# KMP算法

#include <cstdio>

#include <cstring>

#include <algorithm>

using namespace std;

#define N 1000010

int nxt[N];

char S[N],T[N];

void get\_next(char T[N],int tlen)

{

int i=0,j=-1;

nxt[0]=-1;

while(i<tlen)

if(j==-1||T[i]==T[j]) nxt[++i]=++j;

else j=nxt[j];

}

/\*

返回模式串T在主串S中首次出现的位置

返回的位置是从0开始的。

\*/

int kmp\_index(char S[],int slen,char T[],int tlen)

{

get\_next(T,tlen);

int j=0;

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

{

while(j>0&&S[i]!=T[j]) j=nxt[j];

if(S[i]==T[j]) ++j;

if(j==tlen) return i-tlen+1;

}

return -1;

}

/\*

返回模式串T在主串S中出现的次数

\*/

int kmp\_count(char S[],int slen,char T[],int tlen)

{

get\_next(T,tlen);

int ans=0,j=0;

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

{

while(j>0&&S[i]!=T[j]) j=nxt[j];

if(S[i]==T[j]) ++j;

if(j==tlen)

{

++ans;

j=nxt[j];//子串可重叠

//j=0;//子串不可重叠

}

}

return ans;

}

# 扩展KMP算法

/\*\*

拓展kmp是对KMP算法的扩展，它解决如下问题：

定义母串S，和字串T，设S的长度为n，T的长度为m，

求T与S的每一个后缀的最长公共前缀，

也就是说，设extend数组,extend[i]表示T与S[i,n-1]的最长公共前缀，

要求出所有extend[i](0<=i<n)，

设辅助数组nxt[i]表示T[i,m-1]和T的最长公共前缀长度

\*\*/

#include<iostream>

#include<string>

#include<cstring>

#include<cstdio>

using namespace std;

const int N=100010;

int nxt[N],extand[N];

char S[N],T[N];

void get\_next(char T[],int tlen)

{

int a=0;

nxt[0]=tlen;

while(a<tlen-1 && T[a]==T[a+1]) a++;

nxt[1]=a;

a=1;

for(int k=2; k<tlen; k++)

{

int p=a+nxt[a]-1,L=nxt[k-a];

if( (k-1)+L >= p)

{

int j = (p-k+1)>0 ? (p-k+1) : 0;

while(k+j<tlen && T[k+j]==T[j]) j++;

nxt[k]=j;

a=k;

}

else

nxt[k]=L;

}

}

void get\_extand(char S[],int slen,char T[],int tlen)

{

get\_next(T,tlen);

int a=0;

int min\_len = slen < tlen ? slen : tlen;

while(a<min\_len && S[a]==T[a]) a++;

extand[0]=a;

a=0;

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

{

int p=a+extand[a]-1, L=nxt[k-a];

if( (k-1)+L >= p)

{

int j= (p-k+1) > 0 ? (p-k+1) : 0;

while(k+j<slen && j<tlen && S[k+j]==T[j]) j++;

extand[k]=j;

a=k;

}

else

extand[k]=L;

}

}

int main(void)

{

while(scanf("%s%s",S,T)==2)

{

int slen=strlen(S),tlen=strlen(T);

get\_extand(S,slen,T,tlen);

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

printf("%d ",nxt[i]);

puts("");

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

printf("%d ",extand[i]);

puts("");

}

return 0;

}

# manacer算法

/\*

manacher：可以解决最长回文问题。

算法：

首先，将字符串的每个字符左右加入#，

并在s0位置加入\*（如果字符串中本身含有这些，则换成未出现过的字符），

此时字符串的长度为len+len+3，

即加入了len+1个#和一个\*; （比如：aba变成 \*#a#b#a#）

得到一个p数组，该数组是基于新字符串进行的，

p[i]表示以 i 为中心的最长回文半径

p[i] - 1正好是原字符串中最长回文串的长度，如

S \* # a # b # b # a # h # o # p # x # p #

p 1 2 1 2 5 2 1 2 1 2 1 2 1 2 1 4 1 2 1

\*/

void manacher(char S[],int len) //manacher 函数

{

for(int i=len;i>=0;--i) //将s扩大，中间加#，开头加\*

{

S[i+i+2]=S[i];

S[i+i+1]='#';

}

S[0]='\*';

int id,mx=0; //mx代表以id为中心时，到达最远的位置

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

{

if(mx>i) p[i]=min(p[2\*id-i],mx-i); //如果到达最远位置大于当前匹配的地方，则p[i]取min（id的对称点的p，到达最远距离-i）

else p[i]=1; //如果i在mx右方，则p[i]=-1;

while(S[i-p[i]] == S[i+p[i]])++p[i]; //判断i回文长度

if(i+p[i]>mx) //看是否要更新最远距离，如果要，将此点作为中心。

{

id=i;

mx=p[i]+i;

}

}

}

# 最小表示法和最大表示法

/\*

循环字符串的最小表示法的问题可以这样描述：

对于一个字符串S，求S的循环的同构字符串S’中字典序最小的一个。

\*/

//返回字符串S'的首字母的id

int get\_min(char s[],int len)

{

int i = 0, j = 1, l;

while(i < len && j < len)

{

for(l = 0; l < len; l++)

if(s[(i + l) % len] != s[(j + l) % len]) break;

if(l >= len) break;

if(s[(i + l) % len] > s[(j + l) % len])

{

if(i + l + 1 > j) i = i + l + 1;

else i = j + 1;

}

else if(j + l + 1 > i) j = j + l + 1;

else j = i + 1;

}

return i < j ? i : j;

}

//返回字符串S'的首字母的id

int get\_max(char s[],int len)

{

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

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

{

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

if(!t) k++;

else

{

if(t > 0)

{

if(j+k+1 > i) j = j+k+1;

else j = i+1;

}

else if(i+k+1 > j) i = i+k+1;

else i = j+1;

k = 0;

}

}

return i < j ? i : j;

}

# 拉格朗日插值法求自然数幂和

/\*

CF622F

拉格朗日插值法求1^k+2^k+3^k+...+n^k的值

时间复杂度为O(k\*log(mod))，log(mod)为快速幂的复杂度

\*/

#include <cstdio>

#include <cstring>

#include <algorithm>

#include <iostream>

using namespace std;

typedef long long ll;

#define N 1000010

const ll mod=1e9+7;

ll f[N],fac\_inv[N],inv[N];

void init()

{

inv[1] = 1;

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

{

if(i>=mod) break;

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

}

//求前N项阶乘的逆元

fac\_inv[0]=1;

for(int i=1;i<N;++i) fac\_inv[i]=fac\_inv[i-1]\*inv[i]%mod;

}

ll Pow(ll x,ll n)

{

ll ans=1;

while(n)

{

if(n&1) ans=ans\*x%mod;

x=x\*x%mod;

n>>=1;

}

return ans;

}

/\*

求1^k+2^k+3^k+...+n^k的值

时间复杂度为O(k\*log(mod))，log(mod)为快速幂的复杂度

\*/

ll n\_k(int n,int k)

{

f[0]=0;

for(int i=1;i<=k+2;++i)//求自然数幂和的前k+2项

f[i]=(f[i-1]+Pow(i,k))%mod;

if(n<=k+2) return f[n];

ll cur=1,ans=0;

for(int i=1;i<=k+2;++i)//预处理分式的分子

cur=cur\*(n-i)%mod;

for(int i=1;i<=k+2;++i)

{

ll inv1=Pow(n-i,mod-2);//对(n-i)求逆元

ll inv2=fac\_inv[i-1]\*fac\_inv[k+2-i]%mod;//对阶乘求逆元

int sign=(k+2-i)%2?-1:1;//判正负

ans=(ans+sign\*inv1\*inv2%mod\*f[i]%mod\*cur%mod)%mod;

}

ans=ans+mod;

if(ans>=mod) ans-=mod;

return ans;

}

int main()

{

init();

int n,k;

cin>>n>>k;

cout<<n\_k(n,k)<<endl;

return 0;

}

# 斯坦纳树

/\*

斯坦纳树问题是组合优化问题，与最小生成树相似，是最短网络的一种。

最小生成树是在给定的点集和边中寻求最短网络使所有点连通。

而最小斯坦纳树允许在给定点外增加额外的点，使生成的最短网络开销最小。

给一个有n个节点带权完全图，编号从1~n，

现在有一个0节点，可以与m个节点相连，且给出了权值，

求0和这m个节点组成的最小斯坦纳树。

令s表示m个节点的子集，

dp[s][i]表示包含s中节点且以i为根的树的最小权值

dp[s][i]=min(dp[s1][i]+dp[s2][i],dp[s][j]+g[i][j]),

其中s1+s2=s,

\*/

#include <cstdio>

#include <cstring>

#include <algorithm>

using namespace std;

#define N 60

int g[N][N],a[N],dp[(1<<10)+10][N];

int n,m;

const int inf=0x3f3f3f3f;

int main()

{

int ca;

scanf("%d",&ca);

while(ca--)

{

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

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

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

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

int x,w;

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

g[0][i]=g[i][0]=inf;

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

{

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

g[0][x]=g[x][0]=w;

a[i]=x;

}

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

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

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

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

memset(dp,0x3f,sizeof(dp));

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

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

dp[1<<i-1][j]=g[a[i]][j];

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

{

if(s&(s-1)==0) continue;

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

for(int ss=s;ss;ss=(ss-1)&s)

dp[s][i]=min(dp[s][i],dp[ss][i]+dp[s-ss][i]);

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

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

dp[s][i]=min(dp[s][i],dp[s][j]+g[i][j]);

}

printf("%d\n",dp[(1<<m)-1][0]);

}

return 0;

}

# 动态树

/\*

HDU 4010

题目大意

给一棵树，有点权，要求维护4种操作。

操作1：加一条边。

操作2：删一条边。

操作3：将一条路径上的点权增加w。

操作4：询问一条路径上的点权最大值

\*/

#include <bits/stdc++.h>

using namespace std;

/\*

【算法介绍】

所谓动态树（LCT），从数据结构上讲，其实只不过是splay的扩展。

ACM中的很多问题，都是基于一棵固定形态的树。然而LCT的存在，使得就算树形态发生变化，也可以解决很多询问。

<1> 首先要明确，我们在内部是用splay实现LCT，而在splay中，每个点到根节点距离的期望为log级别，这是LCT复杂度有保障的基础

<2> 在LCT中我们会有很多条preferred path（最多n条，即每点都独属于一条preferred path）。

对于每条preferred path，以深度为排序关键字，便可以得到一棵splay（我们把这棵splay称作auxiliary tree）

<3> 我们不需要特殊维护所有点所呈的森林结构，森林结构的所有信息都存在splay中

splay内部的父子关系是由其位置排列的左右来决定的

splay中的关键词是是深度，所以对于任意一个点x，如果把其转为其所在splay的根，则ch[x][0]就是x的祖先，ch[x][1]就是x的子孙

即我们需要注意：ch[x][0]并不一定是x的父节点，last(ch[x][0])才是；ch[x][1]并不一定是x的子节点，first(ch[x][1])才是。

同样，在splay中，fa[x]所指向的节点，并非一定是真实树形态中x的父节点。

总结一下，就是——在splay中，我们只有通过理清深度关系，才可以明确得到所有点在真实树形态中的父子关系。

而splay外部，splay的根节点x中存的fa[x]，在真实树结构中，就恰好对应为x节点的父节点。

【算法实现】

<1> access(x)

该操作的功能是取出从x到真实树根直接的所有节点（这些节点一个也不多一个也不少，设为集合S）并链接在splay中

在该函数中，每个节点将沿着fa[]指针一直爬到其所在splay的根节点。

由之前的理论可知，在access()中，沿着fa[]的爬升过程中，不一定是在同一棵splay()中进行的，而可能需要链接起不同的splay。

因此每爬升到一个节点x，我们都对x做splay操作，使得ch[x][0]为x的祖先，ch[x][1]为x的子孙，把ch[x][1]丢掉，其都不属于集合S

同时因为不设计到在ch[x][1]子树内任何一个节点的修改，所以我们需要使得fa[ch[x][1]]保持不变，

而只是在access的路径中，链接ch[x][1]为son（son为从起点向上爬升来的暂时的到树根的链）。

我们发现，如果x到root的路径很长，我们这里爬升的步数就可能很多，对应一个比较大的时间复杂度，也就是论文中所说的糟糕初势能。

但是，因为采用了splay压缩路径，使得复杂度均摊为log级别。通常，我们在access(x)后再做splay(x)操作，使得x变为根方便后续的处理。

<2> find\_root(x)

对于find\_root(x)，只要先access(x)，再顺着ch[x][0]向左递归，递归的终点便是root.

<3> make\_root(x)

对于make\_root(x)，只要先access(x)，再对x所在整棵splay做reverse，并在翻转整条链之后，把fa[root]赋值给fa[x]，使得该splay还具备向其上的splay做转移的条件。

该操作只会影响(x, root)这条路径的深度，并且把首端点的延展传递。形象而言，就像是一只蜈蚣，我们抓住其"脊锥"，使得(x, root)转了个头

<4> 对于link(x, y)操作，我们先make\_root(x)，把其到实际树根的整条链都取出来，再直接使得fa[x] = y就好了。

<7> 对于cut(x, y)操作，我们先make\_root(x)，再做access(y)操作，使得提取出了链(y, x)，把fa[ch[y][0]]和ch[y][0]设为0，即使得y与y的实际父节点删了边

<8> 对于modify(x, y, val)操作，我们先make\_root(x)，再做access(y)操作，使得提取出了链(y, x)，再add(y, val)即可

<9> 对于query(x, y)操作，我们先make\_root(x)，再做access(y)操作，使得取出了链(y, x)，再return 返回关于y的值即可

【要点】

<1> 既然是splay，因为涉及到pushup操作，于是我们必须使得根节点处恰好维护着完全子树的值，操作的点位也要在根处

同样的道理，只要是设计从根向下遍历，就必须做pushdown操作，LCT中的pushdown写法稍特殊，先找到根再一口气pushdown，可以减少pushdown的次数。

<2> LCT有时候维护的是有根树，此时就不需要执行（也不能执行）make\_root操作，但是通过access，就能得到该有根树的祖先。操作在该祖先点上展开即可。

\*/

#define N 100010

int n;

pair<int, int>edge[N];

vector<int>a[N];

struct LCT

{

int ID;

//当x是伸展树的根节点时，

//fa[x]表示在森林中这颗伸展树所代表的重链的top节点的父节点，

//否则表示x在伸展树中的父节点

int ch[N][2], fa[N];

int v[N];

int mx[N];

bool rv[N];

int ad[N];

void clear(int x)

{

ch[x][0] = ch[x][1] = fa[x] = 0;

rv[x] = ad[x] = 0;

}

int newnode(int val)

{

int x = ++ID; clear(x);

mx[x] = v[x] = val;

return x;

}

inline int D(int x)

{

return ch[fa[x]][1] == x;

}

bool isroot(int x)

{

return ch[fa[x]][0] != x && ch[fa[x]][1] != x;

}

void reverse(int x)

{

if (x == 0)return;

swap(ch[x][0], ch[x][1]);

rv[x] ^= 1;

}

void add(int x, int val)

{

if (x == 0)return;

v[x] += val;

mx[x] += val;

ad[x] += val;

}

void pushup(int x)

{

mx[x] = v[x];

mx[x]=max(mx[x],max(mx[ch[x][0]], mx[ch[x][1]]));

}

void pushdown(int x)

{

if (rv[x])

{

reverse(ch[x][0]);

reverse(ch[x][1]);

rv[x] = 0;

}

if (ad[x])

{

add(ch[x][0], ad[x]);

add(ch[x][1], ad[x]);

ad[x] = 0;

}

}

void rootdown(int x)

{

if (!isroot(x))rootdown(fa[x]);

pushdown(x);

}

void setc(int x, int y, int d)

{

ch[x][d] = y;

if (y)fa[y] = x;

if (x)pushup(x);

}

//旋转操作：旋转使得节点x与其父节点交换位置

void rotate(int x)

{

int f = fa[x];

int ff = fa[f];

bool d = D(x);

bool dd = D(f);

setc(f, ch[x][!d], d);

setc(x, f, !d);

if (ch[ff][dd] == f)setc(ff, x, dd); else fa[x] = ff;

}

//旋转操作：把x旋转到其所在splay的根节点

void splay(int x)

{

rootdown(x);

while (!isroot(x))

{

//if (!isroot(fa[x]))pushdown(fa[fa[x]]); pushdown(fa[x]); pushdown(x); // WHY WA

if (!isroot(fa[x]))rotate(D(x) == D(fa[x]) ? fa[x] : x);

rotate(x);

}

}

//LCT核心操作：在当前的树形态下，取出x到root路径（preferred path）上的所有点，形成auxiliary tree

//我们只需要对当前爬升到的点now做splay(now)操作并丢掉其右子树（子孙），过程中自动把祖先的fa[root]转移为了fa[now]，继续上跳即可。

void access(int x)

{

for (int son = 0, now = x; now; son = now, now = fa[now])

{

splay(now);

setc(now, son, 1);

}

splay(x);

}

//LCT基本操作：先找到x所在的auxiliary tree（即preferred path），并查找返回该条路径最小的节点（即根）

int find\_root(int x)

{

access(x);

while (ch[x][0])pushdown(x), x = ch[x][0];

return x;

}

//LCT基本操作：先找到x所在的auxiliary tree（即preferred path），并把x旋转为这条路径的根

void make\_root(int x)

{

access(x);

reverse(x);

}

//LCT基本操作，先使得x为真实树的根，再提取出y到x这条路径所表示的splay

void getlink(int x, int y)

{

make\_root(x);

access(y);

}

//LCT重要操作：在x与y不在同一棵树的条件下，把x变为子树的根，并把x链接为y的子节点

//如果是有根树，则x为y的祖先

void link(int x, int y)

{

if (x == y || find\_root(x) == find\_root(y)) { puts("-1"); return; }

make\_root(x);

fa[x] = y;

}

//LCT重要操作：在x与y在同一棵树的条件下，把x变为子树的根，并把y与y祖先之间的边彻底断开

//如果x和y不直接相连，则在x到y的路径中，会断开最靠近y的边

void cut(int x, int y)

{

if (x == y || find\_root(x) != find\_root(y)) { puts("-1"); return; }

getlink(x, y);

fa[ch[y][0]] = 0;ch[y][0] = 0;

}

//LCT常用操作：在x与y在同一棵树的条件下，把链(x, y)上每个点的权值都+=val

void modify(int x, int y, int val)

{

if (find\_root(x) != find\_root(y)) { puts("-1"); return; }

getlink(x, y);

add(y, val);

}

//LCT常用操作：在x与y在同一棵树的条件下，询问链(x, y)上的最大点权

int query(int x, int y)

{

if (find\_root(x) != find\_root(y))return - 1;

getlink(x, y);

return mx[y];

}

/\*

//如果权值在边上，使用有根树，把边权值赋给儿子，此时统计x，y到lca路径上的值，

int query(int x, int y)

{

int ans=0;

access(y);

while(true)

{

splay(x);

if(fa[x]==0) break;

ans=max(ans,max(mx[ch[x][0]],v[x]));

x=fa[x];

}

ans=max(ans,mx[ch[x][1]]);

return ans;

}

\*/

//LCT DEBUG

void alldown(int x)

{

pushdown(x);

if (ch[x][0])alldown(ch[x][0]);

if (ch[x][1])alldown(ch[x][1]);

}

//LCT求最近公共祖先

int lca(int x, int y)

{

int ans=0;

access(y);

while(true)

{

splay(x);

if(fa[x]==0) break;

x=fa[x];

}

return x;

}

void solve()

{

mx[ID = 0] = -1e9;

for (int i = 1; i < n; ++i)scanf("%d%d", &edge[i].first, &edge[i].second);

for (int i = 1; i <= n; ++i) { int val; scanf("%d", &val); newnode(val); }

for (int i = 1; i < n; ++i)link(edge[i].first, edge[i].second);

int q, op, x, y, val;

scanf("%d", &q);

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

{

scanf("%d", &op);

if (op == 1)

{

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

link(x, y);

}

else if (op == 2)

{

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

cut(x, y);

}

else if (op == 3)

{

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

modify(x, y, val);

}

else

{

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

printf("%d\n", query(x, y));

}

}

puts("");

}

}lct;

int main()

{

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

{

lct.solve();

}

return 0;

}

/\*

论文 —— 《SPOJ375 QTREE 解法的一些研究》

概念学习:

<1> 访问

称一个点被访问过，如果刚刚执行了对这个点的access操作

<2> preferred child

如果节点x在子树中，最后被访问的节点在子树y中，且y是x的子节点，那么我们称y是x的preferred child

如果最后被访问的节点就是x本身，那么它没有preferred child，每个点到它preferred child之间的边称作preferred edge

<3> preferred path

整棵树被划分成了若干条preferred path，我们对于每条preferred path，用每个点的深度作为关键字，用一棵平衡树来维护。

所以在splay中，每个点左子树中的点，深度都比其小，意为其祖先，都在preferred path中这个点的上方

每个点右子树中的点，深度都比其大，意为其子孙，都在preferred path中这个点的下方

<4> Auxiliary Tree

我们使用splay维护preferred path，把树T分解成若干条preferred path。

我们只需要知道这些路径之间的连接关系，就可以表示出这棵树T

<5> Link-Cut Tree

将要维护的森林中的每棵树T表示为若干个Auxiliary Tree，通过Splay中的深度关系，将这些Auxiliary Tree连接起来 的数据结构

HDU4010

[题意]

有一棵n（3e5）个点的树

每个点有一个权值，权值在任何时候都不会超过int范围

有Q（3e5）个操作，操作包括4种类型

1 x y：Link操作

要求x与y属于两棵不同的树，此时连接x与y，从而使得两棵树合并为一棵

2 x y：Cut操作

要求x与y属于一棵相同的树，先使得x变为该树的根，再切断y与fa[y]之间的边，使得一棵树分裂为两棵

3 val x y：修改操作

使得x与y之间路径上所有点权都加val

4 x y：询问操作

输出x与y路径上点权的最大值

[分析]

LCT的模板题。与之类似的——

SPOJ OTOCI：涉及到单点修改和链上求和，对x做单点修改的时候只要splay(x）即可，并不需要access(x)

HDU5333：涉及到维护链上次大值，虽然代码量大了很多，但是本质都相同

BZOJ2002

[题意]

有1 ~ n共计n（2e5）个点。

对于每个点i，有一个弹跳距离v[i]（正整数），如果i + v[i] <= n，则我们在达到 i 点之后会被弹到 i + v[i]点

否则在到达 i 点之后就会被弹飞。有m（1e5）个操作——

1 x：问你从x出发弹多少次会被弹飞

2 x val：改变v[x]为val

[分析]

通常而言，LCT的树边是双向边，这使得在过程中，我们只要确定好了链接关系，任意一点为树根其实都不太影响

但是，有些题中LCT的树边是单向边，此时就不能再使用LCT的make\_root()函数了，参见此题——

对于这道题，我们可以把每次弹跳的起点和终点之间连边。形成有向树（也是DAG）。于是——

<1> 对于询问操作，就是问你每个点到其所在树根之间的边数。

<2> 对于修改操作，就是涉及到cut与link两种操作。

需要注意的是，这道题与传统的LCT不同，边是有向边。

但是我们可以把其当做无向边来看，心里上明确跳跃过程是从深度较高的点向深度较低的点进行的就好。

可是在LCT中，我们经常会做make\_root(x)这样之类的操作，这实际中是不允许的。

本题中需要实现的函数是cut()和link()，其中——

cut(x, y)的功能是把x与y之间的边删掉，此时我们需要access(x)，取出x所在的整条splay（即从x到root的路径）

然后使得x与其祖先的关联性完全断开，即执行ch[x][0] = fa[ch[x][0]] = 0操作

link(x, y)的功能是把x与y之间连边，因为每个点的出度都最多为1.

所以此时显然x应当是其所在splay的祖先，我们只需要splay(x)，fa[x] = y即可（access(x)实际也一样）。

HDU5967 2016年中国大学生程序设计竞赛（合肥）小R与手机

[题意]

n（2e5）个点，每个点x最多一个出度指向v[x]。

告诉你初始的链接关系，如果v[x] == 0表示无出度

m（2e5）个操作——

1 x y：使得x指向y

2 x：询问如果从x出发，是否能走到一个终止节点，有则输出，无则输出-1

[分析]

问题还是先分析好再做得好。

这道题的动态链接关系显然可以通过LCT实现。

然而，这个链接关系可能会形成环，而LCT显然是不支持环结构的。怎么办？

<1>无环，无环的话，从每个点沿着有向边出发，最终找到的节点，这个节点指向0即可

注意，明确对应关系的话，问题会更好做，要知道——这个点就是根节点！

<2>有环，有环的话，其实我们需要使得一个节点的指向关系存储却不生效。

想想看，这个节点，其实也只能是根节点。

得到之前的结论，问题便变得清晰一些了——

1. 我们只要生效所有链接关系，比如此时要链接(x, y)，x是还没有生效链接关系的，但是发现find\_root(y) == x，那便发现了环。

于是我们在物理结构上，x依然是access()后能得到的根（之一），但同时保留了v[x]，有v[x]不为0。即根的v[]不为0，也就对应着是存在了环

2. 我们过程中还要继续生效链关系，比如此时要链接(x, y)。但是——

对于x，其之前可能也存在着链接的目标z

如果x是根，（x，z）的链接关系可能未生效，我们直接相对于对x做崭新的链接即可

如果（x，z）的链接关系已经生效，首先我们一定要断开（x，z）的链接关系，然后对于z所在的tree的root，如果其root和v[root]之间断边了，则修改v[root]

然后的链接操作与之前无异，是傻瓜式的。

于是link这么写——

void link(int x, int y)

{

int w = find\_root(x);//先找到根节点

if (v[x] && x != w)cut(x, v[x]);//有边且不是根节点，就删边

v[x] = y;//标记

if (y && x != find\_root(y))splay(x), fa[x] = v[x];//根据是否有环决定是否加边，splay（x）而不是access(x)是因为x显然是根，其实都一样

if (v[w] && x != w && w != find\_root(v[w]))splay(w), fa[w] = v[w];//根据是否有环决定是否恢复边，splay（w）是因为w显然是根

//注意，这里我们是有判断x != find\_root(y)和 w != find\_root(v[w])，左侧之所以不用find\_root(x)和find\_root(w)，

//是因为左侧的点必然是根，而如果x == w，则显然x因为指向了y，x就不再是根，即导致最后一句话出错，于是加了条件x != w

//其实可以不加这些判定条件，而把x换成find\_root(x)，把w换成find\_root(w)来解决问题。

}

cut这么写——（其实这里的cut不需要参数y）

void cut(int x, int y)

{

//if (x == y || find\_root(x) != find\_root(y)) { puts("-1"); return; }

access(x);

ch[x][0] = fa[ch[x][0]] = 0;//rgt -> lft

}

HDU5333 LCT动态树 离线询问+树状数组（本题与HDU4677题意和做法完全相同，只是数据范围卡住了分块做法）

[题意]

给你一个n（1e5）点和m（2e5）边的无向图。

同时存在q（1e5）个询问，对于每个询问，有区间[L, R]

我们只连接两个端点都在[L, R]中的所有边，则该图会形成多少个连通块。

[分析]

显然，[1, L - 1] 与 [R + 1, n]中的点都没有任何边连接。

于是，答案其实是：(L - 1 + n - R) + [L, R]区间内起到实质效应的边数。

首先这道题询问众多，我们考虑离线化所有询问。

比如，我们按照右界从小到大的顺序，对所有边做排序，就可以只加入右界不超过R的所有边。

然而，这样子形成的图并不一定是树，可能形成了图，这使得我们无法动态维护splay。

于是，我们在加边的过程中也要动态维护其森林结构。

具体的做法是使用贪心，对于每次新加的边（不考虑自环和重边），如果加入该边会形成环，那么，

对于加边(x, y)之间的链，和该边共成的环，我们删掉环上的任意一条边，整个图的连通性都是一样的。

这里基于后效性最优的贪心原则，我们只要删掉该环上左端点最小的边，该边的用途是最小的。

于是，我们把边抽象为点，原始点的边权为极大，边点的权值为该边所对应的较小节点的编号，于是LCT的查询是树链上的最小节点编号。

因为右边界为R的询问也有很多，对应很多不同的L，我们用树状数组维护此时有从L ~ n的边加了多少条。

每多一条边便少一个连通块，由此更新答案。

void solve()

{

v[ID = 0] = inf;

for (int i = 1; i <= n; ++i) newnode(inf), bit.v[i] = 0;

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

{

scanf("%d%d", &edge[i].l, &edge[i].r);

if (edge[i].r < edge[i].l)swap(edge[i].l, edge[i].r);

}

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

{

scanf("%d%d", &q[i].l, &q[i].r);

q[i].id = i;

}

sort(edge + 1, edge + m + 1);

sort(q + 1, q + Q + 1);

for (int i = 1, p = 1; i <= Q; ++i)

{

while (p <= m && edge[p].r <= q[i].r)

{

int x = edge[p].l;

int y = edge[p++].r;

int o = newnode(x);

if (x == y || x == edge[p - 2].l && y == edge[p - 2].r)continue;

if (find\_root(x) == find\_root(y))

{

getlink(x, y);

int u = mp[y];

if (v[u] >= x)continue;

cut(u, edge[u - n].l);

cut(u, edge[u - n].r);

bit.modify(v[u], -1);

}

link(x, o);

link(y, o);

bit.modify(x, 1);

}

ans[q[i].id] = n - bit.check(q[i].l);

}

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

}

2014-2015 ACM-ICPC(CERC 14) J LCT动态树 + 可持久化线段树 Pork barrel

[题意]

n（1000）个点

m（100000）条边

q（1e6）个询问

对于每个询问[L,R]，问你，如果使用的边的边权在[L,R]范围内，那么——

我们能够得到的最小边权的极大生成森林的边权之和是多少，强制在线。

[分析]

如果可以离线化的话，我们直接把边按照权值从大到小做排序，也把询问按照其左界从大到小做排序.

然后枚举询问，询问的左界递减，我们加的边数也越多。

对于一条边，如果其加入成环了，我们会删掉其所在环上边权最大的一条边。

然后对于查询[L,R]，如果有能够起到同样功效的大边，会被删掉，如果有没有能够起到相同功效的大边，我们也保留了适当的小边。

也就是说，在树状数组中动态维护边权，check(r)就是答案。

然而，这道题需要使得询问在线，于是，如果我们还想要使用离线的方法的话，就要记录下所有可能询问的答案。

该做法是使用可持久化线段树（或者可持久化树状数组），对于每一个左界我们都维护一棵线段树，最后查询就好啦——

void solve()

{

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

v[ID = 0] = -inf;

for (int i = 1; i <= n; ++i) newnode(-inf);

topval = 0; for (int i = 1; i <= m; ++i)

{

scanf("%d%d%d", &edge[i].x, &edge[i].y, &edge[i].v);

rk[++topval] = edge[i].v;

}

sort(rk + 1, rk + topval + 1);

topval = unique(rk + 1, rk + topval + 1) - rk - 1;

sort(edge + 1, edge + m + 1);

pst.sz = 0; pst.rt[topval + 1] = 0;

for(int l = topval, p = 1; l >= 1; --l)

{

pst.rt[l] = pst.rt[l + 1];

while (p <= m && edge[p].v == rk[l])

{

int x = edge[p].x;

int y = edge[p].y;

int o = newnode(edge[p++].v);

if (find\_root(x) == find\_root(y))

{

getlink(x, y);

int u = mp[y];

if (v[u] <= v[o])continue;

cut(u, edge[u - n].x);

cut(u, edge[u - n].y);

pst.addp(pst.rt[l], 1, topval, lower\_bound(rk + 1, rk + topval + 1, v[u]) - rk, -v[u]);

}

pst.addp(pst.rt[l], 1, topval, l, v[o]);

link(x, o);

link(y, o);

}

}

scanf("%d", &Q);

int ans = 0;

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

{

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

l = lower\_bound(rk + 1, rk + topval + 1, l - ans) - rk;

r = upper\_bound(rk + 1, rk + topval + 1, r - ans) - rk - 1;

if (l > r)ans = 0;

else ans = pst.check(pst.rt[l], 1, topval, l, r);

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

}

}

\*/