下限

然后对于增量网络，虚构出新的源点s和汇点t

对于每个点在增量网络中进行操作：如果该点在低保网络中实际上是流入流量，则将s向该点连边，容量大小即为流入流量；反之则将该点向t连边，容量大小即为流出流量，若既不流出流量也不流出流量，则不进行操作

然后对于修改过的增量网络执行一次最大流算法，如果与s、t连接的边的容量均为满流则存在方案

有源汇：即源点和汇点均给定：

将网络中的汇点t向s连接一条容量上限为无穷大，容量下限为0的边，将该网络看作是原网络跑一遍无源汇上下界网络流算法即可

# 网络流最小割

对于一个网络，将其所有点划分为两个集合S，T其中源点s属于S，汇点t属于T，则称其为割

即将某些边删除，这些边删除之后网络被割成了两个块

割的容量表示所有从集合S流向T的边的容量之和

最小割：割容量最小的情况（往往方案不唯一）

最大流最小割定理：最小割的容量等于该网络的最大流

问题一：求最小割的容量：用dinic求最大流即可

问题二：求最小割的划分：跑一遍dinic，然后用残留网从源点s出发跑一遍dfs，将可到达的点划分为S集合，将不可到达的点划分为T集合即可

问题三：求最小割需要删除的边的最小数量：跑一遍dinic，然后将正边中剩余容量为0的边设定为1，其余正边设定为无穷大，全反边设定为0，再跑一遍dinic返回的值即为答案

# 最大流求二分图最大匹配

二分图的最大匹配：无向图的匹配是一个边的集合，其中任意两条边都没有共同的端点。图的最大匹配是一个包含了最多条

边的匹配。图的匹配数是这张图的最大匹配所包含的边的数量。

求法：将原图构建成u、v两个集合（即二分图），然后建立虚拟节点s、t，s向u所有点单向连边，u所有点向t单向连边，若原图为无向图，改成u向v单向连边，然后将所有边容量设置为1，跑一遍最大流即得出答案

补充：加一条边让平衡二分图（两个集合大小均为n，其中1~n为与s直接连边的点，n+1~2n是与t直接连边的点）最大匹配增加方案数：跑一遍最大流，然后在残余网络上跑下列操作：

从源点s出发，执行dfs或bfs（走剩余容量为1的边），计算得出可到达的介于1~n编号的点的数量a，注意：可以通过其它点中转

从汇点t出发，执行dfs或bfs（走剩余容量为0的边），计算得出可到达的介于n+1~2n编号的点的数量b，注意：可以通过其它点中转

答案即为a\*b

# 网络流整理

ek最大流：

#include<iostream>

#include<vector>

#include<queue>

#define ll long long

using namespace std;

struct edge {

ll en, ne, c;

edge(ll \_en,ll \_ne,ll \_c):en(\_en),ne(\_ne),c(\_c){}

};

vector<edge>e;

vector<ll>head;

vector<ll>mf;

vector<ll>pre;

ll s, t;

ll n, m;

ll inf = 1e15;

void add(ll beg, ll en, ll c) {

e.push\_back(edge(en, head[beg], c));

head[beg] = e.size() - 1;

}

bool bfs() {

pre.clear();

pre.resize(n + 1);

mf.clear();

mf.resize(n + 1);

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

mf[q] = inf;

}

queue<ll>have;

vector<bool>check(n + 1);

have.push(s);

while (!have.empty()) {

check[have.front()] = true;

for (ll point = head[have.front()]; point; point = e[point].ne) {

if (!check[e[point].en] && e[point].c) {

pre[e[point].en] = point;

check[e[point].en] = true;

mf[e[point].en] = min(mf[have.front()], e[point].c);

have.push(e[point].en);

if (e[point].en == t) {

return true;

}

}

}

have.pop();

}

return false;

}

ll ek() {

ll answer = 0;

while (bfs()) {

answer += mf[t];

for (ll point = t; point != s; point = e[pre[point] ^ 1].en) {

e[pre[point]].c -= mf[t];

e[pre[point] ^ 1].c += mf[t];

}

}

return answer;

}

int main() {

cin >> n >> m >> s >> t;

head.resize(n + 1);

add(0, 0, 0);

add(0, 0, 0);

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

ll u, v, w;

cin >> u >> v >> w;

add(u, v, w);

add(v, u, 0);

}

cout << ek() << endl;

}

dinic最大流：

#include<iostream>

#include<vector>

#include<queue>

#define ll long long

using namespace std;

struct edge {

ll en, ne, c;

edge(ll \_en,ll \_ne,ll \_c):en(\_en),ne(\_ne),c(\_c){}

};

vector<ll>head;

vector<edge>e;

vector<ll>d;

vector<ll>cur;

ll s, t;

ll n, m;

ll inf = 1e15;

void add(ll beg, ll en, ll c) {

e.push\_back(edge(en, head[beg], c));

head[beg] = e.size() - 1;

}

bool bfs() {

d.clear();

d.resize(n + 1);

queue<ll>have;

have.push(s);

d[s] = 1;

while (!have.empty()) {

for (ll point = head[have.front()]; point; point = e[point].ne) {

if (!d[e[point].en] && e[point].c) {

d[e[point].en] = d[have.front()] + 1;

have.push(e[point].en);

if (e[point].en == t) {

return true;

}

}

}

have.pop();

}

return false;

}

ll dfs(ll pos, ll mf) {

if (pos == t) {

return mf;

}

ll sum = 0;

for (ll point = cur[pos]; point; point = e[point].ne) {

cur[pos] = point;

if (d[e[point].en] == d[pos] + 1 && e[point].c) {

ll f = dfs(e[point].en, min(mf, e[point].c));

e[point].c -= f;

e[point ^ 1].c += f;

sum += f;

mf -= f;

if (!mf) {

break;

}

}

}

if (!sum) {

d[pos] = 0;

}

return sum;

}

ll dinic() {

ll flow = 0;

while (bfs()) {

cur = head;

flow += dfs(s, inf);

}

return flow;

}

int main() {

cin >> n >> m >> s >> t;

head.resize(n + 1);

add(0, 0, 0);

add(0, 0, 0);

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

ll u, v, w;

cin >> u >> v >> w;

add(u, v, w);

add(v, u, 0);

}

cout << dinic() << endl;

}

ek最小费用流：

#include<iostream>

#include<vector>

#include<queue>

#define ll long long

using namespace std;

struct edge {

ll en, ne, c, w;

edge(ll \_en,ll \_ne,ll \_c,ll \_w):en(\_en),ne(\_ne),c(\_c),w(\_w){}

};

vector<edge>e;

vector<ll>head;

vector<ll>pre;

vector<ll>mf;

vector<ll>d;

ll s, t;

ll n, m;

ll inf = 1e15;

ll flow = 0, cost = 0;

void add(ll beg, ll en, ll c, ll w) {

e.push\_back(edge(en, head[beg], c, w));

head[beg] = e.size() - 1;

}

bool spfa() {

pre.clear();

pre.resize(n + 1);

mf.clear();

mf.resize(n + 1);

d.clear();

d.resize(n + 1);

vector<bool>check(n + 1);

check[s] = true;

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

mf[q] = 0;

d[q] = inf;

}

d[s] = 0;

mf[s] = inf;

check[s] = true;

queue<ll>have;

have.push(s);

while (!have.empty()) {

ll u = have.front();

have.pop();

check[u] = false;

for (ll point = head[u]; point; point = e[point].ne) {

if (d[e[point].en] > e[point].w + d[u] && e[point].c) {

d[e[point].en] = d[u] + e[point].w;

mf[e[point].en] = min(mf[u], e[point].c);

pre[e[point].en] = point;

if (!check[e[point].en]) {

check[e[point].en] = true;

have.push(e[point].en);

}

}

}

}

return mf[t] > 0;

}

void ek() {

while (spfa()) {

for (ll point = t; point != s; point = e[pre[point] ^ 1].en) {

e[pre[point]].c -= mf[t];

e[pre[point] ^ 1].c += mf[t];

}

flow += mf[t];

cost += mf[t] \* d[t];

}

}

int main() {

cin >> n >> m >> s >> t;

head.resize(n + 1);

add(0, 0, 0, 0);

add(0, 0, 0, 0);

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

ll u, v, c, w;

cin >> u >> v >> c >> w;

add(u, v, c, w);

add(v, u, 0, -w);

}

ek();

cout << flow << " " << cost << endl;

}

spfa多路增广费用流：

struct node

{

int x, y, c, cc, next;

}a[2100000];//数组模拟链表存储边

int len = 1, last[210000], maxn = 1e9;

int vis[210000]; int dis[210000];

//解释一下各数组的含义：vis两个用处：spfa里的访问标记，増广时候的访问标记，dis是每个点的距离标号

int n, m, s, t, ans = 0, x, y, c, cc, k;

//s是起点，t是终点，ans是费用答案

inline void add(int x, int y, int c, int cc)

{

len++;

a[len].x = x; a[len].y = y; a[len].c = c; a[len].cc = cc;

a[len].next = last[x]; last[x] = len;

len++;

a[len].x = y; a[len].y = x; a[len].c = 0; a[len].cc = -cc;

a[len].next = last[y]; last[y] = len;

}//建边

inline bool spfa(int s, int t)//反向跑最短路，求出距离标号

{

memset(vis, 0, sizeof(vis));

for (int i = 0; i <= n; i++)dis[i] = maxn;

vis[t] = 1; dis[t] = 0;

//首先SPFA我们维护距离标号的时候要倒着跑，这样可以维护出到终点的最短路径

deque<int>p; p.push\_back(t);

//使用了SPFA的SLF优化

while (!p.empty())

{

int now = p.front(); p.pop\_front();

for (int k = last[now]; k; k = a[k].next)

{

if (a[k ^ 1].c > 0)

//首先c[k^1]是为什么呢，因为我们要保证正流，但是SPFA是倒着跑的，所以说我们要求c[k]的对应反向边是正的，这样保证走的方向是正确的

{

int y = a[k].y;

if (dis[y] > dis[now] - a[k].cc)

{

dis[y] = dis[now] - a[k].cc;

//因为已经是倒着的了，我们也可以很清楚明白地知道建边的时候反向边的边权是负的，所以减一下就对了（负负得正）

if (vis[y] == 0)

{

vis[y] = 1;

if (!p.empty() && dis[y] < dis[now])p.push\_front(y);

else p.push\_back(y);

//SLF优化

}

}

}

}

vis[now] = 0;//队头元素退队，设置为未访问

}

if (dis[s] == maxn)return(false);

else return(true);

//判断起点终点是否连通

}

int dfs(int x, int low)//这里就是进行増广了

{

if (x == t) { vis[t] = 1; return low; }

int used = 0, aa; vis[x] = 1;//这边是不是和dinic很像啊

for (int k = last[x]; k; k = a[k].next)

{

int y = a[k].y;

if (vis[y] == 0 && a[k].c > 0 && dis[x] - a[k].cc == dis[y])

//这个条件就表示这条边可以进行増广

{

aa = dfs(y, min(a[k].c, low - used));

if (aa > 0)ans += aa \* a[k].cc, a[k].c -= aa, a[k ^ 1].c += aa, used += aa;

//累加答案，加流等操作都在这了

if (used == low)break;

}

}

return used;

}

inline int costflow()

{

int flow = 0;

while (spfa(s, t))//判断起点终点是否连通，不连通说明满流，做完了退出

{

vis[t] = 1;

while (vis[t])

{

memset(vis, 0, sizeof(vis));

flow += dfs(s, maxn);

//一直増广直到走不到为止（这样也可以省时间哦）

}

}

return(flow);//这里返回的是最大流，费用的答案在ans里

}

不用spfa的费用流：

#include <iostream>

#include <vector>

#include <cstring>

#include <queue>

#define int long long

using namespace std;

typedef pair<int, int> PII;

const int N = 1e5 + 10, M = 4e5 + 10;

const int INF = 1e18;

int h[N], e[M], f[M], w[M], ne[M], idx;

int dis[N], cur[N];

int n, m, S, T;

bool vis[N];

int height[N];

vector<int> dot;

void add(int a, int b, int c, int d) {

if (idx >= M) {

cerr << "Edge index overflow!\n";

exit(1);

}

e[idx] = b, f[idx] = c, w[idx] = d, ne[idx] = h[a], h[a] = idx++;

e[idx] = a, f[idx] = 0, w[idx] = -d, ne[idx] = h[b], h[b] = idx++;

}

bool dijk() {

for (auto& i : dot) dis[i] = INF, cur[i] = h[i];

priority\_queue<PII, vector<PII>, greater<>> pq;

dis[S] = 0;

pq.emplace(0, S);

while (!pq.empty()) {

auto [d, u] = pq.top();

pq.pop();

if (dis[u] != d) continue;

for (int i = h[u]; ~i; i = ne[i]) {

int v = e[i];

if (f[i] && dis[v] > d + height[u] - height[v] + w[i]) {

dis[v] = d + height[u] - height[v] + w[i];

pq.emplace(dis[v], v);

}

}

}

return dis[T] != INF;

}

int find(int u, int limit) {

if (!limit || u == T) return limit;

int flow = 0;

vis[u] = 1;

for (int& i = cur[u]; ~i && flow < limit; i = ne[i]) {

int v = e[i];

if (dis[v] == dis[u] + height[u] - height[v] + w[i] && f[i] && !vis[v]) {

int t = find(v, min(f[i], limit - flow));

f[i] -= t, f[i ^ 1] += t, flow += t;

if (flow == limit) break;

}

}

vis[u] = 0;

return flow;

}

void mcmf(int& flow, int& cost) {

flow = cost = 0;

while (dijk()) {

int r = find(S, INF);

flow += r;

for (auto& i : dot) height[i] += dis[i];

cost += height[T] \* r;

}

}

void solve() {

memset(h, -1, sizeof h);

cin >> n >> m >> S >> T;

for (int i = 1; i <= n; ++i) dot.push\_back(i);

for (int q = 0; q < m; ++q) {

int u, v, w, c;

cin >> u >> v >> w >> c;

add(u, v, w, c);

}

int flow, cost;

mcmf(flow, cost);

cout << flow << " " << cost << "\n";

}

signed main() {

cin.tie(0)->sync\_with\_stdio(0);

int \_ = 1;

while (\_--) solve();

return 0;

}

# 费用流（MinCostFlow 新版）

/\*\* 费用流（MinCostFlow 新版）

\* 2023-11-09: https://qoj.ac/submission/244680

\*\*/

template<class T>

struct MinCostFlow {

struct \_Edge {

int to;

T cap;

T cost;

\_Edge(int to\_, T cap\_, T cost\_) : to(to\_), cap(cap\_), cost(cost\_) {}

};

int n;

std::vector<\_Edge> e;

std::vector<std::vector<int>> g;

std::vector<T> h, dis;

std::vector<int> pre;

bool dijkstra(int s, int t) {

dis.assign(n, std::numeric\_limits<T>::max());

pre.assign(n, -1);

std::priority\_queue<std::pair<T, int>, std::vector<std::pair<T, int>>, std::greater<std::pair<T, int>>> que;

dis[s] = 0;

que.emplace(0, s);

while (!que.empty()) {

T d = que.top().first;

int u = que.top().second;

que.pop();

if (dis[u] != d) {

continue;

}

for (int i : g[u]) {

int v = e[i].to;

T cap = e[i].cap;

T cost = e[i].cost;

if (cap > 0 && dis[v] > d + h[u] - h[v] + cost) {

dis[v] = d + h[u] - h[v] + cost;

pre[v] = i;

que.emplace(dis[v], v);

}

}

}

return dis[t] != std::numeric\_limits<T>::max();

}

MinCostFlow() {}

MinCostFlow(int n\_) {

init(n\_);

}

void init(int n\_) {

n = n\_;

e.clear();

g.assign(n, {});

}

void addEdge(int u, int v, T cap, T cost) {

g[u].push\_back(e.size());

e.emplace\_back(v, cap, cost);

g[v].push\_back(e.size());

e.emplace\_back(u, 0, -cost);

}

std::pair<T, T> flow(int s, int t) {

T flow = 0;

T cost = 0;

h.assign(n, 0);

while (dijkstra(s, t)) {

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

h[i] += dis[i];

}

T aug = std::numeric\_limits<int>::max();

for (int i = t; i != s; i = e[pre[i] ^ 1].to) {

aug = std::min(aug, e[pre[i]].cap);

}

for (int i = t; i != s; i = e[pre[i] ^ 1].to) {

e[pre[i]].cap -= aug;

e[pre[i] ^ 1].cap += aug;

}

flow += aug;

cost += aug \* h[t];

}

return std::make\_pair(flow, cost);

}

struct Edge {

int from;

int to;

T cap;

T cost;

T flow;

};

std::vector<Edge> edges() {

std::vector<Edge> a;

for (int i = 0; i < e.size(); i += 2) {

Edge x;

x.from = e[i + 1].to;

x.to = e[i].to;

x.cap = e[i].cap + e[i + 1].cap;

x.cost = e[i].cost;

x.flow = e[i + 1].cap;

a.push\_back(x);

}

return a;

}

};

# 最大费用最大流

在加边时先加负权再加正权，结果反号输出即可

# 线性基

将某个集合变换得到一个新集合，新集合仍然可以表示旧集合

如：某多元一次方程组的解

应用：异或线性基：

给定n个整数（数字可能重复），求在这些数中选取任意个，使得他们的异或和最大。

将这些数表示成二进制形式，构造成一个矩阵，然后使用高斯消元法得出这组数的基底（初等变换改换成异或）

得到的异或线性基所有数异或起来就是最大的异或和

应用：异或第k大

背景同上，求异或第k大

将算出来的异或线性基排序，然后将k表示为二进制形式，决定是否要异或某个异或线性基

应用：异或种类

给定n个整数（数字可能重复），求在这些数中选取任意个，求异或和的种类数

设异或线性基的维度是k，那么结果就是2^k（某个线性基选或不选）

# 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）

# 斯特林反演

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

# 斯特林反演常用转换

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）！]

# 广义二项式定理

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！

# 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;

}

# Nim博弈

有n堆石头，每次可以选取一堆石头取走其中一定量的石头，不可不取，最后无法取石头的人败北。

若这n堆石头数量分别为a1、a2、……、an则有如下结论：

若a1^a2^……^an==0则先手必败

若a1^a2^……^an！=0则先手必胜

拓展：

游戏有两个人参与，二者轮流做出决策，双方均知道游戏的完整信息；

任意一个游戏者在某一确定状态可以作出的决策集合只与当前的状态有关，而与游戏者无关；

游戏中的同一个状态不可能多次抵达，游戏以玩家无法行动为结束，且游戏一定会在有限步后以非平局结束。

定义：

后继转移状态：可由当前状态一步转移得到的状态

必败态：当前状态不存在后继转移状态

必胜态：当前状态存在一个后继转移状态为必败态

若一个状态的所有后继转移状态均为必胜态则该状态为必败态

定义函数sg(i):i状态下的sg值

定义当sg(i)=0时此状态为必败态

若i状态能一步转化为a1、a2、a3、……、an

则sg(i)=mex(sg(a1),sg(a2),sg(a3),……,sg(n))

sg定理：

一个游戏的sg值为：

sg(a)^sg(b)^……^sg(z)^……

a、b、……、z、……为该游戏的所有起点

可以归纳得出上述石头的sg值为：

sg(n)=n

n为当前堆中石头数量

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

# 分数求模

求a/b%p

求出b的逆元B即可

a/b≡Bb（modp）

根据逆元关系：

Bb≡1（modp）

可以构造出以下方程（该式子要b、p互质时才可使用）

bB+py=1=gcd（b，p）

恰好对应扩展欧几里得算法：ax+by=gcd（a，b）

利用扩展欧几里得算法求出x后将其转换为正数：x=（x%p+p）%p

x的值即为所求逆元

同时由费马小定理也可知道：

若p为质数且b、p互质时存在

b^（p-1）≡1（modp）

而b\*b^（p-2）=b^（p-1）

易得此时b的逆元即为b^（p-2），可用快速幂求解

公式：a/b%m=a\*qpow(b,m-2)%m

# 倍增lca

求某棵树上两点u,v的最近公共祖先。

采用倍增算法：

采用类似st表的原理记录每个点的祖先节点：

f[x][y]表示节点x的第2^y个祖先

从根节点开始采用dfs方式填表

转移方程为：f[x][y]=f[f[x][y-1]][y-1]

将u,v之间的深度差记作dis

利用lowbit将每一位分离，根据大小跳跃，最后将较深的那个点跳跃到与较浅的点相同的深度处

开始二分查找跳跃层数：将跳跃步数逐渐变小，直到该情况：此时两节点不相同，而它们的父节点相同，此时便找到了最近公共祖先。

利用lca，可以计算出树上两点间的最小路径

模板：

#include<map>

#define ll long long

#define lowbit(x) (x&(-x))

using namespace std;

map<ll,map<ll,ll>>tree;//树，记录节点的子节点

map<ll,ll>root;//节点的父节点

map<ll,ll>len;//节点深度，从1开始计数

vector<vector<ll>>st;//记录节点的2^n级祖先

void get\_st(point){

st[point].push\_back(root[point]);//传入point的2^0级祖先

for(ll q=1;(1<<q)<len[point];++q){

st[point].push\_back(st[st[point][i-1]][i-1]);//point的2^i级祖先等于point的2^(i-1)级祖先的2^(i-1)级祖先

}

for(auto it:tree[point]){

get\_st(it.first);//dfs搜索point的子节点并完善st表

}

}//获取st表过程,传入的节点为根节点

ll lca(ll p1,ll p2){

if(len[p1]>len[p2]){

ll temp=p1;

p1=p2;

p2=temp;

}//将较深的节点调为p2

ll dis=len[p2]-len[p1];

while(dis){

p2=st[p2][\_\_lg(lowbit(dis))];

dis-=lowbit(dis);

}//调为同级

if(p1==p2){

return p1;

}//排除未跳跃就相等的情况

while(true){

if(root[p1]==root[p2]){

return root[p1]//找到最近公共祖先

}

else{

ll fin=\_\_lg(len[p1]-1);//最大可跳跃级数

while(stable[p1][fin]==stable[p2][fin]){

--fin;

}//找到此时最大的可跳跃层数，满足跳跃后两点没有重合

p1=stable[p1][fin];//跳跃p1点

p2=stable[p2][fin];//跳跃p2点

}

}

}//lca过程

# 重链剖分lca

求某棵树上两点u,v的最近公共祖先。(一般来说重链剖分的常数时间要优于倍增算法)

采用树链剖分的方法记录每一个节点所在的链的顶点，然后进行跳跃：

若两个点的链顶点相同，两个点中深度较小的即为所求

将两个点中链顶点较深的点跳跃到其链顶点的父节点

#include<iostream>

#include<map>

#include<vector>

#define ll long long

using namespace std;

vector<vector<ll>>re;//存储边

vector<vector<ll>>tree;//存储树

vector<ll>root;//存储父节点

vector<ll>len;//存储深度

vector<ll>num;//存储子树节点数

vector<bool>check;//建树时的判断

vector<ll>list\_root;//

vector<ll>memory\_p;//某节点下最大的子数

ll n, s;//n为节点数，s为根节点

void create(ll p, ll length) {

check[p] = true;

ll all = 1;

len[p] = length;

ll memory\_now = 0;

for (auto it : re[p]) {

if (!check[it]) {

create(it, length + 1);

root[it] = p;

tree[p].push\_back(it);

if (memory\_now) {

if (num[it] > num[memory\_now]) {

memory\_now = it;

}

}

else {

memory\_now = it;

}

all += num[it];

}

}

if (memory\_now) {

memory\_p[p] = memory\_now;

}

num[p] = all;

}

void dfs\_create(ll p) {

if (!tree[p].empty()) {

for (auto it : tree[p]) {

if (it != memory\_p[p]) {

list\_root[it] = it;

dfs\_create(it);

}

else {

list\_root[it] = list\_root[p];

dfs\_create(it);

}

}

}

}

ll lca(ll a, ll b) {

if (a == b) {

cout << a << endl;

}

else {

if (len[a] > len[b]) {

ll temp = a;

a = b;

b = temp;

}

while (true) {

if (list\_root[a] == list\_root[b]) {

return a;

}

else {

if (len[list\_root[a]] > len[list\_root[b]]) {

a = root[list\_root[a]];

}

else {

b = root[list\_root[b]];

}

if (len[a] > len[b]) {

ll temp = a;

a = b;

b = temp;

}

}

}

}

}

int main() {

ios::sync\_with\_stdio(0);

cin.tie(nullptr);

cout.tie(nullptr);

cin >> n >> s;

root.resize(n + 1);

len.resize(n + 1);

num.resize(n + 1);

check.resize(n + 1);

list\_root.resize(n + 1);

tree.resize(n + 1);

re.resize(n + 1);

memory\_p.resize(n + 1);

list\_root[s] = s;

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

cin >> a >> b;

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

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

}

create(s, 0);//建树

root[s] = s;

dfs\_create(s);//树链剖分

}

# 树上最短路径

求树上两点u,v的最短路径。

结合lca求解

模板：

#include<map>

#define ll long long

#define lowbit(x) (x&(-x))

using namespace std;

map<ll,map<ll,ll>>tree;//树，记录节点的子节点

map<ll,ll>root;//节点的父节点

map<ll,ll>len;//节点深度，从1开始计数

vector<vector<ll>>st;//记录节点的2^n级祖先

void get\_st(point){

st[point].push\_back(root[point]);//传入point的2^0级祖先

for(ll q=1;(1<<q)<len[point];++q){

st[point].push\_back(st[st[point][i-1]][i-1]);//point的2^i级祖先等于point的2^(i-1)级祖先的2^(i-1)级祖先

}

for(auto it:tree[point]){

get\_st(it.first);//dfs搜索point的子节点并完善st表

}

}//获取st表过程,传入的节点为根节点

ll lca\_dist(ll p1,ll p2){

if(len[p1]>len[p2]){

ll temp=p1;

p1=p2;

p2=temp;

}//将较深的节点调为p2

ll dis=len[p2]-len[p1];

ll dis1=0,dis2=dis;//记录p1,p2此时走过的路程

while(dis){

p2=st[p2][\_\_lg(lowbit(dis))];

dis-=lowbit(dis);

}//调为同级

if(p1==p2){

return dis2;

}//排除未跳跃就相等的情况

while(true){

if(root[p1]==root[p2]){

return dis1+dis2+2;//找到最近公共祖先

}

else{

ll fin=\_\_lg(len[p1]-1);//最大可跳跃级数

while(stable[p1][fin]==stable[p2][fin]){

--fin;

}//找到此时最大的可跳跃层数，满足跳跃后两点没有重合

dis1+=(1<<fin);

dis2+=(1<<fin);

p1=stable[p1][fin];

p2=stable[p2][fin];

}

}

}//lca过程

# 约瑟夫环

f[n][m][k]，n个人报到k的人出局的前提下第m个出局的人

f[n][m][k]=(f[n-1][m-1][k]+k)%n

f[n][n][k]=(f[n-1][n-1][k]+k)%n

f[n+x][m+x][k]=f[n][m]+k\*x,x＞(n-f[n][m])/(k-1)

# 杂项

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。

c(n,m)=a(n,m)/m!

优化输入：使用scanf/使用cin加速语句：std：：ios：：sync\_with\_stdio(0);cin.tie（0）;（tie函数中的0建议用nullptr替换）

[n,2n]必然存在质数

若n≥3，则[n,3n/2]必然存在一个质数

一个非零整数与其相反数做与运算结果是其二进制下最低位的1所对应的值

例子:5&(-5):

00000000000000000000000000000101

&

11111111111111111111111111111011

结果为:

00000000000000000000000000000001

即1

位运算进行移位操作时注意强制转换：

例子：

(long long)1<<50=1125899906842624

1<<50=0

二者是不一样的

a%b=（a+k\*b）%b

（a\*k）%b=k\*（a%b）

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）

初等数论四大定理：

费马小定理：如果a是一个整数，p是一个质数，并且整数a不是p的倍数（即a、p互质且p为质数），那么a^p-a是p的倍数（即a^（p-1）≡1（mod p））

威尔逊定理：p可整除(p-1)!+1是p为质数的充要条件

欧拉定理：若gcd（a，x）=1，那么a^f（x）≡1（mod x）

中国剩余定理：假设整数m1,m2, ... ,mn两两互质，则对任意的整数：a1,a2, ... ,an，方程组S有解，并可构造得出。

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……

快速幂：计算a^b

如果b为偶数：计算(a^(b/2))^2

如果b为奇数：计算a\*(a^((b-1)/2))^2

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）

判断矩形：

dfs找到该图形的x,y坐标的最小值,并记录面积s。若（x\_max-x\_min+1）\*（y\_max-y\_min+1）与s相等，则可以说明为矩形

高效筛法：埃氏筛法、欧拉筛法

欧拉筛法：

#include<iostream>

#include<vector>

using namespace std;

bool arr[50001]{};

vector<int>prime;

int main() {

for (int q = 2; q <= 50000; ++q) {

if (!arr[q]) {

prime.push\_back(q);

}

for (int i = 0; i < prime.size() && prime[i] \* q <= 50000; ++i) {

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

if (!(q % prime[i])) {

break;

}

}

}

for (int q = 0; q < prime.size(); ++q) {

cout << prime[q] << endl;

}

}

负进制转换：一般步骤与普通进制转换一致，求余出现负数时向商加一再求余数即可（余数负数是向上一位借一的结果）

辗转相除法求最大公约数：

int gcd(int x,int y)

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

推论：若gcd（a，b）=c，则gcd（a/c，b/c）=1

差分操作和前缀和互为逆运算

位运算：

a|b=a&b+a^b

贝祖定理：设多元一次方程：ax+by+cz+……=M

其中a，b，c……为系数，O为所有系数的最大公因数，则若O为M的因子，该方程便有整数解

已知前序遍历与后序遍历，则中序遍历的可能数必为2^n

先序遍历：根左右

后序遍历：左右根

中序遍历：左根右

可知：若先序遍历、后续遍历长度为二时：

先序遍历：根节点

后序遍历：节点根

故此时中序遍历有两种表示：

节点根（右子树为空）

根节点（左子树为空）

例子：先序遍历得字符串a、后序遍历得b，通过以下程序可得到中序遍历的可能数量answer：

long long answer=1;

for(int q=0;q<a.length()-1;++q){

for(int i=1;i<b.length();++i){

if(a[q]==b[i]&&a[q+1]==b[i-1]){

answer\*=2;

}

}

}

avl二叉平衡树：定义某个节点的平衡因子为左右子树节点数只差，若该节点的平衡因子不在-1~1之间，则需要进行旋转

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。

图论寻路：

基本思路：

将一级关系用dp[q][i]储存，表示q到i的距离，无法用一级关系表示的，视为无穷大。随后开始进行转移，判断加入新的节点后，路径是否能变小（dp[q][k]+d[k][i]<dp[q][i]）

floyed：多个起点搜索最短路径

dijkstra：单个起点搜索最短路径：搜索一轮之后，将剩下的最小的dp[q][i]标记为最段路径，随后不再需要改变

spfa：