并查集

基本的并查集

并查集一般用于涉及到集合合并的题。其最简板子如下：

```
const int N=100005;
int F[N];
int find(int x){//查询，自带路径压缩
    return x==F[x]?x:F[x]=find(F[x]);
}
void join(int x,int y){//合并集合
    F[find(x)]=find(y);
}
```

一般板子(未压行版本，可以做出许多变形)：

```
const int N=100005;
int F[N];
int find(int x){
    if(F[x]!=x){
        F[x]=find(F[x]);
    }
    return F[x];
}
void join(int x,int y){
    int Fx=find(x),Fy=find(y);
    if(Fx!=Fy){
        F[Fx]=Fy;
    }
}
```

更一般更通用的并查集： (其实你不一定需要封装成类，可以直接拿到外面正常用，这里只是告诉你对于非整数型间的并查集（比如string）该如何存储关系罢了)

```
template<typename T>
struct BCJ{
    map<T,T> F;//使用map储存集合关系
    BCJ(){}
    void reset(){F.clear();}//重置，进行下一回合操作
    void init(T x){mp[x]=x;}//需要自己手动初始化
    T find(T x){//溯源
        if(F[x]!=x){
            F[x]=find(F[x]);
        }
        return F[x];
```

```
    }
    void join(T x,T y){//合并集合
        T Fx=find(x),Fy=find(y);
        if(Fx!=Fy){
            F[Fx]=Fy;
        }
    }
};
```

例题

P1536 村村通

并查集裸题，直接按题意合并，统计自己是自己亲戚的节点个数即可

参考代码：

```
int F[1003];
void reset(int n){
    rep(i,1,n)F[i]=i;//初始化
}
int find(int x){
    return F[x]==x?x:F[x]=find(F[x]);
}
void join(int x,int y){
    F[find(x)]=find(y);
}

void solve(){
    while(true){
        cin(n);if(n==0)return;
        reset(n);
        cin(m);
        while(m--){
            cin(x);cin(y);
            join(x,y);
        }
        int ans=-1;
        rep(i,1,n){
            ans+=(F[i]==i);
        }
        cout<<ans<<et;
    }
}
```

P2814 家谱

还是并查集裸题，由于询问的是字符串间的关系，所以并查集初始化要点技巧，这里先把操作全部离线下来，初始化完并查集后再处理集合的合并、查询操作。

```
map<string,string> F;

string find(string s){
    return s==F[s]?s:F[s]=find(F[s]);
}
void join(string x,string y){
    F[find(x)]=find(y);
}

void solve() {
    string f;
    vector<string> list;
    while(true){
        scin(s);
        if(s[0]=='$')break;
        list.push_back(s);//将操作全部离线
        F[s.substr(1)]=s.substr(1);//记录操作的同时进行并查集初始化
    }
    for(auto &s:list){//然后就是正常的并查集合并的操作了
        if(s[0]=='#')f=s.substr(1);
        else if(s[0]=='+')join(s.substr(1),f);
        else cout<<s.substr(1)<<" "<<find(s.substr(1))<<et;
    }
}
```

封装一下也行，只是写法上有点麻烦了：

```
template<typename T>
struct BCJ{
    map<T,T> F;
    BCJ(){}
    void reset(){F.clear();}
    void init(T x){F[x]=x;}
    T find(T x){//这里没有压行，压行了后码量和不封装的极简板子差不多
        if(F[x]!=x){
            F[x]=find(F[x]);
        }
        return F[x];
    }
    void join(T x,T y){
        T Fx=find(x),Fy=find(y);
        if(Fx!=Fy){
            F[Fx]=Fy;
        }
    }
};

void solve() {
    BCJ<string> bcj;//存储string的并查集
    string f;
    vector<string> list;
```

```
    while(true){
        scin(s);
        if(s[0]=='$')break;
        list.push_back(s);
        bcj.init(s.substr(1));//初始化并查集
    }
    for(auto &s:list){
        if(s[0]=='#')f=s.substr(1);
        else if(s[0]=='+')bcj.join(s.substr(1),f);
        else cout<<s.substr(1)<<" "<<bcj.find(s.substr(1))<<et;
    }
}
```

扩展域并查集

也称种类并查集，用于维护有多种关系的集合间合并操作，基本板子：

```
int F[20004*2];//扩展成两倍
int find(int x){
    return x==F[x]?x:F[x]=find(F[x]);
}
void join(int x,int y){
    F[find(x)]=find(y);
}
```

可以看到，扩展域并查集和常规并查集代码几乎一致，只是记录元素间关系的F[]数量被扩展了一个倍数（一般来说，有多少种集合，就扩展成多少倍）。

此时F[i]和F[n+i]记录的是数i在两个不同的集合的亲戚。

整数型元素的扩展域并查集可以简单的直接"扩展"其F[]大小得到，但是对于非整数型，就只能老实的多建几个不同名字的map来保存同一元素在不同集合的亲戚关系了（或者说，你建一个map数组，数组大小就是不同种类的集合的数量），此时find函数与join函数都得重新设计，比如用join(int x,int i,int y,int j);将x在第i种集合中的亲戚合并到y在第j种集合中的亲戚中去。

例题

P1525 [NOIP2010 提高组] 关押罪犯

非常典的题，学并查集必经之路。按照贪心的思想，把输入离线下来，从大到小排序，再按照"敌人的敌人就是自己的朋友"的逻辑分配罪犯，如果分配过程中发现了"敌人的敌人还是自己的敌人"的现象，直接输出仇恨值即可。

```
int F[20004<<1];//扩展一次
int find(int x){
    return x==F[x]?x:F[x]=find(F[x]);
}
void join(int x,int y){
```

```
    F[find(x)]=find(y);
}

struct P{
    int i,j,w;
    bool operator<(const P& b){
        return w>b.w;
    }
}ask[100005];

void solve() {
    cin(n);cin(m);
    rep(i,1,n)F[i]=i;
    rep(i,1,m){//离线操作
        cin>>ask[i].i>>ask[i].j>>ask[i].w;
    }
    sort(ask+1,ask+1+m);
    rep(i,1,m+1){
        if(find(ask[i].i)==find(ask[i].j)){//两人已经被分配到同一间监狱了，冲突不可避免，直接输出
            cout<<ask[i].w;
            return;
        }else{
            //如果i还没有树敌，那就给他树个j作为敌人，不然就把i的敌人与j合并
            if(!F[ask[i].i+n])F[ask[i].i+n]=ask[i].j;
            else join(ask[i].i+n,ask[i].j);
            //同理
            if(!F[ask[i].j+n])F[ask[i].j+n]=ask[i].i;
            else join(ask[i].j+n,ask[i].i);
        }
    }
}
```

还有一种二分图染色的方法，不过还没了解，到时候再补

P2024 [NOI2001] 食物链

有点麻烦，要维护三种关系，同类，天敌，食物（我的食物能吃我的天敌（？））。

```
int F[300005];//[同类][天敌+n][食物+2*n]//咋分配意义看自己喜欢，别自己看不懂就行
int find(int x){if(F[x]!=x)F[x]=find(F[x]);return F[x];}
void join(int x,int y){F[find(x)]=find(y);}
int main(){//远古码风，不改了
    int n,k,t,x,y,ans=0;cin>>n>>k;
    for(int i=1;i<=3*n;i++)F[i]=i;
    while(k--){
        cin>>t>>x>>y;
        if(x>n||y>n){ans++;continue;}
        if(t==1){
            if(find(x+n)==find(y)||find(x+2*n)==find(y)){ans++;continue;}
            else{
                join(x,y);
```

```
                join(x+n,y+n);
                join(x+2*n,y+2*n);
            }
        }else{
            if(x==y||find(x)==find(y)||find(x+2*n)==find(y)){ans++;continue;}
            else{
                join(x,y+2*n);
                join(x+n,y);
                join(x+2*n,y+n);
            }
        }
    }
    cout<<ans;
    return 0;
}
```

带权并查集

不管是普通并查集还是扩展域并查集都只能维护元素间的亲戚关系，无法维护元素间的权值关系，这时候就需要用到带权并查集，基本板子如下：

```
const int N=100005;
int F[N],val[N];
int find(int x){
    if(F[x]!=x){
        int t=F[x];//暂存其亲戚
        F[x]=find(F[x]);
        val[x]+=val[t];
        //合并之前是val[x]=x->t, val[t]=t->Ft
        //所以合并后val[x]应+=val[t]（要让x直接连到Ft那边，权值自然得加上val[t]
        //不一定是相加。还要看具体的题目，有些题目甚至要求你不止维护一种权值
    }
    return F[x];
}
void join(int x,int y,int v){
    int Fx=find(x),Fy=find(y);
    if(Fx!=Fy){
        F[Fx]=Fy;
        val[Fx]=v+val[y]-val[x];
        //理解成向量加减法，假设x和y的亲戚均不为自己，那么有如下关系：
        //y->Fy
        //↑  ↑
        //x->Fx
        //由向量加法知：x->y->ry路径上的“权值和”等于x->rx->ry上的“权值和”
        //所以val[Fx]应等于v+val[y]-val[x]
        //不一定是简单加减，只是举个例子方便理解，实际还得具体题目具体分析
    }
}
```

例题

P1196 [NOI2002] 银河英雄传说 设一个权值用于维护当前集合中，该元素是第几个，再设一个值用于维护各个集合大小，把权值转移的关系推出来即可。

```
int F[30004],rnk[30004],siz[30004];
int find(int x){
    if(x!=F[x]){
        int t=F[x];
        F[x]=find(F[x]);
        rnk[x]+=rnk[t]-1;//排名为k的元素前面有k-1个元素
    }
    return F[x];
}
void join(int x,int y,int v){//x移到y后面
    int Fx=find(x),Fy=find(y);
    if(Fx!=Fy){
        F[Fx]=Fy;
        rnk[Fx]+=siz[Fy];//排名向后移动siz个单位，毕竟前面插了一整个队
        siz[Fy]+=siz[Fx];//把原队首管理的元素数更新到新队首管理的元素数那
    }
}

void solve() {
    rep(i,1,30000){
        F[i]=i;
        rnk[i]=siz[i]=1;
    }
    cin(q);
    while(q--){
        scin(tp);cin(i);cin(j);
        if(tp=="M"){
            join(i,j,1);
        }else{
            if(find(i)!=find(j))cout<<-1<<et;
            else cout<<abs(rnk[i]-rnk[j])-1<<et;
        }
    }
}
```

CF 886 div4 H 也是带权并查集

附录

这里放上用到的文件头和main函数

```
#include<iostream>
#include<vector>
#include<cmath>
#include<bits/stdc++.h>
using namespace std;
```

```cpp
using ll = long long;
using ld = long double;
//不要用#define，使用using可以用ll格式转换，但是#define不行
#define et '\n'
//慎用register和inline
#define reg register
#define rep(i,a,b) for(int i=(a);i<=(b);i++)
#define rrep(i,a,b) for(int i=(a);i>=(b);i--)
#define cin(a) ll a;cin>>a
#define dcin(a) ld a;cin>>a
#define scin(a) string a;cin>>a
#define CA cout<<ans<<et
#define CY cout<<"YES"<<et
#define CN cout<<"NO"<<et
#define max(a,b) ((a>b)?(a):(b))
#define min(a,b) ((a<b)?(a):(b))
//#define PP(l,r,CK) *ranges::partition_point(ranges::iota_view((l),(r)+1),(CK))
string ANS[2]={"No\n","Yes\n"};
int M=1e9+7;
//inline ll MO(ll x){return (x%M+M)%M;}

void solve(){

}

int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    //pre();
    cin(t);while(t--)
    solve();
    return 0;
}
```