Hello World ! ---hdu

/\*

构造 二分 规律

题意：0-N有多少个数i满足：i + (i+1) + (i+2) 没有任何进位，如2 + 3 + 4和131 + 132 + 133没有进位，3 + 4 + 5就进位了。

思路：小朱和高阳推出的规律

我是先预处理按位长构造出所有i，然后二分

构造是这样的，会发现，i有这样的规律：

一位: 0, 1, 2

两位: 10, 11, 12

三位: 100, 101, 102

110, 111, 112

120, 121, 122

130, 131, 132 ...

...

易证，不会存在xx9, x10, x11这样的

\*/

void init()

{

for(int i = 0; i < 3; i++) v[1].push\_back( i );

for(VLL it = v[1].begin(); it != v[1].end(); it++) {

for(int i = 1; i < 4; i++) v[2].push\_back( i\*10 + \*it );

}

ll tmp = 100;

for(int i = 3; i <= 10; i++) {

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

for(VLL it = v[j].begin(); it != v[j].end(); it++)

for(int k = 1; k < 4; k++)

v[i].push\_back( k\*tmp + \*it );

tmp \*= 10;

}

idx = 0;

for(int i = 1; i <= 10; i++)

for(VLL it = v[i].begin(); it != v[i].end(); it++)

a[idx++] = \*it;

sort(a, a+idx);

//for(int i = 0; i < 100; i++)printf("%lld\n", a[i]);

}

/\*

二分图最大匹配 经典题

题意：一张nxn的图，图上有的格子有东西挡着，问一个在图上最多放几个炮台能覆盖整张图，且不会火力部重叠。若两个格子同行或同列且中间没有东西挡住的话就会重叠。

思路：经典二分图最大匹配，让我对格子二分图匹配有了更好的理解。

建图：依然还是行元素在二分图的X部，列元素在Y部。不同的是，因为有'#'阻断元素的存在，若某一行被'#'分成两部分，则要把该行分成两部分，对应于二分图的话就是拆成两个点，当然都属于X部。分成几部分就是几个点。

列元素同理。

代码里面用count-row和count\_col统计行/列元素个数。其实就是把n\*n分成多少个区间。

接着是连边：若行区间i 和 列区间j有交集的话则连边<i,j> 代码里面用path[i][j]表示，比较低效，随便了。

\*/

int main()

{

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

init();

for(int row = 0; row < n; row++) cin >> map[row];

for(int row = 0; row < n; row++) { //拆分行区间

int p = 0;

while(p < n) {

if(map[row][p] == 'X') count\_row++;

else R[row][p] = count\_row;

p++;

}

count\_row++;

}

for(int col = 0; col < n; col++) { //拆分列区间

int p = 0;

while(p < n) {

if(map[p][col] == 'X') count\_col++;

else C[p][col] = count\_col;

p++;

}

count\_col++;

}

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

for(int col = 0; col < n; col++) if(map[row][col] == '.')

path[R[row][col]][C[row][col]] = true;

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

}

/\*

最短路

以前对图论的理解只停留在“路径模型”上，这道题让我觉得，所谓的最短路，其实就是最优化问题。

问s-->s能够到达的利率能否大于1,其实就是问利率最优化问题。。路径 = 利率~

题意：若干种货币及其之前的转换利率，问有没有可能经过若干次货币转换取得利益？

思路：可以floyd，这里用的是SPFA，每个点都来一次SPFA，如果dis[s] > 1.0则return true;

最优路径 = 最大利率

尝试了一下SLF优化，可是貌似数据太小，没体现出来

\*/

/\*

分组背包模板

memset(f, 128, sizeof(f));

f[0] = 0;

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

for(int j = m; j >= 1; j--) //逆向

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

f[j] = max(f[j], f[j-k]+a[i][k]);

\*/

/\*

博弈 棋盘博弈 好题 SG的0/1性质

题意：题意：给你一个n\*m的矩形，0表示空着的，1反之，现在两个人轮流放2\*2的矩形，谁不能放了，谁就输了。

思路：sg

从这道题知道了SG在“如果必胜态的所有后继都是必败态”情况时，SG是具有0/1性质的。也就是说，只要知道状态S的某一邻接状态(S->T)为0，则S的sg肯定为1.

不过我不知道怎么找出这个性质的。

\*/

//hdu 1760

bool check(int i, int j)

{

return ! (a[i][j] || a[i][j-1] || a[i-1][j] || a[i-1][j-1]);

}

void label(int i, int j)

{

a[i][j] = a[i][j-1] = a[i-1][j] = (a[i-1][j-1] ^= 1);

}

bool SG()

{

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

for(int j = 2; j <= m; j++) if(check(i, j)) {

label(i, j);

if(!SG()) {

label(i, j);

return true;

}

label(i, j);

}

return false;

}

/\*

2-SAT

构图: 其实可以把一个队的非队长的两个队员看成一个, 所以p[x] = y就够了

\*/

/\*

最短路, 终点集合到s的最远距离最短，求s. 即已知终点集{d}求一s使得Min{ max{ dis(s, di) } }

好题

思路： 多次单源最短路，选出最大值

在对每个x进行分层搜索的过程中, 用max\_d[y]记录每个地区x到达地区y的最短距离中的最大值. 最后求得的Star Value就是max\_d[]中的最小值.

由于题目的特殊性`边权都为1`，所以可以借助这一性质变换一下SPFA使其更快。

说个题外话，在临高时看到有个学弟拓扑排序用到“分层思想”，一直觉得很妙。就是拓扑后我们可以得到floor[i]，如果floor[i] > floor[j]，即说明j是i的前驱节点（层数越小越接近root）; 而floor[i] == floor[j]的话则i，j的相对顺序无所谓，因为他们都在“同一层”。

这里因为边权都为1，所以SPFA可以用到上述的分层思想，层数越高，离source越远。代码里面floors就表示层数，Q是滚动队列，就是一层一层地relax后继节点。

注意！！千万不要以为max\_d[]是最短路算法里面的dis[]，这里的max\_d[i]是到点i到终点集合{di}的最大值！而常规最短路算法里的dis[]已经被省略为“层数”了，不需要记录，所以没开数组。

\*/

int main()

{

int cases, query, id, m, y, x;

scanf("%d", &cases);

while(cases--) {

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

init();

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

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

while(m--) {

scanf("%d", &y);

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

}

}

int when = 0;

while(query--) {

scanf("%d", &m);

while(m--) {

scanf("%d", &x);

SPFA(x, ++when);

}

}

int ans = INF, ans\_id = -1;

for(int i = 1; i < MAXN; i++) if(!v[i].empty() && max\_d[i] < ans) ans = max\_d[i], ans\_id = i;

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

}

return 0;

}

void SPFA(int s, int when)

{

deque<int> Q[2];

int cur = 0;

Q[cur].push\_back(s);

max\_d[s] = max(max\_d[s], 1);

visit[s] = when;

int floors = 1;

do {

floors++;

while(!Q[cur].empty()) {

int at = Q[cur].front();

Q[cur].pop\_front();

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

int to = v[at][i];

if(visit[to] != when) { //是否已入队

//max\_d[to] = max(max\_d[to], max\_d[at]+1); 这句是不对的，因为这个分层跟拓扑排序的分层是不一样的，拓扑排序是要在入度为0时才能加进队Q，所以可以这样写，但是这里只要第一次遇见点to就必须得入队，因为要的是最短路径

max\_d[to] = max(max\_d[to], floors); //不把这句放在if外面，因为这里的max\_d[to]是距离s的最短路径，最短路径也就是最小层数，最小层数在to第一次入队的时候已经得到了

visit[to] = when;

Q[1-cur].push\_back(to);

}

}

}

cur = 1 - cur;

} while(!Q[cur].empty());

}

/\*

最短路 好题 给定两对{s,t}，问这两条最短路上的最多公共点数。

思路：题解说“容易验证相交公共点连续“，可是我没尝试证过。。

我们用dp[i][j] 代表 从点i到点j最短路上最多有多少个点！

那么 map[s1][i]+map[i][j]+map[j][e1]=map[s1][e1] 不就表示i到j的最短路为 s1到e1最短路的子路嘛；

我们只需更新dp[i][j]中的最大值即可

摘抄： 显然，两条最短路径里面公共的那部分子路径一定

是连续的（如果一定要间断的话，那么说明有更短的路

径）。

\*/

void init()

{

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

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

map[i][j] = INF;

dp[i][j] = 2;

}

dp[i][i] = 1;

map[i][i] = 0;

}

}

int main()

{

int m, u, v, w, s1,s2,t1,t2;

while(scanf("%d%d", &n, &m), n+m) {

init();

while(m--) {

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

map[v][u] = map[u][v] = min(map[u][v], w);

}

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

for(int i = 1; i <= n; i++) if(i != k)

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

if(map[i][j] > map[i][k] + map[k][j]) {

map[i][j] = map[i][k] + map[k][j];

dp[i][j] = dp[i][k] + dp[k][j] - 1;

} else if(map[i][j] == map[i][k] + map[k][j] && dp[i][j] < dp[i][k] + dp[k][j] - 1) {

dp[i][j] = dp[i][k] + dp[k][j] - 1;

}

}

scanf("%d%d%d%d", &s1,&t1,&s2,&t2);

int ans = 0;

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

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

if(map[s1][i] + map[i][j] + map[j][t1] == map[s1][t1] &&

map[s2][i] + map[i][j] + map[j][t2] == map[s2][t2])

if(ans < dp[i][j]) ans = dp[i][j];

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

}

return 0;

}

/\*

最短路 好题

题意：给出边建图，然后分别删除各条边，问每一次删边后的所有端点的两两最短路之和，若有一对端点不连通，则返回INF

思路：暴力解法是每次删边后都来n次最短路。这里面的冗余就是删除的边并不影响一些点的最短路树，所以这些点可以不用在删边后都来次dijkstra。标程解法就是在暴力解法上加上一些剪枝。先预处理出所有点的最短路树，记x的最短路树的和为sum[x]。现在来去掉冗余：在预处理时用used[x][u][v]记录点x的最短路树是否用到了边u--v，则删除边u--v的时候，判断点x的最短路树是否用到边u--v的，若用到，则对x做一次dijkstra，用新的sum[x]表示当前最短路树；否则用预处理的sum[x]就可以，不用再dijkstra.

dijkstra是利用`边权为1`这一特性来加快的版本，具体看:http://www.cppblog.com/keroro/archive/2013/05/27/200621.html

这道题有很多重边，这估计也是考点之一，不好好处理重边的话会超时。

\*/

int cnt[MAXN][MAXN]; //cnt[u][v]表u--v的边有多少条，用来处理重边

bool used[MAXN][MAXN][MAXN]; //used[x][u][v]x的最短路树是否用到了边u--v

int main()

{

int when = 0;

int u, v;

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

init();

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

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

vertex[u].push\_back(v);

vertex[v].push\_back(u);

edge[i].u = u, edge[i].v = v;

cnt[u][v]++, cnt[v][u]++;

}

int ans = 0;

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

dijkstra(i, ++when, 1);

ans += sum[i];

}

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

int tmp = ans;

int u = edge[i].u, v = edge[i].v;

//forbid\_u = edge[i].u, forbid\_v = edge[i].v; 因为有重边所以不能用这种方法

for(int j = 1; j <= n; j++) if(used[j][u][v] && cnt[u][v] == 1) { //不加cnt[u][v] == 1会超时。。卡的就是重边，靠！

int tmp = sum[j];

sum[j] = 0;

//vector<int> :: iterator it1 = find(vertex[u].begin(), vertex[u].end(), v);

//vector<int> :: iterator it2 = find(vertex[v].begin(), vertex[v].end(), u);

//\*it1 = u, \*it2 = v;

cnt[u][v]--, cnt[v][u]--;

dijkstra(j, ++when, 2);

cnt[u][v]++, cnt[v][u]++;

//\*it1 = v, \*it2 = u; //本来是用erase的，超时了。 现在换这种方法也超时了，估计find耗时太久。

ans = ans - tmp + sum[j]; //用新的sum[j]来代替旧的tmp

sum[j] = tmp;

int k ;

for(k = 1; k <= n; k++) if(visit[k] != when) break; //如果删边了以后j不能到达k(即k没有进过队)

if(k <= n) {

ans = INF;

break;

}

}

ans == INF ? printf("INF\n") : printf("%d\n", ans);

ans = tmp; //不要把这个tmp和for\_j里的tmp混了..

}

for(int i = 0; i < m; i++) cnt[edge[i].u][edge[i].v] = cnt[edge[i].v][edge[i].u] = 0; //初始化...因为感觉memset(cnt)会不会花更多时间

}

return 0;

}

void dijkstra(int s, int when, int flag)

{

int floors = 1;

int cur = 0;

deque<int> Q[2];

Q[cur].push\_back(s);

visit[s] = when;

do {

while(!Q[cur].empty()) {

int u = Q[cur].front();

Q[cur].pop\_front();

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

int v = vertex[u][i];

if(visit[v] != when && cnt[u][v] > 0) {

if(flag == 1) used[s][u][v] = used[s][v][u] = true; //第一次最短路才标记used...因为懒得写两遍，所以要flag来标记是删边前还收删边后做的最短路，1则是预处理时的最短路，此时要标记used；2则是删边后的最短路，这个时候不能标记used.

visit[v] = when;

sum[s] += floors;

Q[1-cur].push\_back(v);

}

}

}

floors++;

cur = 1 - cur;

} while(!Q[cur].empty());

}

/\*

多数题解是错的，这几个题解比较靠谱。

http://blog.csdn.net/nash142857/article/details/8253913

http://www.cnblogs.com/crisxyj/archive/2013/03/10/2952396.html

http://hi.baidu.com/novosbirsk/item/bfcf0cd201edfc2d39f6f709

两份代码的思想是完全一样的，只是“baidu blog”那份用w[i][e]来判断i的最短路树是否包括边e，而cnblog的那份是用used[x][u][v]来判断边u-->v是否xxx.

\*/

/\*

记忆化搜索 DP 博弈 好题 启发题

题意：一个有向无环点权图，只有一个入度为0的点st，且点st能到达所有点。两个人移动同一个棋子从st点出发，轮流选择下一步的路线，第一次由A选，第二次由B选，第三次A选...需要注意的是，只有一个棋子，只是棋子每一步的路线选择轮流由A，B决定而已。

该棋子最终会停留在一个点，算作结束旅程。该旅程路线上的点权和作为旅程的价值。

现在，A想要旅程价值尽量大，B想要旅程价值尽量小，问有没有可能使得该旅程价值 >= 常量F ? "Victory" : "Glory" ?

思路：dp[u][0] 表示当前路线权在A手上，到达点u的最优值

一开始我是按拓扑序dp一遍，不断更新各点的dp[][]值，狂WA无下限

后来发现按拓扑序dp确实不能保证“路线轮流选择”这个条件，即使我有分情况进行状态转移，但这个还是不能保证最后ans = max(ans, dp[t][0], dp[t][1])的正确性。 顺便说一下，我的错误解法是以st为起点，每到一个出度为0的点我都更新一下最优旅程价值ans

意识到这一点对我启发很大，想了很久也不能补救“路线轮流选择”这一问题----看起来好像拓扑序都完美解决了，其实只是假象=\_=

学习了记忆化搜索的写法。

因为是搜索，所以我们可以加个参数cur 说明当前选择权在A手上还是在B手上。这样就简单巧妙地解决的我的问题。

重要的一点是，记忆化搜索实际上是“从后往前”递归回来的，也就是说，由后继节点的dp值推理出当前点的dp值，但这里的“从后往前”又是我们从前往后递归下去的...又因为我们一直有选择权参数cur保证，所以转移的时候可以直接按情况转移，遇到分叉口也没事。

为什么我的拓扑序解法不能补救呢？因为我是以入度为0的st为起始点，是“从前往后”更新各个后继节点，这种解法遇到分叉口就跪了有木有！！ 那我把边逆向再拓扑不就行了？不行！再看看正解记忆化搜索，它是“从前往后请求需要的状态，然后到了叶子节点就递归回来，就可以知道我希望知道的状态了，类似于client-server模型，且client有把控制参数(cur)一起发送到server”，但拓扑的话就是纯递推模型。

启发：拓扑dp与记忆化搜索的区别

\*/

int dfs(int u, int cur)

{

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

if(head[u] == -1) return dp[u][cur] = val[u];

int \_max = -INF, \_min = INF;

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

int v = edge[k].v;

if(cur) \_min = min(\_min, dfs(v, !cur) + val[u]);

else \_max = max(\_max, dfs(v, !cur) + val[u]);

}

if(cur) return dp[u][cur] = \_min;

else return dp[u][cur] = \_max;

}

int main()

{

int F, u, v;

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

init();

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

while(m--) {

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

add\_edge(u, v);

in[v] = true;

}

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

if(!in[i]) puts(dfs(i, 0) >= F ? "Victory" : "Glory");

}

return 0;

}

/\*

图论 简单图判定 构造判定 思路题 好题

题意：给出一个无向图的度的序列{Degree}，问该图有没有可能是简单图（不存在平行边和自环的图）

思路：高阳推出来的。

按度大到小排序。对于当前度最大的点度x，删掉该点，后面连续x个点度减1。每次执行完后还是按度大小排序一遍，保证接下来要删掉的点是度最大的点。我用优先队列搞。

\*/

/\*

博弈 Anti-SG Anti-Nim

题意：有n堆苹果，每堆有mi个。两人轮流取，每次可以从一堆苹果中取任意连续个苹果，最后取光者为输。Fra先下，问是否可以获胜。

思路：用下下面的代码打了一个sg表，表示我和我的小伙伴们都惊呆了，sg(i) = i...

参看《组合游戏略述——浅谈SG游戏的若干拓展及变形---贾志豪》。

SJ定理：Anti-SG游戏的必胜态：(1)游戏的sg函数不为0且游戏种的某个单一游戏的sg大于1 (2)游戏的sg函数为0且没有单一游戏的sg大于1

\*/

//hdu 2509

#include <stdio.h>

int n, x;

int main()

{

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

int ans = 0;

bool flag = false;

while(n--) {

scanf("%d", &x);

ans ^= x; //因为这道题的sg(i) == i

if(x > 1) flag = true;

}

puts( (ans && flag) || (!ans && !flag) ? "Yes" : "No");

}

return 0;

}

/\*

int SG(int u)

{

if(u == 0) return 0;

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

bool vis[MAXN];

memset(vis, false, sizeof(vis));

for(int i = 1; i <= u; i++) { //取的个数

for(int j = 0; j <= u-i; j++) //左边剩下的个数

vis[ SG(j) ^ SG(u-i-j) ] = true;

}

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

}

\*/

/\*

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`

下面两个解释, 推荐一下. 罗嗦一句, 看代码更容易理解, 人脑模拟一遍更容易理解

http://www.nocow.cn/index.php/Tarjan%E7%AE%97%E6%B3%95

http://blog.chinaunix.net/uid-1721137-id-181005.html

\*/

#include <vector>

#include <stdio.h>

using namespace std;

#define MAXN 40001

#define debug printf("!\n")

vector<int> v[MAXN], w[MAXN], query[MAXN], ans\_num[MAXN];

int father[MAXN], dis[MAXN], ans[201];

bool visit[MAXN];

int n;

void init()

{

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

v[i].clear();

w[i].clear();

ans\_num[i].clear();

query[i].clear();

father[i] = i;

dis[i] = 0;

visit[i] = false;

}

}

int find(int x)

{

return x == father[x] ? x : father[x] = find(father[x]);

}

void Union(int x, int y) { father[find(y)] = find(x); }

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)];

}

}

int main()

{

int cases, Query, x, y, z;

scanf("%d", &cases);

while(cases--) {

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

init();

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

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

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

w[x].push\_back(z);

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

w[y].push\_back(z);

}

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

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

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

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

ans\_num[x].push\_back(i);

ans\_num[y].push\_back(i);

}

Tarjan(1, 0);

for(int i = 0; i < Query; i++) printf("%d\n", ans[i]);

}

return 0;

}

/\*

Tarjan算法是利用并查集来实现的。它按DFS的顺序遍历整棵树。对于每个结点x，它进行以下几步操作：

\* 计算当前结点的层号lv[x]，并在并查集中建立仅包含x结点的集合，即root[x]:=x。

\* 依次处理与该结点关联的询问。

\* 递归处理x的所有孩子。

\* root[x]:=root[father[x]]（对于根结点来说，它的父结点可以任选一个，反正这是最后一步操作了）。

　　现在我们来观察正在处理与x结点关联的询问时并查集的情况。由于一个结点处理完毕后，它就被归到其父结点所在的集合，所以在已经处理过的结点中（包括 x本身），x结点本身构成了与x的LCA是x的集合，x结点的父结点及以x的所有已处理的兄弟结点为根的子树构成了与x的LCA是father[x]的集合，x结点的父结点的父结点及以x的父结点的所有已处理的兄弟结点为根的子树构成了与x的LCA是father[father[x]]的集合……（上面这几句话如果看着别扭，就分析一下句子成分，也可参照右面的图）假设有一个询问(x,y)（y是已处理的结点），在并查集中查到y所属集合的根是z，那么z 就是x和y的LCA，x到y的路径长度就是lv[x]+lv[y]-lv[z]\*2。累加所有经过的路径长度就得到答案。 　　现在还有一个问题：上面提到的询问(x,y)中，y是已处理过的结点。那么，如果y尚未处理怎么办？其实很简单，只要在询问列表中加入两个询问(x, y)、(y,x)，那么就可以保证这两个询问有且仅有一个被处理了（暂时无法处理的那个就pass掉）。而形如(x,x)的询问则根本不必存储。 　　如果在并查集的实现中使用路径压缩等优化措施，一次查询的复杂度将可以认为是常数级的，整个算法也就是线性的了。

\*

/\*

LCA 离线

\*/

void build(int u, int fa)

{

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

int v = edge[k].v;

if(v == fa) continue;

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

build(v, u);

}

}

void LCA(int u, int fa)

{

vis[u] = true;

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

int v = Quetion[u][i].first;

if(!vis[v]) continue;

ans[ Quetion[u][i].second ] = dis[v] + dis[u] - 2 \* dis[ Find(v) ];

}

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

int v = edge[k].v;

if(fa == v) continue;

LCA(v, u);

}

Union(u, fa);

}

int main()

{

int cases, u, v, query, w;

scanf("%d", &cases);

while(cases--) {

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

init();

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

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

add\_edge(u, v, w), add\_edge(v, u, w);

}

dis[1] = 0;

build(1, 0);

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

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

Quetion[u].push\_back( PII(v, i) );

Quetion[v].push\_back( PII(u, i) );

}

LCA(1, 0);

for(int i = 0; i < query; i++) printf("%d\n", ans[i]);

}

return 0;

}

/\*

0/1背包扩展 好题 用“是否合法”来保证“至少取m个”

题意：K种品牌的鞋子，每种品牌都至少要买一双。给出每双鞋子的品牌，价钱，价值，求m元能买到多大价值。

思路：人家只是用了“组”这个概念而已，哪是什么分组背包呀...还是称0/1背包比较靠谱。

难点在于“每种品牌至少一双”，也就是每组至少取一个。

保证每个分组至少取一件，f[K][m]表示前K种品牌话费了m元钱能买多大价值，先初始化f[][]为-1，f[0][]为0。每次更新f[k][j]时判断一下它的前继状态是否为-1，如果不为-1，说明前继合法，可以递推到f[k][j].

f[k][j]的前继有两个（假设现在循环到物品i）：

f[k-1][j-p[i]] != -1 : 在前k-1种品牌上花j-p[i]元 已经满足题目的“前k-1种品牌每种品牌至少一双” ，是合法状态

f[k][j-p[i]] != -1 : 在前k种品牌花j-p[i]元，已经满足题目的条件，合法状态

如果再来扩展“每种品牌至少两双”呢？加一维就好了呗。

用“是否合法”来保证“至少取m个”真是个好方法。

另外这个题的数据有点巨神的地方，就是price能为0！！！为0的话状态转移的时候就要注意顺序。对于f[k][j]，要先从f[k][j-p[i]]转移，然后再从f[k-1][j-p[i]]转移，因为这样才能保证price=0时同一件物品没有被多次取过。

\*/

int f[15][10005];

int main()

{

int n, m, K;

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

for(int i = 1; i <= n; i++) scanf("%d%d%d", &a[i].brand, &a[i].price, &a[i].value);

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

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

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

for(int k = 1; k <= n; k++) if(a[k].brand == i)

for(int j = m; j >= a[k].price; j--) {

if(f[i][j-a[k].price] != -1) //决定取当前第 i 个牌子时要取多少件以及怎样得到最优值

f[i][j] = max(f[i][j], f[i][j-a[k].price] + a[k].value);

if(f[i-1][j-a[k].price] != -1) //用于承接上一个品牌时的状态

f[i][j] = max(f[i][j], f[i-1][j-a[k].price] + a[k].value);

}

if(f[K][m] == -1) puts("Impossible");

else printf("%d\n", f[K][m]);

}

return 0;

}

/\*

ST sparsee table应用 思路题 算好题吧

题意：给你一个1000位的数，要你删掉m个位，求最小结果

思路：在n个位里面删除m个位，也就是顺序找出n-m个位组成最小数

所以，在[0,m]中找最小元素的位置i，作为ans的第一位

在[i+1, m+1]中找最小元素的位置i'，作为ans的第二位

在[i'+1, m+2]...

找出n-m个数

思路我是看题解的，能理解它的正确性，但不知道怎么想出来的orz..

\*/

/\*

最小表示法 KMP应用 KMP的周期问题 KMP求字符串的循环节 好题

题意：求串的最小/最大表示，并输出最大/最小表示串的同构串个数

思路：最小表示简单，学到的是KMP的next原来还可以用来求循环节...

int len = s.length();

int rep = len - next[len];

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

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

\*/

#include <algorithm>

#include <iostream>

#include <string>

using namespace std;

#define MAXN 1000005

int next[MAXN];

int max\_Presentation(string &s)

{

int i = 0, j = 1, k = 0, len = s.length();

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

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

if(!res) k++;

else {

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

else j = j + k + 1;

if(i == j) j++;

k = 0;

}

}

return min(i,j);

}

int min\_Presentation(string &s)

{

int i = 0, j = 1, k = 0, len = s.length();

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

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

if(!res) k++;

else {

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

else j = j + k + 1;

if(i == j) j++;

k = 0;

}

}

return min(i,j);

}

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];

}

}

int main()

{

string s;

while(cin >> s) {

fail(s);

int len = s.length();

int rep = len - next[len];

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

cout << min\_Presentation(s) + 1 << " " << times << " " << max\_Presentation(s) + 1 << " " << times << endl;

}

return 0;

}

/\*

混合背包 好题

题意：3种组别，每种组别有不同要求：组别为0的要求一组内至少取一个；组别为1的要求一组内之多取一个；组别为2的随便

思路：一开始自己按组0分类DP出一个f[i][j]表示组别为0的前i组，花j元钱，能达到的最大价值；其它两个组别DP出一个g[i][j]，含义跟f相似。然后取f[I][j] + g[I'][T-j]的最大值，可能是细节地方写撮了吧，老是WA，但分析这样做是没错的。太惨了=\_= 太弱了>\_<

这里也是按组别不同分别DP

至少取一个：f[i][j]状态的选择可以来自上一组的状态，也可以来自本组的状态

至多取一个：只能来自上一组的状态

随便取 ：0/1背包

注意到，因为后两种组别没有“必须取一个”这种要求，所以在者两种组别DP时可以完全继承上一组的结果先；而“至少取一个”就不能直接继承上一组的状态，而是要判断目前状态的前继状态是否合法，才继续更新状态。

\*/

//hdu 3535

int main()

{

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

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

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

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

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

for(int j = 0; j < num[i]; j++) scanf("%d%d", &job[i][j].cost, &job[i][j].happy);

}

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

if(type[i] == AT\_LEAST\_ONE) {

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

Job & tmp = job[i][k];

for(int j = T; j >= tmp.cost; j--) {

if(f[i][j-tmp.cost] != -1) f[i][j] = max(f[i][j], f[i][j-tmp.cost] + tmp.happy);

if(f[i-1][j-tmp.cost] != -1) f[i][j] = max(f[i][j], f[i-1][j-tmp.cost] + tmp.happy);

}

}

} else if(type[i] == AT\_MOST\_ONE) {

memcpy(f[i], f[i-1], sizeof(f[i]));

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

Job & tmp = job[i][k];

for(int j = T; j >= tmp.cost; j--)

if(f[i-1][j-tmp.cost] != -1) //TODO

f[i][j] = max(f[i][j], f[i-1][j-tmp.cost] + tmp.happy);

}

} else {

memcpy(f[i], f[i-1], sizeof(f[i]));

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

Job & tmp = job[i][k];

for(int j = T; j >= tmp.cost; j--)

if(f[i][j-tmp.cost] != -1)

f[i][j] = max(f[i][j], f[i][j-tmp.cost] + tmp.happy);

}

}

}

printf("%d\n", f[n][T]);

}

return 0;

}

/\*

二分图多重匹配 启发题

题意：题意：末日逃亡，n个人逃往m个星球，一个人只能在某些星球上生存。一个星球最多居住人口有上限。求最后是否可以让所有人都逃离地球

思路：一般的二分图匹配是一对一的，link[]贮存Y部的匹配点。

这道题是多对一的，也就是星球能与多个人同时匹配，方法就是扩展link的含义，给它加一维，贮存与星球匹配的所有人。

link[]含义扩展了，接下来就扩展匈牙利算法，原始的二分图最大匹配匈牙利算法在找未盖点i的增广路时，对于其邻接点j，若j也是未盖点，则i,j匹配，否则尝试删除<link[j], j> 边，向上找增广路，如果找到了，则i,j匹配，link[j] = i.

现在扩展成若星球j已匹配的人数 < capacity[j]，则人类i加到其匹配列表里，即link[j].push\_back(i)，否则尝试删除星球j的已匹配边<j, k>，从k向上寻找增广路，若找到，则用i代替k的位置link[j][k] = i;

\*/

void init()

{

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

}

bool find(int u)

{

for(int v = 0; v < m; v++) if(path[u][v] && visit[v] == false){

visit[v] = true;

if(link[v][0] < cap[v]) {

link[v][++link[v][0]] = v;

return true;

}

for(int i = 1; i <= link[v][0]; i++)

if(find(link[v][i])) {

link[v][i] = u;

return true;

}

}

return false;

}

bool hungary()

{

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

memset(visit, false, sizeof(visit));

if(!find(i)) return false;

}

return true;

}

int main()

{

int x;

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

init();

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

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

scanf("%d", &x);

path[i][j] = x;

}

for(int i = 0; i < m; i++) scanf("%d", &cap[i]);

printf(hungary() ? "YES\n" : "NO\n");

}

return 0;

}

/\*

连通性 反向建图 好题 启发题

题意：题意：给出一个有向图，表示点与点之间的支持关系，支持关系可以传递，A支持B，B支持C，那么A支持C，C获得两个支持。找出哪些人获得最多支持，输出获得支持票数，然后升序点的标号

思路：缩点，统计点u能被多少个点到达。

难点就在于统计这一块儿。得到缩点后的DAG后，问题就变成，一个点能被多少个点到达，要知道一个点能被多少个点到达，一般就是方向建图。

\*\*\*反向建图统计前驱点数\*\*\* : 建立反向图后，对于点u，从u除法遍历整个图，统计能够到达的点。易知点u为当前入度为0的点。

反思：一开始我是边拓扑排序边统计，但是对于除法点x，x能通过i到达y，也能够通过j到达y，这样的话我的拓扑就会统计两次y。而且在我的源代码里没法补救。

\*/

/\*

二分匹配 最小路径覆盖

想了dfs搞，没搞出来

题意：有向图如果2个点可以互达，那么它们属于一个state，然后将图分割成尽量少的部分，在一个部分内，任意2点u，v，都要满足u可达v或者v可达u。

思路：

1.求一次强连通分量，然后重新建图变成一个DAG

2.DAG中一个大点内的小点一定是满足[ u可达v或者v可达u ]，另外如果是在DAG中找一条不分叉的路径，路径上的点也全部满足[ u可达v或者v可达u ]，但是分叉了就不行了。这其实就是最小路径覆盖

最小路径覆盖：用尽量少的不想交的简单路径覆盖有向无环图的所有顶点。

最小路径覆盖 = 顶点数 – 匹配数

3.在DAG上做一次最大匹配求得最大匹配，从顶点数减去即可

变化：题目可以理解为，对于一个有向图，最少需要多少条链可以覆盖整个图？而且这些链不能相交。那就是最小路径覆盖。

那要改为链可以相交呢？我觉得就是max(入度为0的点数，出度为0的点数)，证明略.

\*/

/\*

连通性 树DP 树的构造 好题

题意：给定一个连通的无向图，你只能毁掉一条边，而敌人可以补回一条边（但不会补你毁的那条，是任意地补另外一条）。为了保证你一定能使这个图断裂，你至少需要多少钱。即求出一个最小权值使得任意加一条边后都能去掉一条小于这个权值的边，使得图不连通。

思路：对于缩点后的树选取出边权最小的边，分别以该边的两个端点作为根确立两个不相交的树。然后在这两个树种分别DP。

首先要知道，在我们还没毁掉一条边的时候，敌人补任意一条边就会存在一个环，而上述的最小边肯定会包含在这个环里面。为什么？因为如果最小边不包含在这个环里，那最小边还是一个桥，那我们就毁掉这条最小边好了。

也就是说，敌人加的一条边肯定两端分别在我们新定义的两颗树里。这就是为什么要最小边的两个端点分别建树的原因了。

理解到这里就简单啦，对于新树A，我们在里面找一条以根为一端的链，使得链尽量包含该树中权值最小的边。两颗树两条链，就是敌人要补边连起来的链。但是直接找尽量包含树中权值最小的边比较麻烦，细心发现，既然是一条链，说明树中任何节点的两个不同分支不可能同时存在在链里面。这样，我们又把问题转化为“求所有节点的次小边权的最小值，并且最小边权和次小边权不在同一分支”....有点乱，就是“次小分支”啦。

http://www.cnblogs.com/wuyiqi/archive/2011/11/04/2235671.html

http://www.gonglin91.com/category/acm/dp/treedp/

收获：一开始用树DP找节点的次小边权的最小值，用f[u].fir/f[u].sec来表示最小/次小分支边权，但是这样是求树的最小/次小值，有可能最小/次小在同一分支，所以跪了...

之后成了现在这样，f[u]表示以u为根的子树的最小边权。但是在维护f[u]的时候要同时更新ans，这是考细节的=\_=||，无数WA之后自己出了个数据才找到WA点。

简单记一下dfs的修改过程：

dfs( Node u ) {

for every child edge(u,v,w) do {

dfs(v)

Min = min(w, f[v]) ;当前边E(u,v,w)应该算跟v同一个分支。 我一开始是w更新一次，f[v]更新一次，果断跪

if f[u] >= Min then {

ans = min(ans, f[u]) ;旧最小分支充当次小分支，然后更新u的dp值

f[u] = Min

} else {

ans = min(ans, Min) ;当前是新分支，就算f[u]<Min，也有可能该分支Min就是我们要找的次小分支

}

}

\*/

/\*

树的同构 树的最小表示法 枚举子树

题意：给一棵无根树，问有多少个异构的子树。

思路：判断同构异构我觉得不难，难的是怎么搞出所有子树。

网上的都是暴力枚举所有子树...=\_= 就是枚举所有点集，然后判断这个点集是不是一棵子树（即是不是个连通块，因为森林不算子树）。

对于一棵子树，枚举每个节点作为根，出来的子树最小表示是不一样的，但一棵子树又只能算1。

\*/

bool is\_subtree(int u, int pre) //判断是否连通

{

if(--Node\_Rest == 0) return true;

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

int v = edge[k].v;

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

if(is\_subtree(v, u)) return true;

}

return false;

}

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;

}

int main()

{

int cases, Cas = 0, u, v;

scanf("%d", &cases);

while(cases--) {

scanf("%d", &n);

init();

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

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

add\_edge(u, v), add\_edge(v, u);

}

int ans = 0;

for(int N = (1 << n), status = 1; status < N; status++) {

int idx ;

Node\_Rest = 0;

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

if((1<<(i-1)) & status) vis[i] = true, Node\_Rest++, idx = i;

else vis[i] = false;

if(!is\_subtree(idx, -1)) continue;

bool flag = false;

for(int i = 1; i <= n; i++) if(vis[i]) {

string res = min\_Presentation(i, -1) ;

if(!M[res]) {

M[res] = true; //虽然不同点做根得到的最小表示序列不一样

flag = true; //但是只能算一棵子树。还要记得把每个节点做根节点的时候该点集的最小表示哈希存下来

}

}

ans += flag;

}

printf("Case #%d: %d\n", ++Cas, ans);

}

return 0;

}

/\*

博弈 SG

题意：Nim单游戏，取数只能从有限集合数集S里面取。

思路：dfs，这道题又是什么SG性质的，就是败态后面都是胜态，胜态后面都是败态的。

\*/

/\*

状压DP TSP 旅行商问题 最短路 实现 经典题 好题

题意：无向边权图，从城市1带着M元钱开始出发，遍历给定的H个城市，而且到达这H个城市必须要有足够的钱di买护照，再在这个城市打工赚得ci元。

也就是说这H个城市是给出的，是一定要在这里买护照并且打工一次的，无强制顺序（选定的H个城市须且仅能买一次护照打一次工），其它城市无所谓要不要经过。每个城市可以到达多次。如果城市1在被选定的城市里面，则不一定要先在城市1打工。

问是否有线路可以实现从城市1出发再回到城市1

思路：比赛的时候被题意给坑了...

理解清楚题意就是裸TSP问题了.

看cxlove的题解，floyed搞出最短路

dp[i][x] 表状态S，最后到达的城市为i 的最优值

状态转移不说，就是觉得cxlove的初始化很巧妙。如果城市1不在给出的H个城市{X}里面，他就把城市1添加进{X}里面，c/d值都为0，这样就省去了很多考虑这儿考虑那儿的麻烦

有些细节，我觉得我是想不到的。cxlove实现得很漂亮

!!! 用memset() 初始化会在计算的时候溢出...跪...

TSP问题：http://blog.csdn.net/gfaiswl/article/details/4749713

\*/

void init()

{

//memset(path, 127, sizeof(path)); //!!! 这点很重要啊！这样memset初始化在计算的时候有可能会溢出，然后就跪了！！！

for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) path[i][j] = inf;

//for(int i = 0; i <= n; i++) path[i][i] = 0; //need ? 不需要

}

int main()

{

int cases, u, v, w;

scanf("%d", &cases);

while(cases--) {

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

init();

while(m--) {

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

u--, v--;

if(w < path[u][v]) path[u][v] = path[v][u] = w;

}

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

for(int i = 0; i < n; i++) if(k != i)

for(int j = 0; j < n; j++) if(i != j && k != j)

if(path[i][k] + path[k][j] < path[i][j]) path[i][j] = path[i][k] + path[k][j];

scanf("%d", &H);

int s = -1;

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

scanf("%d %d %d", &choose[i], &c[i], &d[i]); choose[i] --;

if(choose[i] == 0) s = i;

}

if(s == -1) { //这一步好妙 //细节

choose[H] = 0, c[H] = d[H] = 0; //起点

s = H++;

}

memset(dp, -1, sizeof dp);

if(money >= d[s]) dp[s][1 << s] = money - d[s] + c[s]; // 这步很重要也很容易忘记 //细节

dp[s][0] = money; //这一步需要吗？？？？？

//难道说如果点1被选定，可以先不在点1买护照打工，而是先跑到别的城市吗...? yes

for(int S = 0; S < (1 << H); S++)

for(int i = 0; i < H; i++) if(dp[i][S] != -1) //因为第一个工作的城市不一定是1...blabla，所以不能if(S & (1<<i)) // 细节

for(int j = 0; j < H; j++) if(!((1 << j) & S))

if(dp[i][S] >= path[choose[i]][choose[j]] + d[j]) {

int T = (S | (1 << j));

dp[j][T] = max(dp[j][T], dp[i][S] - path[choose[i]][choose[j]] - d[j] + c[j]);

}

bool flag = false;

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

if(dp[i][(1 << H)-1] >= path[choose[i]][0]) flag = true;

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

}

return 0;

}

/\*

合并堆 追逐法 好题

题意：给5个规模为n(n <= 200)的堆，判断能否从这5个堆里各取一个数字，使这5个数和为0

思路：首先问题如果变成(1)“两个有序堆，判断能否各取一个数，使两个数和为0”，我们可以用追逐法来

求：一个指针指向堆1的头，一个指针指向堆2的尾，然后向里收敛。

如果是无序堆呢？当然是可以nlogn排一遍序。不过Neko给了一个hash法比较更好：扫一遍n

把堆1里的数据hash起来，再扫一遍堆2，在hash\_table里找是否存在相应的数即可

那如果是(2)“3个有序堆，判断能否各取一个数，使两个数和为0”呢？我们可以枚举第一个堆的

数，然后问题就变成了问题(1)了

回到原问题，5个堆。我们可以把5个堆变成我们已知的问题(1)(2)：前两个堆的和作为一个堆，

后两个堆的和作为一个堆，这样我们就回到问题(2)了。其实这个思想就是“把多个堆合并成一个

的思想”

扩展开来，有n个堆呢？我们可以把n个堆合并成3堆，前两堆数目为n/3向上取整，最后要枚举的

那一堆n/3向下取整

说完堆合并，说说追逐法。

我感觉追逐法就是不断收敛的过程。比如我把问题改成“从两堆里各取两个数，和在区间[l,r]”

是可以用追逐法来完成的，但是如果是乘法，即积为一个定值或一个区间，是不能用追逐法的

，因为乘法不具有收敛性质。 哦对了，这里所说的追逐法是“一个从头开始，一个从尾巴开始”

的追逐，不是“一个在前跑，一个在后追”。

\*/

void calc(ll \*a, ll \*b, ll \*res, int &m) //合并堆

{

int idx = 0;

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

for(int j = 0; j < n; j++) res[idx++] = a[i] + b[j];

sort(res, res+idx);

m = 0;

for(int i = 1; i < idx; i++) if(res[i] != res[i-1]) res[m++] = res[i];

}

int main()

{

int cases;

scanf("%d", &cases);

while(cases--) {

scanf("%d", &n);

for(int i = 0; i < 5; i++)

for(int j = 0; j < n; j++) scanf("%I64d", &a[i][j]);

calc(a[0], a[1], p0, pe0);

calc(a[2], a[3], p1, pe1);

bool flag = false;

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

ll &x = a[4][i];

int s1 = 0, s2 = pe1-1;

while(s1 < pe0 && s2 >= 0) { //追逐法

if(p0[s1] + p1[s2] + x == 0) { flag = true; break; }

if(p0[s1] + p1[s2] + x > 0) s2--;

else s1++;

}

}

puts(flag ? "Yes" : "No");

}

return 0;

**}**

/\*\*

数论 Lucas定理扩展应用 二进制

题意： C(n,0),C(n,1),C(n,2)...C(n,n)这些数里面有多少个数是奇数

思路：如果C(n,m)是奇数，则C(n,m) mod 2 == 1

这里就是用到了Lucas定理了：Lucas(n,m,p)定义为C(n,m) mod P

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

分析一下会发现，若Lucas(n,m,p) == 0，则说明m,n存在某一二进制位mi = 1, ni = 0(mi表

示m的二进制第i位)；反之若Lucas(n,m,p) == 1，则需要n & m == m

那现在我们就是要找n & m == m的m, 0 <= m <= n

易知m的个数就是2^x，x是n的二进制表示上的 1 的个数

学到了\_\_builtin\_popcount() 统计二进制表示上1的个数

1 << x

\*\*/

/\*

2-SAT

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 pre\_check()

{

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

if(b[i][i] != 0) return false;

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

if(b[i][j] != b[j][i]) return false;

}

return true;

}

bool solve()

{

init();

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

for(int col = 0; col < row; col++) {

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); //x = ai(0,1) y = aj(0,1)

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(int row = 0; row < n; row++)

for(int col = 0; col < row; col++)

b[row][col] >>= 1;

for(int N = hash(n), i = 0; i < N; i++)

if(dfn[i] == -1) Tarjan(i);

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

if(low[i] == low[hash(i)]) return false;

return true;

}

/\*

并查集 好题

题意 : 给出一棵树，找出一个点root，使得所有点到这个点的权值和最大，所谓的权值是指为点到root的路径上所有边权里的最小值。

思路 : 好吧..竟然是并查集...知道是并查集后应该不难, 难的是如何想到并查

边按权值大到小排序, 做并查集.

因为给出的是树, 所以其实每条边e都是是用来连接两个连通块的, 在这里`连通块`即并查集的集合. 我们需要考虑的是要把root放在e的哪一边? 可以肯定, 放在使得新的边权和最大的那一边最优, 详细看代码. 每次连接两个集合, 就要更新权值总和, 这需要知道两个集合各自的权值和, 集合内的点数目. 我觉得这一点是我之前做的并查集没有过的, \*\*亮点\*\*

**\*/**

/\*

DP 好题. 虽然大家都说是水DP, 可是对我来说还是有难度呀...

状态转移的思路还是挺厉害的, 时间卡得紧, 细节也有, 坑不多.

题意 : 给出N, M, K, 问有多少种方法构造出长度为K的序列, 使得sum{A} = N, LCM{A} = M

思路 : 有几点我之前没意识到. 1, 序列中的Ai肯定是M的因子, 不会出现Ak = ai + Aj而M % Ak != 0 的情况, 没想通这一点使得我有点被蒙蔽了双眼, 想复杂了.

2, 如果想明白了(1), 则该DP其实就是对于每一个位置position进行`0/1背包`的感觉, 而不是`完全背包`

以位置为阶段, 把v[i]放在该位置为状态(v[i]表M的第i个因子),

f[pos][sum][lcm] 表前pos个位置, 总和为sum, 最小公倍数为lcm的最优值, 则

f[pos+1] [sum+v[i]] [LCM(lcm, v[i])] = sum{ f[pos][sum][lcm] },

详细解释看代码

优化 :

预处理LCM是必须的

枚举状态时是以`M的第几个因子`来枚举, 而不是从0-N, 这也是必须的

\*/

void init()

{

v.clear();

for(int i = 1; i <= M; i++) if(M % i == 0) v.push\_back(i);

}

int main()

{

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

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

LCM[i][j] = lcm(i, j);

//LCM[j][i] = LCM[i][j] = lcm(j, i);

while(scanf("%d%d%d", &N, &M, &K) != EOF) {

init();

solve();

}

return 0;

}

void solve()

{

int top = v.size();

int cur = 0;

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

for(int j = 0; j < top; j++) f[cur][i][v[j]] = 0; //@优化2 换掉memset

f[cur][0][1] = 1;

for(int position = 1; position <= K; position++) {

cur ^= 1;

for(int sum = 0; sum <= N; sum++) //用memset会超时

for(int i = 0; i < top; i++) f[cur][sum][v[i]] = 0;

for(int sum = position-1; sum <= N; sum++) {//@优化1 : pos-1代替了0, 因为在pos-1个位置的总和肯定 >= pos-1

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

if(f[cur^1][sum][v[i]] == 0) continue;

for(int j = 0; j < top; j++) { //前面for i循环是对上一个位置的状态的枚举, 这里的for j是对当前位置要放什么数的枚举

int ss = sum + v[j];

int ll = LCM[v[i]][v[j]]; //不用判断ll的合法性, 因为ll必定合法

if(ss > N) break; //估计是因为这一点我比kuangbin的快. 因为v[j+1] > v[j], 所以后面的可以不用考虑了

f[cur][ss][ll] += f[cur^1][sum][v[i]];

f[cur][ss][ll] %= MOD;

}

}

}

}

printf("%d\n", f[cur][N][M]);

}

/\*

连通性 割点 好题 AC后花很多时间来思考细节

题意：对于一个无向图删除两个点，使得剩余图连通分量最多。

思路：无向图里的连通分量：只要两点(u,v)有路径到达即为同一的连通分量。

删除一个节点使得剩余连通分量最多：尽量找割点。对于割点u，如果u只存在于一个连通分量中，则删u和删该连通分量的其它点没什么区别。但是若割点u存在于多个连通分量中，就不一样了，原因很简单，略。如果没有割点的话会是怎样呢？答案是删哪个都一样。什么时候没有割点呢？当所有点要么为孤立点，要么属于一个双连通分量中。

再回到上面问题，可能在找割点的过程中，找不到一个割点，那么只能随便删除一个点，那么图的连通分量就是 ANS = max(ANS,CNT)吗？

不一定的！因为一个无向图中，如果没有割点，删除一个点，图的连通分量数不一定保持不变，而是可能减少

例如

1 2

3 2

4 2

5 2

这种数据，删除点2，后得到1 3 4 5这些连通分量，CNT = 4，这些点中没有一个点是割点，那么任意删除一个点即可

但删除了一个点，连通分量数不变吗？不是的，而是-1

所以为了保证正确性，更新答案应该是

ANS = max(ANS,CNT-1);

总结一下，当图中没有割点时：<1>当原图有存在双连通分量，则删除双连通分量里的点，使得删点后的连通分量数保持不变 <2>当原图所有点都是孤立点，则删点后的连通分量数减1

OK! 所以第一步枚举删的第一个点后，问题就转为上述的“删除一个节点使得剩余连通分量最多”的问题了。

\*/

XN], dfn[MAXN], low[MAXN];

void Tarjan(int u)

{

dfn[u] = low[u] = Index++;

in\_stack[u] = true;

int tot = 0;

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

vis[k] = vis[k^1] = true;

int v = edge[k].v;

if(v == del) continue;

if(!dfn[v]) {

Tarjan(v);

tot ++ ;

low[u] = min(low[u], low[v]);

if(u != root && low[v] >= dfn[u]) {

cnt[u] ++ ;

//if(del == 6) printf("# %d %d\n", u, v);

}

} else if(in\_stack[v]) low[u] = min(low[u], dfn[v]);

}

if(u == root) cnt[u] = tot - 1; //为了处理没有割点的第二种情况。

in\_stack[u] = false; //TODO 疑问

}

int main()

{

int m, u, v;

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

init();

while(m--) {

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

u++, v++;

add\_edge(u, v), add\_edge(v, u);

}

int ans = 0;

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

del = i;

init2();

int sum = 0;

for(int j = 1; j <= n; j++) if(!dfn[j] && del != j) {

sum ++ ;

root = j;

Tarjan(j);

}

for(int j = 1; j <= n; j++) if(j != i) ans = max(ans, sum + cnt[j]); //这里一定要判断!! 因为如果是第二种没有割点的情况的话要减1，即cnt[x] = -1，但cnt[del] = 0，这时候就会错

//我觉得定个sum出来把事情弄方便了一些。如果删掉的不是root，而是一个割点，则root所在的连通分量已经被加到sum里了；如果删的是root，则sum-1，即cnt[root] = 分支数 - 1

}

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

}

return 0;

}

/\*

差分约束 好题

题意：问一个图<G,E>是否满足下列两个条件(ai表示i点的权值，<i,j>表示边i-j)：

<1> 所有 |ai| < T

<2> (i,j) ∈ E <====> |ai - aj| >= T

思路：我一开始是想直接判是否有奇环就可以了，后面看题解才知道题意没搞对。题意就不能写清楚明白点吗！ 对于题目的第二个条件，注意符号"<====>" 是表示"充要"，我一开始就只理解为"推出===>"。什么意思呢？如果(i,j) ∈ E，那要符合条件|ai - aj| >= T；反之如果(i,j) ∉ E，则要|ai - aj| < T.

染色法判断一下奇环，有奇环则No.

染色过程中我们默认color[x] == 0的x点点权为正，coloe[x] == 1的ax点点权为负。

然后差分约束判断是否满足条件。

对于e(i,j)属于E，并且假设顶点ai为正，则ai - aj >= T ----> aj <= ai - T

对于e(i,j)不属于E，并且假设顶点ai为正，则ai - aj < T ----> ai < aj + T ----> ai <= aj + T - 1

以a0为参照点，若ai为正，则加边<i, a0, -T>,<a0, i, 0)；若ai为负，则加边<a0, ai, T-1>,<ai, a0, 0>

SPFA看看是否有负圈，有则No。

代码：if(color[i] == color[j]) continue; //去掉这句也对 因为ai,aj同号的话直接可以肯定(ai,aj)满足条件了

color[i] == color[j]即ai和aj同号，假设ai和aj为正，则ai - aj < T肯定满足，加上我们在染色之后，若i,j颜色相同，则说明i,j不相连，所以|ai - aj| >= T肯定不满足。说明color[i]==color[j]这种情况可以忽略，因为它一定合法。

\*/

/\*

堆 优先队列 set 数据结构 曼哈顿距离

距离：最远曼哈顿距离

思路：学会了处理最远曼哈顿和multiset的用法。武森的《浅谈信息学竞赛中的“0”和“1”》

set/multiset是用二叉树来维护的，priority\_queue是用heap来维护的。

两者的区别这里讲得挺好：http://stackoverflow.com/questions/10141841/difference-between-stdset-and-stdpriority-queue

实现：这步的实现代码还是挺好看的:

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

if(j & (1 << i)) sum += x[i];

else sum -= x[i];

\*/

void init()

{

memset(status, 128, sizeof(status)); //为了防止op = 1时删错之前case的东西，把status初始化一下

for(int i = 0; i < (1 << 6); i++) S[i].clear();

}

int main()

{

int query, k;

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

int K = (1 << k); //k维度，K状态数=2\*k

init();

for(int no\_ = 1; no\_ <= query; no\_++) {

if(rdi() == 0) {

for(int i = 0; i < k; i++) scanf("%d", &x[i]);

for(int j = 0; j < K; j++) { //第j种状态

int sum = 0;

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

if(j & (1 << i)) sum += x[i]; //TODO 这个实现不错哟~

else sum -= x[i];

status[no\_][j] = sum;

S[j].insert(sum);

}

} else {

int u = rdi();

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

multiset<int> :: iterator it;

it = S[j].find(status[u][j]);

if(it != S[j].end()) S[j].erase(it);

}

}

int ans = 0;

for(int j = 0; j < K; j++) if(S[j].size()) ans = max(ans, \*(--S[j].end()) - \*(S[j].begin()));

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

}

}

return 0;

}

/\*

树状DP 树的直径 好题 实现妙题

题意：给一棵树，边长为1，有边权ei.w。删掉一条边e把树分成两棵新树，代价为两棵新树的直径的较大值\*e.w，问最小代价。

思路：先求树的直径。

如果边e不是直径上的边，那么去掉该边的毁灭值 = e的权值 \* 直径长度

如果边e是直径上的边，那么去掉该边的毁灭值 = e的权值 \* max(两颗新树的直径)，从直径两端各DP一次就可以了。

这里只讨论e是直径上的边的情况。

边e被删掉以后，我们要算两颗新树的直径。易知原树直径上的两个端点e1,e2被分在不同树上。那我们以e1,e2为根分别做一次树上DP，记录每个节点如下状态：

f[u].dis 以u为根的子树的最长直径。

f[u].fir/f[u].sec u节点的最长/次长子树长度

注意，以u为根的子树直径不一定经过u，所以，f[u].dis = max{ f[v].dis, f[u].fir+f[u].sec }, v is child node of u

仔细想想就可以知道，假设现在是以e1开始做DP的，那么f[u].dis其实就表示“删边后u点和e1不在同一棵树时，以u点为根的子树的直径”，说直白一点，就是“删掉连接u与u的父节点的这条边后，不包含e1的那棵树的直径”

这样，我们经过分别从e1,e2开始的DP且每次DP后计算出每条边的最大毁灭值。

\*/

bool label(int u, int father) //标记直径边

{

if(u == end) return true;

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

int v = edge[k].v;

if(v == father) continue;

if(label(v, u)) {

used[k/2+1] = true;

return true;

}

}

return false;

}

void dfs(int u, int len,int father) //求直径

{

if(len > Max\_len) end = u, Max\_len = len;

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

int v = edge[k].v;

if(v == father) continue;

dfs(v, len+1, u);

}

}

void DP(int u, int father)

{

f[u].dis = f[u].fir = f[u].sec = 0;

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

int v = edge[k].v;

if(v == father) continue;

DP(v, u);

if(f[v].fir+1 > f[u].sec) {

f[u].sec = f[v].fir + 1;

if(f[u].sec > f[u].fir) swap(f[u].fir, f[u].sec);

}

f[u].dis = max(f[u].dis, f[v].dis);

}

f[u].dis = max(f[u].dis, f[u].fir + f[u].sec);

}

void calc(int u, int father)

//我觉得calc过程真是太妙了！！calc的顺序跟DP的顺序是一样的这样就保证了DP()和calc()遍历出来的序列是一模一样的，就保证了e(u,v)一定是u在前v在后。而以不同直径端点e1,e2开始DP()和calc()来遍历，得到的序列肯定是相反的！

//我觉得这是细节吧。要我写可能就直接for循环遍历了，根本没想到有这种细节。

{

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

int v = edge[k].v, id = k/2 + 1;

if(v == father) continue;

calc(v, u);

if(used[id])

ans[id] = max(ans[id], f[v].dis \* edge[k].w);//为什么是v而不是u呢？因为我们的f[]意义为“删掉边id后不包含起始点e'的新树的

//直径”，即u与e'同一棵树，v才是与e'不同树

else ans[id] = max(ans[id], Dia \* edge[k].w);

}

}

int main()

{

int cases, Cas = 0, u, v, w;

scanf("%d", &cases);

while(cases--) {

scanf("%d", &n);

init();

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

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

add\_edge(u, v, w), add\_edge(v, u, w);

}

Max\_len = -1;

dfs(1, 0, 1);

int e1 = end;

Max\_len = -1;

dfs(end, 0, end);

int e2 = end;

Dia = Max\_len;

label(e1, e1);

DP(e1, e1);

calc(e1, e1);

DP(e2, e2);

calc(e2, e2);

int ans\_ = 999999999, ansi\_;

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

if(ans\_ > ans[i]) ans\_ = ans[i], ansi\_ = i;

printf("Case #%d: %d\n", ++Cas,ansi\_);

}

return 0;

}

/\*

区间DP 最长回文子序列

http://fszxwfy.blog.163.com/blog/static/44019308201381585813745/

题意：给定一个串，求从任意两个起点分别朝着两个方向走，形成的两个串的最长公共子序列的最大值是多少。且走到串尾可再回到串头，但不能超过其起始位置。

解法：就是求最长回文子串，枚举起点，针对每个起点，两边的最长回文串的长度之和就是当前情况下的值。

而且显然当两个兔子的起点相邻时肯定可以得到一种情况是最优解，因为假设在取得最优解时两个起点的其距离为len,当len内无最长回文子串时，我们将两个起点移到其中间的两个相邻的位置时显然值不变。

当len内涉及最长回文子串时，得到的结果可能会改变顺序，但至少会有一种情况得到得长度与之前相同，显然当移到两个起点时移过涉及最长回文子串的点数相同时，得到的最值结果只是将这些点改变下顺序。

当移过的点数不同时，同样也是，只需调整配对的顺序即可。假设分别移过了x,y个点，则len里任然有min(x,y)个点可以相互配对，其中一边多的部分任然与之前在len之外的点配对，这过程可能会跳过一些原来需要配对的点，对于这些点交换顺序即可（原来a，b两个位置可以配对，a原来是在向左时被选中，则此次选择在向右时选a，同理b).

\*/

int main()

{

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

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

for(int i = 1; i <= n; i++) dp[i][i] = 1;

for(int i = n; i >= 1; i--)

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

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

if(a[i] == a[j]) dp[i][j] = max(dp[i][j], dp[i+1][j-1] + 2);

}

int ans = 0;

for(int i = 1; i <= n; i++) ans = max(ans, dp[1][i] + dp[i+1][n]);

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

}

return 0;

}

/\*

线段树 区间乱搞 好题 思路题

题意：给定一个长度为 n 的非负数列，定义 mex(l, r) 为 l，r 区间里最小的没有出现的数字。求所有 mex(l, r) 的和

思路：比赛时想了一种分治区间的算法，虽然感觉可行，但是无奈细节巨烦，搞了3个小时都没搞出来！！！简单记录一下：以a[i]=0的i为中间分界线，把区间分成两部分，然后依次对两个区间找1,2,3...不断删区间，不断乱搞...

看了xiaodao的题解，巨赞。

\*\*\*直接合并区间难以进行时。。利用离线先固定一个端点。。沿着另一个端点扫应该是接下来第一个要想的方法了\*\*\*

先搞出mex[i]表示初始的mex(1,i)，已知是单调递增的。

next[x]记录右边第一个和a[x]相同的位置，如果没有则等于n+1

沿着左端开始删除。。删除前要统计出以这个点为左端点的所有mex

考虑删除a[L]，那么将L+1到next[L]-1之间的mex值大于a[i]的全设置成a[i]

于是...我们需要维护...区间求和...区间修改...线段树裸题...

爆搞的话会TLE。

进一步分析发现mex至始至终都是单增的，所以对于上述“i到next[i]-1中mex值大于a[i]的改为a[i]”只要找到(i,next[i]-1)第一个mex值比a[i]大的位置，再区间覆盖即可，那找这个位置也是很简单的，Find()函数.

做完后看别人代码发现其实若a[i] > i则可以不用管它...

\*/

/\*

树状数组 离线查询 好题 启发题 经典题

题意：序列{A}, ai<=200000，若干询问[l,r]，回答区间[l,r]内有多少个ai与区间内其它数都互质

我是觉得这道题跟bzoj那道“求区间内不同的数字个数”很类似，如过我们把“不互质的数Not co-prime”对应着“不同数字”，那就很像了。

但不同点，也是这道题的难点在于，a与b不互质，b与c不互质，但a与c不一定不互质，也就是说，bzoj那道题的“相同关系”是有传递性的，但这里的“不互质关系”是没有传递性的！

那怎么解决这一问题呢？

我先预处理出对于每一个ai，右端第一个不互质的数aj，我用个vector<int> R来存，R[j].push\_back( i )；还预处理出L[i]表示ai的左端第一个与ai不互质的数的位置

然后离线扫描的时候，对于当前i，我把R[i]里面的下标全都减1（思考一下就会发现，减1之后肯定为0），L[i]再减1，i加1（就对应与bzoj那道题“对于之前所有和ai相等的数都为0，然后当前i加1，L[i]减1”）。

说一下为什么这么操作：

R[i]里面的全都减1：易知每个下标x只会存在于一个R[i]里面。这个操作就相当于，把i左端的“右端first Not co-prime的数为ai”的x消为0，因为我们之前遇到每一个x都加了1，这里遇到ai是第一个与ax不互质的数，我们要把x消掉为0

L[i]减1：i的哨兵，不解释

i加1：不解释

还有很重要的一个操作：在我们R[i]里的所有x都减1的时候判断一下 if(L[x] != -1) Update(L[x], +1); //Attention!

这个操作的目的就是把x的哨兵消掉

再联 系一下bzoj 1878，来写一下两题的对应联系

同：

hdu 4777 的Not co-prime关系对应着bzoj 1878的equal相等关系

hdu 4777 的R[i]里面的数其实就是对应着bzoj 1878的L[i]

hdu 4777 的L[i]也是对应着bzoj 1878的L[i]

所以我们的操作都用L[i]作为哨兵, Update(L[i], -1)

异：

<一对一 与 一对多>

bzoj 1878离线扫描时保证前面aj = ai,j < i都已经把j位置上的树状数组搞为0了，当然哨兵为-1除外，所以查询的时候时候可以Query(l,r)

hdu 4777离线扫描时对应着要保证对于gcd(aj, ai) != 1的位置j，j在树状数组里面为0，哨兵为-1除外，所以查询的时候可以Query(l,r).

那对于搞为0的操作，bzoj 1878只要消L[i]即可，因为是“一对一”的关系；hdu 4777需要对于R[i]里的每个x都手动消为0，因为这里是一对多的关系---ai左端有多个不互质的关系

bzoj 1878的一对一关系，使得我们每次只要Update(L[L[i]], +1)就可以消掉之前的影响；hdu 4777的一对多关系，就要求我们得对每个x ∈ R[i]，都得Update(L[x], +1)

<传递性 与 非传递性>

其实传递性就是一对一，非传递性就是一对多

\*/

void Update(int pos, int val)

{

while(pos <= n) {

c[pos] += val;

pos += low(pos);

}

}

int Query(int pos)

{

int sum = 0;

while(pos) {

sum += c[pos];

pos -= low(pos);

}

return sum;

}

void init\_factor()

{

for(int i = 2; i <= 200000; i++) if(!composite[i])

for(int j = i+i; j <= 200000; j += i) composite[j] = true;

for(int i = 2; i <= 200000; i++) if(!composite[i])

for(int j = i; j <= 200000; j += i)

factor[j].push\_back( i );

}

void init()

{

memset(c, 0, sizeof(c));

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

for(int i = 1; i <= n; i++) Q[i].clear(), R[i].clear();

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

int tmp = -1;

for(vector<int> :: iterator it = factor[ a[i] ].begin(); it != factor[ a[i] ].end(); it++) {

tmp = max(tmp, p[\*it]);

p[ \*it ] = i;

}

L[i] = tmp;

}

for(int i = 1; i <= 200000; i++) p[i] = INF;

for(int i = n; i >= 1; i--) {

int tmp = INF;

for(vector<int> :: iterator it = factor[ a[i] ].begin(); it != factor[ a[i] ].end(); it++) {

tmp = min(tmp, p[\*it]);

p[\*it] = i;

}

if(tmp != INF) R[tmp].push\_back( i );

}

}

void solve()

{

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

for(vector<int> :: iterator it = R[i].begin(); it != R[i].end(); it++) {

Update(\*it, -1);

if(L[\*it] != -1) Update(L[\*it], +1); //Attention!

}

if(L[i] != -1) Update(L[i], -1);

Update(i, 1);

for(vector<Pii> :: iterator it = Q[i].begin(); it != Q[i].end(); it++) {

ans[ it->second ] = Query(i) - Query(it->first-1);

}

}

}

int main()

{

int query, l, r;

init\_factor();

while(scanf("%d %d", &n, &query) , n || query) {

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

init();

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

scanf("%d %d", &l, &r);

Q[r].push\_back( Pii(l, i) );

}

solve();

for(int i = 0;i < query; i++) printf("%d\n", ans[i]);

}

return 0;

}

/\*

数学 实现 好题

题意：从给定的[a,b]区间选出一个数x，从[c,d]区间选出数y，问有多少对(x,y)满足(x+y) % p = m

思路：

其实成都现场赛的时候我在网上就看到题了，觉得这道题我自己的算法编码好麻烦，现场这么多大神过，应该有什么简单的解法吧。。然后，然后成都现场赛重现之后看了几个题解，编码都挺复杂的------对我这种一分多种情况考虑就跪的菜鸟来说。于是硬着头皮按自己的思路写了一发，1Y，调试挺久，但代码看题来还挺简单。

我的第一个想法就是把‘模p等于m’ 变成‘模p等于0’，A ≡ m (mod p) A-m ≡ 0 (mod p)

所以，‘模p等于m’变成‘模p等于0’很简单，把区间[a,b]或者[c,d]左移m位即可，变成[a-m, b-m] 或者[c-m, d-m]，我这里移的是[a,b]。要是a-m小于0的话根据模的性质，我们可以再右移p个单位，变成[a - m + p, b - m + p]

然后想到fushuai提出的容斥定理，猜想 f( [a,b], [c,d] ) = f( [0, b], [0,d] ) + f( [0, a-1], [0, c-1] ) - f( [0, a-1], [0, d] ) - f( [0, c-1], [0, b] )

写好点 ans = f(b,d) + f(a-1, b-1) - f(a-1, d) - f(c-1, b)

我也不知道为什么这里容斥是对的，组合学的证明对我来说总是太难了。。。那我们目标就变成了求给定区间[0, a], [0,b]，从中选出x，y，使得(x+y) % p = 0有多少对，记为f(a,b)。

[0,a]区间可以拆成 : 0, [1..p], [1..2p],...,[1..kp], kp+1, kp+2...a

就是按单位长度为p，把区间[0,a]拆开。。中间那部分可以合成一部分，则变成 : 0, [1..kp], kp+1, kp+2...a

那[0,a]就被我们拆成了3部分，第一部分单独一个 0 ，第二部分是k个p的完全剩余类[1..kp]，第三部分是不足p的完全剩余类的部分。

同理拆[0, b]。 为什么要把0出来？我说不清，我也是在调试的时候发现的，一开始是把0归到第一个剩余类里面的。。只能说知道，还不能说理解。。有理解地比较好的大神求教一下。

接下来就是分情况计算啦。。可能我说的比较啰嗦，其实下面大家可以自己画个图推了。。

(1)对于[0,a]的第一部分0，有b/p+1种情况使得0 + y ≡ 0 (mod p)，[0,b]的0有a/p+1种情况使得x + 0 ≡ 0 (mod p)，因为两个0 + 0重复了一次，减1

(2)对于[0,a]的第二部分，有p \* (a/p) \* (b/p) 种情况

(3)对于[0,a]的第三部分，从b的第二部分[1..kp]选出y的话，有(a % p) \* (b/p) 种情况，a%p表第三部分的大小，b/p表第三部分的这些数可以在[0,b]的第二部分分别找到多少个y使得x + y ≡ 0 (mod p)。同理[0,b]有(b%p) \* (a/p)

然后，从[0,a]的第三部分选出x，从[0,b]的第三部分选出y，有多少种情况，我推出来的公式是max{0, b%p - (p-a%p) + 1) } , 其中a%p >= b%p

\*/

ll calc(ll a, ll b)

{

if(a < 0 || b < 0) return 0;

ll ans = 0;

ans += p \* (a/p) \* (b/p) + (a % p) \* (b/p) + (b % p) \* (a/p);

ans += a/p + b/p + 1; //x=0，y=kp和y=0，x=kp的情况

a = a % p, b = b % p;

if(a < b) swap(a, b);

ans += max((ll)0, b - (p-a) + 1); //[0,a],[0,b]第三部分选x/y的情况

return ans;

}

int main()

{

int cases, Cas = 0; ll a, b, c, d;

scanf("%d", &cases);

while(cases--) {

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

ll tot = (b - a + 1) \* (d - c + 1);

if(a >= m) a -= m, b -= m; //左移区间

else a = a - m + p, b = b - m + p;

ll ans = calc(b, d) + calc(a-1, c-1) - calc(a-1, d) - calc(c-1, b);

ll d = gcd(ans, tot);

ans /= d, tot /= d;

printf("Case #%d: %lld/%lld\n", ++Cas, ans, tot);

}

return 0;

}