Hello world ---note\_all

/\* 2-sat \*/

构图：构图的关键是找“冲突关系”，若a与b冲突，则可以连边a->b',b->a'.

每组2个元素，所以要把多于2个元素的情况压缩，如hdu 1824

关于输出解，有两种方法，一种是逆向topsort输出任意解（hdu\_1814\_3.cpp)，另一种是暴力dfs，可以输出最小字典序解（hdu\_1814.cpp）。

逆向topsort法：

(0)建立图G的补图G'，下面的讨论都是针对G'的。

(1)开队列Q1，Q2，遇到未染色点，加到Q1里，来一论topsort染色。

(2)topsort-enter:

从Q1里取出元素i，若它此时还是未染色，则染red，把~i加到Q2里

(3)不断地以Q2种的元素~i扩展开来，把~i的所有后继节点都进Q2，并染black

(4)对于之前加到Q1的i点的子节点都入度减1，若入度为0则加到Q1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(1) 判断是否有解 (AND,OR,XOR) hdu\_4421

/\* 题意：该题给出操作(AND,OR,XOR)和结果(b[i][j]), 判断是否有解

2012长春现场赛的题。

由于不细心，WA了几次。。多是建边的时候粗心

思路：该题给出操作(AND,OR,XOR)和结果(b[i][j])，只要把元素拆成bit形式来做2-SAT就行了。因为\*\*有解就必须每个位bit都有解\*\*，`且每一位bit不互相影响`。如果一下把31位都给拆来建图的话会ME，所以得一位一位地

来，之前我没意识到这一点，原因在没运用到`每一位不互相影响`。

建图时AND->1, 和or->0是重点。问“同组元素连边”有什么意义？达神跟

我解释：如果只是一组元素连边的话没意义，但是有多组元素连边时就有意>义了。举例，A AND B == 1, 连a->a', b->b'. 直观理解为这两组元素一定>得

选a'和b'。如果有解取了a和b，则建连通图的时候会把a也连上，同理有b/b'，则这个子连通图会包括a和a'，矛盾。（因为由于对称性，a组和b组肯定

会有至少两条边相连，所以a/a'/b/b'会在同一环里）

什么时候同组元素连边？--还没有比较好的理解...

不知道pre-check有没有必要...

bool build()

{

init();

FOR(row, 0, n) FOR(col, 0, row) {

int bit = (b[row][col] & 1);

int op;

if((row & 1) && (col & 1)) op = OR;

else if(!(row & 1) && !(col & 1)) op = AND;

else op = XOR;

int x = row, y = col, x1 = hash(row), y1 = hash(col);

if(op == OR) {

if(bit == 0) {

v[x1].push\_back(x); v[y1].push\_back(y);

} else {

v[x].push\_back(y1); v[y].push\_back(x1);

}

} else if(op == AND) {

if(bit == 0) {

v[x1].push\_back(y); v[y1].push\_back(x);

} else {

v[x].push\_back(x1); v[y].push\_back(y1);

}

} else if(op == XOR) {

if(bit == 0) {

v[x].push\_back(y); v[y].push\_back(x);

v[x1].push\_back(y1); v[y1].push\_back(x1);

} else {

v[x].push\_back(y1); v[y].push\_back(x1);

v[x1].push\_back(y); v[y1].push\_back(x);

}

}

}

FOR(row, 0, n) FOR(col, 0, row) b[row][col] >>= 1;

\*/

(2) 2-SAT 二分法求最佳 hdu\_3715

/\*

给出一个递归表达式， 问最多能递归到第几层？

思路： 构图：x表0, x'表1

c = 0, 则矛盾式子为a=0,b=0，所以可以连边a->b',b->a'

c = 1, 矛盾式为a=1,b=0或a=0,b=1,可以连边a'->b',b->a,a->b,b'->a

c = 2, 矛盾式为a=1,b=1,可以连a'->b,b'->a

\*/

/\* LRJ 的 2-SAT 模板 \*/

bool dfs(int u)

{

if(color[hash(u)]) return false;

if(color[u]) return true;

color[u] = 1;

S[S\_top++] = u;

for(int k = head[u]; k != -1; k = edge[k].next) {

int v = edge[k].v;

if(!dfs(v)) return false;

}

return true;

}

bool solve()

{

for(int i = 0; i < 2\*n; i += 2) // i += 2 ???

if(!color[i] && !color[hash(i)]) {

S\_top = 0;

if(!dfs(i)) {

while(S\_top > 0) color[S[--S\_top]] = 0;

if(!dfs(hash(i))) return false;

}

}

return true;

}

/\* AC 自动机 \*/

struct Trie {

Trie \* child[26];

Trie \* fail;

int cnt ;

Trie () {

cnt = 0, fail = NULL, memset(child, 0, sizeof(child));

}

};

Trie \*root, \* Q[MAXN \* 50];

void Insert(char \*str)

{

Trie \*p = root; int len = strlen(str);

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

int idx = str[i] - 'a' ;

if(p->child[idx] == NULL) p->child[idx] = new Trie;

p = p->child[idx];

}

p->cnt++;

}

void Fail() //自顶向下，一个节点一个节点找fail指针

{

int car = 0, cdr = 0;

Q[car++] = root;

while(cdr < car) {

Trie \* p = Q[cdr++];

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

if(p->child[i]) { //p有孩子节点i，则找i的fail指针

if(p == root) p->child[i]->fail = root; //与根节点相连的fail指针都指向根节点

else {

Trie \* tmp = p;

while(tmp->fail) { //父节点fail指针非空

if(tmp->fail->child[i]) {

p->child[i]->fail = tmp->fail->child[i]; //父节点的fail指针指向的节点有孩子i，则可以作为p的i孩子的fail指针

break;

}

tmp = tmp->fail; //到了这一步，说明现在的fail指针指向的节点没有孩子i，则fail指针继续往上跳

}

if(tmp->fail == NULL) p->child[i]->fail = root;

}

Q[car++] = p->child[i];

}

}

}

int Query(char \* str)

{

Trie \* p = root, \* tmp = NULL;

int len = strlen(str), ans = 0;

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

int idx = str[i] - 'a';

while(p != root && p->child[idx] == NULL) p = p->fail;

if(p->child[idx]) { // p 记录当前位置最长后缀匹配，下次从该支继续匹配

p = p->child[idx], tmp = p; //用tmp继续找当前位置较短的后缀匹配

while(tmp != root && tmp->cnt) {

ans += tmp->cnt;

tmp->cnt = 0; //

tmp = tmp->fail; //因为可能有多次匹配是吧~

}

}

}

return ans;

}

void Delete()

{

int car = 0, cdr = 0;

Q[car++] = root;

while(cdr < car) {

Trie \* p = Q[cdr++];

for(int i = 0; i < 26; i++) if(p->child[i]) Q[car++] = p->child[i];

delete p;

}

}

/\* AC自动机 + 矩阵快速幂 \*/

struct Matrix {

ll mat[105][105];

void init() { memset(mat, 0, sizeof(mat)); }

friend Matrix operator \* (const Matrix & a, const Matrix & b) ;

friend Matrix operator ^ (const Matrix & a, int x) ;

} M;

struct Trie {

Trie \* child[4], \* fail;

int isword, id;

} \* root, \*Q[100005], t[100005];

Matrix operator \* (const Matrix & a, const Matrix & b) {

Matrix ans; ans.init();

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

for(int k = 0; k < idx; k++) if(a.mat[i][k])

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

ans.mat[i][j] = (ans.mat[i][j] + a.mat[i][k] \* b.mat[k][j]) % MOD;

return ans;

}

Matrix operator ^ (const Matrix & a, int x) {

Matrix ans, tmp = a; ans.init();

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

for(int j = 0; j < idx; j++) ans.mat[i][j] = (i == j);

while(x) {

if(x & 1) ans = ans \* tmp;

tmp = tmp \* tmp;

x >>= 1;

}

return ans;

}

Trie \* NewTrie() //静态Trie

{

Trie \* p = &t[idx];

p->isword = 0, p->fail = NULL, p->id = idx++;

memset(p->child, 0, sizeof(p->child));

return p;

}

void Insert(char \* str)

{

Trie \* p = root; int len = strlen(str);

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

int idx = hash(str[i]);

if(!p->child[idx]) p->child[idx] = NewTrie();

p = p->child[idx];

}

p->isword = 1;

}

void Fail()

{

int car = 0, cdr = 0;

Q[car++] = root;

while(cdr < car) {

Trie \* p = Q[cdr++];

for(int i = 0; i < 4; i++)

if(p->child[i] == NULL) { //这部分跟一般ac自动机不一样

if(p == root) p->child[i] = root;

else p->child[i] = p->fail->child[i];

} else {

if(p == root) p->child[i]->fail = root;

else {

p->child[i]->fail = p->fail->child[i];

if(p->child[i]->fail->isword) p->child[i]->isword = 1; //! 不能改成直接赋值 //label

}

Q[car++] = p->child[i];

}

}

}

char str[20];

int main()

{

int n, m;

while(scanf("%d %d", &m, &n) != EOF) {

idx = 0;

root = NewTrie();

while(m--) {

scanf("%s", str);

Insert(str);

}

Fail();

memset(M.mat, 0, sizeof(M.mat));

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

for(int j = 0;j < 4; j++) {

Trie \* son = t[i].child[j];

if(!son->isword && !t[i].isword) M.mat[i][son->id] ++;

}

M = M ^ n;

ll ans = 0;

for(int i = 0; i < idx; i++) ans = (ans + M.mat[0][i]) % MOD;

printf("%I64d\n", ans);

}

return 0;

}

/\* 判断奇环 ======================================================================================\*/

void Find(int u, int c)

{

if(color[u] != -1) return ;

color[u] = c;

for(int k = head[u]; k != -1; k = edge[k].next) Find(edge[k].v, !c);

}

bool dye()

{

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

for(int i = 1; i <= n; i++) if(color[i] == -1) Find(i, 0);

for(int i = 0; i < edge\_num; i++) if(color[edge[i].u] == color[edge[i].v]) return false;

return true;

}

/\* ===============================================================================================\*/

/\* 堆优化的SPFA ==================================================================================\*/

typedef pair<int, int> PII;

bool SPFA(int s)

{

memset(times, 0, sizeof(times));

memset(dis, 127, sizeof(dis));

memset(inque, false, sizeof(inque));

dis[s] = 0;

priority\_queue< PII > Q; //默认pair按first增序

Q.push(PII(0, s)); //first表示距离，second才表示标号

while(!Q.empty()) {

int u = Q.top().second;

Q.pop();

inque[u] = false;

for(int k = head[u]; k != -1; k = edge[k].next) {

int v = edge[k].v;

if(dis[v] > dis[u] + edge[k].w) {

dis[v] = dis[u] + edge[k].w;

if(!inque[v]) {

inque[v] = true;

Q.push(PII(dis[v], v));

if(++times[v] > n) return false;

}

}

}

}

return true;

}

/\* ISAP \*/

void add\_edge(int u,int v,int w)

{

edge[tol].from=u;

edge[tol].to=v;

edge[tol].cap=w;

edge[tol].next=head[u];

head[u]=tol++;

edge[tol].from=v;

edge[tol].to=u;

edge[tol].cap=0;

edge[tol].next=head[v];

head[v]=tol++;

}

void BFS(int start,int end)

{

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

memset(gap,0,sizeof(gap));

gap[0]=1;

int que[MAXN];

int front,rear;

front=rear=0;

dep[end]=0;

que[rear++]=end;

while(front!=rear)

{

int u=que[front++];

if(front==MAXN)front=0;

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

{

int v=edge[i].to;

if(edge[i].cap!=0||dep[v]!=-1)continue;

que[rear++]=v;

if(rear>=MAXN)rear=0;

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

++gap[dep[v]];

}

}

}

int SAP(int start,int end)

{

int res=0;

BFS(start,end);

int cur[MAXN];

int S[MAXN];

int top=0;

memcpy(cur,head,sizeof(head));

int u=start;

int i;

while(dep[start]<n)

{

if(u==end)

{

int temp=inf;

int inser;

for(i=0;i<top;i++)

if(temp>edge[S[i]].cap)

{

temp=edge[S[i]].cap;

inser=i;

}

for(i=0;i<top;i++)

{

edge[S[i]].cap-=temp;

edge[S[i]^1].cap+=temp;

}

res+=temp;

top=inser;

u=edge[S[top]].from;

}

if(u!=end&&gap[dep[u]-1]==0)//出现断层，无增广路

break;

for(i=cur[u];i!=-1;i=edge[i].next)

if(edge[i].cap!=0&&dep[u]==dep[edge[i].to]+1)

break;

if(i!=-1)

{

cur[u]=i;

S[top++]=i;

u=edge[i].to;

}

else

{

int min=n;

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

{

if(edge[i].cap==0)continue;

if(min>dep[edge[i].to])

{

min=dep[edge[i].to];

cur[u]=i;

}

}

--gap[dep[u]];

dep[u]=min+1;

++gap[dep[u]];

if(u!=start)u=edge[S[--top]].from;

}

}

return res;

}

/\* KMP \*/

/\*

KMP的next求循环节：http://blog.sina.com.cn/s/blog\_7981299401012xl0.html

int len = s.length();

int rep = len - next[len];

int times = (len%rep == 0 ? len/rep : 1);

\*/

//next[] : 匹配失败时回跳的位置

void fail(string &s)

{

int len = s.length();

int i = 0, j = -1;

next[0] = -1;

while(i < len) {

if(j == -1 || s[i] == s[j]) {

i++, j++;

next[i] = j;

} else j = next[j];

}

}

//PMT[] : 部分匹配值

void fail(string s)

{

int len = s.length();

int j = 0;

PMT[0] = 0;

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

while(j > 0 && s[i] != s[j]) {

j = PMT[j-1];

}

if(s[i] == s[j]) j ++;

PMT[i] = j;

}

}

int KMP(string s1, string s2) //求s2在s1出线的次数（重叠的算两次）

{

int sum = 0, len1 = s1.length(), len2 = s2.length();

fail(s2);

int i = 0, j = 0;

while(i < len1) {

while(i < len1 && j < len2) {

if(s1[i] != s2[j]) j -= PMT[j];

else j++;

i++;

}

if(len2 == j) {

sum++;

j = 0;

i = i - len2 + 1;

}

}

return sum;

}

/\* LCA倍增法模板 \*/

void Get\_Depth(int root, int deep)

{

depth[root] = deep;

for(int Size = child[root].size(), i = 0; i < Size; i++) {

Get\_Depth(child[root][i], deep+1);

}

}

int LCA(int u, int v)

{

if(depth[u] < depth[v]) swap(u, v); //使得v比较接近根

int K = depth[u] - depth[v];

for(int i = 0; i < ANC\_SIZE; i++) //使得depth[u] == depth[v]

if(K & (1<<i)) u = ancester[u][i];

if(u == v) return v; //重要，别忘记了，因为下面的循环只能处理u != v的情况

for(int i = ANC\_SIZE-1; i >= 0; i--)

if(ancester[u][i] != ancester[v][i]) {

u = ancester[u][i];

v = ancester[v][i];

}

return ancester[u][0];

}

void work(int root)

{

Get\_Depth(root, 1); //确定depth关系

for(int j = 1; j < ANC\_SIZE; j++) //确定ancester的关系

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

ancester[i][j] = ancester[ancester[i][j-1]][j-1];

printf("%d\n", LCA(node[0], node[1]));

}

void init()

{

for(int i = 1; i <= n; i++) child[i].clear();

for(int i = 1; i <= n; i++) ancester[i][0] = -1;

}

/\* 中国剩余定理 CRT\*/

假设C ≡ a1 (mod m1)，C ≡ a2 (mod m2)

令C = m1\*x1 + a1，则m1\*x1 ≡ a2-a1(mod m2)，d = gcd(m1, m2)

由斐蜀定理知，若d不能整除(a2-a1)，则方程组无解

用扩展欧几里得可以求出m1\*y1 + m2\*y2 = d的任一解y1， 对应到m1\*x1 + m2\*x2 = a2-a1的解x1

就可以表示为x1 = y1\*(a2-a1)/d，C1 = m1\*x1 + a1

接下来我们要求最小正整数解C0。

我们知道，原方程m1\*x1 ≡ a2-a1(mod m2)的任意解C有C ≡ C1 mod m2/d //不会证

C = C1 + (m2/d)\*x，所以，最小正整数解C0 = [C1 % (m2/d) + m2/d] % (m2/d)

接下来是不断合并上面两个方程C ≡ a1 (mod m1)，C ≡ a2 (mod m2)并继续迭代下去，一直到最后

合并：原二方程可合并为C ≡ C0 (mod LCM(m1,m2))

//这里的C0是前面方程的最小公共解，我们还要继续迭代到只剩一个方程，找到所有方程的公共最

小解

下面摘一段“为什么上面两条方程就可以被C’ ≡ C (mod LCM(m1,m2))代替”的证明，目前看不懂~~

"

http://scturtle.is-programmer.com/posts/19363.html

题意有:C=A1+X1\*B1=A2+X2\*B2

因为B是lcm(B1,B2)所以有:

B=Y1\*B1=Y2\*B2

又C`=C+Z\*B,把C带入得:

C`=A1+(X1+Z\*Y1)\*B1=A2+(X2+Z\*Y2)\*B2

所以C`就符合最初的两个式子了吧,不清楚为何是最小的

"

\*/

/\*

对于m1，m2，…，mk是两两互质的情况，还有一种更加一般性的解法。

x=a1(mod m1);

x=a2(mod m2)

···

x=ak(mod mk)

该方程组在小于等于M = m1 \* m2 \* ··· \* mk的整数中有唯一解。

记Mi = M / mi;因为gcd(Mi,mi)=1，故有两整数pi，qi满足Mi\*pi + mi\*qi=1，如果记ei = Mi\*pi，

那么ei = 0(mod mj)，当j != i 时；ei = 1(mod mj)，j = i 时。

这时很明显，e1\*a1 + e2\*a2 + ··· + ek\*ak就是方程的一个解，加减M的任意整数倍后就可以得到

所有可行的解了。

\*/

//poj 2891 求解一般同余方程组(C≡ai mod mi不一定两两互质)，输入顺序为ai, mi

#define LCM(a1, a2, d) ( a1/d \* a2 )

ll EGCD(ll a, ll b, ll &x, ll &y)

{

if(b == 0) {

x = 1, y = 0;

return a;

}

ll d = EGCD(b, a%b, x, y);

ll t = x; x = y, y = t - a/b\*y;

return d;

}

void CRT(ll &m1, ll &a1, ll m2, ll a2, ll x, ll d) //引用！ x是m1\*x + m2\*y = d的任意解，在外面由扩展欧几里得求出来，d是gcd(m1,m2)

{

ll C, m, t;

t = m2 / d; //

x = (a2-a1)/d \* x; //x1 = x0\*(a1-a2)/d，x0是x\*m1 + y\*m2 = a1-a2的任意解

x = (x % t + t) % t; //这样就能保证x1是最小正整数解 ????????????????

C = x\*m1 + a1; //合并C ≡ a1 (mod m1), C ≡ a2 (mod m2)成 C' ≡ C (mod LCM(m1,m2))

m = LCM(m1, m2, d); //因为x是前两式最小解，所以C = x\*m1 + a1

//??????????? 为啥能合并???

a1 = C, m1 = m;

}

int main()

{

int n;

while(scanf("%d", &n) != EOF) {

ll a1, a2, m1, m2, x, y; bool flag = true;

scanf("%I64d %I64d", &m1, &a1);

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

scanf("%I64d %I64d", &m2, &a2); if(!flag) continue;

ll d = EGCD(m1, m2, x, y);

if((a2 - a1) % d != 0) { flag = false; continue; }

CRT(m1, a1, m2, a2, x, y, d);

}

printf("%I64d\n", flag ? a1 : -1);

}

return 0;

}

//poj 1006 m1，m2，…，mk是两两互质 的另外一种解法

//题意：使等式 （x+d）= p mod 23 = e mod 28 = i mod 33 成立的最小的x，注意x不能为0

int m[5] = {23,28,33,23\*28\*33}, a[5], d;

void EGCD(int a, int b, int &x, int &y)

{

if(b == 0) {

x = 1 , y = 0;

return ;

}

EGCD(b, a%b, x, y);

int t = x; x = y, y = t - a/b\*y;

}

int CRT()

{

int C = 0, M = m[3];

//for(int i = 0; i < 3; i++) M \*= m[i];

for(int i = 0; i < 3; i++) {

int Mi = M/m[i], x, y, ei;

EGCD( Mi, m[i], x, y );

ei = Mi\*x\*a[i];

C = (C + ei) % M;

}

return C;

}

int main()

{

int Cas = 0;

while( scanf("%d%d%d%d", &a[0], &a[1], &a[2], &d) ) {

if(a[0] == -1 && a[1] == -1 && a[2] == -1 && d == -1) break;

int C = CRT() - d;

C = C % m[3]; //要最小正整数解吧

if(C <= 0) C += m[3];

printf("Case %d: the next triple peak occurs in %d days.\n", ++Cas, C);

}

return 0;

}

/\* SPARSE TABLE \*/

int Query(int left, int right)

{

int j = (int)(log2(right-left+1));

return max(f[1][left][j], f[1][right-Bit[j]+1][j]);

}

int Set(int a, int x) //动态修改元素 看起来貌似代价不小耶

{

f[a][0] = x;

for(int j = 1; Bit[j] <= n; j++)

for(int i = max(x+1-Bit[j]) : 0; i < x && i + Bit[j] - 1 < n; i++)

f[i][j] = cmp(f[i][j-1], f[i+Bit[j-1]][j-1]);

}

void build()

{

for(int i = 0; i < n; i++) f[i][0] = a[i];

for(int j = 1; Bit[j] < n; j++)

for(int i = 0; i + Bit[j]-1 < n; i++) {

f[i][j] = max(f[i][j-1], f[i+Bit[j-1]][j-1]);

}

}

/\* KM 二分图带权匹配 \*/

//================ 下面这是改进的slack数组版本的KM ==================================================================

bool find(int u)

{

visx[u] = true;

for(int k = head[u]; k != -1; k = edge[k].next) {

int v = edge[k].v;

if(visy[v]) continue;

int rest = LX[u] + LY[v] - edge[k].w;

if(rest == 0) {

visy[v] = true;

if(link[v] == -1 || find(link[v])) {

link[v] = u;

ans[v] = k; //记录边

return true;

}

} else slack[v] = min(slack[v], rest); //u在交错树，v不在交错树，更新d 也就是说边(u,v)不在相等子图中

}

return false;

}

bool KM()

{

memset(LX, 128, sizeof(LX)); //最大匹配

memset(LY, 0, sizeof(LY));

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

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

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

LX[i] = max(LX[i], edge[k].w);

}

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

for(int j = 1; j <= m; j++) slack[j] = INF;

while(true) {

memset(visx, false, sizeof(visx));

memset(visy, false, sizeof(visy));

if(find(i)) break;

else {

int tmp = INF;

for(int j = 1; j <= m; j++) if(!visy[j]) tmp = min(tmp, slack[j]);

if(tmp == INF) return false; //无法松弛，找不到完备匹配 TODO TODO 这里判断是否有完备匹配

for(int j = 1; j <= n; j++) if(visx[j]) LX[j] -= tmp;

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

if(visy[j]) LY[j] += tmp;

else slack[j] -= tmp; //修改顶标后，要把所有的不在交错树中的Y顶点的slack值都减去d。

}

}

}

return true;

}

/\* 博弈 \*/

/\*

[定理](SJ 定理)

对于任意一个 Anti-SG 游戏,如果我们规定当局面中所有的单一游

戏的 SG 值为 0 时,游戏结束,则先手必胜当且仅当:(1)游戏的 SG 函

数不为 0 且游戏中某个单一游戏的 SG 函数大于 1;(2)游戏的 SG 函数

为 0 且游戏中没有单一游戏的 SG 函数大于 1。

\*/

/\*

sg函数只会用，不会证明啊...

今天做到一道sg，是这样的，一个长度为n的珍珠环，每次只能选连续的m个未染色的珍珠染色，轮到谁不能染就输。

问题很容易变为，一条长度为n-m的珍珠链，轮流选连续m个未染色的珍珠染上色。

假设长度为n的链，我们染上i,i+1,..,i+m-1，则原链n就变成两条子链(i-1),(n-(i+m-1))

也就是说，当前链n可以变成两个后继状态(i-1),(n-i-m+1)，利用sg函数性质抑或之.

for(i = 1; i < n-m; i++) vis[ sg(i-1) ^ (n-i-m+1) ] = true;

\*/

/\*

博弈 二维sg 好题

题意(hdu\_4111)：n堆石子，每次可以有两种操作：1、拿走一颗石子。2、将两堆石子合并

思路：

之前做过一堆可以分成两堆的，但是两堆合成一堆的就不知道怎么做了。看了题解，知道需要分析一些东西，感觉自己在现场赛肯定做不出来

假设所有堆都有多余1的石子，则因为每次可以拿走一颗石子变成sum-1，也可以合并变成sum，即奇偶性试试可以随意确定的（合并a堆和b堆的>石子相当于对一堆(a+b+1)的石子减1）

“然而，特殊情况出现了。当某些堆石子只有一颗的时候呢？只要本来要输的那个人有机会将这颗石子拿走，本来要赢的那个孩子就没有机会将>这一堆合并了！整体奇偶性一改变，输家可能变成了赢家！” （无法逆转奇偶）

sg[x][y]表有x堆石子数为1的堆，y代表剩下的∑ai+tot-1, ai>1, tot为ai>1的堆数

int SG(int x, int y)

{

if(x == 0) return y&1;

if(sg[x][y] != -1) return sg[x][y];

if(y == 1) return sg[x][y] = SG(x+y, 0); //不加这一句就会WA，为什么？

bool vis[7];

memset(vis, false, sizeof vis);

if(x >= 2) vis[ min(5, SG(x-2, y+2+(y?1:0))) ] = true;

if(x >= 1 && y > 0) vis[ min(5, SG(x-1, y+1)) ] = true;

if(x >= 1) vis[ min(5, SG(x-1, y )) ] = true;

if(y >= 1) vis[ min(5, SG(x, y-1)) ] = true;

for(int i = 0; ; i++) if(!vis[i]) return sg[x][y] = i;

}

\*/

/\* Bash/Wyhoff 游戏 ========================================================\*/

Bash Game: 只有一堆石子，一次最多取m个：S % (m+1) == 0，则必败；反之必胜

Wyhoff Game: 两堆石子，一次能从某一堆里取任意多，或者从两堆里取相同多的石子：xi = [i\*(√5+1)], yi = [i\*(√5+3)] = xi+i

bool Wyhoff\_judge(long long A, long long B) //判断(A,B)是否为必胜态

{

#define gold\_cut ((sqrt((double)5.0) + 1) / 2)

double k = B - A;

return (long long)(k \* gold\_cut) != A;

}

/\* 阶梯博弈 ================================================================\*/

阶梯博弈：游戏开始时任意多硬币分布在楼梯上，共n阶楼梯，从地面到最上层楼梯编号为0-n。游戏者轮流将某一阶梯j上任意多的硬币移动到下一个阶梯j-1上，将最后一枚硬币移至地上的人获胜。

解法：将所有奇数阶梯看成N堆石子，做Nim

/\* 差分约束 \*/

差分约束(difference constraints)，对，两个关键字要理解好，“difference”简单理解就是两个节点的“差”，对应的就是图中的边权，而“约束”对应的是

图的边。这个图的边权不一定都是正数，之前我一直很奇怪为什么做最短路的时候初始化dis[]为了0也可以，那是因为我没意识到边权可以为负数，而思维

定势地想初始化dis[]为0，那0不就是最小路径了吗，但这里差分约束的最短路径常常是负数的，所以最短路径可以不是0！！

看网上讲解的时候要小心，很多人把最长路和最短路是不分的，乱死了。

还有很重要的一点很多人没区分开，

求最小可行解 ==> 把不等式划为dv >= dx + Z的形式，即建立<u,v,Z>边 ==> 可行解要最小，其实就是取约束条件中`最大`的约束 ==> 求最长路

解释：为什么求最小可行解要划成dv >= dx + Z形式？因为这个形式暗指了“让dv尽量小”，因为此刻dv的取值区间为[du+Z, ∞]。

为什么可行解最小，即意味着取最大约束条件？这样想，如果有dv >= du + Z1, dv >= du + Z2，(Z1<Z2)，那dv的最小取值就是du+Z2，因为du+Z1不满足第二个约束条件。

最后一步就好理解了，因为建图的边权就是约束值，既然上一步指要取最大约束，那当然是求最长路啦。

网上很多讲解没有区分开所谓的最大/最小，一会儿指可行解的最，一会儿指约束条件的最，弄得我乱了好久。

顺便贴一下：

求最大可行解 ==> 把不等式划为dv <= dx + Z的形式，即建立<u,v,Z>边 ==> 其实就是取约束条件中`最小`约束 ==> 求最短路

关于源点：很多时候额外价格源点可以帮我们把一个非连通图变成连通图，而对于源点的不等式，一定要和你之前建边时的不等式形式一样，如果之前

是dv >= du + Z，那源点也要dv >= d0 + xxx。这个xxx就是dis[]的初始值，关于如何选取xxx，下面两句话摘自百度百科：

“

1.如果将源点到各点的距离初始化为0，最终求出的最短路满足 它们之间相互最接近了

2.如果将源点到各点的距离初始化为INF(无穷大)，其中之1为0，最终求出的最短路满足 它们与该点之间相互差值最大。

”

差分约束题目我一般是用SPFA+栈，为什么不用dijkstra+heap？因为dijkstra不能处理负环，而我们的题目可能有负环，所以干脆都用SPFA了，多数条

件下，用stack的SPFA比用queue的快，why？因为常常地，用最先更新了的点去更新其它点，效果比用以前已经更新了的点(在queue的tail)好。

poj 1201 差分约束

题意：求符合题意的最小集合Z的元素个数，约束条件:i j C，表区间[i,j]至少有C个Z集合的元素。

隐含条件是，S区间是个连续的数字区间，0 <= s[i+1] - s[i] <= 1，其中s[i]表0~i中有多少数字是Z集合元素。下面是隐含条件的建边。

for(int i = 0; i < 50001; i++) { //@

vert[i].push\_back(i+1); edge[i].push\_back(0);

vert[i+1].push\_back(i); edge[i+1].push\_back(-1);

}

poj 1364 差分约束

题意：约束条件：i, n, op, K --> op分greater和less，需要满足Si + S[i+1] + S[i+2] + ... + S[i+n] > K （或小于）

因为我是用dis[i]表示S0+S1+...+Si的和，所以<u,v,w>应该表示的意思是sum[v]-sum[u-1] = w，所以这里0也是一个点，所以源点不能取0！

//@ ?? 我在SPFA后面输出了dis[]数组来看，这些值并不符合题目的要求，那为什么整个程序是对的？如果要输出一个解的话怎么写？

答：因为这里的dis[i]表示的是s1+s2+...+si的和，用第一个样例来说，

sample input:

4 2

1 2 gt 0

2 2 lt 2

输出的dis[0--n] : -1 0 0 0 0

s1+s2+s3 = sum[3] - sum[1-1] = dis[3] - dis[0] = 1 , 满足gt 0

s2+s3+s4 = sum[4] - sum[2-1] = dis[4] - dis[1] = 0 , 满足lt 2

所以，{si}的一组解应该为0,1,0,0,0.

poj 1983 差分约束

题意：给出两种约束条件，一种是P A B X，意为精确地约束A比B远X个单位；另一种V A B，意为模糊地约束A至少比B远1个单位。是否有可行解？

好点：两种约束条件，其中Precise约束可以转换为X <= A-B <= X

有V i i 这种数据，这种数据在SPFA里会WA，在ballman\_ford里AC，不过预处理一下就可以了，还是用SPFA.

/\* LCA 最小公共祖先 Tarjan \*/

/\*

最小公共祖先

题意: 给出一颗无向有边权树, 询问若干个(u,v)对的距离.

所谓LCA 的Tarjan算法, 实际上就是在建树的过程中把query中的lca给计算出来, 所以称为`离线算法` . 是的, 本质就是这么简单, 好多解释都搞复>杂了.

步骤略, 自己google.

理解这个算法一定要抓住`递推`的思想(也有递归在里面, 也要抓住).

Tarjan是利用并查集实现的, 在递推过程中, 一棵树的root节点代表这棵树(联系并查集来想), 这样做的好处是, 如果问点i与j的lca, 我们只要找i,j都属于的最小的哪棵子树就行了, 因为该子树就是我们的答案. 那如何确定是那颗子树呢? 这一点后面再解释一下.

下面说Tarjan最巧妙的点了. 因为是在建树的过程中计算所有query, 也就表示我们此刻是否能计算某一query对(u,v)的条件是 : u和v是否都已经遍历

过. 所以我们可以在遍历到点v(假设经历v的时间比u晚)的时候把query给计算出来. 比如lcm(u,v)就是find(u). 那此刻的find(v)和lcm(u,v)相不相等呢? 答案是不相等, 至少在我的代码实现上不相等. 因为father[x]的更新是在`递归回去`的时候更新的, 而此刻在遍历v点, 还没递归回去呢, father[v]当然>也就没更新啦.

其实上一段就已经回答了`如何确定哪棵子树是我们想要的答案`这一问题了. 就是find(u)所代表的子树! 注意, 是find(u), 不是find(v)! 因为u是在

v之前已经被遍历过了, 并且递归回去过sub\_root过了, 也就是father[u]被更新为sub\_root了, 所以find(u)可以代表当前的sub\_tree, 即`最小包含(u,v)>子树`

void Tarjan(int now, int value)

{

visit[now] = true;

dis[now] = value;

for(int Size = v[now].size(), i = 0; i < Size; i++) {

int tmp = v[now][i];

if(visit[tmp] != 0) continue;

Tarjan(tmp, value + w[now][i]);

Union(now, tmp); //注意顺序, 先Tarjan子节点tmp, 再更新其father[tmp], 因为要保证在递推tmp所代表的子树时, father[tmp] = tmp, 而与当前子树无关. 递归回来的时候再把tmp代表的子树`并入`到当前树里

}

for(int Size = query[now].size(), i = 0; i < Size; i++) {

int tmp = query[now][i];

if(!visit[tmp]) continue; //若visit[tmp] == true, 即表示tmp节点已经遍历过, 此时可计算相应的query

ans[ans\_num[now][i]] = dis[now] + dis[tmp] - 2 \* dis[find(tmp)];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MCMF模板\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int solve()

{

int ans = 0;

while(SPFA(src, sink)) {

int min\_f = INF;

for(int k = pre[sink]; k != -1; k = pre[edge[k].u]) //寻找颈项边

min\_f = min(min\_f, edge[k].f);

for(int k = pre[sink]; k != -1; k = pre[edge[k].u]) {

edge[k].f -= min\_f;

edge[k^1].f += min\_f; //因为互为相反边是两两相近的。 因此在建图的时候要注意一下

}

ans += min\_f \* dis[sink];

}

return ans;

}

bool SPFA(int src, int sink)

{

memset(in\_queue, false, sizeof(in\_queue));

//memset(pre, -1, sizeof(pre)); //不用初始化pre

memset(dis, 127, sizeof(dis));

pre[src] = -1;

in\_queue[src] = true;

dis[src] = 0;

deque<int> Q;

Q.push\_back(src);

while(!Q.empty()) {

int u = Q.front();

Q.pop\_front();

in\_queue[u] = false;

for(int k = head[u]; k != -1; k = edge[k].next) {

int v = edge[k].v;

if(edge[k].f > 0 && dis[v] > dis[u] + edge[k].value) {

dis[v] = dis[u] + edge[k].value;

pre[v] = k; //记录其前驱边k

if(!in\_queue[v]) {

Q.push\_back(v);

in\_queue[v] = true;

}

}

}

}

return dis[sink] != INF;

}

/\*

串的最小表示法

令i=0,j=1

如果S[i] > S[j] i=j, j=i+1

如果S[i] < S[j] j++

如果S[i]==S[j] 设指针k，分别从i和j位置向下比较，直到S[i] != S[j]

如果S[i+k] > S[j+k] i=i+k

否则j++

返回i和j的小者

注意到上面两个算法唯一的区别是粗体的一行。这一行就把复杂度降到O(n)了。

值得一提的是，与KMP类似，最小表示法处理的是一个字符串S的性质，而不是看论文时给人感觉的处理两个字符串。

应用最小表示法判断两个字符串同构，只要将两个串的最小表示求出来，然后从最小表示开始比较。剩下的工作就不用多说了。

\*/

int min\_Presentation(char \*s, int len)

{

int i = 0, j = 1, k = 0;

while(i < len && j < len && k < len) {

int res = s[(i+k)%len] - s[(j+k)%len];

if(res == 0) k++;

else {

if(res > 0) i = i + k + 1;

else j = j + k + 1;

if(i == j) j++;

k = 0;

}

}

return min(i,j);

}

//树的最小表示 树的同构

string min\_Presentation(int u, int pre) //树的最小表示

{

vector<string> sub;

string res = "(";

for(int k = head[u]; k != -1; k = edge[k].next) {

int v = edge[k].v;

if(v == pre || !vis[v]) continue;

sub.push\_back( min\_Presentation(v, u) );

}

sort(sub.begin(), sub.end());

for(int Size = sub.size(), i = 0; i < Size; i++)

res += sub[i];

res += ")";

return res;

}

/\* 差值博弈 \*/

/\*

差值博弈 极大极小 博弈DP

题意：一堆数，两个小朋友，轮流从堆里面取数，要求第一个人取的数在[a,b], 0 <= a <= b区间内，每次取的数x和前一次(也就是对方)取的数y满足

a <= x-y <= b。两人都想使得自己取得的数总和减去对方的总和的差值尽量大，问最后第一个小朋友减去第二个小朋友的差值最大为多少？

思路：差值博弈

易推出后面取的数总是比前面取的数大，所以先排序

dp[i]表当前选手确定选了i之后，可以达到的最大分差

两人都想让自己的与对方的分差尽量大，而当前选手选了i之后，是轮到对方选。假设当前选手A选了i之后，先手变成了对方B，那B当然在后面>的状态种也是要使得自己与A的分差尽量大，所以A选了i的最优分差A-B = num[i] - max{DP(j)}

即dp[i] = num[i] - max{ dp[j] } , a <= num[j] - num[i] <= b

算法是，枚举先手取的第一个数，然后记忆化dp。

感觉这种差值博弈就是，假设先手A取了i，接着后面B对方因为要使得A-B最小，所以B就会选一个最大的dp[j]，使得A-B = min{ num[i] - dp[j] } = num[i] - max{dp[j]}

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*

\* 题目要求先手的最大值，可以转化为求先手-后手的最大值 \*

\* \*

\* dp[i]表示先手面临状态i时，先手进行决策后，先手-后手的最\*

\* 大值 \*

\* 程序在进行递推时，状态要从后往前进行枚举。也可以采用记 \*

\* 忆化搜索的方式，正向思考比较方便 \*

\* \*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*

\*/

int DP(int now)

{

if(dp[now] != -inf) return dp[now];

int ans = -inf;

for(int i = now+1; i < n; i++) if(num[i] - num[now] >= a) {

if(num[i] - num[now] > b) break;

ans = max(ans, DP(i));

}

if(ans == -inf) ans = 0; //说明后面没有数可以选了

return dp[now] = num[now] - ans;

}

/\*

差值博弈 极大极小博弈 好题

题意：盒子里有n个圆饼，两个人轮流从盒子中拿出一些圆饼，至少拿一个，至多m个。当一个人在某一轮拿走盒子中所有的圆饼后，他吃掉自己拿走的

圆饼，另一个人把他拿到的圆饼全部放回盒子中。然后游戏重新开始，由没有吃圆饼的那个人先开始。当所有的圆饼全部被吃掉后，游戏结束。假设双方都

采取最优策略，求先手吃掉圆饼的最大值。

思路：题目要求先手的最大值，可以转换成求先手-后手的最大值，因为总的圆饼数量为n，所有先手的最大值 = (ans+n) / 2

设dp[n][j][k]为还没被吃掉的饼为n个，前后之前取了j个，后手之前取了k个，先手-后手的最大值

(1) 先手可以一次性拿走盒子中所有饼，即盒子中剩余饼不足n

f[n][j][k] = max{-f[k][0][0] + n-k} //不足n，所以先手肯定直接取完，先手就能吃掉j+(n-j-k) = n-k个饼。然后接下来是后手作为

总数为k的新一轮的先手。 易证明此时先手肯定直接取完，这是DP无后效性和最优性的关键

(2) f[n][j][k] = max{-f[n][k][j+x]}, 1 <= x <= n, j+k+x < i，先手取x个，轮到后手

最后ans = (f[n][0][0] + n) / 2

感觉这个状态表示好奇葩......

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*

\* 题目要求先手的最大值，可以转化为求先手-后手的最大值 \*

\* \*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*

\*/

int DP(int n, int j, int k)

{

if(dp[n][j][k] != oo) return dp[n][j][k];

if(n-j-k <= m) return -DP(k, 0, 0) + n-k;

int ans = oo;

for(int i = 1; i <= m && j+k+i <= n; i++) {

ans = max(ans , -DP(n, k, j+i));

}

return dp[n][j][k] = ans;

}

int main()

{

memset(&oo, 128, sizeof(oo));

int n;

while(scanf("%d %d", &n, &m) != EOF) {

memset(dp, 128, sizeof dp);

dp[0][0][0] = 0;

printf("%d\n", (DP(n, 0, 0)+n)/2);

}

return 0;

}

/\*

差值博弈 极大极小 状压DP 好题 实现

题意：题目大意：有G种颜色的宝石，B（B<=21）个背包，每个背包中有一些宝石。Alice和Bob轮流每次拿走一个背包，把背包中的宝石放进熔炉中。>当一名角色X把一个背包中的宝石放进熔炉中后，熔炉中某种颜色的宝石有S个，那么这S个宝石变成一颗魔法石，角色X获得这颗魔法石（如果形成了多颗魔

法石，角色X获得多颗魔法石）。如果角色X在某一轮中获得了魔法石，那么他可以接着额外进行一轮游戏，如果在额外的一轮游戏中，又获得了魔法石，那

么他又可以继续进行一轮游戏。游戏一共进行B轮，若两人都采取最优策略，求游戏结束后，Alice获得魔法石数量-Bob获得魔法石数量的最大值。Alice先>进行游戏。

思路：

实现比较好看，我是抄的

dp[i]表还没拿走的状态为i，先手-后手的最大差值

dp[(1<<B)-1] = 0

分情况转移，tp = get(j->i), get(j->i)表示在状态j的情况下取了背包x能获得的魔法石数量

tp > 0 :

dp[i] = max{dp[j] + get(j->i)} //j状态取了背包x后变成状态i

tp = 0 :

dp[i] = max{-dp[j]}

状态转移用的是递推

我觉得最好的部分是预处理出get(j->i)这部分，我就是死活写不出来，看题解的。

low(x) : x为1的最低位

x ^ low(x) : x去掉为1的最低位

get[i][j] 表状态i中共有j类宝石的个数，get[i][j] = get[pre][j] + num[cur][j]，其中pre状态取了cur包后变成状态i

sum[i] 表示状态i一共可以拿的魔法石个数，sum[i] = ∑(get[i][j] / S), j = 1..G

\*/

while(scanf("%d %d %d", &G, &B, &S), G || B || S) {

memset(num, 0, sizeof num);

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

scanf("%d", &c);

while(c--) {

scanf("%d", &x);

num[i][x]++;

}

}

memset(get[0], 0, sizeof get[0]);

for(int oo = (1 << B), i = 1; i < oo; i++) {

int pre = i ^ low(i), cur = Log[low(i)];

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

get[i][j] = get[pre][j] + num[cur][j];

}

for(int oo = (1 << B), i = 0; i < oo; i++) {

sum[i] = 0;

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

sum[i] += (get[i][j] / S);

}

memset(dp, 128, sizeof dp);

dp[(1<<B) - 1] = 0;

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

for(int j = i; j; j -= low(j)) {

int pre = i ^ low(j);

int tp = sum[i] - sum[pre];

if(tp) dp[pre] = max(dp[pre], dp[i] + tp);

else dp[pre] = max(dp[pre], -dp[i]);

}

}

printf("%d\n", dp[0]);

}