数 学....................................................... 4

最大公约数、最小公倍数................................... 4

最大公约数——欧几里得算法O(n)............................ 4

Stein算法O( log(max(a,b)) ).............................. 4

最小公倍数：.......................................... 4

素数相关........................................... 5

普通素数判断................... 5

筛法求素数[1，N].................... 5

二次筛法求素数[L，R]..................... 6

Miller-Rabbin素数测试方法............... 7

算术基本定理的定义和性质：................ 8

同余方程[组] 乘法模逆元 中国剩余定理....................... 9

扩展欧几里得，求一组解x，y，使得gcd(a,b) = d = a \* x + b \* y ................. 9

扩展欧几里得，求所有解x，y，使得c = a \* x + b \* y

扩展欧几里得，求a关于n的逆元a^-1，使得a \* a^-1 ≡ 1(mod n)

扩展欧几里得，求解x，满足同余方程组x ≡ Ri(mod Ai) ................................... 10

扩展欧几里得，求解x，满足高次同余方程A^x ≡ B(mod C).... 11

中国剩余定理：............................... 13

中国剩余定理最小非负数解的算法：...................... 14

求解a\*x + b\*y = c的其中一组解，使得|x| + |y|尽可能小，若相等，则a|x| +b|y|尽可能小。............. 15

整数快速幂.............................. 16

矩阵快速幂................................ 16

整数分解........................... 18

试除法整数分解....................... 18

筛法整数分解.............................. 18

PollardRho大整数分解............................ 19

欧拉函数..................................... 22

直接欧拉函数.................................. 22

递推快速求欧拉函数...................... 23

容斥原理......................................... 23

母函数........................................... 24

普通母函数........................................ 24

指数型母函数...................................... 25

其他相关............................................ 27

九余数定理：一个数N各位数字的和，对9取余等于这个数对9取余 ................. 27

给你一个奇数N，求1~N的奇数平方和： S = N\*(N+1)\*(N+2)/6 .......................... 27

约瑟夫问题：有N个人，编号为1~N，按顺时针围成一个圈，每数k个人，就将这个人从圈中消除，问：最终只留下一个人的编号。..... 27

给你整数x和y的和以及x和y的积，是否能找到满足这两个式子的整数x和整数y。................................ 29

Fibonacci数列大于40前四位数字................... 29

小于N的素数个数为π(N),π(N)的位数是多少？............ 29

Stirling公式：当N足够大时，N! = (N/e) \* N \* sqrt(2\*pi\*N)。 .................. 29

有N张卡，将N张卡分成若干不同的集合，集合不能为空。问：总共有多少种分法。........................ 29

字符串............................................. 29

KMP............................................. 29

1. 既能做前缀又能做后缀的子串长度..................... 30

2.求最短子串长度................... 30

3.求模式串pat在主串str中匹配次数.............. 30

4.求N行M列的字符矩阵，最小的字符子矩阵的面积................ 30

5.求字符串s的循环前缀的长度和循环的次数................... 30

字典树........................... 31

数据结构..................................................... 33

并查集.............................................. 33

树状数组....................................... 33

单点更新，区间求值：树状数组代表区间的和。................. 33

区间更新，单点求值：树状数组代表单个元素的变化.......... 34

求逆序数： 数组Tree[i]表示数字i是否在序列中出现过，如果数字i已经存在于序列中，Tree[i] = 1，否则Tree[i] = 0。按序列从左到右将值为a的元素当作下标为a，赋值为1插入树状数组里，这时，比a的数个数就是i - Query(a)。

将全部结果累加起来就是逆序数了。............................. 35

二维树状数组：.................. 36

线段树................................... 36

单点更新.............................. 37

成段更新....................................... 43

区间合并........................................ 52

扫描线........................................... 54

图 论................................................... 58

图的五种种存储方式............................ 58

方式1：邻接矩阵........................... 58

方式2：前向星.......................... 58

方式3：邻接表——动态建表................................. 59

方式4：邻接表——vector模拟链表实现 ........................ 60

方式5：邻接表——链式前向星★...................... 61

拓扑排序.................................. 62

最小生成树............................... 63

Kruskal算法：.......................... 63

Prim算法：............................. 64

次小生成树.......................................... 66

最小树形图....................................... 68

最近公共祖先LCA—Tarjan-LCA算法：........................ 72

树的最小支配集、最小点覆盖、最大独立集..................... 74

最小支配集：指从所有顶点中取尽量少的点组成一个集合，使得剩下的所有点都与取出来的点有边相连。顶点个数最小的支配集被称为最小支配集。这里用贪心法来

求。........................... 74

最小点覆盖：指从所有顶点中取尽量少的点组成一个集合，使得集合中所有的边都与取出来的点有边相连。顶点个数最小的覆盖集被称为最小点覆盖。 ................. 76

最大独立集：指从所有顶点中取尽量多的点组成一个集合，使得这些点之间没有边相连。顶点个数最多的独立集被称为最大独立集。........... 77

单源最短路径Dijkstra、BellmanFord、SPFA...................... 77

Dijkstra算法：........................................ 77

BellmanFord算法：.................................. 79

SPFA算法：................................................ 81

多源最短路径Floyd、Floyd求最小环........................ 84

Floyd算法：用来找出每对点之间的最短距离。图可以是无向图，也可以是有向图，边权可为正，也可以为负，唯一要求是不能有负环。........... 84

Floyd求最小环....................... 85

K短路—A\*+SPFA算法：................................... 86

差分约束系统............................................. 89

强连通分量Kosaraju、Tarjan................................ 92

Kosaraju算法：........................................ 92

Tarjan算法：............................................ 94

2-SAT............................................ 97

二分图.......................................... 100

二分图最大匹配——匈牙利算法DFS版：................... 100

二分图多重匹配：................................... 101

二分图最佳匹配——KM算法：.............. 102

网络流最大流EdmondKarp、SAP.................... 104

EdmondKarp算法：.................................. 104

SAP算法：................................................ 106

输入输出外挂—仅适合纯数字输入....................... 109

数 学

最大公约数、最小公倍数

最大公约数、最小公倍数性质：

1.若a | m，b | m，则lcm(a，b) | m。

2.若d | a，d | b，则d | gcd(a，b)。

3.lcm(a，b) = a \* b / gcd(a，b)。

4.设m，a，b是正整数，则lcm(m\*a，m\*b) = m \* gcd(a，b)

5.若m是非零整数a1，a2，…，an的公倍数，则lcm(a1，a2，…，an) | m用素因子分解法求a和b的最大公约数和最小公倍数：

a = p1^r1 \* p2^r2 \* … \* pk^rk

b = p1^s1 \* p2\*s2 \* … \* pk^sk

gcd(a，b) = p1^min(r1,s1) \* p2^min(r2,s2) \* … \* pk^min(rk,sk) lcm(a，b) = p1^max(r1,s1) \* p2^max(r2,s2) \* … \* pk^max(rk,sk) 最大公约数——欧几里得算法O(n)

int GCD(int a,int b)

{

if(a < b)

int temp = a, a = b, b = temp;

if(b == 0) return a;

return GCD(b,a%b);

}

Stein算法O( log(max(a,b)) )

int KGCD(int a,int b)

{

if(a == 0) return b;

if(b == 0) return a;

if(~a & 1)

{

if(b & 1) return KGCD(a>>1, b);

else return KGCD(a>>1, b>>1) << 1;

}

if(~b & 1) return KGCD(a, b>>1);

if(a > b) return KGCD((a-b)>>1, b);

return KGCD((b-a)>>1, a);

}

最小公倍数：

int LCM(int a,int b)

{

return a/KGCD(a,b)\*b;

}

素数相关

素数和合数共同的性质：

1.a > 1是合数，当且仅当a = b \* c，其中1 < b < a，1 < c < a。

2.合数必有素数因子。

3.如果d > 1，p是素数，且d | q，则d = p。

4.设p是素数且p | a\*b，则必有p | a或者p | b。

素数的性质：

1.存在无穷多个素数。

2.每个大于1的正整数都有一个素因子。

素数的分布：

素数定理：用π(x)估计小于正整数x的素数有多少个。随着x的增长，π(x) / (x/ln x) = 1。

推论：令pn是第n个素数，其中n是正整数，那么pn ~ n\*ln n。

素数的猜想：

1.波特兰猜想：对于任意给定的正整数n，其中n > 1，存在一个素数p，使得n < p < 2\*n。

2.孪生素数猜想：存在无穷多的形如p和p+2的素数对。

3.哥德巴赫猜想：每个大于2的正偶数可以写成两个素数的和。可推出任一大于7的奇数都可写成三个质数之和的猜想。

普通素数判断

int IsPrime(int N) //注意#include<cmath>

{

if(N <= 1) return 0;

int i;

for(i = 2; i <= sqrt(N\*1.0); ++i)

if(N%i == 0) return 0;

return 1;

}

筛法求素数[1，N]

const int MAXN = 1000000;

bool Prime[MAXN+100]; //Prime[i] == true表示i为素数

void IsPrime()

{

for(int i = 2; i <= MAXN; ++i)

Prime[i] = true;

for(int i = 2; i <= MAXN; ++i)

if(Prime[i])

for(int j = i+i; j <= MAXN; j+=i)

Prime[j] = false;

}

二次筛法求素数[L，R]；

boolPrime[50010];//存5000；intPrimer[1000010];//存放区；boolPrime1[1000010];//判断；intIsPrime()//第一次筛50000内；intnum=0;；for(inti=2;i<=50000;i；Prime[i]=true;；for(inti=2;i<=

二次筛法求素数[L，R]

bool Prime[50010]; //存50000内素数判断结果

int Primer[1000010]; //存放区间[L,R]之间的素数

bool Prime1[1000010]; //判断区间[L,R]中的数是否为素数

int IsPrime()//第一次筛50000内的素数

{

int num = 0;

for(int i = 2; i <= 50000; i++)

Prime[i] = true;

for(int i = 2; i <= 50000; i++)

{

if(Prime[i])

{

Primer[num++] = i;

for(int j = i+i; j <= 50000; j+=i)

Prime[j] = false;

}

}

return num; //num为50000范围内的素数个数

}

int IsPrime2(\_\_int64 a,\_\_int64 b)

/\*

在第一次筛素数的基础上，利用50000以内的素数，筛去范围【a,b】之间的素数倍数，

剩下则为素数

\*/

{

int num = IsPrime();

memset(Prime1,true,sizeof(Prime1));

//Prime1数组用来存放范围【a,b】的素性判断

if(a == 1) Prime1[0] = 0; //这里表示0+1不为素数

for(\_\_int64 i = 0; i < num && Primer[i] \* Primer[i] <= b; i++) {

\_\_int64 begin = a/Primer[i] + (a%Primer[i] != 0);

//上边的a/Primer算出应a为素数Primer[i]的多少倍

//(a%Primer[i]!=0)表示应从Primer[i]的a/Primer[i]倍开始筛，还是a/Primer[i]+1倍筛

if(begin == 1)//若得出结果为所被筛素数的1倍，则从该素数的2倍开始筛

begin++;

for(begin = begin\*Primer[i]; begin <= b; begin += Primer[i]) Prime1[begin - a] = false;

}

//这里重新利用Primer数组，用来存放区间【a,b】间的素数，num为素数个数

memset(Primer,0,sizeof(Primer));

num = 0;

for(\_\_int64 i = a; i <= b; i++)

if(Prime1[i-a]==1) Primer[num++] = i-a;

return num; //num为区间[a,b]的素数个数

}

Miller-Rabbin素数测试方法

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<math.h>

#define MAX\_VAL (pow(2.0,60))

#define LL \_\_int64

/\*LL mod\_mul(LL x,LL y,LL mo) //计算x \* y % mo

{

LL t;

x %= mo;

for(t = 0; y; x = (x<<1)%mo,y>>=1)

if(y & 1)

t = (t+x) %mo;

return t;

}\*/

LL mod\_mul(LL x,LL y,LL mo) //计算x \* y % mo

{

LL t,T,a,b,c,d,e,f,g,h,v,ans;

T = (LL)(sqrt(double(mo)+0.5));

t = T\*T - mo;

a = x / T;

b = x % T;

c = y / T;

d = y % T;

e = a\*c / T;

f = a\*c % T;

v = ((a\*d+b\*c)%mo + e\*t) % mo;

g = v / T;

h = v % T;

ans = (((f+g)\*t%mo + b\*d)% mo + h\*T)%mo;

while(ans < 0) ans += mo;

return ans;

}

LL mod\_exp(LL num,LL t,LL mo) //计算num^t % mo

{

LL ret = 1, temp = num % mo;

for(; t; t >>=1,temp=mod\_mul(temp,temp,mo))

if(t & 1) ret = mod\_mul(ret,temp,mo);

return ret;

}

bool miller\_rabbin(LL n) //miller\_rabbin素性测试

{

if(n == 2) return true;

if(n < 2 || !(n&1)) return false;

int t = 0;

LL a,x,y,u = n-1;

while((u & 1) == 0)

{

t++;

u >>= 1;

}

for(int i = 0; i < 50; i++)

{

a = rand() % (n-1)+1;

x = mod\_exp(a,u,n);

for(int j = 0; j < t; j++)

{

y = mod\_mul(x,x,n);

if(y == 1 && x != 1 && x != n-1) return false; x = y;

}

if(x != 1) return false;

}

return true;

}

算术基本定理的定义和性质：

定理：每个大于1的正整数N都可以被唯一地写成素数的乘积，在乘积中的素因子按照非降序排列。

正整数N的分解式 N = p1^α1 \* p2^α2 \* p3^α3 \* … \* pk^αk 称为N的标准分解式，其中p1，p2，…，pk是素数，p1 < p2 < p3 < … < pk，且α1，α2，α3，…，αk是正整数。

性质：

(1)若N的标准素因子分解表达式为：N = p1^α1 \* p2^α2 \* p3^α3 \* … \* pk^αk，设d(N)

为N的挣银子个数， Φ(N)为N的所有因子之和，则有

d(N) = (α1 + 1) \* (α2 + 1) \* (α3 + 1) \* … \* (αk + 1)

Φ(N) = ( p1^(α1 + 1) )/(p1 - 1) \* ( p2^(α2 + 1) )/(p2 - 1) \* … \* ( pk^(α

k + 1) )/(pk - 1)

(2)设a = p1^α1 \* p2^α2 \* … \* pk^αk，b = p1^ β1 \* p2^β2 \* … \* pk^βk，则有

gcd(a,b) = p1^min(α1，β1) \* p2^min(α2，β2) \* … \* pk^min(αk，βk) lcm(a,b) = p1^max(α1，β1) \* p2^max(α2，β2) \* … \* pk^max(αk，βk)

(3)如果a和b为实数，则

max( gcd(a，b) ) + min( gcd(a，b) ) = a + b

(4)如果a和b是正整数，则

lcm(a，b) = a\*b/gcd(a，b)

(5)N!的素因子分解中的素数p的幂为

[N/p] + [N/p^2] + [N/p^3] + …

同余方程[组] 乘法模逆元 中国剩余定理

同余定义：给定正整数M，若用M去除两个整数 a 和 b 所得余数相同，称 a 和 b 对模 m 同余，记作a ≡ b (mod m)，并称该式为同余式；否则，称 a 和 b 对模 m 不同余。

同余定理：

1.a ≡ b (mod m)，当且仅当m | (a - b)。

2.a ≡ b (mod m)，当且仅当存在整数 k，使得a = b + k\*m。

3.a ≡ a (mod m)。

4.若 a ≡ b (mod m)，则 b ≡ a (mod m)。

5.若 a ≡ b (mod m)，b ≡ c (mod m)，则 a ≡ c (mod m).

6.若 a、b、c 是整数，m 是正整数，且 a ≡ b (mod m)，则

a + c ≡ b + c (mod m)；a - c ≡ b - c (mod m)；a\*c ≡ b\*c (mod m)。

7.设 a、b、c、d 为整数，m 为正整数，若 a ≡ b (mod m)，则

a\*x + c\*y = b\*x + d\*y (mod m)，其中 x，y 为任意整数；

a\*c ≡ b\*d (mod m)；

a^n ≡ b^n (mod m)，其中n > 0；

f(a) ≡ f(b) (mod m)，其中 f(x) 为任一整系数多项式。

8.设a、b、c、d为整数，m为正整数，则：

若 a ≡ b (mod m)，且d | m，则 a ≡ b (mod d)；

若 a ≡ b (mod m)，则gcd(a，m) ≡ gcd(b，m)；

a ≡ b (mod mi) (1 <= i <= n)同时成立，当且仅当 a ≡ b (mod lcm(m1，m2，…，mn))；

9.若 a\*c ≡ b\*c (mod m)，且gcd(c，m) = d，则 a ≡ b (mod m/d)。

扩展欧几里得，求一组解x，y，使得gcd(a,b) = d = a \* x + b \* y

void ExGcd(int a,int b,int &d,int &x,int &y)

{

if(b == 0)

{

x = 1;

y = 0;

d = a;

}

else

{

ExGcd(b,a%b,d,y,x);

y -= x\*(a/b);

}

}

扩展欧几里得，求所有解x，y，使得c = a \* x + b \* y

bool ModeEqual(int a,int b,int c) //解a\*x + b\*y = c;a\*x ≡ c(mod b) {

int x,y,d,x0;

ExGcd(a,b,d,x,y);

if(c%d) return false; //无解

x0 = x \* (c/d) % b;

for(int i = 1; i < d; ++i) //输出所有解

printf("%d\n",(x0+i\*(b/d))%b);

return true;

}

扩展欧几里得，求a关于n的逆元a^-1，使得a \* a^-1 ≡ 1(mod n) int ModInverse(int a,int n,int &d,int &x,int &y)

{

ExGcd(a,n,d,x,y);

if(1 % d != 0)

return -1; //不存在模逆元

int ans = x/d < 0 ? x/d + n : x/d;

return ans; //返回模逆元a^-1

}

扩展欧几里得，求解x，满足同余方程组x ≡ Ri(mod Ai) int ModEquals(int N) //解方程组x ≡ Ri(mod Ai)

{

int a,b,d,x,y,c,A1,R1,A2,R2;

bool flag = 1; //标记是否有解

scanf("%d%d",&A1,&R1);

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

{

scanf("%d%d",&A2,&R2);

a = A1, b = A2, c = R2 - R1;

ExGcd(a,b,d,x,y);

if(c % d != 0)

flag=0;；intt=b/d;；x=(x\*(c/d)%t+t)%t;；R1=A1\*x+R1;；A1=A1\*(A2/d);；if(!flag)R1=-1;；returnR1;//求出解，-1表示无解；扩展欧几里得，求解x，满足高次同余方程A^x≡B；constintMAXN=65535;；structHASH；inta;；intb;；intnext;；}H

flag = 0;

int t = b/d;

x = (x\*(c/d)%t + t) % t;

R1 = A1 \* x + R1;

A1 = A1 \* (A2 / d);

}

if( !flag ) R1 = -1;

return R1; //求出解，-1表示无解

}

扩展欧几里得，求解x，满足高次同余方程A^x ≡ B(mod C) #define LL \_\_int64

const int MAXN = 65535;

struct HASH

{

int a;

int b;

int next;

}Hash[MAXN\*2];

int flag[MAXN+66];

int top,idx;

void ins(int a,int b)

{

int k = b & MAXN;

if(flag[k] != idx)

{

flag[k] = idx;

Hash[k].next = -1;

Hash[k].a = a;

Hash[k].b = b;

return;

}

while(Hash[k].next != -1)

{

if(Hash[k].b == b)

return;

k = Hash[k].next;

}

Hash[k].next = ++top;

Hash[top].next = -1;

Hash[top].a = a;

Hash[top].b = b;

}

int Find(int b)

{

int k = b & MAXN;

if(flag[k] != idx)

return -1;

while(k != -1)

{

if(Hash[k].b == b)

return Hash[k].a;

k = Hash[k].next;

}

return -1;

}

int GCD(int a,int b)

{

if(b == 0)

return a;

return GCD(b,a%b);

}

void ExGcd(int a,int b,int &d,int &x,int &y)

{

if(b == 0)

{

x = 1;

y = 0;

d = a;

}

else

{

ExGcd(b,a%b,d,y,x);

y -= x\*(a/b);

}

}

int Inval(int a,int b,int n)

{

int x,y,d,e;

ExGcd(a,n,d,x,y);

e = (LL)x\*b%n;

return e < 0 ? e + n : e;

}

int PowMod(LL a,int b,int c)

{

LL ret = 1%c;

a %= c;

while(b)

{

if(b&1)

ret = ret\*a%c;

a = a\*a%c;

b >>= 1;

}

return ret;

}

int BabyStep(int A,int B,int C) //解A^x ≡ B(mod C)

{

top = MAXN;

++idx;

LL buf = 1%C,D = buf,K;

int d = 0,temp,i;

for(i = 0; i <= 100; buf = buf\*A%C,++i)

{

if(buf == B)

return i;

}

while((temp = GCD(A,C)) != 1)

{

if(B % temp)

return -1;

++d;

C /= temp;

B /= temp;

D = D\*A/temp%C;

}

int M = (int)ceil(sqrt((double)C));

for(buf = 1%C,i = 0; i <= M; buf = buf\*A%C,++i)

ins(i,buf);

for(i = 0,K = PowMod((LL)A,M,C); i <= M; D = D\*K%C,++i)

{

temp = Inval((int)D,B,C);

int w;

if(temp >= 0 && (w = Find(temp)) != -1)

return i \* M + w + d;

}

return -1; //无解

}

中国剩余定理：

简单描述：已知 n%3=2，n%5=3，n%7=2，求n。

因为 n%3=2，n%5=3，n%7=2 且 3，5，7互质 （互质可以直接得到这三个数的

最小公倍数）

令 x = n % 3 = 2，y = n%5=3，z = n%7=2。

使5\*7\*a 被3除余1，有35×2=70，即 a = 2；

使3\*7\*b 被5除余1，有21×1=21，即 b = 1；

使3\*5\*c 被7除余1，有15×1=15，即 c = 1。

那么 n =(70×x+21×y+15×z) % lcm(3，5，7) = 23 这是n的最小解

其中，a 为 (5\*7)^-1 mod 3，b 为 (3\*7)^-1 mod 5，c 为 (3\*5)^-1 mod 7。 一般描述：若m1，m2，m3，…，mr是两两互素的正整数，则同余方程组 x ≡ a1 (mod m1)

x ≡ a2 (mod m2)

x ≡ a3 (mod m3)

……

x ≡ ar (mod mr)

有模M = m1\*m2\*m3\*…\*mr的唯一解，即为中国剩余定理。

解法：令Mi = M/mi，因为m1，m2，m3，…，mr两两互素，因此 gcd(Mi，mi) = 1，即MiPi ≡ 1 (mod mi)，那么解为 a1\*M1\*P1 + a2\*M2\*P2 + a3\*M3\*P3 + … + ar\*Mr\*Pr。

中国剩余定理最小非负数解的算法：

int m[110],a[110]; //m[]和a[]对应x ≡ ai (mod mi)

void ExGcd(int a,int b,int &d,int &x,int &y)

{

if(b == 0)

{

x = 1;

y = 0;

d = a;

}

else

{

ExGcd(b,a%b,d,y,x);

y -= x\*(a/b);

}

}

int China(int r) // r为同于方程组个数

{

int M = 1,Mi,x0,y0,d,ans = 0;

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

M \*= m[i];

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

{

Mi = M / m[i];

ExGcd(Mi,m[i],d,x0,y0); //调用扩展欧几里得，见之前

ans = (ans + Mi\*x0\*a[i]) % M;

}

if(ans < 0)

ans += M;

return ans;

}

中国剩余定理推论：

1.若a，b是正整数，那么gcd(2^a-1，2^b-1) = 2^gcd(a，b) - 1.

2. 2\*a - 1，2\*b - 1是互素的，当且仅当正整数a，b是互素的。

求解a\*x + b\*y = c的其中一组解，使得|x| + |y|尽可能小，若相等，则a|x| + b|y|尽可能小。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

#define LL \_\_int64

using namespace std;

LL GCD(LL a,LL b)

{

if(b == 0)

return a;

return GCD(b,a%b);

}

void ExGCD(LL a,LL b,LL &d,LL &x,LL &y)

{

if(!b)

{

x = 1;

y = 0;

d = a;

}

else

{

ExGCD(b,a%b,d,y,x);

y -= x\*(a/b);

}

}

int main()

{

LL a,b,c,temp,d;

while(cin >> a >> b >> c && (a||b||c))

{

LL x0,y0;

LL gcd = GCD(a,b);

a /= gcd

b/=gcd;；c/=gcd;；ExGCD(a,b,d,x0,y0);；LLx,y,x1,y1,x2,y2;；x1=x0\*c;；x1=(x1%b+b)%b;；y1=(c-a\*x1)/b;；if(y1<0)；y1=-y1;；y2=y0\*c;；y2=(y2%a+a)%a;；x2=(c-b\*y2)/a;；if(x2<0)；x2=-x2;；if(x1+y

b /= gcd;

c /= gcd;

ExGCD(a,b,d,x0,y0);

LL x,y,x1,y1,x2,y2;

x1 = x0\*c;

x1 = (x1%b + b)%b;

y1 = (c - a\*x1)/b;

if(y1 < 0)

y1 = -y1;

y2 = y0\*c;

y2 = (y2%a + a)%a;

x2 = (c - b\*y2)/a;

if(x2 < 0)

x2 = -x2;

if(x1+y1 < x2+y2)

x = x1,y = y1;

else

x = x2,y = y2;

cout << x << ' ' << y << endl;

}

return 0;

}

整数快速幂

LL QuickMod(LL a,LL b,LL m) //a^b % m

{

LL ans = 1%m;

a %= m;

while(b > 0)

{

if(b&1)

ans = ans\*a%m;

a = a\*a%m;

b >>= 1;

}

return ans;

}

矩阵快速幂

const int MAXN = 110;

struct Matrax

{

int m[MAXN][MAXN];

}a,per;

int N,M;

void Init()

{

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

for(int j = 0; j < N; ++j)

{

scanf("%d",&a.m[i][j]);

a.m[i][j] %= M;

per.m[i][j] = (i == j);

}

}

Matrax Multi(Matrax a,Matrax b) //a \* b

{

Matrax c;

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

for(int j = 0; j < N; ++j)

{

c.m[i][j] = 0;

for(int k = 0; k < N; ++k)

c.m[i][j] += a.m[i][k]\*b.m[k][j];

c.m[i][j] %= M;

}

return c;

}

Matrax Power(int k) //a^k % M

{

Matrax c,p,ans = per;

p = a;

while(k)

{

if(k&1)

ans = Multi(ans,p);

p = Multi(p,p);

k >>= 1;

}

return ans;

}

Matrax Add(Matrax a,Matrax b) // a + b

{

Matrax c;

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

for(int j = 0; j < N; ++j)

c.m[i][j] = (a.m[i][j] + b.m[i][j]) % M;

return c;

}

Matrax MatraxSum(int k) //a + a^2 + a^3 + … + a^k

{

if(k == 1)

return a;

Matrax temp,b;

temp = MatraxSum(k/2);

if(k&1)

{

b = Power(k/2+1);

temp = Add(temp,Multi(temp,b));

temp = Add(temp,b);

}

else

{

b = Power(k/2);

temp = Add(temp,Multi(temp,b));

}

return temp;

}

整数分解

试除法整数分解

int factor[11000]; //记录素因子个数

int ct; //记录每项素因子个数factor[0] = 2 表示第一个素因子2的个数为2 void Divide(int N)

{

ct = 0;

for(int i = 2; i <= sqrt(N\*1.0); ++i)

{

while(N % i == 0)

{

factor[ct++] = i;

N /= i;

}

}

if(N != 1) factor[ct++] = N;

}

筛法整数分解

const int MAXN = 11000;

int Prime[MAXN],NPrime,ct;

//记录每项素因子个数factor[0] = 2 表示第一个素因子2的个数为2

//Prime[]存放素数，NPrime为素数个数，ct为素因子个数

bool IsPrime[MAXN];

int factor[11000];//存放素因子个数

void GetPrime()

{

NPrime = 0;

for(int i = 2; i <= MAXN; ++i)

IsPrime[i] = 1;

for(int i = 2; i <= sqrt(MAXN-1.0); ++i)

{

if(IsPrime[i])

{

Prime[NPrime++] = i;

for(int j = i\*i; j < MAXN; j+=i)

IsPrime[j] = 0;

}

}

}

void Divide(int N)

{

int temp = sqrt(N\*1.0);

ct = 0;

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

{

if(Prime[i] > temp)

break;

while(N%Prime[i] == 0)

{

factor[ct++] = Prime[i];

N /= Prime[i];

}

}

if(N != 1) factor[ct++] = N;

}

PollardRho大整数分解

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<math.h>

#define MAX\_VAL (pow(2.0,60))

//miller\_rabbin素性测试

//\_\_int64 mod\_mul(\_\_int64 x,\_\_int64 y,\_\_int64 mo)

//{

// \_\_int64 t;

// x %= mo;

// for(t = 0; y; x = (x<<1)%mo,y>>=1)

// if(y & 1)

// t = (t+x) %mo;

//

// return t;

//}

\_\_int64 mod\_mul(\_\_int64 x,\_\_int64 y,\_\_int64 mo) //x \* y % mo {

\_\_int64 t,T,a,b,c,d,e,f,g,h,v,ans;

T = (\_\_int64)(sqrt(double(mo)+0.5));

t = T\*T - mo;

a = x / T;

b = x % T;

c = y / T;

d = y % T;

e = a\*c / T;

f = a\*c % T;

v = ((a\*d+b\*c)%mo + e\*t) % mo;

g = v / T;

h = v % T;

ans = (((f+g)\*t%mo + b\*d)% mo + h\*T)%mo;

while(ans < 0)

ans += mo;

return ans;

}

\_\_int64 mod\_exp(\_\_int64 num,\_\_int64 t,\_\_int64 mo) //num^t % mo {

\_\_int64 ret = 1, temp = num % mo;

for(; t; t >>=1,temp=mod\_mul(temp,temp,mo))

if(t & 1)

ret = mod\_mul(ret,temp,mo);

return ret;

}

bool miller\_rabbin(\_\_int64 n) //MillerRabbin素数测试 {

if(n == 2)

return true;

if(n < 2 || !(n&1))

return false;

int t = 0;

\_\_int64 a,x,y,u = n-1;

while((u&1)==0)；t++;；u>>=1;；for(inti=0;i<50;i++)；a=rand()%(n-1)+1;；x=mod\_exp(a,u,n);；for(intj=0;j<t;j++)；y=mod\_mul(x,x,n);；if(y==1&&x!=1&am；returnfalse;；x=y

while((u & 1) == 0)

{

t++;

u >>= 1;

}

for(int i = 0; i < 50; i++)

{

a = rand() % (n-1)+1;

x = mod\_exp(a,u,n);

for(int j = 0; j < t; j++)

{

y = mod\_mul(x,x,n);

if(y == 1 && x != 1 && x != n-1)

return false;

x = y;

}

if(x != 1)

return false;

}

return true;

}

//PollarRho大整数因子分解

\_\_int64 minFactor; //最小的素因子

\_\_int64 gcd(\_\_int64 a,\_\_int64 b)

{

if(b == 0)

return a;

return gcd(b, a % b);

}

\_\_int64 PollarRho(\_\_int64 n, int c)

{

int i = 1;

srand(time(NULL));

\_\_int64 x = rand() % n;

\_\_int64 y = x;

int k = 2;

while(true)

{

i++;

x = (mod\_exp(x,2,n) + c) % n;

\_\_int64 d = gcd(y-x,n);

if(1 < d && d < n)

return d;

if(y == x)

return n;

if(i == k)

{

y = x;

k \*= 2;

}

}

}

int Ans[1100],cnt;//Ans来存放素因子，cut用来计数

void getSmallest(\_\_int64 n, int c) //拆分n

{

if(n == 1)

return;

if(miller\_rabbin(n)) //如果n为素数

{

if(n < minFactor) //Ans[cnt++] = n;

minFactor = n;

return;

}

//n不为素数

\_\_int64 val = n;

while(val == n)

val = PollarRho(n,c--); //得到约数val

getSmallest(val,c); //尝试拆分n的约束val

getSmallest(n/val,c); //尝试拆分n的另一个约数n/val

}

欧拉函数

直接欧拉函数

int Euler(int n)

{

int ret = n;

for(int i = 2; i\*i <= n; ++i)

{

if(n % i == 0)

{

n /= i;

ret = ret - ret/i;

}

while(n % i == 0)

n /= i;

}

if(n > 1)

ret = ret - ret/n;

return ret;

}

递推快速求欧拉函数

int prime[100010],phi[1000010];

bool unprime[1000010];

\_\_int64 sum[1000010];

void Euler()

{

int i,j,k = 0;

for(i = 2; i <= 1000000; i++)

{

if(!unprime[i])

{

prime[k++] = i;

phi[i] = i-1;

}

for(j = 0; j < k && prime[j]\*i <= 1000000; j++)

{

unprime[prime[j] \*i] = true;

if(i % prime[j] != 0)

{

phi[prime[j]\*i] = phi[i]\*(prime[j]-1);

}

else

{

phi[prime[j]\*i] = phi[i]\*prime[j];

break;

}

}

}

}

容斥原理

容斥原理：在计数时，必须注意无一重复，无一遗漏。为了使重叠部分不被重复计算，人们研究出一种新的计数方法，这种方法的基本思想是：先不考虑重叠的情况，把包含于某内容中的所有对象的数目先计算出来，然后再把计数时重复计算的数目排斥出去，使得计算的结果既无遗漏又无重复，这种计数的方法称为容斥原理。

LL Q[100010],factor[110],num;

//Q数组存放的就是右边边各项的因子数以及正负情况，factor[]存放对应对象的数目，num为有几个对象

void Divid(LL n) //n的素因子分解,得到每项素因子的个数

{

num = 0;

for(LL i = 2; i\*i <= n; ++i)

{

if(n%i==0)

{

while(n%i==0)

n /= i;

factor[num++] = i;

}

}

if(n != 1)

factor[num++] = n;

}

LL solve(LL n) //容斥定理

{

LL k,t,ans;

t = ans = 0;

Q[t++] = -1;

for(LL i = 0; i < num; ++i)

{

k = t;

for(LL j = 0; j < k; ++j)

Q[t++] = -1\*Q[j]\*factor[i];

}

//A∪B∪C = A+B+C - A∩B - B∩C - C∩A + A∩B∩C

//Q数组存放的就是A∪B∪C右边边各项的因子数以及正负情况。 for(LL i = 1; i < t; ++i)

ans += n/Q[i];

//n/Q[i]累加起来就是A∪B∪C

return ans;

}

母函数

普通母函数

1.根据题目要求得到母函数(生成函数)

2.把第一个括号的表达式的系数赋值到c1中。

3.从第二个括号开始计算每一项乘积。

4.迭代得到最终母函数结果。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

int c1[1100],c2[1100],A[1100];

int main()

{

int T,N,K;

cin >> T;

while(T--)

{

cin >> N >> K;

memset(A,0,sizeof(A));

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

{

int a,b;

cin >> a >> b;

A[a] = b;

}

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

c1[i] = c2[i] = 0;

c1[0] = 1;

for(int i = 1; i <= K; ++i) //第i项表达式(1+x^2+...) {

for(int j = 0; j <= N; ++j)

//前面i个表达式累乘的表达式(前一表达式)里第j个变量 {

/\*如(1+x)(1+x^2)(1+x^3)，j先指示的是1和x的系数，i=2执行完之后变为（1+x+x^2+x^3）(1+x^3)，这时候j应该指示的是合并后的第一个括号的四个变量的系数。\*/

for(int k = 0; j+i\*k <= N && k <= A[i]; ++k) c2[j+i\*k] += c1[j];//c2[]保存结果

}// k表示的是将要计算的表达式的每项。(因为第i个表达式的增量是i，所以k每次增i)。

for(int j = 0; j <= N; ++j)

{

c1[j] = c2[j];//迭代

c2[j] = 0;

}

}

cout << c1[N] << endl;

}

return 0;

}

指数型母函数

#include<iostream>；#include<algorithm>；#include<cstdio>；#include<cstring>；usingnamespacestd;；doublef[]={1,1,2,6,24,12；intmain()；intN,M;；while(cin>>N>

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

double f[] = {1,1,2,6,24,120,720,5040,40320,362880,3628800}; double num[11],c1[11],c2[11];

int main()

{

int N,M;

while(cin >> N >> M)

{

memset(c1,0,sizeof(c1));

memset(c2,0,sizeof(c2));

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

cin >> num[i];

for(int i = 0; i <= num[0]; ++i)

c1[i] = 1.0/f[i];

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

{

for(int j = 0; j <= M; ++j)

for(int k = 0; k <= num[i] && (k+j)<=M; ++k) c2[k+j] += (c1[j]/f[k]);

for(int j = 0; j <= M; ++j)

{

c1[j] = c2[j];

c2[j] = 0;

}

}

double s = 1.0\*c1[M]\*f[M];

printf("%.0lf\n",s);

}

return 0;

}

其他相关

九余数定理：一个数N各位数字的和，对9取余等于这个数对9取余 给你一个奇数N，求1~N的奇数平方和： S = N\*(N+1)\*(N+2)/6 约瑟夫问题：有N个人，编号为1~N，按顺时针围成一个圈，每数k个人，就将这个人从圈中消除，问：最终只留下一个人的编号。

num = 0;

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

num = (num + m) % i;

快速求约瑟夫问题

#include<iostream>

#include<algorithm>

#include<math.h>

using namespace std;

int main()

{

\_\_int64 n,k,ans;

while(~scanf("%I64d%I64d",&n,&k))

{

\_\_int64 num = 0;

if(k==1)

num = n-1;

else if(n == 1)

num = 0;

else

{

for(\_\_int64 i = 2; i <= n;)

{

if(num + k < i)

{

\_\_int64 temp =

(i-1-num)%(k-1)?(i-1-num)/(k-1):(i-1-num)/(k-1)-1;

if(i + temp > n)

{

num += (n-(i-1))\*k;//n-(i-1)次

break;

}

num += temp\*k;

i += temp;

}

else

{

num = (num + k)%i;

++i;

}

}

num %= n;

}

printf("%I64d\n",num+1);

}

return 0;

}

模拟求约瑟夫问题

#include<stdio.h>

#include<string.h>

#include<vector>

#include<iostream>

#include<algorithm>

using namespace std;

vector<int> v;

int main()

{

int n,k;

while(~scanf("%d%d",&n,&k) &&(n||k))

{

v.clear();

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

v.push\_back(i);

int m = (k-1)%v.size(),t;

while(v.size() > 1)

{

t = (m + (k-1)) % (v.size()-1);

if(t >= m)

t = (t+1) % v.size();

else

t = t % v.size();

v[m] = v[t];

v.erase(v.begin()+t);

if(t >= m)

m = (m + k)%v.size();

else

m = (m + k-1)%v.size();

}

printf("%d\n",(n-(v.front()-1))%n+1);

}

return 0;

}

给你整数x和y的和以及x和y的积，是否能找到满足这两个式子的整数x和整数y。

满足if(num >= 0 && (int)sqrt(num\*1.0)\*(int)sqrt(num\*1.0) == num && (int)(n + sqrt(num\*1.0))%2==0 && (int)(n - sqrt(num\*1.0)) %2==0) Fibonacci数列大于40前四位数字

1/sqrt(5.0) + N\*log10(1+(sqrt(5.0))/2.0)

小于N的素数个数为π(N),π(N)的位数是多少？

double M = double(N) - log10(N) - log10(log(10));

cout << (int)M + 1 << endl;

Stirling公式：当N足够大时，N! = (N/e) \* N \* sqrt(2\*pi\*N)。 有N张卡，将N张卡分成若干不同的集合，集合不能为空。问：总共有多少种分法。

for(int i = 1; i <= 2000; ++i)

stir[i][1] = stir[i][i] = 1;

for(int i = 2; i <= 2000; ++i)

for(int j = 2; j <= i; ++j)

stir[i][j] = (stir[i-1][j-1] + stir[i-1][j]\*j)%1000; for(int i = 1; i <= 2000; ++i)

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

Bell[i] = (Bell[i] + stir[i][j]) % 1000; //Bell[N]为所求

字符串

KMP

当字符串匹配失败时，模式串的指针并没有指向0从头比较，而是指向了一个特定的位置，因为这个Next[j]指向的位置pos前长度为Next[pos]的子串，同模式串第j位前的长度为Next[j]的子串是相同的。

即S[0]~S[Next[j]]一定与S[len-1-Next[j]]~S[j-1]匹配。

1.既能做前缀又能做后缀的子串长度

ans[0] = len;

int id = 1,i = len;

while(Next[i] > 0)

{

ans[id++] = Next[i];

i = Next[i];

}

2.求最短子串长度

给定的字符串s，最短的重复子串a是s[Next[len]] s[Next[len+1]] … s[len-1]。

当len % Next[len] = 0时，说明字符串s有不是它本身重复子串a，最大的n为len/Next[len]。当len % Next[len] != 0时，除了s，没有重复子串a满足要求，这时n = 1。

if(len % (len-Next[len]) == 0)

cout << len/(len-Next[len]) << endl;

else

cout << 1 << endl;

3.求模式串pat在主串str中匹配次数

可重叠匹配：当一次匹配成功后，j继续回退到Next[j]向后进行匹配，直到主串str的末尾。

不可重叠匹配：当一次匹配成功后，j继续回退到0向后进行匹配，直到主串str的末尾。

4.求N行M列的字符矩阵，最小的字符子矩阵的面积

每一行s[i]来说，M-Next[M]是最小的，对于每一列s[j]来说，N-Next[N]是最小的。

分别求每一行的最小公倍数lcmn和每一列的最小公倍数lcmm。最后结果就是lcmn\*lcmm。

5.求字符串s的循环前缀的长度和循环的次数

对于当前前缀S[1]~S[i]，如果i % (i-Next[i]) == 0，则i - Next[i]就是最小重复子串，即循环节长度为i - Next[i]。循环次数为i / (i - Next[i])。 char str[1000010],pat[1000010];//pat为模式串，str为主串

int Next[1000010]; //Next[x]下标x表示匹配失败处字符下标

//模式串pat的前缀与x位置的后缀的最大匹配字符个数-1

void GetNext(char \*pat)

intLenPat=strlen(pat);；inti=0,j=-1;；Next[0]=-1;；while(i<LenPat)；if(j==-1||pat[i]==pat[j]；i++,j++;；Next[i]=j;；else；j=Next[j];；intKMP()//返回模式串pat在str中第；intLenStr=strlen(str);；intL

{

int LenPat = strlen(pat);

int i = 0,j = -1;

Next[0] = -1;

while(i < LenPat)

{

if(j == -1 || pat[i] == pat[j])

{

i++,j++;

Next[i] = j;

}

else

j = Next[j];

}

}

int KMP()//返回模式串pat在str中第一次出现的位置

{

int LenStr = strlen(str);

int LenPat = strlen(pat);

GetNext(pat);

int i = 0,j = 0;

int ans = 0;//计算模式串在主串匹配次数

while(i < LenStr)

{

if(j == -1 || str[i] == pat[j])

i++,j++;

else

j = Next[j];

if(j == LenPat)

{

//ans++; ans存放匹配次数，去掉return,最后返回ans return i - LenPat + 1;

}

}

return -1;//没找到匹配位置

//return ans;//返回匹配次数。

}

字典树

struct TrieNode

{

int Count; //前缀单词出现次数

struct TrieNode\* Next[26]; //26个字母节点

}Tree,\*Trie;

TrieNode \*root;

void Create() //初始化

{

root = new TrieNode;

memset(root->Next,NULL,sizeof(root->Next));

root->Count = 0;

}

void Insert(char \*s) //插入字符串s[]，O(M\*L)

{

TrieNode \*p, \*q;

p = root;

while(\*s)

{

if(p->Next[\*s-'a'] == NULL)

{

q = new TrieNode;

memset(q->Next,NULL,sizeof(q->Next));

q->Count = 1;

p->Next[\*s-'a'] = q;

}

else

p->Next[\*s-'a']->Count++;

p = p->Next[\*s-'a'];

s++;

}

}

int Find(char \*s) //查找字符串s[] O(N\*L)

{

TrieNode \*p, \*q;

p = root;

while(\*s)

{

if(p->Next[\*s-'a'] == NULL)

return 0;

p = p->Next[\*s-'a'];

s++;

}

return p->Count;

}

int Dele(TrieNode \*p)

{

if(p == NULL) return 0;

for(int i = 0; i < 26; ++i)

if(p->Next[i] != NULL) Dele(p->Next[i]);

free(p);

}

数据结构

并查集

# include<stdio.h>

# include<string.h>

int father[500010];

int find(int x)

{

if (father[x]!=x) father[x]=find(father[x]);

return father[x];

}

int main()

{

int n,m,p;

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

for (int i=1;i<=n;i++) //初始化集合

father[i]=i;

for (int i=1;i<=m;i++) //合并集合

{

scanf("%d%d",&a,&b);

a=find(a);

b=find(b);

father[a]=b;

}

for(int i=1;i<=p;i++) //询问关系

{

scanf("%d%d",&a,&b);

a=find(a);

b=find(b);

if(a==b) printf("Yes");

else printf("No");

}

return 0;

}

树状数组

Tree[N] = A[N-2^k+1] + … + A[N]

单点更新，区间求值：树状数组代表区间的和。

单点更新，将第i点的值加上v：Update(i,v)

区间求值，求区间(u，v)的和：Query(v) - Query(u-1)

const int MAXN = 100010;

int N,Tree[MAXN];

int Lowbit(int i)

{

return i & (-i);

}

void Update(int i,int x)

{

while(i <= N)

{

Tree[i] = Tree[i] + x;

i += Lowbit(i);

}

}

int Query(int N)

{

int sum = 0;

while(N > 0)

{

sum += Tree[N];

N -= Lowbit(N);

}

return sum;

}

//初始化

memset(Tree,0,sizeof(Tree));

scanf("%d",&N);

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

{

int x;

scanf("%d",&x);

Update(i,x);

}

区间更新，单点求值：树状数组代表单个元素的变化

区间更新，将区间(u，v)的元素增加d：Update(u,d),Update(v+1,-d) 单点求值，查找第i个元素的值：Query(i)

const int MAXN = 100010;

int N,Tree[MAXN];

int Lowbit(int i)

{

return i & (-i);

}

void Updata(int i,int x)

{

while(i <= N)

{

Tree[i] = Tree[i] + x;

i = i + Lowbit(i);

}

}

int Query(int n)

{

int sum = 0;

while(n > 0)

{

sum += Tree[n];

n = n - Lowbit(n);

}

return sum;

}

求逆序数： 数组Tree[i]表示数字i是否在序列中出现过，如果数字i已经存在于序列中，Tree[i] = 1，否则Tree[i] = 0。按序列从左到右将值为a的元素当作下标为a，赋值为1插入树状数组里，这时，比a的数个数就是i - Query(a)。将全部结果累加起来就是逆序数了。

const int MAXN = 100010;

int N,Tree[MAXN];

int Lowbit(int i)

{

return i & (-i);

}

void Update(int i,int x)

{

while(i <= N)

{

Tree[i] = Tree[i] + x;

i = i + Lowbit(i);

}

}

int Query(int n)

{

int sum = 0;

while(n > 0)

{

sum += Tree[n];

n = n - Lowbit(n);

returnsum;；//求逆序数：；memset(Tree,0,sizeof(Tre；for(inti=1;i<=N;++i)；cin>>a;；Update(a,1);；ans+=i-Query(a);；二维树状数组：；单点更新，将坐标(x，y)上的元素增加d：Upd；区间求值，求左上角(x2，y2)到右下角(x1，；constintMA

}

return sum;

}

//求逆序数：

memset(Tree,0,sizeof(Tree));

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

{

cin >> a;

Update(a,1);

ans += i - Query(a);

}

二维树状数组：

单点更新，将坐标(x，y)上的元素增加d：Update(x,y,d);

区间求值，求左上角(x2，y2)到右下角(x1，y2)的和： Query(x1,y1) - Query(x1,y2-1) - Query(x2-1,y1) + Query(x2-1,y2-1)

const int MAXN = 1010;

int Tree[MAXN][MAXN],N;

bool Mark[MAXN][MAXN];

int Lowbit(int i)

{

return i & (-i);

}

void Update(int x,int y,int num)

{

for(int i = x; i <= MAXN; i += Lowbit(i))

for(int j = y; j <= MAXN; j += Lowbit(j))

Tree[i][j] += num;

}

int Query(int x,int y)

{

int sum = 0;

for(int i = x; i > 0; i -= Lowbit(i))

for(int j = y; j > 0; j -= Lowbit(j))

sum += Tree[i][j];

return sum;

}

线段树

maxn是题目给的最大区间,而节点数要开4倍,确切的来说节点数要开大于maxn的最小2x的两倍

lson和rson分辨表示结点的左儿子和右儿子,由于每次传参数的时候都固定是这几个变量,所以可以用预定于比较方便的表示

以前的写法是另外开两个个数组记录每个结点所表示的区间,其实这个区间不必

保存,一边算一边传下去就行,只需要写函数的时候多两个参数,结合lson和rson的预定义可以很方便

PushUP(int rt)是把当前结点的信息更新到父结点

PushDown(int rt)是把当前结点的信息更新给儿子结点

rt表示当前子树的根(root),也就是当前所在的结点

整理这些题目后我觉得线段树的题目整体上可以分成以下四个部分: 单点更新

最最基础的线段树,只更新叶子节点,然后把信息用PushUP(int r)这个函数更新上来

hdu1166 敌兵布阵 题意:O(-1) 思路:O(-1)

线段树功能:update:单点增减 query:区间求和

# include<stdio.h>

# include<iostream>

# include<algorithm>

using namespace std;

const int INF = 0xffffff0;

const int MAXN = 50010;

int sum[MAXN<<2];

void pushup(int root)

{

sum[root] = sum[root<<1] + sum[root<<1|1];

}

void build(int root,int L,int R)

{

if(L == R)

{

scanf("%d",&sum[root]);

return;

}

int mid = (L+R)>>1;

build(root<<1,L,mid);

build(root<<1|1,mid+1,R);

pushup(root);

}

void update(int root,int L,int R,int i,int v)

{

if(L==R)

{

sum[root] += v;

return ;

}

int mid = (L+R)>>1;

if(i <= mid) update(root<<1,L,mid,i,v);

else update(root<<1|1,mid+1,R,i,v);

pushup(root);

}

int query(int root,int L,int R,int s,int e)

{

if(s==L && e==R) return sum[root];

int mid = (L+R)>>1;

int res = 0;

if(e <= mid) res += query(root<<1,L,mid,s,e);

else if(s > mid) res += query(root<<1|1,mid+1,R,s,e); else

{

res += query(root<<1,L,mid,s,mid);

res += query(root<<1|1,mid+1,R,mid+1,e);

}

return res;

}

int main()

{

int T, N, a, b, kase = 1;

char cmp[10];

scanf("%d", &T);

while(T--)

{

printf("Case %d:\n",kase++);

scanf("%d",&N);

build(1,1,N);

while(scanf("%s",cmp) && cmp[0]!='E')

{

scanf("%d%d",&a, &b);

if(cmp[0]=='Q')

printf("%d\n",query(1,1,N,a,b));

else if(cmp[0]=='A')

update(1,1,N,a,b);

else if(cmp[0]=='S')

update(1,1,N,a,-b);

}

}

return 0;

}

hdu1754 I Hate It 题意:O(-1) 思路:O(-1)

线段树功能:update:单点替换 query:区间最值

# include<stdio.h>

# include<iostream>

# include<algorithm>

using namespace std;

const int INF = 0xffffff0;

const int MAXN = 200010;

int MAX[MAXN<<2];

void pushup(int root)

{

MAX[root] = max(MAX[root<<1],MAX[root<<1|1]);

}

void build(int root,int L,int R)

{

if(L==R)

{

scanf("%d",&MAX[root]);

return;

}

int mid = (L+R)>>1;

build(root<<1,L,mid);

build(root<<1|1,mid+1,R);

pushup(root);

}

void updata(int root,int L,int R,int i,int v)

{

if(L==R)

{

MAX[root] = v;

return;

}

int mid = (L+R)>>1;

if(i<=mid) updata(root<<1,L,mid,i,v);

else updata(root<<1|1,mid+1,R,i,v);

pushup(root);

}

int query(int root,int L,int R,int s,int e)

{

if(s==L && e==R) return MAX[root];

int mid = (L+R)>>1;

int res = 0;

if(e <= mid) res = max(res,query(root<<1,L,mid,s,e));

else if(s > mid) res = max(res,query(root<<1|1,mid+1,R,s,e)); else

{

res = max(res,query(root<<1,L,mid,s,mid));

res = max(res,query(root<<1|1,mid+1,R,mid+1,e));

}

return res;

}

int main()

{

int N,M,a,b;

char cmp[3];

while(~scanf("%d%d",&N,&M))

{

build(1,1,N);

while(M--)

{

scanf("%s%d%d",cmp,&a,&b);

if(cmp[0]=='Q')

printf("%d\n",query(1,1,N,a,b));

else if(cmp[0]=='U')

updata(1,1,N,a,b);

}

}

return 0;

}

hdu1394 Minimum Inversion Number 题意:求Inversion后的最小逆序数 思路:用O(nlogn)复杂度求出最初逆序数后,就可以用O(1)的复杂度分别递推出其他解

线段树功能:update:单点增减 query:区间求和

# include<iostream>

# include<algorithm>

using namespace std;

const int MAXN = 5010;

int sum[MAXN<<2];

void pushup(int root)

{

sum[root] = sum[root<<1] + sum[root<<1|1];

}

void build(int root,int L,int R)

{

sum[root] = 0;

if(L==R)

return;

int mid = (L+R)>>1;

build(root<<1,L,mid);

build(root<<1|1,mid+1,R);

voidupdata(introot,intL,；if(L==R)；sum[root]++;；return;；intmid=(L+R)>>1;；if(i<=mid)updata(root；elseupdata(root<<1；pushup(root);；intquery(introot,intL,in；if(s==L&&

}

void updata(int root,int L,int R,int i,int v)

{

if(L==R)

{

sum[root]++;

return;

}

int mid = (L+R)>>1;

if(i <= mid) updata(root<<1,L,mid,i,v);

else updata(root<<1|1,mid+1,R,i,v);

pushup(root);

}

int query(int root,int L,int R,int s,int e)

{

if(s==L && e==R)

return sum[root];

int mid = (L+R)>>1;

int res = 0;

if(e <= mid) res += query(root<<1,L,mid,s,e);

else if(s > mid) res += query(root<<1|1,mid+1,R,s,e);

else

{

res += query(root<<1,L,mid,s,mid);

res += query(root<<1|1,mid+1,R,mid+1,e);

}

return res;

}

int x[MAXN];

int main()

{

int n;

while(~scanf("%d", &n))

{

build(1,0,n-1);

int sum = 0;

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

{

scanf("%d", &x[i]);

sum += query(1,0,n-1,x[i],n-1);

updata(1,0,n-1,x[i],1);

}

int res = sum;

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

{

sum += n - x[i] -x[i] -1;

res = min(res, sum);

}

printf("%d\n",res);

}

return 0;

}

hdu2795 Billboard

题意:h\*w的木板,放进一些1\*L的物品,求每次放空间能容纳且最上边的位子 思路:每次找到最大值的位子,然后减去L

线段树功能:query:区间求最大值的位子(直接把update的操作在query里做了) # include<stdio.h>

# include<iostream>

# include<algorithm>

using namespace std;

const int MAXN = 200010;

int MAX[MAXN<<2];

void pushup(int root)

{

MAX[root] = max(MAX[root<<1],MAX[root<<1|1]);

}

void build(int root,int L,int R,int w)

{

MAX[root] = w;

if(L==R)

{

return;

}

int mid = (L+R)>>1;

build(root<<1,L,mid,w);

build(root<<1|1,mid+1,R,w);

pushup(root);

}

int query(int root,int L,int R,int v)

{

if(L==R)

{

MAX[root] -= v;

return L;

}

int mid = (L+R)>>1;

int res;

if(MAX[root<<1] >= v) res = query(root<<1,L,mid,v);

else res = query(root<<1|1,mid+1,R,v);

pushup(root);

return res;

}

int main()

{

int w,h,n,a;

while(~scanf("%d%d%d",&h,&w,&n))

{

if(h > n)

h = n;

build(1,1,h,w);

while(n--)

{

scanf("%d", &a);

if(MAX[1] < a)

printf("-1\n");

else

printf("%d\n",query(1,1,h,a));

}

}

return 0;

}

成段更新

(通常这对初学者来说是一道坎),需要用到延迟标记(或者说懒惰标记),简单来说就是每次更新的时候不要更新到底,用延迟标记使得更新延迟到下次需要更新or询问到的时候

hdu1698 Just a Hook 题意:O(-1) 思路:O(-1)

线段树功能:update:成段替换 (由于只query一次总区间,所以可以直接输出1结点的信息)

# include<stdio.h>

# include<iostream>

const int MAXN = 100010;

int sum[MAXN<<2],col[MAXN<<2];

void pushup(int root)

{

sum[root] = sum[root<<1] + sum[root<<1|1];

}

void pushdown(int root,int mid)

{

if(col[root])

{

col[root<<1] = col[root<<1|1] = col[root]; sum[root<<1] = col[root]\*(mid-(mid>>1));

sum[root<<1|1] = col[root]\*(mid>>1);

col[root] = 0;

}

}

void build(int root,int L,int R)

{

col[root] = 0;

sum[root] = 0;

if(L == R)

{

sum[root] = 1;

return;

}

int mid = (L+R)>>1;

build(root<<1,L,mid);

build(root<<1|1,mid+1,R);

pushup(root);

}

void updata(int root,int L,int R,int s,int e,int c) {

if(s==L && e==R)

{

col[root] = c;

sum[root] = c\*(R-L+1);

return;

}

pushdown(root,R-L+1);

int mid = (L+R)>>1;

if(e <= mid) updata(root<<1,L,mid,s,e,c);

else if(s > mid) updata(root<<1|1,mid+1,R,s,e,c); else

{

updata(root<<1,L,mid,s,mid,c);

updata(root<<1|1,mid+1,R,mid+1,e,c);

}

pushup(root);

}

int main()

{

int T,kase = 1;

scanf("%d", &T);

while(T--)

{

int N,Q;

scanf("%d%d", &N, &Q);

build(1,1,N);

while(Q--)

{

int s,e,c;

scanf("%d%d%d",&s,&e,&c);

updata(1,1,N,s,e,c);

}

printf("Case %d: The total value of the hook is %d.\n",kase++,sum[1]);

}

return 0;

}

poj3468 A Simple Problem with Integers 题意:O(-1) 思路:O(-1) 线段树功能:update:成段增减 query:区间求和

# include<stdio.h>

# include<iostream>

# include<algorithm>

using namespace std;

const int MAXN = 100010;

\_\_int64 sum[MAXN<<2],add[MAXN<<2];

void pushup(int root)

{

sum[root] = sum[root<<1] + sum[root<<1|1];

}

void pushdown(int root,int len)

{

if(add[root])

{

add[root<<1] += add[root];

add[root<<1|1] += add[root];

sum[root<<1] += (add[root]\*(len-(len>>1))); sum[root<<1|1] += (add[root]\*(len>>1));

add[root] = 0;

}

}

void build(int root,int L,int R)

{

add[root]=0;；if(L==R)；scanf("%I64d",；return;；intmid=(L+R)>>1;；build(root<<1,L,mi；build(root<<1|1,mi；pushup(root);；voidupdata(introot,intL,；if(s<=L&am

add[root] = 0;

if(L == R)

{

scanf("%I64d", &sum[root]);

return;

}

int mid = (L+R)>>1;

build(root<<1,L,mid);

build(root<<1|1,mid+1,R);

pushup(root);

}

void updata(int root,int L,int R,int s,int e,\_\_int64 v)

{

if(s<=L && e>=R)

{

add[root] += v;

sum[root] += (\_\_int64)(v\*(R-L+1));

return;

}

pushdown(root,R-L+1);

int mid = (L+R)>>1;

if(s <= mid) updata(root<<1,L,mid,s,e,v);

if(e > mid) updata(root<<1|1,mid+1,R,s,e,v);

pushup(root);

}

\_\_int64 query(int root,int L,int R,int s,int e)

{

if(s<=L && e>=R) return sum[root];

pushdown(root,R-L+1);

\_\_int64 res = 0;

int mid = (L+R)>>1;

if(s <= mid) res += query(root<<1,L,mid,s,e);

if(e > mid) res += query(root<<1|1,mid+1,R,s,e);

return res;

}

int main()

{

int N,Q,s,e,a,b;

\_\_int64 v;

char cmd[10];

scanf("%d%d", &N,&Q);

build(1,1,N);

while(Q--)

{

scanf("%s",cmd);

if(cmd[0]=='Q')

{

scanf("%d%d",&s,&e);

printf("%I64d\n",query(1,1,N,s,e));

}

else if(cmd[0]=='C')

{

scanf("%d%d%I64d",&a,&b,&v);

updata(1,1,N,a,b,v);

}

}

return 0;

}

poj2528 Mayor’s posters

题意:在墙上贴海报,海报可以互相覆盖,问最后可以看见几张海报

思路:这题数据范围很大,直接搞超时+超内存,需要离散化:

离散化简单的来说就是只取我们需要的值来用,比如说区间[1000,2000]

[1990,2012] 我们用不到[-∞,999] [1001,1989] [1991,1999] [2001,2011]

[2013,+∞]这些值,所以我只需要1000,1990,2000,2012就够了,将其分别映射到0,1,2,3,在于复杂度就大大的降下来了，所以离散化要保存所有需要用到的值,排序后,分别映射到1~n,这样复杂度就会小很多很多。而这题的难点在于每个数字其实表示的是一个单位长度(并且一个点),这样普通的离散化会造成许多错误(包括我以前的代码,poj这题数据奇弱)

给出下面两个简单的例子应该能体现普通离散化的缺陷:

1-10 1-4 5-10

1-10 1-4 6-10

为了解决这种缺陷,我们可以在排序后的数组上加些处理,比如说[1,2,6,10] 如果相邻数字间距大于1的话,在其中加上任意一个数字,比如加成

[1,2,3,6,7,10],然后再做线段树就好了.

线段树功能:update:成段替换 query:简单hash

#include <cstdio>

#include <cstring>

#include <algorithm>

using namespace std;

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

const int maxn = 11111;

bool hash[maxn];

int li[maxn] , ri[maxn];

int X[maxn\*3];

int col[maxn<<4];

int cnt;

void PushDown(int rt) {

if (col[rt] != -1) {

col[rt<<1] = col[rt<<1|1] = col[rt];

col[rt] = -1;

}

}

void update(int L,int R,int c,int l,int r,int rt) {

if (L <= l && r <= R) {

col[rt] = c;

return ;

}

PushDown(rt);

int m = (l + r) >> 1;

if (L <= m) update(L , R , c , lson);

if (m < R) update(L , R , c , rson);

}

void query(int l,int r,int rt) {

if (col[rt] != -1) {

if (!hash[col[rt]]) cnt ++;

hash[ col[rt] ] = true;

return ;

}

if (l == r) return ;

int m = (l + r) >> 1;

query(lson);

query(rson);

}

int Bin(int key,int n,int X[]) {

int l = 0 , r = n - 1;

while (l <= r) {

int m = (l + r) >> 1;

if (X[m] == key) return m;

if (X[m] < key) l = m + 1;

else r = m - 1;

}

return -1;

}

int main() {

int T , n;

scanf("%d",&T);

while (T --) {

scanf("%d",&n);

int nn = 0;

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

scanf("%d%d",&li[i] , &ri[i]);

X[nn++] = li[i];

X[nn++] = ri[i];

}

sort(X , X + nn);

int m = 1;

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

if (X[i] != X[i-1]) X[m ++] = X[i];

}

for (int i = m - 1 ; i > 0 ; i --) {

if (X[i] != X[i-1] + 1) X[m ++] = X[i] + 1; }

sort(X , X + m);

memset(col , -1 , sizeof(col));

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

int l = Bin(li[i] , m , X);

int r = Bin(ri[i] , m , X);

update(l , r , i , 0 , m , 1);

}

cnt = 0;

memset(hash , false , sizeof(hash));

query(0 , m , 1);

printf("%d\n",cnt);

}

return 0;

}

poj3225 Help with Intervals 题意:区间操作,交,并,补等

思路:我们一个一个操作来分析:(用0和1表示是否包含区间,-1表示该区间内既有包含又有不包含)

U:把区间[l,r]覆盖成1

I:把[-∞,l)(r,∞]覆盖成0

D:把区间[l,r]覆盖成0

C:把[-∞,l)(r,∞]覆盖成0 , 且[l,r]区间0/1互换

S:[l,r]区间0/1互换

成段覆盖的操作很简单,比较特殊的就是区间0/1互换这个操作,我们可以称之为异或操作

很明显我们可以知道这个性质:当一个区间被覆盖后,不管之前有没有异或标记都没有意义了。所以当一个节点得到覆盖标记时把异或标记清空。而当一个节点得到异或标记的时候,先判断覆盖标记,如果是0或1,直接改变一下覆盖标记,不然的话改变异或标记。开区间闭区间只要数字乘以2就可以处理(偶数表示端点,奇数表示两端点间的区间)

线段树功能:update:成段替换,区间异或 query:简单hash

#include <cstdio>

#include <cstring>

#include <cctype>

#include <algorithm>

using namespace std;

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

const int maxn = 131072;

bool hash[maxn];

int cover[maxn<<2];

int XOR[maxn<<2];

void FXOR(int rt) {

if (cover[rt] != -1) cover[rt] ^= 1;

else XOR[rt] ^= 1;

}

void PushDown(int rt) {

if (cover[rt] != -1) {

cover[rt<<1] = cover[rt<<1|1] = cover[rt]; XOR[rt<<1] = XOR[rt<<1|1] = 0;

cover[rt] = -1;

}

if (XOR[rt]) {

FXOR(rt<<1);

FXOR(rt<<1|1);

XOR[rt] = 0;

}

}

void update(char op,int L,int R,int l,int r,int rt) { if (L <= l && r <= R) {

if (op == 'U') {

cover[rt] = 1;

XOR[rt] = 0;

} else if (op == 'D') {

cover[rt] = 0;

XOR[rt] = 0;

} else if (op == 'C' || op == 'S') { FXOR(rt);

}

return ;

}

PushDown(rt);

int m = (l + r) >> 1;

if (L <= m) update(op , L , R , lson);

else if (op == 'I' || op == 'C') {

XOR[rt<<1] = cover[rt<<1] = 0;

}

if (m < R) update(op , L , R , rson);

elseif(op=='I'||；XOR[rt<<1|1]=cover；voidquery(intl,intr,intr；if(cover[rt]==1){；for(intit=l;it<=r;it+；hash[it]=true;；return;；}elseif(cover[rt]==0)ret；if(l==r)return

else if (op == 'I' || op == 'C') {

XOR[rt<<1|1] = cover[rt<<1|1] = 0;

}

}

void query(int l,int r,int rt) {

if (cover[rt] == 1) {

for (int it = l ; it <= r ; it ++) {

hash[it] = true;

}

return ;

} else if (cover[rt] == 0) return ;

if (l == r) return ;

PushDown(rt);

int m = (l + r) >> 1;

query(lson);

query(rson);

}

int main() {

cover[1] = XOR[1] = 0;

char op , l , r;

int a , b;

while ( ~scanf("%c %c%d,%d%c\n",&op , &l , &a , &b , &r) ) { a <<= 1 , b <<= 1;

if (l == '(') a ++;

if (r == ')') b --;

if (a > b) {

if (op == 'C' || op == 'I') {

cover[1] = XOR[1] = 0;

}

} else update(op , a , b , 0 , maxn , 1);

}

query(0 , maxn , 1);

bool flag = false;

int s = -1 , e;

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

if (hash[i]) {

if (s == -1) s = i;

e = i;

} else {

if (s != -1) {

if (flag) printf(" ");

flag = true;

printf("%c%d,%d%c",s&1?'(':'[' , s>>1 , (e+1)>>1 , e&1?')':']');

s = -1;

}

}

}

if (!flag) printf("empty set");

puts("");

return 0;

}

区间合并

这类题目会询问区间中满足条件的连续最长区间,所以PushUp的时候需要对左右儿子的区间进行合并

poj3667 Hotel

题意:1 a:询问是不是有连续长度为a的空房间,有的话住进最左边

2 a b:将[a,a+b-1]的房间清空

思路:记录区间中最长的空房间

线段树操作:update:区间替换 query:询问满足条件的最左断点

#include <cstdio>

#include <cstring>

#include <cctype>

#include <algorithm>

using namespace std;

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

const int maxn = 55555;

int lsum[maxn<<2] , rsum[maxn<<2] , msum[maxn<<2];

int cover[maxn<<2];

void PushDown(int rt,int m) {

if (cover[rt] != -1) {

cover[rt<<1] = cover[rt<<1|1] = cover[rt];

msum[rt<<1] = lsum[rt<<1] = rsum[rt<<1] = cover[rt] ? 0 : m - (m >> 1);

msum[rt<<1|1] = lsum[rt<<1|1] = rsum[rt<<1|1] = cover[rt] ? 0 : (m >> 1);

cover[rt] = -1;

}

}

void PushUp(int rt,int m) {

lsum[rt] = lsum[rt<<1];

rsum[rt] = rsum[rt<<1|1];

if (lsum[rt] == m - (m >> 1)) lsum[rt] += lsum[rt<<1|1]; if (rsum[rt] == (m >> 1)) rsum[rt] += rsum[rt<<1];

msum[rt] = max(lsum[rt<<1|1] + rsum[rt<<1] , max(msum[rt<<1] , msum[rt<<1|1]));

}

void build(int l,int r,int rt) {

msum[rt] = lsum[rt] = rsum[rt] = r - l + 1;

cover[rt] = -1;

if (l == r) return ;

int m = (l + r) >> 1;

build(lson);

build(rson);

}

void update(int L,int R,int c,int l,int r,int rt) {

if (L <= l && r <= R) {

msum[rt] = lsum[rt] = rsum[rt] = c ? 0 : r - l + 1; cover[rt] = c;

return ;

}

PushDown(rt , r - l + 1);

int m = (l + r) >> 1;

if (L <= m) update(L , R , c , lson);

if (m < R) update(L , R , c , rson);

PushUp(rt , r - l + 1);

}

int query(int w,int l,int r,int rt) {

if (l == r) return l;

PushDown(rt , r - l + 1);

int m = (l + r) >> 1;

if (msum[rt<<1] >= w) return query(w , lson);

else if (rsum[rt<<1] + lsum[rt<<1|1] >= w) return m - rsum[rt<<1] + 1;

return query(w , rson);

}

int main() {

int n , m;

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

build(1 , n , 1);

while (m --) {

int op , a , b;

scanf("%d",&op);

if (op == 1) {

scanf("%d",&a);

if (msum[1] < a) puts("0");

else {

int p = query(a , 1 , n , 1);

printf("%d\n",p);

update(p , p + a - 1 , 1 , 1 , n , 1);

}

} else {

scanf("%d%d",&a,&b);

update(a , a + b - 1 , 0 , 1 , n , 1);

}

}

return 0;

}

扫描线

这类题目需要将一些操作排序,然后从左到右用一根扫描线(当然是在我们脑子里)扫过去

最典型的就是矩形面积并,周长并等题

hdu1542 Atlantis

题意:矩形面积并

思路:浮点数先要离散化;然后把矩形分成两条边,上边和下边,对横轴建树,然后从下到上扫描上去,用cnt表示该区间下边比上边多几个

线段树操作:update:区间增减 query:直接取根节点的值

#include <cstdio>

#include <cstring>

#include <cctype>

#include <algorithm>

using namespace std;

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

const int maxn = 2222;

int cnt[maxn << 2];

double sum[maxn << 2];

double X[maxn];

struct Seg {

double h , l , r;

int s;

Seg(){}

Seg(double a,double b,double c,int d) : l(a) , r(b) , h(c) , s(d) {}

bool operator < (const Seg &cmp) const {

return h < cmp.h;

}

}ss[maxn];

void PushUp(int rt,int l,int r) {

if (cnt[rt]) sum[rt] = X[r+1] - X[l];

else if (l == r) sum[rt] = 0;

else sum[rt] = sum[rt<<1] + sum[rt<<1|1];

}

void update(int L,int R,int c,int l,int r,int rt) { if (L <= l && r <= R) {

cnt[rt] += c;

PushUp(rt , l , r);

return ;

}

int m = (l + r) >> 1;

if (L <= m) update(L , R , c , lson);

if (m < R) update(L , R , c , rson);

PushUp(rt , l , r);

}

int Bin(double key,int n,double X[]) {

int l = 0 , r = n - 1;

while (l <= r) {

int m = (l + r) >> 1;

if (X[m] == key) return m;

if (X[m] < key) l = m + 1;

else r = m - 1;

}

return -1;

}

int main() {

int n , cas = 1;

while (~scanf("%d",&n) && n) {

int m = 0;

while (n --) {

double a , b , c , d;

scanf("%lf%lf%lf%lf",&a,&b,&c,&d); X[m] = a;

ss[m++] = Seg(a , c , b , 1); X[m] = c;

ss[m++] = Seg(a , c , d , -1); }

sort(X , X + m);

sort(ss , ss + m);

int k = 1;

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

if (X[i] != X[i-1]) X[k++] = X[i]; }

memset(cnt , 0 , sizeof(cnt));

memset(sum , 0 , sizeof(sum));

double ret = 0;

for (int i = 0 ; i < m - 1 ; i ++) { int l = Bin(ss[i].l , k , X);

intr=Bin(ss[i].r,k,X)-1;；if(l<=r)update(l,r,ss；printf("Testcase#%d；area:%.2lf\n\n",cas；return0;；hdu1828Picture；题意:矩形周长并；思路:与面积不同的地方是还要记录竖的边有几个(n；线段树操作:update:区间增减query

int r = Bin(ss[i].r , k , X) - 1;

if (l <= r) update(l , r , ss[i].s , 0 , k - 1, 1); ret += sum[1] \* (ss[i+1].h - ss[i].h);

}

printf("Test case #%d\nTotal explored

area: %.2lf\n\n",cas++ , ret);

}

return 0;

}

hdu1828 Picture

题意:矩形周长并

思路:与面积不同的地方是还要记录竖的边有几个(numseg记录),并且当边界重合的时候需要合并(用lbd和rbd表示边界来辅助)

线段树操作:update:区间增减 query:直接取根节点的值

#include <cstdio>

#include <cstring>

#include <cctype>

#include <algorithm>

using namespace std;

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

const int maxn = 22222;

struct Seg{

int l , r , h , s;

Seg() {}

Seg(int a,int b,int c,int d):l(a) , r(b) , h(c) , s(d) {} bool operator < (const Seg &cmp) const {

return h < cmp.h;

}

}ss[maxn];

bool lbd[maxn<<2] , rbd[maxn<<2];

int numseg[maxn<<2];

int cnt[maxn<<2];

int len[maxn<<2];

void PushUP(int rt,int l,int r) {

if (cnt[rt]) {

lbd[rt] = rbd[rt] = 1;

len[rt] = r - l + 1;

numseg[rt] = 2;

} else if (l == r) {

len[rt] = numseg[rt] = lbd[rt] = rbd[rt] = 0;

} else {

lbd[rt] = lbd[rt<<1];

rbd[rt] = rbd[rt<<1|1];

len[rt] = len[rt<<1] + len[rt<<1|1];

numseg[rt] = numseg[rt<<1] + numseg[rt<<1|1];

if (lbd[rt<<1|1] && rbd[rt<<1]) numseg[rt] -= 2;//两条线重合

}

}

void update(int L,int R,int c,int l,int r,int rt) {

if (L <= l && r <= R) {

cnt[rt] += c;

PushUP(rt , l , r);

return ;

}

int m = (l + r) >> 1;

if (L <= m) update(L , R , c , lson);

if (m < R) update(L , R , c , rson);

PushUP(rt , l , r);

}

int main() {

int n;

while (~scanf("%d",&n)) {

int m = 0;

int lbd = 10000, rbd = -10000;

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

int a , b , c , d;

scanf("%d%d%d%d",&a,&b,&c,&d);

lbd = min(lbd , a);

rbd = max(rbd , c);

ss[m++] = Seg(a , c , b , 1);

ss[m++] = Seg(a , c , d , -1);

}

sort(ss , ss + m);

int ret = 0 , last = 0;

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

if (ss[i].l < ss[i].r) update(ss[i].l , ss[i].r - 1 , ss[i].s , lbd , rbd - 1 , 1);

ret += numseg[1] \* (ss[i+1].h - ss[i].h); ret += abs(len[1] - last);

last = len[1];

}

printf("%d\n",ret);

}

return 0;

}

图 论

图的五种种存储方式

方式1：邻接矩阵

邻接矩阵是表示图的数据结构中最简单也是最常用的一种。

实现：二维数组Map[MAXN][MAXN]，Map[i][j]表示点i到点j的距离。

初始化：Map[i][i] = 0,Map[i][j] = INF(i!=j)，读入数据Map[i][j] = w。 时间复杂度：初始化O(n^2)，建图需要O(m)，总时间复杂度O(n^2)。

优缺点：简单直观，可直接查询点i和点j之间是否有边；遍历效率低，不能存储重边；初始化效率低；大图开销大，适合存储点少的稠密图。

方式2：前向星

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 100;

const int MAXM = 100;

int head[MAXN];//存储起点为i的第一条边的位置

struct node

{

int from;//起点

int to;//终点

int w;//权值

};

node edge[MAXM];

bool cmp(node a,node b)//边排序

{

if(a.from == b.from && a.to == b.to) return a.w < b.w; if(a.from == b.from) return a.to < b.to;

return a.from < b.from;

}

int main()

{

int n,m;

cin >> n >> m;

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

cin >> edge[i].from >> edge[i].to >> edge[i].w;

sort(edge,edge+m,cmp);//边排序

memset(head,-1,sizeof(head));

head[edge[0].from] = 0;

for(int i = 1; i < m; i++)//确定起点为i的第一条边位置

{

if(edge[i].from != edge[i-1].from)

head[edge[i].from] = i;

}

for(int i = 1; i <= n; i++)//遍历图

{

//k为第i个结点边的起始位置

for(int k = head[i]; edge[k].from == i && k < m; k++)

cout << edge[k].from << ' ' << edge[k].to << ' ' << edge[k].w << endl;

}

return 0;

}

/\*

7 8

1 1 1

1 3 1

1 3 2

3 5 1

3 6 1

4 6 1

2 4 1

1 2 1

\*/

方式3：邻接表——动态建表

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 100;

struct EdgeNode //邻接表节点

{

int to; //终点

int w; //权值

EdgeNode \*next; //指向下一条边的指针

};

struct VNode //起点表节点

{

int from; //起点

EdgeNode \*first; //邻接表头指针

};

VNode Adjlist[MAXN] //整个图的邻接表

int main()

{

int n,m,x,y,w;

cin >> n >> m;

for(int i = 0; i < m; i++)//读入图

{

cin >> x >> y >> w;

EdgeNode \*p = new EdgeNode();

p->to = y;

p->w = w;

p->next = Adjlist[x].first;

Adjlist[x].first = p;

}

cout << endl;

for(int i = 1; i <= n; i++)//遍历图

{

for(EdgeNode \*k = Adjlist[i].first; k != NULL; k = k->next) cout << i << ' ' << k->to << ' ' << k->w << endl; }

return 0;

}

方式4：邻接表——vector模拟链表实现

#include<iostream>

#include<algorithm>

#include<vector>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 100;

const int MAXM = 100;

struct EdgeNode //边表节点类型

{

int to; //终点

int w; //权值

};

vector<EdgeNode> Map[MAXN];

int main()

{

EdgeNode e;

int n,m,x,y,w;

cin >> n >> m;

for(inti=0;i<m;i++)//；cin>>x>>y>；e.to=y;；e.w=w;；Map[x].push\_back(e);；cout<<endl;；for(inti=1;i<=n;i++)/；for(vector<EdgeNode&g；k!=Map[i].end();k++)；EdgeN

for(int i = 0; i < m; i++) //读入图

{

cin >> x >> y >> w;

e.to = y;

e.w = w;

Map[x].push\_back(e);

}

cout << endl;

for(int i = 1; i <= n; i++) //遍历图

{

for(vector <EdgeNode>::iterator k = Map[i].begin();

k!=Map[i].end(); k++)

{

EdgeNode t = \*k;

cout << i << ' ' << t.to << ' ' << t.w << endl;

}

}

return 0;

}

方式5：邻接表——链式前向星★

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 100;

const int MAXM = 100;

int head[MAXN];

struct EdgeNode

{

int to;

int w;

int next;

};

EdgeNode Edges[MAXM];

int main()

{

int n,m,x,y,w;

cin >> n >> m;

memset(head,-1,sizeof(head));

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

{

cin >> x >> y >> w;

Edges[i].to = y;

Edges[i].w = w;

Edges[i].next = head[x];

head[x] = i;

}

cout << endl;

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

{

for(int k = head[i]; k!= -1; k = Edges[k].next)

cout << i << ' ' << Edges[k].to << ' ' << Edges[k].w << endl; }

return 0;

}

拓扑排序

1.普通判断拓扑排序、输出路径：queue<int> Q

2.满足字典序的拓扑排序：priority\_queue<int,vector<int>, greater<int> > Q; 保证了权值小的优先级高，取出的时候保证序号是队列中最小的。

3.满足最小的尽量在前面(不保证是字典序)：priority\_queue<int> Q;

反向建图，用优先队列存储将入度为0的点，再遍历权值大的结点，从大到小存入数组ans[]，最后ans[]逆序就是满足要求的拓扑排序

时间复杂度：O(MAXM)

const int MAXN = 110;

const int MAXM = 10010;

int head[MAXN],indegree[MAXN],ans[MAXN],N,M,T;

//indegree[]存放入度，ans[]逆序存放拓扑序列

struct EdgeNode

{

int to, w, next;

};

EdgeNode Edges[MAXM];

int toposort()

{

queue<int> Q;//拓扑排序

int u;

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

{

if(indegree[i] == 0)

Q.push(i);

} //将没有前驱(入度为0)的点加入队列

int id = 0;

//使用队列中的点，更新indegree数组，并生成拓扑序列

while(!Q.empty())

{

u = Q.front();

ans[id++] = u; //记录拓扑序列

Q.pop();

//删除从该顶点出发的全部有向边，更新indegree[]

for(int i = head[u]; i != -1; i = Edges[i].next)

{

indegree[Edges[i].to]--;

if(indegree[Edges[i].to]==0)

Q.push(Edges[i].to);

//如果indegree数组为0，则新的没有前驱的店被找到，加入队列 }

}

if(id == N)//如果有N个点进队，出队，则能生成拓扑序列

{

for(int i = N; i >= 1; i--)//逆序输出

if(i != 1) cout << ans[i] << " ";

else cout << ans[i] << endl;

return true;

}

else return false; //否则构不成拓扑序列

}

最小生成树

Kruskal算法：

const int MAXN = 1010;

const int MAXM = 200020;

struct Edge

{

int from, to, w;

}Edges[MAXM];

int father[MAXN];

int find(int x)

{

if(x != father[x]) father[x] = find(father[x]);

return father[x];

}

int cmp(Edge a,Edge b) //将边按权值排序

{

if(a.w != b.w) return a.w < b.w; //return a.w > b.w;求最大生成树

if(a.from != b.from) return a.from < b.from;

return a.to < b.to;

}

int Kruskal(int N,int M)

{

sort(Edges,Edges+M,cmp);//先将边按权值排序

int ans = 0,Count = 0;//Count表示合并的变数,ans存放MST大小 for(int i = 0; i < M; i++)

{

int u = find(Edges[i].from);

int v = find(Edges[i].to);

if(u != v)

{

ans += Edges[i].w;

father[v] = u;

Count++;

if(Count == N-1)//合并了N-1条边，已经找到了最小生成树 break;

}

}

if(Count == N-1) return ans; //找到最小生成树

else return -1; //图不连通

}

//初始化及构图

for(int i = 0; i < N; i++) father[i] = i;

Sum = 0;

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

{

scanf("%d%d%d",&x,&y,&w);

Edges[i].from = x;

Edges[i].to = y;

Edges[i].w = w;

Sum += w;

}

int ans = Kruskal();

Prim算法：

int G[110][110],vis[110],low[110],pre[110];

//G[][]存放图，vis[]表示Va、Vb集合，low[]表示生成树边长集合 //pre[i]对应Dist[i]中所表示的最短边，记录当前点的父节点

void Prim(int N)

{

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

int ans = 0,pos = 1;//ans记录MST大小，pos记录加点位置 vis[pos] = 1; //

low[pos] = 0; //无意义

for(int i = 1; i <= N; i++) //初始化

{

if(i != pos)

{

low[i] = G[pos][i];

pre[i] = 1;

}

}

for(int i = 1; i < N; i++) //循环N-1次，每次加入一个点 {

int Min = 0xffffff0;

pos = 0;

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

{

if(!vis[j] && low[j] < Min)

{

Min = low[j];

pos = j;

}

}

if(pos == 0) return; //没有点可以扩展，图G不连通，可用于判断联通

ans += Min; //添加边

vis[pos] = 1;

for(int j = 1; j <= N; j++) //对每个与pos点相邻的未遍历的点j，更新到j点最近的点及距离

{

if(!vis[j] && low[j] > G[pos][j])

{

low[j] = G[pos][j];

pre[j] = pos;

}

}

}

printf("%d\n",ans);

}

//初始化调用

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

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

G[i][j] = 0xffffff0;

for(int i = 1; i <= n\*(n-1)/2; i++)

{

scanf("%d%d%d",&x,&y,&d);

map[x][y] = map[y][x] = d;

}

次小生成树；给一个图，判断图的最小生成树是否唯一；End[]记录邻接表尾节点的位置；#include<algorithm>；#include<cstdio>；#include<cstring>；usingnamespacestd;；constintMAXN=1010;//元素个数；constintMAXM=100010;；i

次小生成树

给一个图，判断图的最小生成树是否唯一。

End[]记录邻接表尾节点的位置。Len[x][y]表示对x，y两点在最小生成树上最长边的计算。MST表示最小生成树的大小，SecMST表示次小生成树的大小。 #include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 1010; //元素个数

const int MAXM = 100010;

int father[MAXN];

int find(int x) //查找x所在等价类的代表元素

{

if(x != father[x]) father[x] = find(father[x]); return father[x];

}

struct Node

{

int from, to, w;

bool vis; //标记为是否是最小生成树上的边

};

Node Edges[MAXM];//存储边信息

bool cmp(Node a, Node b) //按边的权值排序

{

return a.w < b.w;

}

//链式前向星记录每个集合有哪些点

struct Node1

{

int to, next;

};

Node1 Vertex[MAXN];//边数组，表示结点连向的边

int N,M;

int head[MAXN]; //邻接表头结点位置

int End[MAXN]; //邻接表尾结点位置，方便合并

int Len[MAXN][MAXN];//图中两点之间在最小生成树上路径最长的边 void Kruskal()

{

int x,y,k = 0;

int ans = 0;

//初始化邻接表，每个节点初始的时候添加一条指向自己的边，表示结点i各自为一个集合

memset(head,-1,sizeof(head));

memset(End,-1,sizeof(End));

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

{

Vertex[i].to = i;

Vertex[i].next = head[i];

End[i] = i;

head[i] = i;

}

sort(Edges,Edges+M,cmp);//边按权值排序

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

{

if(k == N-1) break; //构成生成树

if(Edges[i].w < 0) continue;

x = find(Edges[i].from);

y = find(Edges[i].to);

if(x != y)

{

//遍历两个节点所在的集合

for(int w = head[x]; w != -1; w = Vertex[w].next) {

for(int v = head[y]; v != -1; v = Vertex[v].next) {

Len[Vertex[w].to][Vertex[v].to] =

Len[Vertex[v].to][Vertex[w].to] = Edges[i].w;

//当前加入的边一定是加(x,y)边成环后删去的除(x,y)外长度最大的边

}

}

//合并两个邻接表，表示两点已连边连在一个集合中，最终连成一个最小生成树

Vertex[End[y]].next = head[x];

head[x] = head[y];

End[y] = End[x];

father[y] = x;

k++;

Edges[i].vis = true;

}

}

}

int main()

{

int T,x,y,w;

scanf("%d",&T);

while(T--)

{

scanf("%d%d",&N,&M);

for(int i = 1; i <= N; i++) father[i] = i;

memset(Len,0x7f,sizeof(Len));

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

{

scanf("%d%d%d",&x,&y,&w);

Edges[i].from = x;

Edges[i].to = y;

Edges[i].w = w;

Edges[i].vis = false;

}

int MST,SecMST;//MST最小生成树大小,SecMST次小生成树大小 Kruskal();

MST = 0;//最小生成树长度

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

{

if(Edges[i].vis) MST += Edges[i].w;

}

SecMST = 0xfffff0;

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

{

if(!Edges[i].vis)//加边，并删去最小生成树上的边

SecMST = min(SecMST,MST+Edges[i].w -

Len[Edges[i].from][Edges[i].to]);

}

if(SecMST == MST) printf("Not Unique!\n");//次小生成树不唯一

else printf("%d\n",MST);

}

return 0;

}

最小树形图

基于贪心和缩点的思想。

假设根的顶点是V0。

(1)除了根结点外，所有的点Vi，找到以Vi为终点的最短的边，加入集合中 （pre[v]存放的是终点v的起点,In[v]存放终点为v的最短的边）

(2)检查集合中有没有有向环和收缩点。若没有有向环和收缩点，结束计算；若没有有向环、但含收缩边，则跳至步骤(4)；若含有有向环，则跳至步骤(3)。Ps：如果出现重边，将忽略权值较高的。

(3)含有有向环，则收缩有向环(缩点)，把有向环收缩为一个点，其有向环内的边被收缩掉，而环外的边被保留，构建新图，重复步骤(1)、(2)。

(4)没有有向环，有收缩边，则展开收缩边。

不定根：如果没有固定根，就需要建立一个虚根。假设一个根结点，也就是虚根，虚根节点可以为N+1。让所有边都有虚根指向的边，权值大小看题目要求。 若要使每一个原节点都有可能当根，则是权值全部相等，且权值应大于所有边的总权值。如果虚根到每个点的权值和该点相关，则权值应为对应大小。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

#include<cmath>

#include<limits.h>

using namespace std;

const int MAXN = 110;

const int MAXM = 10010;

struct Node//存放图

{

int from, to;

double w;

}Edges[MAXM];

struct Node1

{

double x, y;

};

Node1 Point[MAXN];

double Dist(Node1 a, Node1 b)

{

double x = a.x - b.x;

double y = a.y - b.y;

return sqrt(x\*x+y\*y);

}

int pre[MAXN],vis[MAXN],flag[MAXN];

//flag标记缩点，标记为true，则该点被缩掉

double In[MAXN],sum;//一般为int

double ZhuLiu(int root,int N,int M)

{

sum = 0; //存放最小树形图

while(true)

{

//求最短弧集合

for(int i = 0; i < N; ++i) In[i] = INT\_MAX;

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

{

int u = Edges[i].from;

int v = Edges[i].to;

if(Edges[i].w < In[v] && u != v)

{

pre[v] = u;//v为终点，pre[v]存放起点

In[v] = Edges[i].w;//权值最小的边

}

}

for(int i = 0; i < N; ++i)//如果存在除root以外的孤立点，则不存在最小树形图

{

if(i == root) continue;

if(In[i] == INT\_MAX) return -1;

}

int CntNode = 0; //新图编号

memset(flag,-1,sizeof(flag));

memset(vis,-1,sizeof(vis));

In[root] = 0;

for(int i = 0; i < N; ++i) //找环,标记每个环

{

sum += In[i];

int v = i;

while(vis[v]!=i && flag[v]==-1 && v!=root)//每个点寻找其前序点，要么最终寻找至根部，要么找到一个环

vis[v]=i;；v=pre[v];；if(v!=root&&flag；for(intu=pre[v];u!=v;u=p；flag[u]=CntNode;；flag[v]=CntNode++;；if(CntNode==0)break;//无环；for(inti=0;i<N;++i)//；if(flag[i]==-1)flag[i]=C

{

vis[v] = i;

v = pre[v];

}

if(v != root && flag[v] == -1) //新图重新编号

{

for(int u = pre[v]; u != v; u = pre[u])

flag[u] = CntNode;

flag[v] = CntNode++;

}

}

if(CntNode == 0) break; //无环，跳出

for(int i = 0; i < N; ++i) //缩点

if(flag[i] == -1) flag[i] = CntNode++;

for(int i = 0; i < M; ++i) //建立新图，更新其他点到环的距离 {

int v = Edges[i].to;

Edges[i].from = flag[Edges[i].from];

Edges[i].to = flag[Edges[i].to];

if(Edges[i].from != Edges[i].to) Edges[i].w -= In[v]; }

N = CntNode;

root = flag[root];

}

return sum;

}

int main()

{

int x,y,N,M;

while(~scanf("%d%d",&N,&M))

{

int id = 0;

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

scanf("%lf%lf",&Point[i].x,&Point[i].y);

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

{

scanf("%d%d",&x,&y);

if(x == y)

continue;

x--;

y--;

Edges[id].from = x;

Edges[id].to = y;

Edges[id++].w = Dist(Point[x],Point[y]); //建立有向图

}

double ans = ZhuLiu(0,N,id);

if(ans == -1) //不能构成最小树形图

printf("poor snoopy\n");

else

printf("%.2lf\n",ans);

}

return 0;

}

最近公共祖先LCA—Tarjan-LCA算法：

对于每一点u：

1.建立以u为代表元素的集合。

2.遍历与u相连的节点v，如果没有被访问过，对于v使用Tarjan-LCA算法，结束后，将v的集合并入u的集合。

3.对于与节点u相关的询问(u，v)，如果v被访问过，则结果就是v所在集合的所代表的元素。

求(u,v)的最近公共祖先节点:则询问调用QEdges[k].lca =

find(QEdges[k].to);

求(u,v)在树上的距离：Dist(u，v) = Dist(1，u) + Dist(1，v) - 2\*Dist( 1，LCA(u，v) )，即u到根结点的距离 + v到根结点的距离 - 2\*(u，v)最近公共祖先到根结点的距离，则询问时调用QEdges[k].lca = Dist[u] +

Dist[QEdges[k].to] - 2\*Dist[find(QEdges[k].to)];

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 80080;

const int MAXQ = 20020;

int father[MAXN],Head[MAXN],QHead[MAXN],Dist[MAXN];

//Head[]和Eges[]用来存储原图；QHead[]和QEdges[]用来存储询问 struct EdgeNode

{

int to, next, lca;

}Edges[MAXN],QEdges[MAXN];

int find(int x)

{

if(x != father[x]) father[x] = find(father[x]);

return father[x];

}

bool vis[MAXN];

void LCA(int u)

{

father[u] = u;

vis[u] = true;

for(int k = Head[u]; k != -1; k = Edges[k].next)

{

if(!vis[Edges[k].to])

{

Dist[Edges[k].to] = Dist[u] + Edges[k].lca;

LCA(Edges[k].to);

father[Edges[k].to] = u;

}

}

for(int k = QHead[u]; k != -1; k = QEdges[k].next)

{

if(vis[QEdges[k].to])

{

//QEdges[k].lca = find(QEdges[k].to);

QEdges[k].lca = Dist[u] + Dist[QEdges[k].to] -

2\*Dist[find(QEdges[k].to)];

QEdges[k^1].lca = QEdges[k].lca;

}

}

}

int main()

{

int N,M,K,u,v,w,a,b;

while(~scanf("%d%d",&N,&M))

{

memset(father,0,sizeof(father));

memset(Head,-1,sizeof(Head));

memset(QHead,-1,sizeof(QHead));

memset(vis,false,sizeof(vis));

memset(Edges,0,sizeof(Edges));

memset(QEdges,0,sizeof(QEdges));

memset(Dist,0,sizeof(Dist));

int id = 0;

for(int i = 0; i < M; ++i)//插入图的M条边

{

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

Edges[id].to = v;

Edges[id].lca = w;

Edges[id].next = Head[u];

Head[u] = id++;

Edges[id].to = u;

Edges[id].lca = w;

Edges[id].next = Head[v];

Head[v] = id++;

} //(u,v)和(v,u)都要存，表示双向边

scanf("%d",&K);//K条询问

int iq = 0;

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

{

scanf("%d%d",&a,&b);

QEdges[iq].to = b;

QEdges[iq].next = QHead[a];

QHead[a] = iq++;

QEdges[iq].to = a;

QEdges[iq].next = QHead[b];

QHead[b] = iq++;

} //同理(u,v)和(v,u)都要存，但是询问时只对一条边回答 LCA(1); //跟结点。

for(int i = 0; i < iq; i+=2) //回答询问

printf("%d\n",QEdges[i].lca);

}

return 0;

}

树的最小支配集、最小点覆盖、最大独立集

最小支配集：指从所有顶点中取尽量少的点组成一个集合，使得剩下的所有点都与取出来的点有边相连。顶点个数最小的支配集被称为最小支配集。这里用贪心法来求。

1.以1号点深度优先搜索整棵树，求出每个点在DFS中的编号和每个点的父亲节点编号。

2.按DFS的反向序列检查，如果当前点既不属于支配集也不与支配集中的点相连，且它的父亲也不属于支配集，将其父亲点加入支配集，支配集个数加1。

3.标记当前结点、当前结点的父节点(属于支配集)、当前结点的父节点的父节点(与支配集中的点相连)。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 20020;

struct EdgeNode

{

int to, next;

}Edges[MAXN];

int Head[MAXN],father[MAXN],NewPos[MAXN];

bool vis[MAXN];

//NewPos[]表示深度优先遍历序列的第i个点是哪个点

//now表示当前深度优先遍历序列中已经有多少个点了

//vis[]用来深度优先遍历的判重

//father[]表示点i的父亲节点编号

int N,M,now;

void DFS(int x) //得到深度优先队列的反向序列

{

NewPos[now++] = x;

for(int k = Head[x]; k != -1; k = Edges[k].next)

{

if(!vis[Edges[k].to])

{

vis[Edges[k].to] = true;

father[Edges[k].to] = x;

DFS(Edges[k].to);

}

}

}

//S[i]为true，表示第i个点被覆盖了

//Set[i]表示点i属于要求的点集

bool S[MAXN],Set[MAXN];

int Greedy()//贪心求最小支配集

{

memset(S,0,sizeof(S));

memset(Set,0,sizeof(Set));

int ans = 0;

for(int i = N-1; i >= 1; i--)//反向序列检查

{

int t = NewPos[i];

if(!S[t])//当前点未被覆盖，也就是当前点既不属于支配集，也不与支配集中的点相连

{

if(!Set[father[t]])//当前点的父亲结点不属于支配集， {

Set[father[t]] = true; //将父节点加入支配集 ans++; //顶点个数加1

}

S[t] = true;

S[father[t]] = true;

S[father[father[t]]] = true;

//标记当前点、当前结点的父节点、当前结点的父节点的父节点 }

}

return ans;

intmain()；intu,v;；while(~scanf("%d&qu；{//初始化；memset(Edges,0,sizeof(Ed；memset(Head,-1,sizeof(He；memset(father,0,sizeof(f；memset(vis,false,sizeof(；memset(NewPos,0,sizeof(N；intid

}

int main()

{

int u,v;

while(~scanf("%d",&N))

{ //初始化

memset(Edges,0,sizeof(Edges));

memset(Head,-1,sizeof(Head));

memset(father,0,sizeof(father));

memset(vis,false,sizeof(vis));

memset(NewPos,0,sizeof(NewPos));

int id = 0;

for(int i = 0; i < N-1; ++i)

{

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

Edges[id].to = v;

Edges[id].next = Head[u];

Head[u] = id++;

Edges[id].to = u;

Edges[id].next = Head[v];

Head[v] = id++;

}

now = 0;

vis[1] = true;

father[1] = 1;

DFS(1);

printf("%d\n",Greedy());

}

return 0;

}

最小点覆盖：指从所有顶点中取尽量少的点组成一个集合，使得集合中所有的边都与取出来的点有边相连。顶点个数最小的覆盖集被称为最小点覆盖。

贪心策略：如果当前点和当前点的父节点都不属于顶点覆盖集合，则将父节点加入到顶点覆盖集合中，并标记当前节点和其父节点都被覆盖。

int Greedy()//贪心求最小点覆盖

{

memset(S,0,sizeof(S));

memset(Set,0,sizeof(Set));

int ans = 0;

for(int i = N-1; i >= 1; i--)//反向序列检查

{

int t = NewPos[i];

if(!S[t] && !S[father[t]])//当前点和当前点的父节点都不属于顶点覆盖集合

{

Set[father[t]] = true; //当前点的父节点加入到顶点覆盖集合中

ans++; //顶点个数+1

S[t] = true;

S[father[t]] = true;

//标记当前点、当前结点的父节点

}

}

return ans;

}

最大独立集：指从所有顶点中取尽量多的点组成一个集合，使得这些点之间没有边相连。顶点个数最多的独立集被称为最大独立集。

贪心策略：如果当前节点没有被覆盖，则将当前节点加入独立集，并标记当前节点和其父节点都被覆盖。

int Greedy()//贪心求最大独立集

{

memset(S,0,sizeof(S));

memset(Set,0,sizeof(Set));

int ans = 0;

for(int i = N-1; i >= 1; i--)//反向序列检查

{

int t = NewPos[i];

if(!S[t])//当前点未被覆盖

{

Set[t] = true; //将当前点加入独立集

ans++; //独立集点个数加1

S[t] = true;

S[father[t]] = true;

//标记当前点、当前结点的父节点都被覆盖

}

}

return ans;

}

单源最短路径Dijkstra、BellmanFord、SPFA

Dijkstra算法：

将所有点分为两个集合。如果源点s到u的最短路径已经确定，点u就属于集合Va，否则属于集合Vb。

1.将源点s到图中各点的直接距离当做初始值记录为s到各点的最短距离，不能到达的记为INF。S到S距离为0。

2.在集合Vb中的点中找一个点u，使得源点s到该点u路径长度最短，将u从Vb中除去，加到V1中。这时候求出了当前S到u的最短路径。

3.把新确定的点u更新s到集合Vb中每一个点v的距离，如果s到u的距离加上u到v的直接距离小于当前s到v的距离，则表示新找到的最短路径长度比之前的更短，那么更新这个距离，并更新最短路径。

4.重复步骤2.3，直到集合Vb中已经没有点，或是Vb没有从源点s能达到的点。 如果有带花费的最短路径，建立一个二维数组cost[][]表示花费图，建立一维数组Value[]来更新最小花费。则对于每个与k相邻的在Vb中的点j，更新s到j的最短距离时，如果距离相等，则更新花费。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 110;

const int INF = 1000000000;

int Map[MAXN][MAXN],pre[MAXN],Dist[MAXN];

bool vis[MAXN];

//Map[]来存储图，pre[]来保存结点前驱、源点、终点

//Dist[i]为源点s到节点i的最短距离

//vis[i]记录点i是否属于集合Va

void Dijkstra(int N,int s)

{

int Min;

for(int i = 1; i <= N; ++i) //初始化

{

vis[i] = false;

if(i != s)

{

Dist[i] = Map[s][i];

pre[i] = s;

}

}

Dist[s] = 0;

vis[s] = true;

for(int i = 1; i <= N-1; ++i) //循环N-1次，求源点s到其他N-1个点最短路径

{

Min = INF;

int k = 0;

for(int j = 1; j <= N; ++j) //在Vb中的点钟取一个s到其距离最小的点k

{

if(!vis[j] && Dist[j] < Min)

{

Min = Dist[j];

k = j;

}

}

if(k == 0) return;//如果没有点可以扩展，即剩余的点不可达，返回

vis[k] = true; //将k从Vb重除去，加入到Va中

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

{ //对于每个与k相邻的在Vb中的点j，更新s到j的最短距离 if(!vis[j] && Map[k][j] != INF && Dist[j] > Dist[k] + Map[k][j])

{

Dist[j] = Dist[k] + Map[k][j];

pre[j] = k;

}

}

}

}

int main()

{

int N,M,a,b,w;

while(~scanf("%d%d",&N,&M) && (N||M))

{ //初始化注意

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

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

Map[i][j] = INF;

//memet(Map,INF,sizeof(Map);这样是错的。不能这样子初始化。。。 memset(Dist,INF,sizeof(Dist));

memset(pre,0,sizeof(pre));

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

{

scanf("%d%d%d",&a,&b,&w);

Map[a][b] = Map[b][a] = w;

}

Dijkstra(N,1);

printf("%d\n",Dist[N]);

}

return 0;

}

BellmanFord算法：

处理带负权边的单元最短路径问题。

1.将Dist[]赋值为INF，出发点为s，Dist[s] = 0。

2.对于每条边(u，v)，如果Dist[u] != INF，且Dist[v] > Dist[u] + Map[u][v]，则Dist[v] = Dist[u] + Map[u][v]。

3.重复步骤(2) N-1次或直到某次中不再更新，进入步骤(4)。

4.对于每条边(u，v)，如果Dist[u] != INF，且Dist[v] > Dist[u] + Map[u][v]，则存在负权回路。

寻找负权回路：将Dist[]赋值为INF，寻找最短路径。

寻找正权回路：将Dist[]赋值为0，寻找最长路径。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 550;

const int MAXM = 5500;

const int INF = 0xffffff0;

struct EdgeNode

{

int to, w, next;

}Edges[MAXM];

int Head[MAXN],Dist[MAXN];

bool BellmanFord(int N,int M)

{

Dist[1] = 0;

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

{

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

{

if(Dist[j] == INF) continue;

for(int k = Head[j]; k != -1; k = Edges[k].next) //寻找最短路径

{

if(Edges[k].w != INF && Dist[Edges[k].to] > Dist[j] + Edges[k].w)

Dist[Edges[k].to] = Dist[j] + Edges[k].w; }

}

}

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

{

if(Dist[j] == INF) continue;

//Dist[u] != INF 且Dist[v] > Dist[u] + Map[u][v]，则存在负权回路。寻找正权回路，则改变符号为<，去掉Dist[u] != INF

for(int k = Head[j]; k != -1; k = Edges[k].next)

{

if(Edges[k].w!=INF&&；returnfalse;//不存在负权回路；intmain()；intF,N,M,W,S,E,T;；scanf("%d",&am；while(F--)；//初始化；memset(Dist,INF,sizeof(D；memset(Head,-1,sizeof(He；memset(Edges,

if(Edges[k].w != INF && Dist[Edges[k].to] > Dist[j] + Edges[k].w) return true; //存在负权回路

}

}

return false; //不存在负权回路

}

int main()

{

int F,N,M,W,S,E,T;

scanf("%d",&F);

while(F--)

{

//初始化

memset(Dist,INF,sizeof(Dist));

memset(Head,-1,sizeof(Head));

memset(Edges,0,sizeof(Edges));

scanf("%d%d%d",&N,&M,&W);

int id = 0;

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

{

scanf("%d%d%d",&S,&E,&T);

Edges[id].to = E;

Edges[id].w = T;

Edges[id].next = Head[S];

Head[S] = id++;

Edges[id].to = S;

Edges[id].w = T;

Edges[id].next = Head[E];

Head[E] = id++;

}

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

{

scanf("%d%d%d",&S,&E,&T);

Edges[id].to = E;

Edges[id].w = -T;

Edges[id].next = Head[S];

Head[S] = id++;

}

if(BellmanFord(N,M)) printf("YES\n");

Else printf("NO\n");

}

return 0;

}

SPFA算法：

在计算带负边权图的单源最短路径基础上，降低时间复杂度。

1.初始化Dist[] = INF，Dist[s] = 0，新建一个队列，将源点S入队，标记S已经在队列中。

2.入队首取出一个点top，标记top已经出队，对top出队的次数进行检查，如果大于N，说明出现负环，算法结束。否则遍历top所连接的边，如果边k的另一端节点to的距离可以更新，即Dist[Edges[k].b] > Dist[top] + Edges[k].w，则更新Dist[Edges[k].b] = Dist[top] + Edges[k].w，检查b是否在队列中，如果不在，加入队列。

3.重复步骤(2)，直到队列为空。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXM = 1000100;

const int MAXN = 1000100;

struct EdgeNode

{

int to, w, next;

}Edges[2][MAXM];

int Head[2][MAXN],vis[MAXN],queue[MAXN],outque[MAXN];

\_\_int64 Dist[MAXN];

//两个链式前向星，一个存正边，一个存反边(特殊题目要求，平常用一个)，求1~N的最短路和N~1的最短路。

bool SPFA(int S,int N,int flag)

{

for(int i = 2; i <= N; ++i) Dist[i] = 0xffffffff;

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

memset(outque,0,sizeof(outque));

int iq = 0;

queue[iq++] = S;

vis[S] = 1;

Dist[S] = 0;

int i = 0,top,k;

while(i != iq)

{

top = queue[i];

vis[top] = 0;

outque[top]++;

if(outque[top] > N) return false;

//如果出队次数大于N，则说明出现负环

k = Head[flag][top];

while(k >= 0) //遍历top链接的边

{ //如果边k的另一端点to的距离可以更新，则更新

if(Dist[Edges[flag][k].to] - Edges[flag][k].w > Dist[top]) {

Dist[Edges[flag][k].to] = Dist[top] + Edges[flag][k].w; if( !vis[Edges[flag][k].to]) //检查to是否在队列中，不在则加入队列

{

vis[Edges[flag][k].to] = 1;

queue[iq++] = Edges[flag][k].to;

}

}

k = Edges[flag][k].next;

}

i++;

}

return true;

}

int main()

{

int T,N,M,u,v,w;

scanf("%d",&T);

while(T--)

{

scanf("%d%d",&N,&M);

memset(Head,-1,sizeof(Head));

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

{

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

Edges[0][i].to = v;

Edges[0][i].w = w;

Edges[0][i].next = Head[0][u];

Head[0][u] = i;

Edges[1][i].to = u;

Edges[1][i].w = w;

Edges[1][i].next = Head[1][v];

Head[1][v] = i;

}

\_\_int64 ans = 0;

SPFA(1,N,0);

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

if(Dist[i] != 0xffffffff)

ans += Dist[i];

SPFA(1,N,1);

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

if(Dist[i] != 0xffffffff)

ans += Dist[i];

printf("%I64d\n",ans);

}

return 0;

}

多源最短路径Floyd、Floyd求最小环

Floyd算法：用来找出每对点之间的最短距离。图可以是无向图，也可以是有向图，边权可为正，也可以为负，唯一要求是不能有负环。

1.初始化:将Map[][]中的数据复制到Dist[][]中作为每对顶点之间的最短路径的初值，Pre[i][j] = i 表示 i 到 j 路径中 j 的前一节点。

2. k 从 1 到 N 循环 N 次，每次循环中，枚举图中不同的两点 i，j，如果Dist[i][j] > Dist[i][k] + Dist[k][j]，则更新Dist[i][j] = Dist[i][k] + Dist[k][j]，更新Pre[i][j] = Pre[k][j]。

只要图中不存在负环就可以得出正确的答案，关于Floyd算法对负环的判定，参考下边Floyd求最小环。

const int MAXN = 110;

const int INF = 0xffffff0;

int Map[MAXN][MAXN], Dist[MAXN][MAXN],Pre[MAXN][MAXN];

//Pre[i][j] = i表示i到j路径中j的前一节点

void Floyd(int N)

{ //初始化

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

{

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

{

Dist[i][j] = Map[i][j];

Pre[i][j] = i;

}

}

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

{

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

{

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

{ //如果Dist[i][j] > Dist[i][k] + Dist[k][j]，则更新 if(Dist[i][k] != INF && Dist[k][j] != INF && Dist[i][k] + Dist[k][j] < Dist[i][j])

{

Dist[i][j] = Dist[i][k] + Dist[k][j];

Pre[i][j] = Pre[k][j]; //更新Pre[i][j] }

}

}

}

}

求点u到点v能达到的最长边尽可能短的路径上最长边为多少:将循环内部改为如下代码：

int tMax; //这里边求的是能达到的路径上最长边最小为多少

if(Dist[i][k] > Dist[k][j]) tMax = Dist[i][k];

else tMax = Dist[k][j];

if(Dist[i][j] > tMax) Dist[i][j] = tMax;

Floyd求最小环

不能在Map[][]数组上直接计算，因为判断过程中用到了Map[][]原始值。 const int MAXN = 110;

const int INF = 0xffffff0;

Int temp, Map[MAXN][MAXN], Dist[MAXN][MAXN], pre[MAXN][MAXN], ans[MAXN\*3];

void Solve(int i,int j,int k)

{

temp = 0; //回溯，存储最小环

while(i != j)

{

ans[temp++] = j;

j = pre[i][j];

}

ans[temp++] = i;

ans[temp++] = k;

}

void Floyd(int N)

{

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

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

{

Dist[i][j] = Map[i][j];

pre[i][j] = i;

}

int MinCircle = INF; //最小环

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

{

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

{

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

{

if(i != j && Dist[i][j] != INF && Map[i][k] != INF && Map[k][j] != INF

&& Dist[i][j] + Map[i][k] + Map[k][j] < MinCircle)

MinCircle=min(MinCircle,；Solve(i,j,k);//回溯存储最小环；for(inti=1;i<=N;++i)；for(intj=1;j<=N;++j)；if(Dist[i][k]!=INF&&；Dist[i][k]+Dist[k][j]<；Dist[i][j]=Dist[i][k]+Di；pre[i][j]

{

MinCircle = min(MinCircle, Dist[i][j] + Map[i][k] + Map[k][j]);

Solve(i,j,k); //回溯存储最小环

}

}

}

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

{

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

{

if(Dist[i][k] != INF && Dist[k][j] != INF &&

Dist[i][k] + Dist[k][j] < Dist[i][j])

{

Dist[i][j] = Dist[i][k] + Dist[k][j];

pre[i][j] = pre[k][j]; //记录点i到点j的路径上，j前边的点

}

}

}

}

if(MinCircle == INF) //不存在环

{

printf("No solution.\n"); return;

}

//如果求出最小环为负的，原图必定存在负环

for(int i = 0;i < temp; ++i) //输出最小环

if(i != temp-1) printf("%d ",ans[i]);

else printf("%d\n",ans[i]);

}

K短路—A\*+SPFA算法：

(1)将有向图的所有边正向、反向分别存入两个不同的边集(Edges，Edges1)中。用反向边集，以所求终点t为源点，利用SPFA或Dijkstra求解出所有点到t的最短路径，用Dist[i]数组来表示点i到点t的最短距离。

(2)建立一个优先队列，将源点s加入到队列中。

(3)从优先队列中取出最小的点p，如果点p == t，则计算t出队的次数。如果当前路径长度就是s到t的第k短路长度，算法结束。否则遍历与p相连的所有的边，将扩展出的到p的邻接点信息加入到优先队列中取。

注意： 当s == t的时候，需要计算第k+1短路。因为s到t这条距离为0的路不能算是这k短路里边，当s == t的时候，只需要将k = k+1后再求第k短路就可以了。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

#include<queue>

using namespace std;

const int MAXN = 1100;

const int MAXM = 110000;

const int INF = 0xffffff0;

struct EdgeNode

{

int to, w, next;

}Edges[MAXM],Edges1[MAXM];

int Head[MAXN],Head1[MAXN];

struct Node

{

int to;

int g,f;

bool operator < (const Node &r) const

{

if(r.f == f)

return r.g < g;

return r.f < f;

}

};

int vis[MAXN],Dist[MAXN];

int A\_Star(int start,int end,int N,int k)

{

Node e,ne;

int Cnt = 0;

priority\_queue<Node> que;

if(start == end) k++;

if(Dist[start] == INF) return -1;

e.to = start;

e.g = 0;

e.f = e.g + Dist[e.to];

que.push(e);

while( !que.empty() )

{

e = que.top();

que.pop();

if(e.to == end) Cnt++;

if(Cnt == k) return e.g;

for(int i = Head[e.to]; i != -1; i = Edges[i].next)

{

ne.to = Edges[i].to;

ne.g = e.g + Edges[i].w;

ne.f = ne.g + Dist[ne.to];

que.push(ne);

}

}

return -1;

}

void SPFA(int s,int N)

{

for(int i = 0; i <= N; ++i) Dist[i] = INF;

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

vis[s] = 1;

Dist[s] = 0;

queue<int> Q;

Q.push(s);

while( !Q.empty() )

{

int u = Q.front();

Q.pop();

vis[u] = 0;

for(int i = Head1[u]; i != -1; i = Edges1[i].next)

{

int temp = Dist[u] + Edges1[i].w;

if(temp < Dist[Edges1[i].to])

{

Dist[Edges1[i].to] = temp;

if(!vis[Edges1[i].to])

{

vis[Edges1[i].to] = 1;

Q.push(Edges1[i].to);

}

}

}

}

}

int main()

{

int N,M,u,v,w,s,t,k;

while(~scanf("%d%d",&N,&M))

{

memset(Edges,0,sizeof(Edges));

memset(Edges1,0,sizeof(Edges1));

memset(Head,-1,sizeof(Head));

memset(Head1,-1,sizeof(Head1));

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

{

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

Edges[i].to = v;

Edges[i].w = w;

Edges[i].next = Head[u];

Head[u] = i;

Edges1[i].to = u;

Edges1[i].w = w;

Edges1[i].next = Head1[v];

Head1[v] = i;

}

scanf("%d%d%d",&s,&t,&k);

SPFA(t,N);

int kthlenth = A\_Star(s,t,N,k);

printf("%d\n",kthlenth);

}

return 0;

}

差分约束系统

差分约束系统：如果一个系统由n个变量和m个约束条件组成，其中每个约束条件形如xj - xi<= bk ( i , j ∈ [1，n]，k ∈ [1，m])，则称其为差分约束系统。

例如如下的约束条件：

X1 - X2 <= 0 X1 - X5 <= -1

X2 - X5 <= 1 X3 - X1 <= 5

X4 - X1 <= 4 X4 - X3 <= -1

X5 - X3 <= -3 X5 - X4 <= -3

全都是两个未知数的差小于等于某个常数（大于等于也可以，因为左右乘以-1就可以化成小于等于）。这样的不等式组就称作差分约束系统。

差分约束系统求解过程：

1.新建一个图，N个变量看作N个顶点，M个约束条件作为M条边。每个顶点Vi分别对于一个未知量，每个有向边对应两个未知量的不等式。

2.为了保证图的连通性，在图中新加一个节点Vs，图中每个节点Vi都能从Vs可达，建立边w(Vs，Vi) = 0。

3.对于每个差分约束Xj-Xi <= Bk(这里是小于等于号)，则建立边w(Xi，Xj) = Bk。

4.初始化Dist[] = INF，Dist[Vs] = 0.

5.求解以Vs为源点的单源最短路径，推荐用SPFA，因为一般可能存在负值。 如果图中存在负权回路，则该差分约束系统不存在可行解。

Vs到某点如果不存在最短路径，即最短路为INF，则对于该点表示的变量可以取任意值，都能满足差分约束的要求，如果存在最短路径，则得到该变量的最大值。 上述过程最终得到的解为满足差分约束系统各项的最大值。

注意点：

1. 如果要求最大值想办法把每个不等式变为标准 x - y <= k 的形式,然后建立一条从 y 到 x 权值为 k 的边，变得时候注意 x - y < k => x - y <= k-1。

2. 如果要求最小值的话，变为 x - y >= k 的标准形式，然后建立一条从 y到 x 权值为 k 的边，求出最长路径即可。

3. 如果权值为正，用Dijkstra，SPFA，BellmanFord都可以，如果为负不能用Dijkstra，并且需要判断是否有负环，有的话就不存在。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

#include<queue>

#define INF 0x7fffffff

using namespace std;

const int MAXN = 1100;

const int MAXM = 30030;

struct EdgeNode

{

int to, w, next;

}Edges[MAXM];

int Head[MAXN],Dist[MAXN],vis[MAXN],outque[MAXN],id;

void AddEdges(int u,int v,int w)

{

Edges[id].to = v;

Edges[id].w = w;

Edges[id].next = Head[u];

Head[u] = id++;

}

void SPFA(int s,int N)

{

int ans = 0;

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

memset(outque,0,sizeof(outque));

for(int i = 1; i <= N; ++i) Dist[i] = INF;

Dist[s] = 0;

vis[s] = 1;

queue<int> Q;

Q.push(s);

while( !Q.empty() )

{

int u = Q.front();

Q.pop();

vis[u] = 0;

outque[u]++;

if(outque[u] > N+1) //如果出队次数大于N，则说明出现负环

{ans=-1;break;}；for(inti=Head[u];i!=-1;i；inttemp=Dist[u]+Edges[i]；if(temp<Dist[Edges[i]；Dist[Edges[i].to]=temp;；if(!vis[Edges[i].to])；vis[Edges[i].to]=1;；Q.push(Edges[i].to);；if

{ ans = -1; break; }

for(int i = Head[u]; i != -1; i = Edges[i].next)

{

int temp = Dist[u] + Edges[i].w;

if(temp < Dist[Edges[i].to])

{

Dist[Edges[i].to] = temp;

if( !vis[Edges[i].to])

{

vis[Edges[i].to] = 1;

Q.push(Edges[i].to);

}

}

}

}

if(ans == -1) printf("-1\n");//出现负权回路，不存在可行解

else if(Dist[N] == INF) printf("-2\n");//可取任意值，都满足 else printf("%d\n",Dist[N]); //求使得源点 s 到 终点 t 的最大的值

}

int main()

{

int N,ML,MD,u,v,w;

while(~scanf("%d%d%d", &N, &ML, &MD))

{

memset(Head,-1,sizeof(Head));

id = 0;

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

{

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

AddEdges(u,v,w);//建边 u - v <= w

}

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

{

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

AddEdges(v,u,-w);//建边 v - u <= w

}

//这里不加也可以

// for(int i = 1; i < N; ++i)

// AddEdges(i+1,i,0);

SPFA(1,N); //求使得源点 s 到 终点 t 的最大的值

}

return 0;

}

强连通分量Kosaraju、Tarjan

强连通分量：在有向图G中，如果两个顶点vi,vj间（vi>vj）有一条从vi到vj的有向路径，同时还有一条从vj到vi的有向路径，则称两个顶点强连通。如果有向图G的每两个顶点都强连通，称G是一个强连通图。有向图的极大强连通子图，称为强连通分量。

把一个图变为一个强连通图需要添加边数：先求出原图的强连通分量，缩点后变为有向无环图，计算新图入度为0的点的个数SumIn和出度为0的点的个数SumOut，答案就是max(SumIn，SumOut)。

Kosaraju算法：

1.对原图进行第一遍深度优先遍历，记录下每个节点的离开时间num[i]。

2.对原图的反向边构成的图进行第二遍深度优先遍历，从步骤(1)中离开时间最晚的点开始。第(2)步中每搜索到一棵树都是一个强连通分量Hash[]把同一连通分量上的点缩成一个点。

3.缩点之后的图就构成了DAG(有向无环图)，树的个数就是强连通分量的个数。 #include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 10010;

const int MAXM = 50050;

struct EdgeNode

{

int to, next1;

int fr, next2;

}Edges[MAXM];

int Head1[MAXN],Head2[MAXN],vis[MAXN];

int num[MAXN],Hash[MAXN],Count[MAXN],outdegree[MAXN];

int id;

//Head1[],Head2[]分别存原图和反图

//num[i]记录每个节点离开时间

//vis[i]用来记录某个点是否被访问过

//Count[sig]统计每个连通分量sig中的点个数

//Hash[cur]用来将同一个联通分量中的点cur(都属于一个sig)缩成一个点sig void AddEdges(int u,int v)

{

Edges[id].to = v;

Edges[id].next1 = Head1[u];

Head1[u] = id;

Edges[id].fr = u;

Edges[id].next2 = Head2[v];

Head2[v] = id++;

}

//DFS第一遍，求出记录每个节点离开时间num[i]

void DfsOne(int cur,int& sig)

{

vis[cur] = 1;

for(int i = Head1[cur]; i != -1; i = Edges[i].next1)

{

if( !vis[Edges[i].to] )

DfsOne(Edges[i].to,sig);

}

num[++sig] = cur;

}

//DFS第二遍，求出双联通分量

void DfsTwo(int cur,int sig)

{

vis[cur] = 1;

Hash[cur] = sig; //Hash[]用来将同一个联通分量中的点(都所欲一个sig)缩成一个点

Count[sig]++; //Count[]统计每个连通分量中的点个数，sig为强连通分量个数

for(int i = Head2[cur]; i != -1; i = Edges[i].next2)

{

if( !vis[Edges[i].fr]) DfsTwo(Edges[i].fr,sig);

else if(Hash[Edges[i].fr] != Hash[cur]) //outdegree判断缩点后新图各点是否有出度，特殊题目要求

outdegree[Hash[Edges[i].fr]] = 1;

}

}

int Kosaraju(int N)

{

int sig = 0,ans;

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

//第一次深度优先搜索

for(int i = 1; i <= N; ++i) //DFS求得拓扑序列num[]

if( !vis[i] ) DfsOne(i,sig); //sig为强连通个数 memset(vis,0,sizeof(vis));

memset(Count,0,sizeof(Count));

memset(outdegree,0,sizeof(outdegree));

//第二次深度优先搜索

int i = sig;

sig = 0;

for(; i >= 1; --i) //按照拓扑序列进行第二次dfs

if( !vis[num[i]])

DfsTwo(num[i],++sig);

//算法结束，以下为特殊题目要求

int temp = 0;

for(int i = 1; i <= sig; i++) //新图只有一个点出度为0才算有解，特殊题目要求

if(!outdegree[i])

{ temp++; ans = Count[i]; }

if(temp == 1) return ans;

else return 0;

}

int main()

{

int N,M,u,v;

while(~scanf("%d%d",&N,&M))

{

id = 0;

memset(Head1,-1,sizeof(Head1));

memset(Head2,-1,sizeof(Head2));

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

{

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

AddEdges(u,v);

}

int ans = Kosaraju(N);

printf("%d\n",ans);

}

return 0;

}

Tarjan算法：

1.访问一个没有被访问过的节点v；否则结束。

2.初始化dfn[v]和low[v]。

对于节点v的所有邻接顶点u：

1.如果没有访问过，转到步骤(2)，同时维护low[v]。

2.如果访问过，但没有删除，维护low[v]。

如果low[v] == dfn[v]，那么取出相应的强连通分量。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 10010;

const int MAXM = 50050;

struct EdgeNode

{

int to, next;

}Edges[MAXM];

int Head[MAXN],vis[MAXN],low[MAXN];

int dfn[MAXN],Stack[MAXN],outdegree[MAXN],Count[MAXN],m,id;

//dfn[v]表示顶点v被访问的时间

//low[v]为与顶点v邻接的未删除的顶点u的low[u]和low[v]的最小值 void AddEdges(int u,int v)

{

Edges[id].to = v;

Edges[id].next = Head[u];

Head[u] = id++;

}

int TarBFS(int pos,int lay,int &scc)

{

vis[pos] = 1;

low[pos] = lay; //初始为开始时间

dfn[pos] = lay; //初始为开始时间

Stack[++m] = pos;

//将当前未处理节点入栈，回溯时可以判断栈顶到栈中的结点是否为同一个强连通分量

//如果当前节点是一个强连通分量的根，它的强连通分量一定是以该根为根节点的(剩下节点)子树

for(int i = Head[pos]; i != -1; i = Edges[i].next) //枚举每一条边 {

if(!vis[Edges[i].to]) TarBFS(Edges[i].to,++lay,scc); if(vis[Edges[i].to] == 1)low[pos] =

min(low[pos],low[Edges[i].to]);

}

//如果dfn[pos] == low[pos]，则当前顶点就是一个强连通分量

if(dfn[pos] == low[pos]) //缩点，low[]相同的结点属于同一个强连通分量

{

++scc;//强连通分量个数

do

{

Count[scc]++; //记录每个强连通分量内的点数 low[Stack[m]] = scc;

vis[Stack[m]] = 2;

}while(Stack[m--] != pos);

}

return 0;

}

int Tarjan(int N)

intscc=0,temp=0,ans,lay=；m=0;；memset(vis,0,sizeof(vis)；memset(low,0,sizeof(low)；memset(dfn,0,sizeof(dfn)；for(inti=1;i<=N;++i)/；if(vis[i]==0)TarBFS(i,la；//下边为特殊题目要求，如果只有一个入度为0的点；

{

int scc = 0, temp = 0, ans, lay = 1;

m = 0;

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

memset(low,0,sizeof(low));

memset(dfn,0,sizeof(dfn));

for(int i = 1; i <= N; ++i) //一次DFS求出强连通分量

if(vis[i] == 0) TarBFS(i,lay,scc); //scc得出强连通分量个数

//下边为特殊题目要求，如果只有一个入度为0的点，则得到

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

{

for(int j = Head[i]; j != -1; j = Edges[j].next)

if(low[i] != low[Edges[j].to])

{

outdegree[low[i]] = 1;//标记入度不为0的点

break;

/\*

outdegree[low[i]]++;

indegree[low[Edges[j].to]]++;

//记录入度和出度不为0的点

\*/

}

}

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

{

if(! outdegree[i]) //得到入度不为0的点个数

{

if(++temp > 1) break;

ans = Count[i];

}

}

if(temp != 1) return 0;

return ans;

}

int main()

{

int N,M,u,v;

while(~scanf("%d%d",&N,&M))

{

memset(Head,-1,sizeof(Head));

memset(outdegree,0,sizeof(outdegree));

memset(Count,0,sizeof(Count));

id = 0;

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

{

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

AddEdges(u,v);

}

int ans = Tarjan(N);

printf("%d\n",ans);

}

return 0;

}

2-SAT

现有一个由N个布尔值组成的序列A，给出一些限制关系，比如A[x] && A[y] = 0、A[x] || A[y] || A[z]=1等，要确定A[0..N-1]的值，使得其满足所有限制关系。这个称为SAT问题，特别的，若每种限制关系中最多只对两个元素进行限制，则称为2-SAT问题。

在2-SAT问题中，最多只对两个元素进行限制，所以可能的限制关系共有11种： A[x] 、NOT A[x] 、A[x] AND A[y]、 A[x] AND NOT A[y]、 A[x] OR A[y] 、A[x] OR NOT A[y]、 NOT (A[x] AND A[y])、 NOT (A[x] OR A[y]) 、A[x] XOR A[y] 、NOT (A[x] XOR A[y]) 、A[x] XOR NOT A[y]

进一步，A[x] AND A[y]相当于(A[x]) AND (A[y])（也就是可以拆分成A[x]与A[y]两个限制关系），NOT(A[x] OR A[y])相当于NOT A[x] AND NOT A[y]（也就是可以拆分成NOT A[x]与NOT A[y]两个限制关系）。因此，可能的限制关系最多只有9种。

2-SAT问题在大多数时候表现成以下形式：有N对物品，每对物品中必须选取一个，也只能选取一个，并且它们之间存在某些限制关系（如某两个物品不能都选，某两个物品不能都不选，某两个物品必须且只能选一个，某个物品必选）等，这时，可以将每对物品当成一个布尔值（选取第一个物品相当于0，选取第二个相当于1），如果所有的限制关系最多只对两个物品进行限制，则它们都可以转化成9种基本限制关系，从而转化为2-SAT模型。

建模：可以构造有向图G，G中包含2\*N个顶点，前N个顶点(1~N)表示第i个元素能被选择，后N个顶点(N+1~2\*N)表示第i个元素不能被选择。Ai和A(i+N)不能同时被选择。同理Ai + Bj = ~( A(i+N) + B(j+N) )，A(i+N)和B(j+N)不能同时被选。选中~A，必须选择B；如果选择~B，必须选择A。

若图中i到j有路径，则若i选，则j也要选；或者说，若j不选，则i也不能选。

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

#include<cmath>

#include<queue>

using namespace std;

const int MAXN = 2200;

const int MAXM = MAXN\*MAXN;

struct EdgeNode

{

int to, next;

}Edges[MAXM];

int Head[MAXN];

int dfn[MAXN],low[MAXN],belong[MAXN],Stack[MAXN],vis[MAXN];

int m,id,lay,scc,N,M;

//belong[]来判断i和i+N是否在一个强连通分量里

void AddEdges(int u,int v)

{

Edges[id].to = v;

Edges[id].next = Head[u];

Head[u] = id++;

}

void TarBFS(int pos)

{

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

Stack[m++] = pos;

vis[pos] = 1;

for(int i = Head[pos]; i != -1; i = Edges[i].next)

{

int v = Edges[i].to;

if( !dfn[v] )

{

TarBFS(v);

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

}

else if(vis[v]) low[pos] = min(low[pos],low[v]); }

int v;

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

{

++scc;

do

{

v = Stack[--m];

belong[v] = scc;

vis[v] = 0;

}while(v != pos);

}

}

int main()

{

int u,v;

while(~scanf("%d%d",&N,&M))

{

id = m = scc = lay = 0;

memset(Head,-1,sizeof(Head));

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

memset(low,0,sizeof(low));

memset(dfn,0,sizeof(dfn));

memset(belong,0,sizeof(belong));

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

{

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

int a = abs(u);

int b = abs(v);

if(u > 0 && v > 0)

{ //a、b至少一个被选中

AddEdges(a+N,b); //如果a不被选，b就必须被选 AddEdges(b+N,a); //如果b不被选，a就必须被选 }

if(u < 0 && v < 0)

{ //a、b至少有一个不被选中

AddEdges(a,b+N); //如果a被选，b就必须不被选 AddEdges(b,a+N); //如果b被选，a就必须不被选 }

if(u > 0 && v < 0)

{ //a被选中和b不被选中两件事至少发生一件

AddEdges(a+N,b+N); //如果a不被选中，b必须不被选中 AddEdges(b,a); //如果b被选中，那么a必须被选中 }

if(u < 0 && v > 0)

{ //a不被选中和b被选中至少发生一件

AddEdges(a,b); //如果a被选中，b必须被选中 AddEdges(b+N,a+N); //如果b不被选中，a必须不被选中 }

}

for(int i = 1; i <= 2\*N; ++i)

if( !dfn[i] ) TarBFS(i);

int ans = 1;

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

{

if(belong[i] == belong[i+N]) //如果i和i+N同在一个连通分量里，则2-SAT不满足

{ ans = 0; break; }

}

printf("%d\n",ans); //不存在就满足2-SAT

}

return 0;

}

二分图

二分图：原图G的顶点可以分类两个集合X和Y，所有的边关联的两个顶点恰好一个属于集合X，另一个属于集合Y，则称该图为二分图。

二分图匹配：给定一个二分图G，在G的一个子图M中，M的边集中的任意两条边都不依附于同一个顶点，即一个顶点最多只有一条边。则称M是一个匹配。 二分图最大匹配：图中包含边数最多的匹配称为图的最大匹配。

二分图完美匹配：如果所有点都在匹配边上，则称这个最大匹配是完美匹配。 二分图多重匹配：二分图匹配一对一匹配，这里允许集合Y中的一个元素和集合X中的多个元素匹配(一般有最大限制N)，但是集合X中的元素只能和集合Y中的多个元素匹配。

二分图最佳匹配：将二分图加权，在图中找到一个总权值最大的匹配。

二分图最小点覆盖： 最小覆盖要求用最少的点（Ｘ集合或Ｙ集合的都行）让每条边都至少和其中一个点关联。 二分图最小点覆盖 = 二分图最大匹配 二分图最小边覆盖：选择最少的边，使得能够覆盖图中所有的点。

二分图最小边覆盖 = 无向图被重复计算两次，ans = N - 最大匹配数/2 DAG图的最小路径覆盖：用尽量少的不相交简单路径覆盖有向无环图 (DAG)G的所有顶点，这就是DAG图的最小路径覆盖问题。

最小路径覆盖 ＝ 节点数N－最大匹配数

二分图的最大独立集：在Ｎ个点的图G中选出若干个点，使这若干个点两两之间没有边，求点数最大值。二分图的最大独立集 = 节点数n - 最大匹配数m 一般是双向边，则被重复计算了两次，ans = N - 最大匹配数/2

二分图最大匹配——匈牙利算法DFS版：

const int MAXN = 220;

bool Map[MAXN][MAXN];

bool bMask[MAXN];

int NX,NY;

int cx[MAXN],cy[MAXN];

int FindPath(int u)

{

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

{

if(Map[u][i] && !bMask[i])

{

bMask[i] = 1;

if(cy[i] == -1 || FindPath(cy[i]))

{

cy[i] = u;

cx[u]=i;；return1;；return0;；intMaxMatch()；intres=0;；for(inti=1;i<=NX;++i)；cx[i]=-1;；for(inti=1;i<=NY;++i)；cy[i]=-1;；for(inti=1;i<=NX;++i)；if(cx[i]==-1)；for(intj=1;j<=NY;

cx[u] = i;

return 1;

}

}

}

return 0;

}

int MaxMatch()

{

int res = 0;

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

cx[i] = -1;

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

cy[i] = -1;

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

{

if(cx[i] == -1)

{

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

bMask[j] = 0;

res += FindPath(i);

}

}

return res; //返回最大匹配数

}

二分图多重匹配：

const int MAXN = 33; //最大顶点数

int Map[MAXN][MAXN]; //二分图

bool Mask[MAXN]; //寻找增广路径时的标志数组

int NX,NY,N; //NX左集合顶点数，NY右集合顶点数

int vcy[MAXN]; //vcy[i]表示右集合i顶点匹配到左集合的顶点数目 int cy[MAXN][MAXN]; //cy[i][j]表示与右集合i顶点匹配的第j个元素 int limit[MAXN]; //每个右集合各顶点最多匹配左集合顶点的个数 //也可以为limit，表示最多匹配左集合顶点的共同限制数，如果为待求元素，可二分搜索查找答案

bool FindPath(int u) //寻找增广路径

{

for(int i = 1; i <= 5; ++i)

{

if(Map[u][i] && !Mask[i])

{

Mask[i] = 1;

if(vcy[i] < limit[i]) //vcy[i] < limit

{

cy[i][vcy[i]++] = u;

return true;

}

for(int j = 0; j < vcy[i]; ++j) //j < vcy[i]

{

if(FindPath(cy[i][j]))

{ cy[i][j] = u; return true; }

}

}

}

return false;

}

void MulMatch() //求多重匹配

{

int Ans = 0;

memset(vcy,0,sizeof(vcy));

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

{

memset(Mask,0,sizeof(Mask));

Ans += FindPath(i); //计算右边能匹配点个数

/\*

if(!FindPath(i))

return false;

\*/

}

//return true;

if(Ans == N)

printf("T-shirts rock!\n");

else

printf("I'd rather not wear a shirt anyway...\n");

}

二分图最佳匹配——KM算法：

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<cstring>

using namespace std;

const int MAXN = 330;

const int INF = 0xffffff0;

int N,NX,NY;

int link[MAXN],lx[MAXN],ly[MAXN],slack[MAXN];

int visx[MAXN],visy[MAXN]; //标记

int Map[MAXN][MAXN]; //存放权值

//lx[],ly[]顶标; link[]记录匹配值

int FindPath(int u) //回溯寻找最优解

{

visx[u] = 1;

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

{

if(visy[i]) continue;

int temp = lx[u] + ly[i] - Map[u][i];

if(temp == 0)

//if(Map[u][i] == lx[u] + ly[i]) //说明是相等子图 {

visy[i] = 1;

if(link[i] == -1 || FindPath(link[i]))

{

link[i] = u;

return 1;

}

}

else if(slack[i] > temp) slack[i] = temp;

}

return 0;

}

int KM() //求权值最大的最佳匹配

{

memset(ly,0,sizeof(ly));

memset(link,-1,sizeof(link));

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

{

lx[i] = -INF; //求最小权匹配则 lx[i] = INF

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

if(Map[i][j] > lx[i]) //最小权匹配则更改符号

lx[i] = Map[i][j];

}

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

{

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

slack[j] = INF;

while(1)

{

memset(visx,0,sizeof(visx));

memset(visy,0,sizeof(visy));

if(FindPath(i)) break;

int d = INF;

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

if(!visy[j] && d > slack[j]) d = slack[j]; if(d == INF) return ;

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

if(visx[j]) lx[j] -= d;//求最小权值则改为 lx[j] += d; for(int j = 1; j <= NY; ++j)

if(visy[j]) ly[j] += d;//求最小权值则改为 ly[j] -= d; else slack[j] -= d;

}

}

int res = 0;

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

if(link[i] > -1) res += Map[link[i]][i];

return res; //输出最佳匹配的最大权值和

}

int main()

{

int N;

while(~scanf("%d",&N))

{

NX = NY = N;

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

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

scanf("%d",&Map[i][j]);

printf("%d\n",KM());

}

return 0;

}

网络流最大流EdmondKarp、SAP

EdmondKarp算法：

const int MAXN = 220; //最大点个数

int Map[MAXN][MAXN],pre[MAXN]; //Pre[]前驱数组

int N,NP,NC,M;

bool EkBFS(int start,int end) //宽度优先搜索寻找增广路

{

queue<int> Q; //宽度优先搜索队列

bool vis[MAXN]; //标记数组

memset(vis,false,sizeof(vis));

memset(pre,-1,sizeof(pre)); //初始化

Q.push(start);

vis[start] = true;

while( !Q.empty() )

{

int u = Q.front();

if(u == end) //当队头弹出的点位终点时即可判增广路 return true;

Q.pop();

for(int i = 0; i <= N+1; ++i)

{

if(Map[u][i] && !vis[i]) //当边容量为非零，且增广路点未标记

{

vis[i] = true;

pre[i] = u; //记录前驱

Q.push(i); //入队

}

}

}

return false;

}

int EkMaxFlow(int start,int end) //网络流的源点和汇点

{

int v,Ans = 0,MinN; //初始化最大流Ans = 0

while(EkBFS(start,end)) //当增广成功

{

MinN = 0xffffff0;

v = end;

while( pre[v] != -1)

{

MinN = min(MinN,Map[pre[v]][v]);

v = pre[v];

} //寻找"瓶颈"边，并且标记容量

Ans += MinN; //累加进最大流

v = end;

while( pre[v] != -1)

{

Map[pre[v]][v] -= MinN;

Map[v][pre[v]] += MinN;

v = pre[v];

} //修改路径上的边容量

}

return Ans;

}

//初始化

memset(Map,0,sizeof(Map)；scanf("%d%d",&；for(inti=0;i<M;++i)；scanf("%d%d%d"；Map[u][v]+=w;；printf("Case%d:%d\n；SAP算法：；constintMAXN=100010;//最大；constintMAX

memset(Map,0,sizeof(Map));

scanf("%d%d",&N,&M);

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

{

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

Map[u][v] += w;

}

printf("Case %d: %d\n",++kase, EkMaxFlow(1,N));

SAP算法：

const int MAXN = 100010; //最大点个数

const int MAXM = MAXN\*4;

const int INF = 0xffffff0;

struct EdgeNode

{

int to;

int next;

int w;

}Edges[MAXM];

int Head[MAXN],id; //链式前向星

void AddEdges(int u,int v,int w)

{

Edges[id].to = v;

Edges[id].w = w;

Edges[id].next = Head[u];

Head[u] = id++;

Edges[id].to = u;

Edges[id].w = 0;

Edges[id].next = Head[v];

Head[v] = id++;

}

int Numh[MAXN],h[MAXN],curedges[MAXN],pre[MAXN];

//Numh:用于GAP优化的统计高度数量数组;

//h:距离标号数组;

//curedges:当前弧数组

//pre:前驱数组

void BFS(int end,int N) //BFS求出每个点的距离标号值

{

memset(Numh,0,sizeof(Numh));

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

Numh[h[i]=N]++;

h[end] = 0;

Numh[N]--;

Numh[0]++;

queue<int> Q;

Q.push(end);

while(!Q.empty())

{

int v = Q.front();

Q.pop();

int i = Head[v];

while(i != -1)

{

int u = Edges[i].to;

if(h[u] < N)

{

i = Edges[i].next;

continue;

}

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

Numh[N]--;

Numh[h[u]]++;

Q.push(u);

i = Edges[i].next;

}

}

}

int SAPMaxFlow(int start,int end,int N)

{

int CurFlow,FlowAns = 0,temp,neck; //FlowAns:最大流，初始化为0 memset(h,0,sizeof(h));

memset(pre,-1,sizeof(pre));

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

curedges[i] = Head[i]; //初始化当前弧为第一条邻接边 //Numh[0] = N;

BFS(end,N);

int u = start;

while(h[start] < N) //当h[start] >= N时，网络中肯定出现了GAP

{

if(u == end)

{

CurFlow = INF;

for(int i = start; i != end; i = Edges[curedges[i]].to) {

if(CurFlow > Edges[curedges[i]].w)

{

neck = i;

CurFlow = Edges[curedges[i]].w;

}

} //增广成功，寻找"瓶颈"边

for(int i = start; i != end; i = Edges[curedges[i]].to) {

temp = curedges[i];

Edges[temp].w -= CurFlow;

Edges[temp^1].w += CurFlow;

} //修改路径上的边容量

FlowAns += CurFlow;

u = neck; //下一次增广从瓶颈边开始

}

int i;

for(i = curedges[u]; i != -1; i = Edges[i].next)

if(Edges[i].w && h[u] == h[Edges[i].to]+1) break;

//寻找可行弧

if(i != -1) //GAP优化

{

curedges[u] = i;

pre[Edges[i].to] = u;

u = Edges[i].to;

}

else

{

if(0 == --Numh[h[u]]) break;

curedges[u] = Head[u];

for(temp = N,i = Head[u]; i != -1; i = Edges[i].next) if(Edges[i].w) temp = min(temp,h[Edges[i].to]); h[u] = temp + 1;

++Numh[h[u]];

if(u != start) u = pre[u]; //重新标号并从当前点前驱重新增广

}

}

return FlowAns;

}

//初始化

memset(Head,-1,sizeof(Head));

id = 0;

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

{

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

AddEdges(u,v,w);

}

printf("%d\n",SAP(1,M,M));

输入输出外挂—仅适合纯数字输入 int Scan() //输入外挂

{

int res=0,ch,flag=0;

if((ch=getchar())=='-')

flag=1;

else if(ch>='0'&&ch<='9')

res=ch-'0';

while((ch=getchar())>='0'&&ch<='9') res=res\*10+ch-'0';

return flag?-res:res;

}

void Out(int a) //输出外挂

{

if(a>9)

Out(a/10);

putchar(a%10+'0');

}