**并查集小结**

并查集大体分为三个：普通的并查集，带种类的并查集，扩展的并查集（主要是必须指定合并时的父子关系，或者统计一些数据，比如此集合内的元素数目。）

POJ-1182

经典的种类并查集

POJ-1308

用并查集来判断一棵树。。注意空树也是树，死人也是人。

POJ-1611

裸地水并查集

POJ-1703

种类并查集

POJ-1988

看上去似乎和种类并查集无关，但其实仔细想想，就是种类并查集。。。  
只不过是种类数目无穷大，通过合并，可以确定两个物品之间的种类差（即高度差）

POJ-2236

裸地并查集，小加一点计算几何

POJ-2492

裸地种类并查集

POJ-2524

又是裸地并查集

POJ-1456

常规思想是贪心+堆优化，用并查集确实很奇妙。。。下面的文章中有详细介绍。

POJ-1733

种类并查集，先要离散化一下，不影响结果。。。

HDU-3038

上一道题的扩展，也是种类并查集，种类无穷大。。。。

POJ-1417

种类并查集，然后需要背包原理来判断是否能唯一确定“好人”那一堆

POJ-2912

baidu的题，AC了，不过有点乱，有时间【【【再看看】】】

ZOJ-3261   NUAA-1087

逆向使用并查集就可以了。。。

POJ-1861  POJ-2560

Kruskal并查集

**另外这个文章很好：**

转自：<http://hi.baidu.com/czyuan%5Facm/blog/item/531c07afdc7d6fc57cd92ab1.html>

      继续数据结构的复习，本次的专题是：并查集。

      并查集，顾名思义，干的就是“并”和“查”两件事。很多与集合相关的操作都可以用并查集高效的解决。

       两个操作代码：  
       int Find(int x)  
       {  
          if (tree[x].parent != x)  
          {  
              tree[x].parent = Find(tree[x].parent);  
          }  
          return tree[x].parent;  
       }

       void Merge(int a, int b, int p, int q, int d)  
       {  
          if (tree[q].depth > tree[p].depth) tree[p].parent = q;  
          else  
          {  
              tree[q].parent = p;  
              if (tree[p].depth == tree[q].depth) tree[p].depth++;  
          }  
       }  
       其中Find()函数用了路径压缩优化，而Merge()函数用了启发式合并的优化(个人感觉有了路径压缩，启发式合并优化的效果并不明显，而经常因为题目和代码的限制，启发式合并会被我们省略)。

       提到并查集就不得不提并查集最经典的例子：食物链。  
**POJ 1182 食物链**  
       <http://acm.pku.edu.cn/JudgeOnline/problem?id=1182>  
       题目告诉有3种动物，互相吃与被吃，现在告诉你m句话，其中有真有假，叫你判断假的个数(如果前面没有与当前话冲突的，即认为其为真话)  
这题有几种做法，我以前的做法是每个集合(或者称为子树，说集合的编号相当于子树的根结点，一个概念)中的元素都各自分为A, B, C三类，在合并时更改根结点的种类，其他点相应更改偏移量。但这种方法公式很难推，特别是偏移量很容易计算错误。  
下面来介绍一种通用且易于理解的方法：  
首先，集合里的每个点我们都记录它与它这个集合(或者称为子树)的根结点的相对关系relation。0表示它与根结点为同类，1表示它吃根结点，2表示它被根结点吃。  
那么判断两个点a, b的关系，我们令p = Find(a), q = Find(b)，即p, q分别为a, b子树的根结点。  
       1. 如果p != q，说明a, b暂时没有关系，那么关于他们的判断都是正确的，然后合并这两个子树。这里是关键，如何合并两个子树使得合并后的新树能保证正确呢？这里我们规定只能p合并到q(刚才说过了，启发式合并的优化效果并不那么明显，如果我们用启发式合并，就要推出两个式子，而这个推式子是件比较累的活…所以一般我们都规定一个子树合到另一个子树)。那么合并后，p的relation肯定要改变，那么改成多少呢？这里的方法就是找规律，列出部分可能的情况，就差不多能推出式子了。这里式子为 : tree[p].relation = (tree[b].relation – tree[a].relation + 2 + d) % 3; 这里的d为判断语句中a, b的关系。还有个问题，我们是否需要遍历整个a子树并更新每个结点的状态呢？答案是不需要的，因为我们可以在Find()函数稍微修改，即结点x继承它的父亲(注意是前父亲，因为路径压缩后父亲就会改变)，即它会继承到p结点的改变，所以我们不需要每个都遍历过去更新。  
       2. 如果p = q，说明a, b之前已经有关系了。那么我们就判断语句是否是对的，同样找规律推出式子。即if ( (tree[b].relation + d + 2) % 3 != tree[a].relation ), 那么这句话就是错误的。  
       3. 再对Find()函数进行些修改，即在路径压缩前纪录前父亲是谁，然后路径压缩后，更新该点的状态(通过继承前父亲的状态，这时候前父亲的状态是已经更新的)。  
       核心的两个函数为：  
       int Find(int x)  
       {  
           int temp\_p;  
          if (tree[x].parent != x)  
          {  
              // 因为路径压缩，该结点的与根结点的关系要更新(因为前面合并时可能还没来得及更新).  
              temp\_p = tree[x].parent;  
              tree[x].parent = Find(tree[x].parent);  
              // x与根结点的关系更新(因为根结点变了)，此时的temp\_p为它原来子树的根结点.  
              tree[x].relation = (tree[x].relation + tree[temp\_p].relation) % 3;  
          }  
          return tree[x].parent;  
       }

       void Merge(int a, int b, int p, int q, int d)  
       {  
          // 公式是找规律推出来的.  
          tree[p].parent = q; // 这里的下标相同，都是tree[p].  
          tree[p].relation = (tree[b].relation – tree[a].relation + 2 + d) % 3;  
       }

       而这种纪录与根结点关系的方法，适用于几乎所有的并查集判断关系(至少我现在没遇到过不适用的情况…可能是自己做的还太少了…)，所以向大家强烈推荐～～

       搞定了食物链这题，基本POJ上大部分基础并查集题目就可以顺秒了，这里仅列个题目编号: **POJ 1308 1611 1703 1988 2236 2492 2524。**

       下面来讲解几道稍微提高点的题目:  
**POJ 1456 Supermarket**  
       <http://acm.pku.edu.cn/JudgeOnline/problem?id=1456>  
       这道题贪心的思想很明显，不过O(n^2)的复杂度明显不行，我们可以用堆进行优化，这里讲下并查集的优化方法(很巧妙)。我们把连续的被占用的区间看成一个集合(子树)，它的根结点为这个区间左边第一个未被占用的区间。  
先排序，然后每次判断Find(b[i])是否大于0，大于0说明左边还有未被占用的空间，则占用它，然后合并(b[i], Find(b[i]) – 1)即可。同样这里我们规定只能左边的子树合并到右边的子树(想想为什么～～)。

**POJ 1733 Parity game**  
       <http://acm.pku.edu.cn/JudgeOnline/problem?id=1733>  
       这题同样用类似食物链的思想。  
首先我们先离散化，因为原来的区间太大了(10^9)，我们可以根据问题数目离散成(10^4)。我们要理解，这里的离散化并不影响最终的结果，因为区间里1的奇偶个数与区间的大小无关(这句话有点奇怪，可以忽略…)，然后每次输入a, b，我们把b++，如果他俩在一个集合内，那么区间[a, b]里1的个数相当于b.relation ^ a.relation，判断对错即可。如果不在一个集合内，合并集合(这里我们规定根结点小的子树合并根结点大的，所以要根据不同情况推式子)，修改子树的根结点的状态，子树的其他结点状态通过Find()函数来更新。

**hdu 3038 How Many Answers Are Wrong**  
       <http://acm.hdu.edu.cn/showproblem.php?pid=3038>  
       上面那题的加强版，不需要离散化，因为区间的和与区间的大小有关(和上面的那句话对比下，同样可以忽略之…)，做法与上面那题差不多，只是式子变了，自己推推就搞定了。但这题还有个条件，就是每个点的值在[0, 100]之间，那么如果a, b不在一个子树内，我们就合并，但在合并之前还要判断合并后会不会使得区间的和不合法，如果会说明该合并是非法的，那么就不合并，同样认为该句话是错误的。

**POJ 1417 True Liars(难)**  
       <http://acm.pku.edu.cn/JudgeOnline/problem?id=1417>  
       并查集 + DP(或搜索)。  
       题目中告诉两种人，一种只说真话，一种只说假话。然后告诉m条语句，问是否能判断哪些人是只说真话的那类人。  
       其实并查集部分跟食物链还是相似，而且种类变少了一种，更容易了。我们可以通过并查集把有关系的一些人合并到一个集合内(具体方法参见食物链讲解)。  
       现在的问题转化为，有n个集合，每个集合都有a, b连个数字，现在要求n个集合中各跳出一个数(a或者b)，使得他们之和等于n1(说真话的人数)。而这个用dp可以很好的解决，用f[i][j]表示到第i个集合和为j个的情况数，我们还用过pre[i][j]记录当前选的是a还是b，用于后面判断状态。方程为f[i][j] = f[i – 1][j – a] + f[i – 1][j – b], j >= a, j >= b。如果最后f[n][n1] == 1说明是唯一的情况，输出该情况，否则输出 “no”(多解算no)  
       注意点 :  
       1. 这题的m, n1, n2都有可能出现0，可以特殊处理，也可以一起处理。  
       2. 按上面的dp写法，f[i][j]可能会很大，因为n可以达到三位数。其实我们关心的只是f[i][j] 等于0，等于1，大于1三种情况，所以当f[i][j] > 1时，我们都让它等于2即可。

**POJ 2912 Rochambeau(难)**  
       <http://acm.pku.edu.cn/JudgeOnline/problem?id=2912>  
       Baidu Star 2006 Preliminary的题目，感觉出的很好，在并查集题目中算是较难的了。其实这题跟食物链完全一个磨子，同样三类食物，同样的互相制约关系。所以食物链代码拿过来改都不需要改。但这题有个judge，他可以出任意手势。于是我们的做法是，枚举每个小孩为judge，判断他为judge时在第几句话出错err[i](即到第几句话能判断该小孩不是judge)。  
       1. 如果只有1个小孩是judge时全部语句都是正确的，说明该小孩是judge，那么判断的句子数即为其他小孩的err[i]的最大值。如果  
       2. 如果每个小孩的都不是judge(即都可以找到出错的语句)，那么就是impossible。  
       3. 多于1个小孩是judge时没有找到出错的语句，就是Can not determine。**ZOJ 3261 Connections in Galaxy War**        <http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemId=3563>  
**nuaa 1087 联通or不连通**  
        <http://acm.nuaa.edu.cn/acmhome/problemdetail.do?&method=showdetail&id=1087>  
        两题做法差不多，都是反过来的并查集题目，先对边集排序，然后把要删去的边从二分在边集中标记。然后并查集连接没有标记的边集，再按查询反向做就可。第一题合并结点时按照题目要求的优先级合并即可。  
　     
       这里介绍的并查集题目，主要都是处理些集合之间的关系(这是并查集的看家本领～～)，至于并查集还有个用处就在求最小生成树的Kruskal算法中，那个是图论中求最小生成树的问题(一般这个难点不在于并查集，它只是用于求最小生成树的一种方法)，就不在这里赘述了～～

czyuan原创，转载请注明出处。

**种类并查集报告**

POJ-1703    POJ-1182    POJ-2492都是这种题。

别人的报告，讲得很好

原始地址：<http://www.cppblog.com/tortoisewu/archive/2009/07/14/85501.html>

——————————————————————————

[poj 1182 解题报告](http://www.cppblog.com/tortoisewu/archive/2009/05/23/85501.html)

本来要先写搜索和DP的报告的，因为前面做的都是搜索和DP，正好昨天做了这道并查集的题目，所以就顺手写下。这道题也让我理解了好长时间，还是很有意义的，网上也没有很详细的解题报告。  
题目：  
<http://acm.pku.edu.cn/JudgeOnline/problem?id=1182>

因为我是按网上通用分类做的，所以做之前就知道用并查集做，不过这道题是并查集的深入应用，没有其它的那么直观。这里我采用的是和网上主流方法一样的方法，这里先贴出源码再深入解释下。

#include <iostream>  
using namespace std;  
struct point{  
    int parent;  
    int kind;  
} ufind[50010];  
void init(int n);  
int find(int index);  
void unions(int rootx, int rooty, int x, int y, int dkind);  
int main()  
{  
    int n,k,count=0,d,x,y;  
    scanf(“%d%d”,&n,&k);  
      
        init(n);  
        while(k–)  
        {  
            scanf(“%d%d%d”,&d,&x,&y);  
            if(x>n || y>n)  
                count++;  
            else if(d==2 && x==y)  
                    count++;  
            else  
            {  
                int rx=find(x);  
                int ry=find(y);  
                if(rx!=ry)  
                    unions(rx,ry,x,y,d-1);  
                else  
                {  
                    if(d==1 && ufind[x].kind!=ufind[y].kind)  
                        count++;  
                    if(d==2 && (ufind[x].kind-ufind[y].kind+3)%3!=1)  
                        count++;  
                }  
            }  
        }  
        printf(“%d\n”,count);  
      
      
    return 0;  
}  
void init(int n)  
{  
    for(int i=1;i<=n;i++)  
    {  
        ufind[i].parent=i;  
        ufind[i].kind=0;  
    }  
}  
int find(int index)  
{  
    int temp;  
    if(index==ufind[index].parent)  
        return index;  
    temp=ufind[index].parent;  
    ufind[index].parent=find(ufind[index].parent);  
    ufind[index].kind=(ufind[temp].kind+ufind[index].kind)%3;  
    return ufind[index].parent;  
}  
void unions(int rootx, int rooty, int x, int y, int dkind)  
{  
    ufind[rooty].parent=rootx;  
    ufind[rooty].kind=(-dkind+(ufind[x].kind-ufind[y].kind)+3)%3;  
}

1.这里并查集(ufind)的一个同类集合存放的是根据已有正确推断有关联的点，这里的关联包括了吃，被吃，同类三个关系；  
2.关系是通过kind来表示的，其值为0，1，2，如果ufind[1].kind==ufind[2].kind，说明1，2点同类；如果  
(ufind[1].kind-ufind[2].kind+3)%3==1，说明1吃2；  
3.并查集是按索引部分组织起来的，即同一类的点都有共同的根结点；  
4.并查集包括初始化(init)，查找(find)，合并(unions)操作，其中有很多关键点，我都在代码中用红色标记。下面逐一解释这些关键点：  
(1)ufind[i].kind=0;种类初始化为0，这个看似很简单，但它其实保证了并查集中每一个类的根结点的kind属性为0，这是后面两个关键式推导的基础；  
(2)ufind[rooty].kind=(-dkind+(ufind[x].kind-ufind[y].kind)+3)%3;这句出现在合并操作里面，这里要解释的是，在合并之前每个类的集合中所有父节点为根结点的点以及根结点，它们之间的关系都是正确的，合并之后只保证了合并前原两个集合的根结点之间的关系正确，即在新的合并后的集合中仍保证所有父节点为根结点的点以及根结点之间的关系正确。这样我们在做合并操作时，是通过三个关系推到出预合并的两个根结点(rootx,rooty)之间的正确关系的：x和rootx的关系，y和rooty的系，x和y的关系。这就是这个式子的由来，其中用到了前面说过的rootx和rooty为0的结论。  
(3)ufind[index].kind=(ufind[temp].kind+ufind[index].kind)%3;这句出现在查找操作里，作用是将待查找的点到它的根结点所经过的所有点进行两个操作，一是把它们的父节点都设为根结点；二是按照从离根结点最近的那个点开始到待查找的点的顺序把它们与根结点的关系设置成正确值(原先有可能是错误的，因为合并操作只保证了所有父节点为根结点的点以及根结点之间的关系正确)。这样之后这个集合中仍然保证了所有父节点为根结点的点以及根结点之间的关系正确，并且待考察的点的父节点为根结点。下面来解释下为什么要按照从离根结点最近的那个点开始到待查找的点的顺序，这也是这个式子为什么这么写的原因：假设1为根结点，Kind为0，其子节点为2，kind为k2，2的子节点为3，kind为k3；因为每次合并只合并根结点，所以3在1，2合并前的根结点一定是2，即若2的kind为0，则3和2的关系就正确了，但合并时2的kind加上了k2，保证了1与2的关系正确而并没有更新k3(这是因为并查集集合中无法从父节点找到子结点，所以等到查找时，要用到该点时再更新也不迟)，所以此时只要将(k2+k3)%3就可以得到正确的以1为基准的3的kind。接下来，k3的kind修正了，k4可以以k3为基础一样通过此方法修正，直到要查的结点，并把它们全部挂到根结点上。  
解释就到这里，我想理解时只要画个图就能容易理解些，即以index和kind为结点做出集合的树状图。这里恰巧是3个关系，若是4个关系我想就要更新并查集单位集合的组织形式了，即要可以从根结点遍历全集和。