12. 并查集

并查集常用于实现集合的合并和查询某元素属于哪个集合.

并查集的常用扩展:

①记录每个集合的大小:因集合的大小只与该集合本身有关,故将其绑定到根节点上.

②记录每个节点到根节点的距离,常用于维护同一集合中的多个类别,如根据节点与根节点的距离来判断其与根节点的关系.事实上维护同一集合中的k个类别还可用扩展域的并查集实现,但这样每一步的时间复杂度是$O(k)$,而用带权的并查集的时间复杂度与k无关.k不大时可认为两者时间复杂度相同.

12.1 并查集

12.1.1 合并集合

题意

设有编号分别为$1\sim n$的n $(1\le n\le 1e5)$个数, 初始时每个数各自在一个集合中. 现进行m $(1\le m\le 1e5)$次操作, 操作命令有如下两种:

①$M\ a\ b$, 表示将编号为a和b的两数所在的集合合并. 若两数已在同一集合中, 则忽略本操作.

②$Q\ a\ b$, 表示询问编号为a和b的两数是否在同一集合中, 若是则输出"Yes", 否则输出"No".

思路

朴素做法: 开一个数组存每个元素属于哪个集合, 操作①时间复杂度$O(1)$, 但操作②时间复杂度$O(m)$, 无法接受.

并查集可在几乎$O(1)$的时间内完成集合的合并和查询操作.

并查集用树存每个集合, 每个集合的标号是其根节点的编号, 开一个数组$fa[]$存每个节点的父亲节点, 约定根节点的父亲节点是它本身.

显然一个节点是根节点的充要条件是: 其父节点是它本身.

①操作, 将一个集合的根节点置为另一集合的根节点的儿子节点.

②操作, 依次查询每个节点的父亲节点, 最终将查询到根节点, 根节点的编号即其所在的集合.

但②操作因需依次查询每个节点的父亲节点, 则其时间复杂度与树的高度成正比. 可采用路径压缩优化, 即让某个元素到根节点的路径上的所有节点都直接指向所在集合的根节点, 这样几乎是$O(1)$的时间复杂度.

事实上并查集的路径压缩、按秩合并各自优化后的时间复杂度都为$O(\log n)$, 若两者同时使用, 理论时间复杂度为$O(\alpha(n))$, 在$n\le 2^{2^{2^{1987}}}$时$\alpha(n)\le 5$, 故可认为是线性的. 实际应用中一般只写路径压缩优化.

代码

```cpp
struct DSU {
    int n;  // 元素个数
    vector<int> fa;

    DSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    bool check(int x, int y) {
        return find(x) == find(y);
    }

    bool merge(int x, int y) {
        if ((x = find(x)) == (y = find(y))) return false;

        fa[x] = y;
        return true;
    }
};

void solve() {
    int n; cin >> n;

    DSU dsu(n);
    CaseT {
        char op; int a, b; cin >> op >> a >> b;

        if (op == 'M') dsu.merge(a, b);
        else cout << (dsu.check(a, b) ? "Yes" : "No") << endl;
    }
}

int main() {
    solve();
}
```

12.1.2 连通块点的数量

题意

给定一个包含n $(1 \leq n \leq 1e5)$个点, 编号分别为$1 \sim n$的无向图, 初始时图中无边. 现进行m $(1 \leq m \leq 1e5)$个操作, 操作命令有如下三种:

①C a b, 表示在点a和b间连一条边, a和b可能相等.

②$Q1$ ab, 询问点a和b是否在同一个联通块中, a和b可能相等. 若是则输出"Yes", 否则输出"No".

③$Q2$ a, 输出点a所在连通块点的数量.

思路

用集合维护连通块.

开一个数组$cnt[]$记录每个集合内点的数量, 只需保证每个集合的根节点的cnt有意义. 合并时, 一个集合的$cnt+=$另一集合的cnt.

代码

```cpp
struct DSU {
    int n;  // 元素个数
    vector<int> fa;
    vector<int> siz;  // 集合元素大小

    DSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        siz = vector<int>(n + 1, 1);
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    bool check(int x, int y) {
        return find(x) == find(y);
    }

    bool merge(int x, int y) {
        if ((x = find(x)) == (y = find(y))) return false;

        fa[x] = y, siz[y] += siz[x];
        return true;
    }

    int getSize(int x) {
        return siz[find(x)];
    }
};

void solve() {
    int n; cin >> n;

    DSU dsu(n);
    CaseT {
        string op; cin >> op;

        if (op == "C") {
            int a, b; cin >> a >> b;
            dsu.merge(a, b);
        }
        else if (op == "Q1") {
            int a, b; cin >> a >> b;
            cout << (dsu.check(a, b) ? "Yes" : "No") << endl;
        }
        else {
            int a; cin >> a;
            cout << dsu.getSize(a) << endl;
```

```
        }
    }
}

int main() {
    solve();
}
```

12.1.3 食物链

题意

有三类动物A、B、C,它们的食物链关系构成环:A吃B, B吃C, C吃A.

现有n $(1 \leq n \leq 5e4)$个动物,编号分别为$1 \sim n$,每个动物都是A、B、C中的一种,但未知是哪一种.用如下两种说法描述这n个动物的食物链关系:

①$1$ x y,表示编号x和y的动物是同类.

②$2$ x y,表示编号x的动物吃编号y的动物.

某人用上述两种说法依次说出k $(1 \leq k \leq 1e5)$句话,其中有真有假.当一句话满足下列条件之一时为假话,否则为真话:①当前的话与前面的某句真话冲突;②当前的话中x或$y > n$;③当前的话表示x吃x.

给定n和k,输出假话的总数.

思路

A吃B记作$A \to B$.注意到若$3 \to 2 \to 1$,则$1 \to 3$,故若给出两动物的关系,无论它们是什么关系都将它们放在同一集合中,则可推断出集合中所有两两关系,这可通过记录每个节点与根节点的关系来实现.为描述方便,称根节点为第0代,吃根节点的为第1代,吃第1代的为第2代,第2代被根节点吃.

因三种动物食物链成环状,可用模3意义下每个节点到根节点的距离来表示每个节点与根节点的关系:

①若某节点到根节点的距离为1,则该节点吃根节点.

②若某节点到根节点的距离为2,因距离为1的节点可吃根节点,则距离为2的节点被根节点吃.

③若某节点到根节点的距离为3,则该节点与根节点是同类.

用并查集维护一个数组$dis[]$表示每个节点到根节点的距离,各距离在模3意义下,上述的①、②、③分别对应余1、2、0的情况,根节点到根节点的距离为0.若两节点在同一集合中,可根据它们离根节点的距离之差模3的余数来推断它们的关系.

实现时,$dis[]$数组记录每个节点到其父亲节点的距离,路径压缩时将该距离加上其父亲节点到根节点的距离,注意更新距离之后$fa[]$数组对应位置存的不是该节点的父亲节点而是根节点,故需在更新距离前先存下该节点的父亲节点.初始时每个节点都是根节点,故$dis[]$初始化为0.

①操作:

1)若两动物在同一集合中,则判断它们是否是同类,即它们到根节点的距离是否满足$dis[x] - dis[y] \equiv 0 \pmod 3$.

2)若它们不在一个集合中,则将它们所在的集合合并.合并时,需给连接两根节点的边定义一个距离.不妨设将x所在集合合并到y所在集合,则$fa[x]$与$fa[y]$间边的距离s满足$dis[x] + s - dis[y] \equiv 0 \pmod 3$,故$s := dis[y] - dis[x]$.

②操作:

1)若两动物在同一集合中,因x吃y,则x离根节点的距离比y多1,即$dis[x] - 1 - dis[y] \equiv 0 \pmod 3$.

2)若它们不在一个集合中,则将它们所在的集合合并.合并时,需给连接两根节点的边定义一个距离.不妨设将x所在集合合并到y所在集合,则$fa[x]$与$fa[y]$间边的距离s满足$dis[x]+s-1-dis[y]\equiv 0\pmod 3$,故$s:=dis[y]+1-dis[x]$.

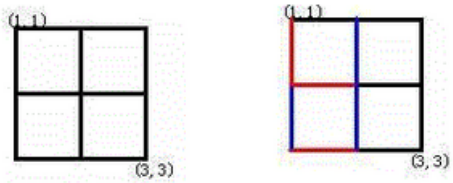
代码

```cpp
const int MAXN = 1e5 + 5;
int fa[MAXN];  // 存每个节点的父亲节点
int dis[MAXN] = { 0 };  // 存每个节点与根节点的距离

void init(int n) {
  for (int i = 1; i <= n; i++) fa[i] = i;  // 将每个节点的父亲节点初始化为自己
}

int find(int x) {  // 求x的祖先节点
  if (fa[x] != x) {  // x非根节点
    int tmp = find(fa[x]);
    dis[x] += dis[fa[x]];  // 将x到其父亲节点的距离更新为到根节点的距离
    fa[x] = tmp;  // 注意顺序
  }

  return fa[x];
}

int main() {
  init(n);

  int ans = 0;  // 假话数量
  while (m--) {
    int op, x, y; cin >> op >> x >> y;

    if (x > n || y > n) ans++;
    else {
      int tmpx = find(x), tmpy = find(y);
      switch (op) {
        case 1:  // x和y是同类
          if (tmpx == tmpy && (dis[x] - dis[y]) % 3) ans++;  // x和y在同一集合中但不是同类
          else if (tmpx != tmpy) {  // 若x和y不在同一集合,将它们合并
            fa[tmpx] = tmpy;
            dis[tmpx] = dis[y] - dis[x];  // 更新两根节点连边的距离
          }
          break;
        case 2:  // x吃y
          if (tmpx == tmpy && (dis[x] - 1 - dis[y]) % 3) ans++;  // x和y在同一集合中但不是同类
          else if (tmpx != tmpy) {  // 若x和y不在同一集合,将它们合并
            fa[tmpx] = tmpy;
            dis[tmpx] = dis[y] + 1 - dis[x];  // 更新两根节点连边的距离
          }
          break;
      }
    }
  }
  cout << ans;
}
```

12.1.4 格子游戏

题意

两人在玩游戏:在一个$n \times n$的点阵上,两人轮流在相邻的两点间连红边或蓝边,直至围成一个封闭的圈(面积不必为1),最后封圈的人胜.下图以$n = 3$为例:



输入两个整数n $(1 \leq n \leq 200)$和m $(1 \leq m \leq 2.4e4)$分别表示点阵的大小和一共画了m条线,接下来m行每行先有两个数字(x, y)表示画线起点的坐标,接着是一个用空格隔开的字符,若为D,则向下连一条边;若为R,则向右连一条边.输出第几步时游戏结束,若m步后未结束,输出"draw".

数据保证无重复的边且操作合法.

思路

可用Tarjan算法求强连通分量,但没必要.

若两点已再同一连通块中,则再将它们连通会出现环,即连边后会不会成环等价于连边前两端点是否在同一集合中.

并查集一般用一维的坐标,故要先把二维坐标转化为一维坐标.若坐标(x, y)中x和y都从0开始,则可用$x * n + y$映射到一维坐标.

代码

```
const int MAXN = 4e+5;
int n;  // 点阵大小
int fa[MAXN];  // 并查集的fa数组

int get(int x, int y) {  // 将二维坐标映射到一维坐标,x和y从0开始
  return x * n + y;
}

int find(int x) {
  return fa[x] == x ? x : fa[x] = find(fa[x]);
}

int main() {
  for (int i = 0; i < n * n; i++) fa[i] = i;  // 初始化并查集,下标从0开始

  int ans = 0;
  for (int i = 1; i <= m; i++) {
    int x, y; char D; cin >> x >> y >> D;

    x--, y--;  // 让x和y从0开始
    int a = get(x, y), b = D == 'D' ? get(x + 1, y) : get(x, y + 1);

    int tmpa = find(a), tmpb = find(b);
    if (tmpa == tmpb) {
      ans = i;
      break;
```

```
    }
    fa[tmpa] = tmpb;
  }

  if (!ans)cout << "draw";
  else cout << ans;
}
```

12.1.5 搭配购买

题意

有编号分别为$1\sim n$的n $(1\leq n\leq 1e4)$个物品,每个物品的价格为c_i $(1\leq c_i\leq 5000)$,价值为d_i $(1\leq d_i\leq 100)$.物品中有m $(1\leq 5000)$个搭配,买一个物品则与其搭配的所有物品都要买.某人有w $(1\leq w\leq 1e4)$的钱,输出他能买到的物品的最大价值.

m个搭配用m行输入描述,每行包含两个整数u_i和v_i $(1\leq u_i,v_i\leq n)$,表示买u_i必须买v_i,同理买v_i必须买u_i.

思路

因搭配是双向的,所以是无向边.将每个连通块看成一个物品,则每个物品只有买和不买两种情况,是$0-1$背包问题.

用并查集找出所有连通块后做一次$0-1$背包即可.

合并集合时,维护这一组物品的总价值和总体积,将其绑定在根节点,dp时只用根节点.

合并后最多有$1e4$个物品,背包的总体积w最大为$1e4$,则最坏需计算$1e8$次.但并查集和$0-1$背包的常数很小,故1 s可过.

代码

```
const int MAXN = 1e5 + 5;
int n, m, vol;  // 物品数、搭配树、背包体积
int v[MAXN], w[MAXN];  // 物品的体积和价值
int fa[MAXN];  // 并查集的fa数组
int dp[MAXN];  // 0-1背包的dp数组

int find(int x) {
  return fa[x] == x ? x : fa[x] = find(fa[x]);
}

int main() {
  cin >> n >> m >> vol;
  for (int i = 1; i <= n; i++) cin >> v[i] >> w[i];  // 物品的体积和价值

  for (int i = 1; i <= n; i++) fa[i] = i;  // 初始化并查集

  while (m--) {
    int a, b; cin >> a >> b;

    int tmpa = find(a), tmpb = find(b);
    if (tmpa != tmpb) {  // 未在同一连通块中才合并
      fa[tmpa] = tmpb;  // 将a合并到b中
      v[tmpb] += v[tmpa];
```

```
      w[tmpb] += w[tmpa];
    }
  }

  for (int i = 1; i <= n; i++) {  // 0-1背包
    if (fa[i] == i) {  // 只对根节点dp
      for(int j=vol;j>=v[i];j--)
        dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
    }
  }
  cout << dp[vol];
}
```

12.1.6 程序自动分析

题意

设$x_1, x_2, \cdots$为程序的变量.给定n个形如$x_i = x_j$或$x_i \neq x_j$的约束条件,判定上述约束条件是否可同时成立.

输入一个正整数t,表示需判定的问题的个数,问题间相互独立.

每个问题包含若干行:第一行包含一个正整数n $(1 \leq n \leq 1e5)$,表示该问题中约束条件的个数,接下来n行每行3个整数i, j, e $(1 \leq i, j \leq 1e9)$描述一个约束条件,相邻整数用空格隔开.$e = 1$表示约束条件为$x_i = x_j$; $e = 0$表示约束条件为$x_i \neq x_j$.

对每个问题输出一行,若约束条件可同时成立则输出"YES",否则输出"NO".

思路

虽$1 \leq i, j \leq 1e9$,但最多有$1 \leq n \leq 1e5$个约束条件,每个约束条件有两个变量,则最多有$2e5$个变量,故先离散化.

显然约束条件的顺序与结论无关,则可先考虑所有的相等的约束条件,这样必然不出现矛盾,若$x_i = x_j$,则合并它们所在的集合.再考虑所有不等的约束条件,若$x_i = x_j$,但x_i和x_j在同一集合中,则矛盾.采用离线做法.

总时间复杂度$O(\log n)$.

代码

```
const int MAXN = 2e5 + 5;
int n;  // 约束条件个数
int fa[MAXN];  // 并查集的fa数组
unordered_map<int, int> ha;  // 哈希表
struct Constraint { int x, y, e; }cons[MAXN];

int get(int x) {  // 离散化
  if (!ha.count(x)) ha[x] = ++n;
  return ha[x];
}

int find(int x) {
  return fa[x] == x ? x : fa[x] = find(fa[x]);
}

int main() {
  while (T--) {
    n = 0;
```

```
    ha.clear();  // 清空

    int m; cin >> m;
    for (int i = 0; i < m; i++) {  // 把所有的约束条件存起来
      int x, y, e; cin >> x >> y >> e;
      cons[i] = { get(x),get(y),e };
    }

    for (int i = 1; i <= n; i++) fa[i] = i;  // 初始化fa数组

    for (int i = 0; i < m; i++) {
      if (!cons[i].e) continue;  // 先考虑相等约束
      int tmpx = find(cons[i].x), tmpy = find(cons[i].y);
      fa[tmpx] = tmpy;
    }

    bool flag = false;  // 记录是否有矛盾
    for (int i = 0; i < m; i++) {
      if (cons[i].e) continue;  // 检查不等约束
      int tmpx = find(cons[i].x), tmpy = find(cons[i].y);
      if (tmpx == tmpy) {
        flag = true;
        break;
      }
    }

    if (flag) cout << "NO" << endl;
    else cout << "YES" << endl;
  }
}
```

12.1.7 奇偶游戏

题意

A、B两人在玩游戏.A先写一个由0和1组成的长度为$n\ \ (1 \leq n \leq 1e9)$的序列S,然后B向A提出$m\ \ (1 \leq m \leq 5000)$个问题,每个问题包含两个数$l, r$,A回答$S[l \cdots r]$中有奇数个1还是偶数个1,若有奇数个则回答为"odd",否则回答为"even",A可能会撒谎.检查这m个回答,输出至少多少个回答后可确定A一定在撒谎,即求出一个最小的$k\ s.t.\ S$满足第$1 \sim k$个回答,但不满足第$1 \sim (k+1)$个回答.若S满足所有回答,则输出问题的总数量.

思路I

$1 \leq n \leq 1e9$,但最多只有5000个问题,故离散化,每个问题包含两个数,故最多用到$1e4$个数.本题用在线离散化,不用把所有问题存下来.

维护前缀和$pre[]$,则$s[l \cdots r]$有奇数个1即$pre[r] - pre[l-1]$为奇数,亦即$pre[r]$与$pre[l-1]$奇偶性不同,同理$s[l \cdots r]$有偶数个1即$pre[r]$与$pre[l-1]$奇偶性相同.

下证前缀和的奇偶性无矛盾等价于该问题无矛盾:对每个奇偶性无矛盾的前缀和序列$pre[]$,构造01序列$a_i = |pre[i] - pre[i-1]|$,即$pre[i]$与$pre[i-1]$同奇偶时$a_i = 0$,不同奇偶时$a_i = 1$.故等价.

对每个节点,维护其与其父亲节点的距离$dis[]$,它在模2意义下表示该节点与其父亲节点的关系:0表示同奇偶,1表示不同奇偶.定义节点到其根节点的距离为它到根节点的路径的边权和,则可根据边权和的奇偶性判断该节点与根节点是否同类.已知同一集合内节点x到根节点的距离$dis[x]$和节点y到根节点的距离$dis[y]$,若$dis[x]+dis[y]$为奇数,则节点x和y不同类,否则同类.显然若已知节点x和节点y的关系、节点y和节点z的关系,则可推断出节点x与节点z的关系.

$dis[x]\ \hat{}\ dis[y]=0$表示$dis[x]$和$dis[y]$奇偶性相同,为同类;$dis[x]\ \hat{}\ dis[y]\neq 0$表示$dis[x]$和$dis[y]$奇偶性不同,为不同类.

合并同类节点x和节点y的两集合时,设两根节点间的边权为s,则$dis[x]+s+dis[y]$为偶数,进而$s=偶-dis[x]-dis[y]=dis[x]\ \hat{}\ dis[y]$,同理合并不同类的节点$x$和节点$y$的两集合时,两根节点间的边权$s=dis[x]\ \hat{}\ dis[y]\ \hat{}\ 1$.

下面代码中$dis[]$统一用异或,也可同一用加,但if ((dis[a] ^ dis[b]) != t)应改为if (((dis[a] + dis[b]) % 2 + 2) % 2 != t),即保证取模后是正数;或在find()中将dis[x] ^= dis[fa[x]];改为dis[x] += dis[fa[x]] % 2,防止溢出.

代码I:带边权并查集

```cpp
const int MAXN = 1e4 + 5;
int n, m;  // 序列长度
int fa[MAXN];  // 并查集的fa数组
int dis[MAXN];  // 记录每个节点到其父亲节点的距离
unordered_map<int, int> ha;  // 哈希表

int get(int x) {  // 离散化
  if (!ha.count(x)) ha[x] = ++n;
  return ha[x];
}

int find(int x) {
  if (fa[x] != x) {  // 不是根节点
    int root = find(fa[x]);
    dis[x] ^= dis[fa[x]];
    fa[x] = root;
  }
  return fa[x];
}

int main() {
  cin >> n >> m;

  n = 0;
  for (int i = 0; i < MAXN; i++) fa[i] = i;  // 初始化fa数组

  int ans = m;
  for (int i = 1; i <= m; i++) {
    int a, b; string type; cin >> a >> b >> type;
    a = get(a - 1), b = get(b);  // s[l...r]对应前缀和pre[r]-pre[l-1]

    int t = type == "odd" ? 1 : 0;

    int tmpa = find(a), tmpb = find(b);
    if (tmpa == tmpb) {
      if ((dis[a] ^ dis[b]) != t) {
        ans = i - 1;
        break;
      }
```

```
    }
    else {
      fa[tmpa] = tmpb;
      dis[tmpa] = dis[a] ^ dis[b] ^ t;
    }
  }
  cout << ans;
}
```

思路II

将每个x拆成x和$x+n$两个条件,前者,后者表示x是偶数,后者表示x是奇数.注意节点要开两倍.

(1)若节点x和节点y同类,即①x为奇,y为奇;②x为偶,y为偶,则合并$x+n$和$y+n$、x和y.

(2)若节点x和节点y不同类,即①x为奇,y为偶;②x为偶,y为奇,则合并$x+n$和y、x和$y+n$.

注意下面的代码中n是动态变化的,故要加一个固定的$base$.

代码II:扩展域并查集

```
const int MAXN = 2e4 + 5, base = MAXN / 2;
int n, m;  // 序列长度
int fa[MAXN];  // 并查集的fa数组
unordered_map<int, int> ha;  // 哈希表

int get(int x) {  // 离散化
  if (!ha.count(x)) ha[x] = ++n;
  return ha[x];
}

int find(int x) {
  return fa[x] == x ? x : fa[x] = find(fa[x]);
}

void merge(int x, int y) {
  fa[find(x)] = find(y);
}

int main() {
  cin >> n >> m;

  n = 0;
  for (int i = 0; i < MAXN; i++) fa[i] = i;  // 初始化fa数组

  int ans = m;
  for (int i = 1; i <= m; i++) {
    int a, b; string type; cin >> a >> b >> type;
    a = get(a - 1), b = get(b);  // s[l...r]对应前缀和pre[r]-pre[l-1]

    if (type == "even") {
      if (find(a + base) == find(b)) {
        ans = i - 1;
        break;
      }

      merge(a, b);
```

```
      merge(a + base, b + base);
    }
    else {
      if (find(a) == find(b)) {
        ans = i - 1;
        break;
      }

      merge(a + base, b);
      merge(a, b + base);
    }
  }
  cout << ans;
}
```

12.1.8 银河英雄传说 (2 s)

题意

有一个划分为n $(1 \leq n \leq 3e4)$列的战场,各列编号依次为$1 \sim n$.有编号为$1 \sim n$的n艘战舰,其中第i号战舰位于第i列.现有T $(1 \leq T \leq 5e5)$条指令,指令的形式有如下两种:

①$M\ i\ j$,表示让第i号战舰所在列的所有战舰保持原有顺序接在第j号战舰所在列的尾部.

②$C\ i\ j$,询问第i号战舰和第j号战舰是否在同一列,若是,输出它们间隔了多少艘战舰;否则输出-1.

思路

判断是否在同一列只需判断它们是否在同一集合中.

若记录任两艘战舰间的关系,则时间复杂度$O(n^2)$,会T.只需维护当前战舰到排头战舰的距离$dis[]$,即集合中节点到根节点的距离,则在同一列的战舰x和y的距离为$\begin{cases} |dis[x]-dis[y]|-1, x \neq y \\ 0, x=y \end{cases}$,可合在一起写作$\max\{0, |dis[x]-dis[y]|-1\}$.

合并集合时,让每列的排头作根节点.将A列接到B列后面时,A列的每个节点到根节点的距离加上$length[B]$.因每个节点存它与其父节点的距离,若定义节点到根节点的距离为它到根节点的路径的边权和,则只需让A的排头离根节点的距离$dis[pa]=length[B]$.

初始化时,因未告知战舰数量,则将$1 \sim 3e4$都初始化.

代码

```
const int MAXN = 3e4 + 5;
int fa[MAXN];  // 并查集的fa数组
int length[MAXN];  // 记录每一列的长度
int dis[MAXN];  // 记录每个节点到其父亲节点的距离

int find(int x) {
  if (fa[x] != x) {  // 若不是根节点
    int root = find(fa[x]);
    dis[x] += dis[fa[x]];
    fa[x] = root;
  }
```

```
12
13    return fa[x];
14  }
15
16  int main() {
17    for (int i = 1; i < MAXN; i++) {  // 初始化fa和length数组
18      fa[i] = i;
19      length[i] = 1;
20    }
21
22    int m; cin >> m;
23    while (m--) {
24      int a, b; char op; cin >> op >> a >> b ;
25
26      int tmpa = find(a), tmpb = find(b);
27      if (op == 'M') {
28        dis[tmpa] = length[tmpb];
29        length[tmpb] += length[tmpa];
30        fa[tmpa] = tmpb;
31      }
32      else {
33        if (tmpa != tmpb) cout << "-1" << endl;
34        else cout << max(0, abs(dis[a] - dis[b]) - 1) << endl;
35      }
36    }
37  }
```

12.2.9 City

题意

有一张包含n个节点和m条边的无向图,每条边有一个权值.每进行一次伤害为x的攻击时,所有边权$<x$的边都会切断.问每次攻击后还有多少对节点连通.

有t $(1 \le t \le 10)$组测试数据.每组测试数据第一行输入三个整数n, m, q $(2 \le n \le 1e5, 1 \le m, q \le 2e5)$,分别表示节点数、边数、询问数.接下来$m$行每行输入三个整数$x, y, k$ $(1 \le x, y \le n, 1 \le k \le 1e9)$,表示节点$x$与$y$间存在权值为$k$的无向边.接下来$q$行每行包含一个整数$x$ $(1 \le x \le 1e9)$,表示询问进行一次伤害为x的攻击时有多少对节点连通.

思路

显然可用并查集维护连通块,但并查集做删除操作不方便.考虑离线,先将所有边按边权降序排列,将所有询问按x降序排列,则全程只需考虑加边.

含n个节点的连通块对答案的贡献为$\frac{n(n-1)}{2}$.合并包含x、y个节点的两连通块时,答案的净增量为

$$\frac{(x+y)(x+y)-1}{2} - \frac{x(x-1)}{2} - \frac{y(y-1)}{2} = xy.$$

代码

```
1  const int MAXN = 2e5 + 5;
2  int n, m, q;  // 节点数、边数、询问数
3  struct Edge {
4    int u, v, w;
5
6    bool operator<(const Edge& B) { return w > B.w; }
```

```
}edges[MAXN];
pii ques[MAXN];  // 询问的x、编号
int fa[MAXN], siz[MAXN];  // 并查集的fa[]数组、集合的大小
ll ans[MAXN];  // 每个询问的答案

int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

void solve() {
  cin >> n >> m >> q;
  for (int i = 0; i < m; i++) {
    int x, y, z; cin >> x >> y >> z;
    edges[i] = { x,y,z };
  }
  for (int i = 0; i < q; i++) {
    int x; cin >> x;
    ques[i] = { x,i };
  }

  for (int i = 1; i <= n; i++) fa[i] = i, siz[i] = 1;  // 初始化

  sort(edges, edges + m);
  sort(ques, ques + q, greater<pii>());

  ll res = 0;  // 每个询问的答案
  int idx = 0;  // 当前用到的边的下标
  for (int i = 0; i < q; i++) {  // 枚举询问
    while (idx < m && edges[idx].w >= ques[i].first) {
      int u = find(edges[idx].u), v = find(edges[idx].v);
      if (u != v) {
        res += (ll)siz[u] * siz[v];
        fa[v] = u;
        siz[u] += siz[v];
      }
      idx++;
    }
    ans[ques[i].second] = res;
  }

  for (int i = 0; i < q; i++) cout << ans[i] << endl;
}

int main() {
  CaseT  // 单测时注释掉该行
  solve();
}
```

12.2.10 cocktail with swap

原题指路:https://codeforces.com/gym/103115/problem/D

题意

给定一个长度为n的字符串s,下标从1开始.现有操作:选择下标$i,j\in[1,n]\ s.t.\ l\le|i-j|\le r$,交换$s_i$和$s_j$.求若干次(可能为零次)操作后能得到的字典序最小的字符串.

第一行输入三个整数$n,l,r\ \ (2\le n\le 1e5,1\le l\le r<n)$.

思路

用一个并查集$dsu2$维护每个起点开始的区间中能动的下标,枚举每个起点i,将以其开始的区间中能动的下标j对应的集合合并.朴素做法时间复杂度$O(n^2)$,会TLE.

考虑优化,注意到可用另一个并查集$dsu1$维护指针j下一步跳到的位置,保证每次j跳到下一段的开头,时间复杂度$O(n)$.

代码

```cpp
const int MAXN = 1e5 + 5;
int n, l, r;
string s;

struct DSU {
  int n;  // 元素个数
  int fa[MAXN];  // fa[]数组
  int siz[MAXN];  // 集合大小

  void init() {  // 初始化fa[]、siz[]
    for (int i = 1; i <= n; i++) fa[i] = i, siz[i] = 1;
  }

  int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

  bool merge(int x, int y) {  // 返回合并是否成功,即初始时是否不在同一集合中
    x = find(x), y = find(y);
    if (x != y) {
      if (siz[x] > siz[y]) swap(x, y);  // 保证x所在的集合小

      fa[x] = y;
      siz[y] += siz[x];
      return true;
    }
    else return false;
  }
}dsu1, dsu2;

void solve() {
  cin >> n >> l >> r >> s;

  dsu1.n = n; dsu1.init();  // 维护维护j跳的位置
  dsu2.n = n; dsu2.init();  // 维护每个起点开始的区间中能动的下标

  vb vis(n);
  for (int i = 0; i < n; i++) {  // 枚举起点
    for (int j = i + l; j <= i + r && j < n; j = dsu1.find(j) + 1) {  // j跳到下一段的开头
      dsu2.fa[dsu2.find(j)] = dsu2.fa[dsu2.find(i)];
      vis[j] = true;

      if (vis[j - 1]) dsu1.fa[dsu1.find(j - 1)] = dsu1.fa[dsu1.find(j)];
    }
```

```cpp
  }

  // 预处理每个起点开始的区间中出能动的字符的下标
  vector<vi> son(n);
  for (int i = 0; i < n; i++) son[dsu2.find(i)].push_back(i);

  for (int i = 0; i < n; i++) {
    if (son[i].size()) {
      // 预处理出能动的字符
      string t;
      for (auto j : son[i]) t.push_back(s[j]);
      sort(all(t));

      // 结果放回原串
      int idx = 0;
      for (auto j : son[i]) s[j] = t[idx++];
    }
  }
  cout << s;
}

int main() {
  solve();
}
```

12.2.11 Mocha and Diana (Hard Version)

原题指路:https://codeforces.com/problemset/problem/1559/D2

题意 (2 s)

有分别包含编号$1 \sim n$的n个节点的两个森林.现有操作:选择一对节点(u, v) $(1 \leq u, v \leq n, u \neq v)$,在两森林中分别加边$(u, v)$,使得加边后的图仍为森林,即不出现环.求最大操作次数,输出任一组合法方案.

第一行输入三个整数n, m_1, m_2 $(1 \leq n \leq 1e5, 0 \leq m_1, m_2 < n)$,分别表示节点数、初始时第一个森林的边数、初始时第二森林的边数.接下来m_1行每行输入两个整数u, v $(1 \leq u, v \leq n, u \neq v)$,表示初始时第一个森林有边$(u, v)$.接下来$m_2$行每行输入两个整数$u, v$ $(1 \leq u, v \leq n, u \neq v)$,表示初始时第二个森林有边$(u, v)$.

第一行输出一个整数h,表示最大操作次数.接下来h行每行输入两个整数u, v $(1 \leq u, v \leq n, u \neq v)$,表示一次操作.

思路

称两个森林分别为森林A和森林B,其中包含的树数分别为cnt_A和cnt_B.

[**定理**] 最终会将某个森林变为一棵树.

[**证**] 不妨设$cnt_A \geq cnt_B$.

(1)若$cnt_B = 1$,此时无法操作,且森林B分别包含一棵树.

(2)若$cnt_B \geq 2$,则$cnt_A \geq 2$.

取森林B中属于不同树的两节点u, v,下证至少存在一种取法 $s.t.$ u与v在森林A中属于不同树.

①$cnt_B \geq 3$时,取森林B中属于不同树的三节点u, v, w,则它们在A中两两连通,即有环,矛盾.

②$cnt_B = 2$时,若结论不成立,设森林B中的两棵树分别包含a、b个节点.

(i)$n = 2$时,显然最终森林A和森林B中至少有一个森林只包含一棵树,结论成立.

(ii)下证$n \geq 3$时,森林B中的属于同一棵树的节点在森林A中不直接连通.

若不然,则$\exists$森林B中属于同一棵树的两节点$u, v\ s.t.\ u$和v在森林A中直接连通.

考察森林B中属于另一棵树的节点w,则u, v, w在森林A中属于同一棵树.

考察森林B中所有的a个节点和b个节点知:森林A只包含一棵树,与$cnt_A \geq 2$矛盾.

这表明:森林A是左侧包含a个节点$u_1, \cdots, u_a$、右侧包含b个节点$v_1, \cdots, v_b$的完全二分图,

进而森林A中每个节点可能有多个前驱节点,与森林A中只包含树矛盾.

综上,对森林B中属于不同树的两节点u, v,至少存在一种取法 $s.t.\ u$与v在森林A中属于不同树,

则每次取森林B中合法的属于不同树的两节点,操作后可使得森林B包含的树数-1,最终森林B变为一棵树.

暴力枚举所有连边方式并检查是否合法,时间复杂度$O(n^2)$,会TLE.

考虑优化,类似于图论中建边优化边数的思路,考虑取一个中间节点$center$,让森林A和森林B中的其他所有节点向$center$尝试连边,设操作后森林A和森林B中与$center$不连通的节点集分别为L和R,则$L \bigcap R = \varnothing$,否则$L \bigcap R$中的节点还能与$center$连边.

考察节点$l \in L, r \in R$,则森林A中l与$center$不连通,r与$center$连通;森林B中l与$center$连通,r与$center$不连通,故任取一个节点为$center$后任意配对L和R中的节点l和r即可,时间复杂度$O(n)$.

代码

```cpp
struct DSU {
  int n;  // 元素个数
  vector<int> fa;

  DSU(int _n) :n(_n) {
    fa.resize(n + 1);
    iota(all(fa), 0);
  }

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }

  int& operator[](int x) {
    return fa[x];
  }

  bool check(int x, int y) {
    return find(x) == find(y);
  }

  bool merge(int x, int y) {
    if ((x = find(x)) == (y = find(y))) return false;

    fa[x] = y;
    return true;
  }
};
```

```
29
30  void solve() {
31    int n, m1, m2; cin >> n >> m1 >> m2;
32    DSU dsu1(n), dsu2(n);
33    while (m1--) {
34      int u, v; cin >> u >> v;
35      dsu1.merge(u, v);
36    }
37    while (m2--) {
38      int u, v; cin >> u >> v;
39      dsu2.merge(u, v);
40    }
41
42    vector<pair<int, int>> ans;
43    for (int i = 2; i <= n; i++) {  // 取节点1为center
44      if (!dsu1.check(1, i) && !dsu2.check(1, i)) {
45        ans.push_back({ 1,i });
46        dsu1.merge(1, i), dsu2.merge(1, i);
47      }
48    }
49
50    vector<int> L, R;  // 森林A、B中与center不连通的节点集
51    for (int i = 2; i <= n; i++) {
52      if (!dsu1.check(1, i)) {
53        L.push_back(i);
54        dsu1.merge(1, i);
55      }
56      else if (!dsu2.check(1, i)) {
57        R.push_back(i);
58        dsu2.merge(1, i);
59      }
60    }
61    int cnt = min(L.size(), R.size());
62    for (int i = 0; i < cnt; i++) ans.push_back({ L[i], R[i] });
63
64    cout << ans.size() << endl;
65    for (auto [u, v] : ans) cout << u << ' ' << v << endl;
66  }
67
68  int main() {
69    solve();
70  }
```

12.2.12 Path Queries

原题指路: https://codeforces.com/problemset/problem/1213/G

题意 (3 s)

给定一棵包含n $(1 \le n \le 2e5)$个节点的树, 边的边权$w \in [1, 2e5]$. 现有m $(1 \le m \le 2e5)$个询问, 每个询问给定一个整数q $(1 \le q \le 2e5)$, 表示询问有多少个节点对(u, v) $(1 \le u < v \le n)$ $s.t.$ 节点u到节点v的简单路径上的最大边权不超过q.

思路I

若在线处理每个询问, 则每个询问需在$O(\log^p n)$的时间复杂度内求解, 但显然无数据结构可维护该信息, 考虑预处理出所有边权w的答案$ans[w]$.

注意到对询问q_i, q_j, 若$q_i < q_j$, 则$ans[q_i] \le ans[q_i]$. 对边权w, 考察$ans[w]$相对于$ans[w-1]$的增量.

类似于Kruskal算法, 将边按边权非降序排列后依次加边. 考察加入边权为w的边(u, v)时对答案$ans[w]$的贡献. 加入该边时会合并节点u所在的连通块和节点v所在的连通块, 设两连通块分别包含$siz[u]$和$siz[v]$个节点, 则贡献为$siz[u] \cdot siz[v]$.

升序枚举边权i, 将当前未加入集合中的边权$\le i$的边加入集合中, 更新贡献即可.

总时间复杂度$O(n \log n + m)$.

代码I

```cpp
const int MAXW = 2e5 + 5;

struct DSU {
    int n;
    vector<int> fa;
    vector<int> siz;

    DSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        siz = vector<int>(n + 1, 1);
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    void merge(int x, int y) {
        fa[x = find(x)] = (y = find(y));
        siz[y] += siz[x];
    }

    int getSize(int x) {
        return siz[find(x)];
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<pair<int, int>>> edges(n + 1);
    vector<tuple<int, int, int>> eds;  // w, u, v
    for (int i = 1; i < n; i++) {
        int u, v, w; cin >> u >> v >> w;
        edges[u].push_back({ v, w }), edges[v].push_back({ u, w });
        eds.push_back({ w, u, v });
    }
    sort(all(eds));

    vector<ll> ans(MAXW);
    DSU dsu(n);
    for (int i = 1, idx = 0; i < MAXW; i++) {  // 枚举边权
```

```
        ans[i] = ans[i - 1];
        while (idx < n - 1 && get<0>(eds[idx]) <= i) {
            auto [w, u, v] = eds[idx];
            ans[i] += (ll)dsu.getSize(u) * dsu.getSize(v);
            dsu.merge(u, v);

            idx++;
        }
    }

    while (m--) {
        int q; cin >> q;
        cout << ans[q] << ' ';
    }
    cout << endl;
}

int main() {
    solve();
}
```

思路II

将询问离线后按q值非降序排列后, 依次枚举询问. 将边按边权非降序排列后, 对每个询问q, 将当前未加入集合中的边权$\leq q$的边加入集合中, 并计算对当前答案res的贡献. 加入边权为w的边(u, v)时, 设节点u、v所在的连通块的大小分别为$siz[u]$和$siz[v]$, 则对res的贡献为$C^2_{siz[u]+siz[v]} - C^2_{siz[u]} - C^2_{siz[v]}$.

总时间复杂度$O(n \log n + m \log m)$.

代码II

```
ll C(int n) {  // 组合数C(n, 2)
    return (ll)n * (n - 1) / 2;
}

struct DSU {
    int n;
    vector<int> fa;
    vector<int> siz;
    ll res = 0;

    DSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        siz = vector<int>(n + 1, 1);
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    void merge(int x, int y) {
        fa[x = find(x)] = (y = find(y));
        res -= C(siz[x]) + C(siz[y]);
        siz[y] += siz[x];
        res += C(siz[y]);
```

```
    }

    int getSize(int x) {
        return siz[find(x)];
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<pair<int, int>>> edges(n + 1);
    vector<tuple<int, int, int>> eds;  // w, u, v
    for (int i = 1; i < n; i++) {
        int u, v, w; cin >> u >> v >> w;
        edges[u].push_back({ v, w }), edges[v].push_back({ u, w });
        eds.push_back({ w, u, v });
    }
    sort(all(eds));

    vector<pair<int, int>> ques(m);  // w, id
    for (int i = 0; i < m; i++) {
        cin >> ques[i].first;
        ques[i].second = i;
    }
    sort(all(ques));

    vector<ll> ans(m);
    DSU dsu(n);
    for (int i = 0, idx = 0; i < m; i++) {
        while (idx < n - 1 && get<0>(eds[idx]) <= ques[i].first) {
            auto [_, u, v] = eds[idx++];
            dsu.merge(u, v);
        }
        ans[ques[i].second] = dsu.res;
    }

    for (int i = 0; i < m; i++)
        cout << ans[i] << " \n"[i == m - 1];
}

int main() {
    solve();
}
```

思路III

对每条边(u, v), 设节点u、v所在的连通块的大小分别为$siz[u]$和$siz[v]$, 则简单路径包含该边的节点的对数为$siz[u] \cdot siz[v]$. 但若直接对每条边累加贡献, 则会重复计算贡献. 考察如何不重复地计算贡献.

建出该树的Kruskal重构树, 对每个节点u, 预处理出Kruskal重构树中以u为根节点的子树中属于原树的节点的个数$siz[u]$.

将新建的节点按边权非降序排列, 则对每个新建的节点u, 若它在Kruskal重构树中存在父亲节点$father$, 则点权$weights[u] \leq weights[father]$. 为使得对每个新建的节点统计贡献时不重复地计算贡献, 只需令节点u只对权值区间$[weights[u], weights[father] - 1]$的答案计算贡献即可, 显然贡献为$C_{siz[u]}^2$, 用差分实现区间加即可.

总时间复杂度$O(n \log n + m)$.

代码III

```cpp
const int MAXA = 2e5 + 5;
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct KruskalTree {
    int n, m;
    vector<tuple<int, int, int, int>> edges;  // w, u, v, id
    vector<int> fa;
    vector<int> used;  // 记录边是否在生成树中
    int idx;  // 当前用到的节点编号
    vector<vector<int>> tr;  // Kruskal重构树
    vector<int> weights;  // Kruskal重构树中节点的点权

    KruskalTree(int _n, int _m, const vector<tuple<int, int, int, int>>& _edges)
        :n(_n), m(_m), edges(_edges), idx(n) {
        fa.resize(2 * n + 1), used.resize(m + 1);
        tr.resize(2 * n + 1), weights.resize(2 * n + 1);
        iota(all(fa), 0);
        sort(all(edges));
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto [w, u, v, id] : edges) {
            int tmpu = find(u), tmpv = find(v);
            if (tmpu != tmpv) {
                // 新建节点并连边
                fa[tmpu] = fa[tmpv] = ++idx;
                tr[tmpu].push_back(idx), tr[idx].push_back(tmpu);
                tr[tmpv].push_back(idx), tr[idx].push_back(tmpv);
                weights[idx] = w;

                used[id] = true;
                res += w, cnt++;
            }
        }
        return cnt == n - 1 ? res : INFF;
    }
};

ll C(int n) {  // 组合数C(n, 2)
    return (ll)n * (n - 1) / 2;
}

void solve() {
    int n, m; cin >> n >> m;
    vector<tuple<int, int, int, int>> edges(n - 1);  // w, u, v, id
    for (auto& [w, u, v, _] : edges) cin >> u >> v >> w;

    KruskalTree solver(n, n - 1, edges);
```

```
    solver.kruskal();
    auto weights = solver.weights;
    int idx = solver.idx;
    sort(weights.begin() + n + 1, weights.begin() + idx + 1);

    auto tr = solver.tr;
    // siz[u]表示Kruskal重构树中以u为根节点的子树中属于原树的节点的个数
    vector<int> siz(2 * n + 1);
    vector<int> fa(2 * n + 1);  // fa[u]表示Kruskal重构树中节点u的父亲节点

    function<void(int, int)> dfs = [&](int u, int pre) {
        siz[u] = u <= n;  // 只统计属于原树的节点
        fa[u] = pre;

        for (auto v : tr[u]) {
            if (v != pre) {
                dfs(v, u);
                siz[u] += siz[v];
            }
        }
    };

    dfs(idx, 0);  // 从Kruskal重构树的根节点开始搜

    vector<ll> diff(MAXA);  // ans[]的差分数组

    for (int u = n + 1; u <= idx; u++) {  // 对Kruskal重构树中新建的节点统计答案
        auto father = fa[u];
        if (father && weights[u] == weights[father]) continue;  // 防止对区间[l, l - 1]计算贡献

        // 节点u只在权值区间[weights[u], weights[father] - 1]中计算贡献, 差分实现区间加
        diff[weights[u]] += C(siz[u]);
        if (father) diff[weights[father]] -= C(siz[u]);
    }

    for (int i = 1; i < MAXA; i++) diff[i] += diff[i - 1];  // 求ans[]

    while (m--) {
        int q; cin >> q;
        cout << diff[q] << ' ';
    }
    cout << endl;
}

int main() {
    solve();
}
```

12.2.13 Swaps in Permutation

原题指路: https://codeforces.com/problemset/problem/691/D

题意 (5 s)

给定一个$1 \sim n$ $(1 \le n \le 1e6)$的排列$p = [p_1, \cdots, p_n]$. 给定m $(1 \le m \le 1e6)$个数对(a_i, b_i) $(1 \le i \le m, 1 \le a, b \le n)$. 现有操作: 选择给定的数对中的任一数对, 交换$p[]$中对应的下标. 经若干次(可能为零)操作, 问可得到的字典序最大的$p[]$.

思路

用并查集维护可交换的下标构成的集合, 将同一集合内的元素升序排列. 对每个p_i $(1 \le i \le n)$, 输出p_i所在的集合$find(i)$中的最大元素.

代码

```cpp
struct DSU {
    int n;
    vector<int> fa;

    DSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    void merge(int x, int y) {
        fa[find(x)] = find(y);
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<int> p(n + 1);
    for (int i = 1; i <= n; i++) cin >> p[i];

    DSU dsu(n);
    while (m--) {
        int a, b; cin >> a >> b;
        dsu.merge(a, b);
    }

    vector<vector<int>> ans(n + 1);
    for (int i = 1; i <= n; i++)
        ans[dsu.find(i)].push_back(p[i]);
    for (int i = 1; i <= n; i++)
        sort(all(ans[i]));

    for (int i = 1; i <= n; i++) {
        int tmp = dsu.find(i);
        cout << ans[tmp].back() << " \n"[i == n];
        ans[tmp].pop_back();
    }
```

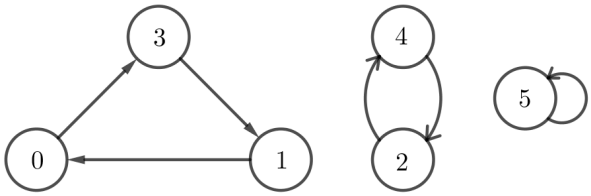
```
}

int main() {
    solve();
}
```

12.2 置换环

若一个有向图包含n个节点和n条边,且所有节点的入度和出度都为1,则这样的图由若干个环(普通环或自环)构成,称为**环图**.

考察$0 \sim (n-1)$的一个排列$a_0, \cdots, a_{n-1}$,若数i在下标p_i处,则连一条从节点i到节点p_i的有向边.显然这样得到的图是个环图,如:

下标$p[i]$	0	1	2	3	4	5
数值i	1	3	4	0	2	5



交换排列中两元素的位置有如下两种情况:

①交换同一环中的两节点时,该环会分裂为两个环.

②交换不同环中的两节点时,两环会合并为一个环.

通过交换元素将$a[]$升序排列等价于通过交换图中的节点使得所有节点都形成自环.

对一个大小为siz的环,至少需交换$(siz-1)$次才能使得该环中的所有节点都形成自环.

12.2.1 交换瓶子

题意

给定一个整数n $(1 \le n \le 1e4)$和一个$1 \sim n$的排列.现有操作:交换排列中的任意两元素.求将排列升序排列所需的最小操作次数.

思路

若限制只能交换相邻的元素,则该过程等价于冒泡排序,所需的步数为排列的逆序对的个数.

本题中的操作允许交换任意两元素,考虑置换环,设建图后图中有cnt个环,则$ans = n - cnt$.

代码

```cpp
void solve() {
  int n; cin >> n;
  vector<int> nxt(n + 1);
  for (int i = 1; i <= n; i++) {
    int x; cin >> x;
    nxt[x] = i;
  }

  int cnt = 0;
  vector<bool> vis(n + 1);
  for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
      cnt++;
      for (int j = i; !vis[j]; j = nxt[j]) vis[j] = true;
    }
  }
  cout << n - cnt << endl;
}

int main() {
  solve();
}
```

12.2.2 用Swap(0,i)排序

题意 (0.2 s)

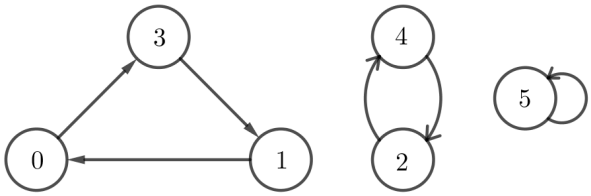
给定一个$0 \sim (n-1)$的排列,要求只用$swap(0,i)$操作(将数字0与另一数字交换位置)将该排列变为升序序列,求最小步数.

第一行输入n $(1 \leq n \leq 1e5)$,表示排列的元素个数.第二行输入$0 \sim (n-1)$的一个排列.

思路

考察置换环,如:

下标$p[i]$	0	1	2	3	4	5
数值i	1	3	4	0	2	5

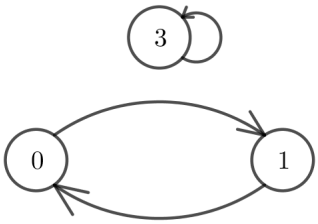


$swap(0,i)$操作分为三种:

①若节点i是与节点0相邻的点,如$swap(0,3)$时,序列变为:

下标$p[i]$	0	1	3
数值i	1	0	3

此时0指向的节点形成自环,0所在的环减小:

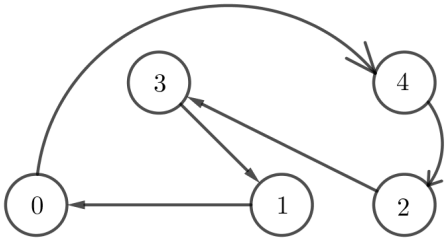


②若节点i非与节点0的相邻点,但与0在同一环内,显然这样会将0所在的环拆开成两个小环.注意到拆开成两个小环后,仍需交换0与其相邻的节点,让小环中的节点形成自环,显然不如直接交换0与其相邻的节点,故非最优解.

③若节点i与节点0不在同一环中,如$swap(0,2)$时,序列变为:

下标$p[i]$	0	1	2	3	4
数值i	1	3	4	2	0

此时节点0和i所在的环合并:



故最优解只需做操作①③,在任意时刻将不含节点0的非自环与节点0所在环合并,每次$swap$节点0与其相邻的节点产生自环.

最坏情况是n为奇数时,0号节点自环,其余节点两两形成共$\frac{n-1}{2}$个环,此时需进行$\frac{n-1}{2}$次合并使得所有节点与节点0在同一环中,再至多进行$(n-1)$次$swap$节点0与其相邻的节点使得其他节点形成自环,总时间复杂度$O(n)$.

代码

```cpp
const int MAXN = 1e5 + 5;
int n;
int p[MAXN];  // p[i]表示数i的下标

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    p[x] = i;
  }

  int ans = 0;
  int idx = 1;  // 将要与节点0交换的节点
  while (idx < n) {
    while (p[0]) swap(p[0], p[p[0]]), ans++;  // 将节点0所在的环都变成自环
```

```
    while (idx < n && p[idx] == idx) idx++;  // 跳过已经是自环的节点
    if (idx < n) swap(p[0], p[idx]), ans++;  // 将不含节点0的环与节点0合并
  }
  cout << ans;
}
```

12.2.3 数组补全

原题指路:https://codeforces.com/problemset/problem/1283/C

题意

有编号$1\sim n$的n个人交换礼物,每个人恰送出一份并收到一份礼物,每个人不能给自己送礼物.定义序列$f_1,\cdots,f_n$,有如下两种情况:①$f_i=0$表示i号人不知道他要把礼物送给谁;②$f_i\in[1,n]$表示i号人将礼物送给f_i号人.给定至少包含两个0的序列$f_1,\cdots,f_n$,构造一个不包含0的序列$f_1,\cdots,f_n$使得每个人恰送出一份并收到一份礼物,且每个人不能给自己送礼物.数据保证有解.若有多组解,输出任一组.

第一行输入一个整数n $(2\le n\le 2e5)$.第二行输入一个长度为n的序列
$f_1,\cdots,f_n$ $(0\le f_i\le n,f_i\neq i,$非零的所有$f_i$相异$)$.

思路

问题即将给定的$f_1,\cdots,f_n$构造为一个$1\sim n$的不包含不动点$(p_i=i)$的排列.

考虑置换环,但此时的图缺少了若干边.问题转化为给图添加若干条边,使得每个节点都在且仅在一个环中,且图中不存在自环.

考察图中的孤立点,不妨设有cnt个孤立点.

①若$cnt=0$,则跳过该步骤.

②若$cnt=1$,则将孤立点合并到任一条链上.

③若$cnt>1$,则将所有孤立点连成一个环.

注意到通过上述步骤处理孤立点后,零入度的节点数和零出度的节点数相等,将它们连成环即可满足要求.

若$f_{idx}=x$,则$a[x]=idx,p[idx]=x$.

代码I

```
void solve() {
  int n; cin >> n;
  vector<int> a(n + 1), p(n + 1);  // 若f[idx]=x,则a[idx]=x,p[x]=idx
  for (int i = 1; i <= n; i++) {
    int x; cin >> x;
    a[i] = x, p[x] = i;
  }

  bool ok = false;  // 记录是否已填完f[]中的所有0
  vector<bool> state(n + 1);  // 记录节点是否已在环中
  for (int i = 1; i <= n; i++) {
    if (state[i] || !a[i]) continue;  // 节点已在环中或该位置为零

    state[i] = true;
    int u = i, v = i;  // 链首、链尾
    for (; p[u] && !state[p[u]]; u = p[u]) state[p[u]] = true;
```

```
    for (; a[v] && !state[a[v]]; v = a[v]) state[a[v]] = true;

    if (a[v] == u) continue;  // 成环

    if (!ok) {  // 将f[]中的所有0接到节点y之后
      for (int j = 1; j <= n; j++) {
        if (!a[j] && !p[j]) {  // 孤立点
          state[j] = true;
          a[v] = j, v = j;  // 将节点j作为链尾
        }
      }
      ok = true;
    }

    a[v] = u;  // 成环
  }

  if (!ok) {  // f[]中非零的元素对应的节点都在环中
    int u = 0, v = 0;  // 链首、链尾
    for (int i = 1; i <= n; i++) {
      if (!a[i]) {  // 零出度的节点
        if (!u && !v) u = v = i;
        else a[v] = i, v = i;  // 将节点i作为链尾
      }
    }

    a[v] = u;  // 成环
  }

  for (int i = 1; i <= n; i++) cout << a[i] << " \n"[i == n];
}

int main() {
  solve();
}
```

代码II

```
void solve() {
  int n; cin >> n;
  vector<int> f(n + 1);
  vector<int> in(n + 1), out(n + 1);  // 节点的入度、出度
  for (int i = 1; i <= n; i++) {
    cin >> f[i];
    if (f[i]) out[i]++, in[f[i]]++;
  }

  vector<int> loops;
  for (int i = 1; i <= n; i++)
    if (!in[i] && !out[i]) loops.push_back(i);  // 孤立点
  int cnt = loops.size();  // 孤立点数
  if (cnt == 1) {  // 只有一个孤立点,将其合并到任一条链上
    int u = loops[0];
    for (int i = 1; i <= n; i++) {
      if (!in[i] && i != u) {  // 非u的零入度节点可作为链首
        f[u] = i;  // 将节点u作为链首
```

```
        in[i]++, out[u]++;
        break;
      }
    }
  }
  else if (cnt > 1) {  // 将所有孤立点连成一个环
    for (int i = 0; i < cnt; i++) {
      int cur = loops[i], nxt = loops[(i + 1) % cnt];
      f[cur] = nxt;
      in[nxt]++, out[cur]++;
    }
  }

  vector<int> in0, out0;  // 零入度、零出度的节点
  for (int i = 1; i <= n; i++) {
    if (!in[i]) in0.push_back(i);
    if (!out[i]) out0.push_back(i);
  }

  cnt = in0.size();
  for (int i = 0; i < cnt; i++)  // 保证零入度的节点数和零出度的节点数相等
    f[out0[i]] = in0[i];  // 将零入度的节点和零出度的节点连成环

  for (int i = 1; i <= n; i++) cout << f[i] << " \n"[i == n];
}

int main() {
  solve();
}
```

12.2.4 情侣牵手

原题指路:https://leetcode.cn/problems/couples-holding-hands/

题意

有n对情侣坐在编号$1 \sim 2n$的$2n$个座位上,给定一个长度为$2n$的数组$row_1, \cdots, row_{2n}$,其中row_i $(1 \le i \le 2n)$表示坐在i号座位的人的编号.每对情侣按顺序编号,其中第一对情侣编号$(0, 1)$,第二对情侣编号$(2, 3)$,$\cdots$.现有操作:交换两人的位置.问使得每一对情侣的座位都相邻所需的最小步数.

第一行输入一个整数n $(1 \le n \le 30)$.第二行输入一个长度为$2n$的数组$row_1, \cdots, row_{2n}$ $(0 \le row_i < 2n)$.数据保证$row[]$中的所有元素都相异.

思路

考虑置换环,问题转化为将图变为n个二元环,其中每个二元环内的两个节点对应一对情侣.

考察交换操作.

①交换同一环中的两个节点时,该环会分裂为两个环.

②交换不同环中的两个节点时,会将两个环合并为一个环,进而环数-1.

综上,每次操作至多增加一个环.设有n对情侣和cnt个环,则$ans = n - cnt$.

代码

```cpp
class Solution {
public:
  vector<int> fa;

  void init(int n) {
    fa.resize(n);
    iota(fa.begin(), fa.end(), 0);
  }

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }

  bool merge(int x, int y) {
    if ((x = find(x)) == (y = find(y))) return false;
    else {
      fa[x] = y;
      return true;
    }
  }

  int minSwapsCouples(vector<int>& row) {
    int n = row.size() / 2;
    init(n);

    int cnt = n;
    for (int i = 0; i < 2 * n; i += 2) {
      int u = row[i] / 2, v = row[i + 1] / 2;  // 两人所属的情侣的编号
      cnt -= merge(u, v);
    }
    return n - cnt;
  }
};
```

12.2.5 Lucky Permutation

原题指路:https://codeforces.com/contest/1768/problem/D

题意

给定一个$1 \sim n$的排列$p_1, \cdots, p_n$.现有操作:交换两元素.问使得该排列恰有一个逆序对所需的最小操作次数.

有t $(1 \leq t \leq 1e4)$组测试数据.每组测试数据第一行输入一个整数n $(2 \leq n \leq 2e5)$.第二行输入一个$1 \sim n$的排列$p_1, \cdots, p_n$.数据保证所有测试数据的n之和不超过$2e5$.

思路I

先考虑使得该排列无逆序对所需的最小操作次数,考虑置换环,设有cnt个环,则所需的最小操作次数为$(n - cnt)$.

为使得该排列恰有一个逆序对,

①若使得该排列无逆序对的交换过程中存在一对逆序,即一个环中存在相邻的两节点时,

则无需交换它们,此时$ans = n - cnt - 1$.

②若使得该排列无逆序对的交换过程中不存在逆序,即所有环中都不存在相邻的两节点时,

则需多交换一次来产生一个逆序对,此时$ans = n - cnt + 1$.

代码I

```cpp
void solve() {
  int n; cin >> n;
  vector<int> p(n + 1);
  for (int i = 1; i <= n; i++) cin >> p[i];

  bool adjacent = false;  // 记录一个环中是否存在相邻的节点
  int cnt = 0;  // 环数
  vector<bool> vis(n + 1);
  set<int> s;  // 存环中的节点

  function<void(int)> dfs = [&](int u) {
    if (vis[u]) return;  // 成环

    vis[u] = true;
    s.insert(u);
    if (s.count(u - 1) || s.count(u + 1)) adjacent = true;
    dfs(p[u]);
  };

  for (int u = 1; u <= n; u++) {
    if (!vis[u]) {
      cnt++;
      s.clear();
      dfs(u);
    }
  }

  cout << (n - cnt + (adjacent ? -1 : 1)) << endl;
}

int main() {
  CaseT
    solve();
}
```

思路II

用DSU的思想,记录记录节点所在的环的代表节点后,枚举数值即可确定一个环中是否存在相邻的节点.

代码II

```cpp
void solve() {
  int n; cin >> n;
  vector<int> p(n + 1);
  for (int i = 1; i <= n; i++) cin >> p[i];

  int ans = n;
  vector<int> loops(n + 1, -1);  // 记录节点所在的环的代表节点
  for (int u = 1; u <= n; u++) {
    if (~loops[u]) continue;  // 该节点已在环中
```

```
10
11      ans--;
12      for (int v = u; loops[v] == -1; v = p[v])
13        loops[v] = u;  // 将节点v合并入节点u所在的环中
14    }
15
16    bool adjacent = false;  // 记录是否存在相邻的节点
17    for (int i = 2; i <= n; i++) adjacent |= loops[i] == loops[i - 1];
18
19    cout << (ans + (adjacent ? -1 : 1)) << endl;
20  }
21
22  int main() {
23    CaseT
24      solve();
25  }
```

思路III

设有cnt个环,其中第$i\ \ (1 \leq i \leq cnt)$个环包含siz_i个节点,则使得该排列升序所需的最小步数为$\sum_{i=1}^{cnt}(siz_i - 1)$.

与**思路I**类似,若存在一个环中包含相邻的节点,则步数-1;否则步数$+1$.

代码III

```
1   void solve() {
2     int n; cin >> n;
3     vector<int> p(n + 1);
4     for (int i = 1; i <= n; i++) cin >> p[i];
5
6     int cnt = 0;  // 环数
7     vector<vector<int>> loops(n + 1);  // 存每个环中的节点
8     vector<bool> vis(n + 1);
9     for (int u = 1; u <= n; u++) {
10      if (p[u] == u || vis[u]) continue;  // 该节点形成自环或已访问过
11
12      vis[u] = true;
13      loops[++cnt].push_back(u);
14      for (int v = u; !vis[p[v]]; v = p[v]) {
15        vis[p[v]] = true;
16        loops[cnt].push_back(p[v]);
17      }
18    }
19
20    int ans = 0;
21    bool adjacent = false;  // 记录是否存在相邻的节点
22    for (int i = 1; i <= cnt; i++) {
23      int siz = loops[i].size();
24      ans += siz - 1;
25
26      sort(all(loops[i]));
27      for (int j = 0; j + 1 < siz; j++)
28        adjacent |= loops[i][j] + 1 == loops[i][j + 1];
29    }
30
```

```
  cout << (ans + (adjacent ? -1 : 1)) << endl;
}

int main() {
  CaseT
    solve();
}
```

12.2.6 Koxia and Game

原题指路:https://codeforces.com/contest/1770/problem/D

题意

A和B玩一个关于三个长度为n的序列$a[]$、$b[]$和$c[]$的游戏,其中每个序列的元素范围为$[1,n]$.游戏包括n轮,第i轮中,设可重集$S\{a_i,b_i,c_i\}$,该轮有两步:①A从S中移除一个元素;②B从S的剩余元素中选择一个元素.设第i轮B选的数为d_i,若$d[]$是$1\sim n$的排列,则A胜,否则B胜.两人都采取最优策略.给定$a[]$和$b[]$,求使得A胜的序列$c[]$的个数,答案对998244353取模.

有t $(1\le t\le 2e4)$组测试数据.每组测试数据第一行输入一个整数n $(1\le n\le 1e5)$.第二行输入n个整数$a_1,\cdots,a_n$ $(1\le a_i\le n)$.第三行输入n个整数$b_1,\cdots,b_n$ $(1\le b_i\le n)$.数据保证所有测试数据的n之和不超过$1e5$.

思路I

对每个$i\in[1,n]$,建边(a_i,b_i),这样得到的图由若干条链和若干个环构成.

①$a_i=b_i$,即节点a_i处形成自环时,A删去c_i即可迫使B选择a_i,故c_i在$[1,n]$中任取,对答案的贡献为n.

②图中存在单独的链时,WLOG,考虑长度为2的链,此时无论A删去哪个元素,该链的两端点至少有一个不能被选到,

此时B总能选择一个不使得$d[]$构成$1\sim n$的排列的元素,故无解.

③对一个孤立的普通环(非自环),选择的过程可视为在环上顺时针或逆时针绕行,对答案的贡献为2.

注意每个普通环只对代表元素统计答案,防止重复计数.

③对一条链上的一个普通环,选择的过程可视为先沿着链行进,后在环上顺时针或逆时针绕行.

链的部分对答案的贡献为1,环的部分对答案的贡献为2,故可将链与环合并,避免后续统计答案时遗漏该环.

④对一条链上的一个自环,选择的过程可视为沿着链和自环行进,链的部分和自环的部分对答案的贡献都为1.

⑤对环套环,WLOG,考虑两个相切的环,在切点处有两种绕行方向,

此时B总能选择一个不使得$d[]$构成$1\sim n$的排列的元素,故无解.

用DSU维护合并过程即可,注意开一个数组$selfLoop[]$记录节点是否存在自环,防止多次经过自环时重复计数.

代码I

```
const int MAXN = 2e5 + 5;
namespace DSU {
  int n;
  int fa[MAXN];
  int state[MAXN];  // 记录节点的状态,0表示在链上,1表示在普通环中,2表示在链上的自环中
  bool selfLoop[MAXN];  // 记录节点是否存在自环
  Z ans;

```

```
  void init() {
    for (int i = 1; i <= n; i++) {
      fa[i] = i;
      state[i] = 0, selfLoop[i] = false;
    }
    ans = 1;
  }

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }

  void merge(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) {
      if (!state[x]) state[x] = 1;  // 链变成普通环
      else ans = 0;  // 环套环,无解
    }
    else if (state[x] && state[y]) ans = 0;  // 环套环,无解
    else {
      if (!state[y]) swap(x, y);
      fa[x] = y;  // 将链合并到环中
    }
  }
}
using namespace DSU;

void solve() {
  cin >> n;
  init();
  vector<int> a(n + 1), b(n + 1);
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i <= n; i++) cin >> b[i];

  for (int i = 1; i <= n; i++) {
    int u = a[i], v = b[i];

    if (u != v) merge(u, v);
    else {  // 形成自环
      if (!selfLoop[u]) {  // 防止重复计数
        selfLoop[u] = true;
        ans *= n;  // 自环贡献为n
      }
      else ans = 0;  // 环套环,无解
    }
  }

  for (int i = 1; i <= n; i++)  // 链上的自环无贡献
    if (a[i] == b[i]) state[find(a[i])] = 2;

  for (int i = 1; i <= n; i++) {
    if (!selfLoop[i] && fa[i] == i) {  // 每个普通环只在代表节点处计算贡献
      if (state[i] == 1) ans *= 2;  // 普通环贡献为2
      else if (!state[i]) ans = 0;  // 存在单独的链
    }
  }
  cout << ans << endl;
}
```

```
67
68 int main() {
69   CaseT
70     solve();
71 }
```

思路II

思路I中的讨论可总结为:有解当且仅当每个连通块的节点数与边数相等.

代码II:并查集写法

```
1  const int MAXN = 2e5 + 5;
2  namespace DSU {
3    int n;
4    int fa[MAXN];
5    int cntV[MAXN], cntE[MAXN];  // 连通块包含的节点数、边数
6    bool selfLoop[MAXN];  // 记录节点是否存在自环
7    bool vis[MAXN];
8
9    void init() {
10     for (int i = 1; i <= n; i++) {
11       fa[i] = i;
12       cntV[i] = 1, cntE[i] = 0;
13       selfLoop[i] = vis[i] = false;
14     }
15   }
16
17   int find(int x) {
18     return x == fa[x] ? x : fa[x] = find(fa[x]);
19   }
20
21   void merge(int x, int y) {  // 将节点y所在的连通块合并到节点x所在的连通块
22     if ((x = find(x)) == (y = find(y))) return;
23
24     fa[y] = x;
25     cntV[x] += cntV[y], cntE[x] += cntE[y];
26     selfLoop[x] |= selfLoop[y];
27   }
28 }
29 using namespace DSU;
30
31 void solve() {
32   cin >> n;
33   init();
34   vector<int> a(n + 1), b(n + 1);
35   for (int i = 1; i <= n; i++) cin >> a[i];
36   for (int i = 1; i <= n; i++) cin >> b[i];
37
38   for (int i = 1; i <= n; i++) {
39     merge(a[i], b[i]);
40
41     int tmp = find(a[i]);
42     cntE[tmp]++;
43     if (a[i] == b[i]) selfLoop[tmp] = true;
44   }
```

```
45
46    Z ans = 1;
47    for (int i = 1; i <= n; i++) {
48      int tmp = find(i);
49      if (!vis[tmp]) {
50        vis[tmp] = true;
51        if (cntV[tmp] != cntE[tmp]) ans = 0;  // 连通块的节点数与边数不相等时无解
52        else ans *= selfLoop[tmp] ? n : 2;  // 自环对答案的贡献为n,普通环对答案的贡献为2
53      }
54    }
55    cout << ans << endl;
56  }
57
58  int main() {
59    CaseT
60      solve();
61  }
```

代码III:搜索写法

```
1   void solve() {
2     int n; cin >> n;
3     vector<int> a(n + 1), b(n + 1);
4     for (int i = 1; i <= n; i++) cin >> a[i];
5     vector<vector<int>> edges(n + 1);
6     for (int i = 1; i <= n; i++) {
7       cin >> b[i];
8       edges[a[i]].push_back(b[i]), edges[b[i]].push_back(a[i]);
9     }
10
11    Z ans = 1;
12    vector<bool> vis(n + 1);
13    for (int i = 1; i <= n; i++) {
14      if (vis[i]) continue;
15
16      queue<int> que;
17      que.push(i);
18      vis[i] = true;
19
20      int cntV = 0, cntE = 0;  // 连通块中的节点数和边数
21      while (que.size()) {
22        auto u = que.front(); que.pop();
23        cntV++, cntE += edges[u].size();
24
25        for (auto v : edges[u]) {
26          if (!vis[v]) {
27            vis[v] = true;
28            que.push(v);
29          }
30        }
31      }
32
33      // 连通块中的节点数与边数相等时有解,先不考虑自环,将所有环对答案的贡献视为2
34      ans *= (cntE /= 2) == cntV ? 2 : 0;
35    }
36
```

```
  for (int i = 1; i <= n; i++)
    if (a[i] == b[i]) ans /= 2, ans *= n;  // 自环对答案的贡献为n
  cout << ans << endl;
}

int main() {
  CaseT
    solve();
}
```

12.2.7 Bertown Subway

原题指路: https://codeforces.com/problemset/problem/884/C

题意

有编号$1\sim n$个n $(1\le n\le 1e5)$个站点. 对节点i, 有: ①恰存在一个下一个站点p_i, 且p_i可为i; ②恰存在一个站点$j\ s.t.\ p_j=i$. 定义序列$p=[p_1,\cdots,p_n]$ $(1\le p_i\le n)$的权值为整数对(x,y) $(1\le x,y\le n)$的对数, 使得存在一条从站点x出发到达站点y到达路径. 现可至多改变序列$p[]$中的两个p_i值, 要求改变后仍满足性质①和②. 求可达到的最大权值.

思路

[**定理**] 该图由若干个环构成, 则$p[]$是$1\sim n$的一个排列.

[**证**] 性质①表明: 每个节点的出度都为1. 性质②表明: 每个节点的入度都为1.

考察一个包含len个节点的环, 显然其中任一节点x与其自身或其他节点y都能构成满足权值要求的数对(x,y), 故该环对答案的贡献为len^2.

为使得修改p_i值后图仍由若干个环构成, 要么不修改, 要么修改恰两次, 且这两次修改等价于交换p_i与p_j.

考察置换环, 交换$p[]$中的两个元素要么使得一个环分裂为两个环, 要么将两个环合并为一个环.

注意到合并长度(包含的节点数)为a和b的两环时权值的增量$(a+b)^2-a^2-b^2=2ab>0$, 故应贪心地合并图中长度最大的两环.

代码

```
struct DSU {
    int n;
    vector<int> fa;
    vector<int> siz;

    DSU(int _n) :n(_n) {
        fa.resize(n + 1), siz.resize(n + 1, 1);
        iota(all(fa), 0);
    }

    int find(int x) {
        return x == fa[x] ? fa[x] : fa[x] = find(fa[x]);
    }

```

```
    void merge(int x, int y) {
        if ((x = find(x)) != (y = find(y))) {
            siz[y] += siz[x];
            fa[x] = y;
        }
    }
};

void solve() {
    int n; cin >> n;
    DSU dsu(n);
    for (int i = 1; i <= n; i++) {
        int p; cin >> p;
        dsu.merge(p, i);
    }

    vector<int> lengths;  // 环的长度
    for (int u = 1; u <= n; u++) {
        if (dsu.fa[u] == u)
            lengths.push_back(dsu.siz[u]);
    }
    sort(all(lengths));

    int siz = lengths.size();
    if (siz >= 2) {  // 合并长度最大的两环
        lengths[siz - 2] += lengths[siz - 1];
        lengths.pop_back();
    }

    ll ans = 0;
    for (auto len : lengths)
        ans += (ll)len * len;
    cout << ans << endl;
}

int main() {
    solve();
}
```

12.3 可撤销并查集与线段树分治

12.3.1 Unique Occurrences

原题指路: https://codeforces.com/contest/1681/problem/F

题意 (6 s)

给定一棵包含编号 $1 \sim n$ 的 n $(2 \le n \le 5e5)$ 的有权树, 边权 $w \in [1, n]$. 对节点 $u, v \in [1, n]$, 定义 $f(u, v)$ 为 u 到 v 的简单路径上恰出现一次的权值的数量. 求 $\sum_{1 \le u < v \le n} f(u, v)$.

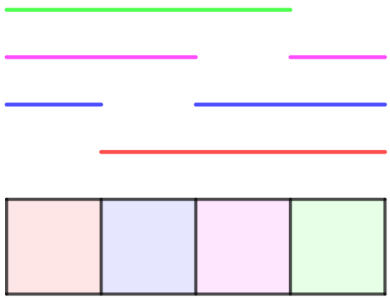
思路

因边权值域较小, 考虑对每个边权 w 统计答案. 删除树上边权为 w 的边(若存在)后, 树分裂为若干个连通块, 且边权为 w 的边在不同连通块的两节点间的简单路径上恰出现一次, 则任意两连通块的大小之积的和即 w 的贡献.

考虑如何快速删除一条边后维护树分裂成的各连通块的大小. 考虑可撤销并查集, 具体地, 用并查集维护连通性, 初始时将树边逐个合并, 用栈记录各版本的节点的父亲节点 $hisFa[]$ 和 连通块大小 $hisSiz[]$, 删边时将操作回退即可.

本题中对每种边权的边都需删除、统计答案、添加, 暴力回退操作的时间复杂度为 $O(n^2 \log n)$, 其中 $O(\log n)$ 是无路径压缩、只有按秩合并的并查集的find()的时间复杂度, 会TLE.

考虑对时间分治. 具体地, 考察每条边在 $[1, n]$ 时间段内存在的时间段, 如下图所示, 其中不同颜色表示不同的权值, 方块表示在对应时刻删除对应颜色的边, 线段表示对应的权值的边存在的时间段:



用线段树维护各边存在的时间段, 在统计答案(查询)时加边和回退即可.

总时间复杂度 $O(n \log^2 n)$.

代码

```cpp
struct RevocableDSU {
    int n;
    vector<int> fa;
    vector<int> siz;
    vector<pair<int&, int>> hisSiz;
    vector<pair<int&, int>> hisFa;

    RevocableDSU() {}
    RevocableDSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        siz = vector<int>(n + 1, 1);
    }

    int find(int x) {  // 无路径压缩
        while (x != fa[x]) x = fa[x];
        return x;
    }

    bool merge(int x, int y) {  // 按秩合并, 返回合并是否成功
        if ((x = find(x)) == (y = find(y))) return false;

        if (siz[x] < siz[y]) swap(x, y);
        hisSiz.push_back({ siz[x], siz[x] });
        siz[x] += siz[y];
        hisFa.push_back({ fa[y], fa[y] });
        fa[y] = x;
        return true;
    }
```

```
30
31      int getSize(int x) {
32          return siz[find(x)];
33      }
34
35      bool check(int x, int y) {
36          return find(x) == find(y);
37      }
38
39      int getVersion() {  // 当前版本的版本号
40          return hisFa.size();
41      }
42
43      void rollback(int ver) {  // 回退到第ver个版本
44          while (hisFa.size() > ver) {
45              hisFa.back().first = hisFa.back().second;
46              hisFa.pop_back();
47              hisSiz.back().first = hisSiz.back().second;
48              hisSiz.pop_back();
49          }
50      }
51  };
52
53  struct SegmentTreeDivide {
54      int n;
55      vector<vector<pair<int, int>>> edges;  // edges[w]存权值为w的边
56      RevocableDSU dsu;
57      struct Node {
58          vector<int> weights;  // 在该节点的时间段存在的边的权值
59      };
60      vector<Node> SegT;
61      ll ans = 0;
62
63      SegmentTreeDivide(int _n) :n(_n) {
64          edges.resize(n + 1);
65          dsu = RevocableDSU(n);
66          SegT.resize(n + 5 << 2);
67      }
68
69      void insert(int u, int l, int r, int L, int R, int w) {
70          if (L <= l && r <= R) {
71              SegT[u].weights.push_back(w);
72              return;
73          }
74
75          int mid = l + r >> 1;
76          if (L <= mid) insert(u << 1, l, mid, L, R, w);
77          if (R > mid) insert(u << 1 | 1, mid + 1, r, L, R, w);
78      }
79
80      void insert(int L, int R, int w) {  // 权值为w的边在[L, R]时间段存在
81          insert(1, 1, n, L, R, w);
82      }
83
84      void query(int rt, int l, int r) {
85          int ver = dsu.getVersion();
86          for (auto w : SegT[rt].weights) {
87              for (auto [u, v] : edges[w])
```

```
                dsu.merge(u, v);
        }

        if (l == r) {
            for (auto [u, v] : edges[l])
                ans += (ll)dsu.getSize(u) * dsu.getSize(v);
        }
        else {
            int mid = l + r >> 1;
            query(rt << 1, l, mid), query(rt << 1 | 1, mid + 1, r);
        }

        dsu.rollback(ver);
    }
};

void solve() {
    int n; cin >> n;

    SegmentTreeDivide solver(n);
    for (int i = 1; i < n; i++) {
        int u, v, w; cin >> u >> v >> w;
        solver.edges[w].push_back({ u, v });
    }

    for (int w = 1; w <= n; w++) {
        if (w == 1) solver.insert(2, n, w);
        else if (w == n) solver.insert(1, n - 1, w);
        else solver.insert(1, w - 1, w), solver.insert(w + 1, n, w);
    }

    solver.query(1, 1, n);
    cout << solver.ans << endl;
}

int main() {
    solve();
}
```

12.3.2 Connect and Disconnect

原题指路: https://codeforces.com/gym/100551/problem/A

题意 (3 s)

维护一张包含编号 $1 \sim n$ 的 n $(1 \le n \le 3e5)$ 的无向图, 初始时图中无边. 现有 m $(0 \le m \le 3e5)$ 个操作, 操作有如下三种:

① $+ \ u \ v$ $(u \ne v)$, 表示加入边 (u, v) . 数据保证该操作前该边不存在.

② $- \ u \ v$ $(u \ne v)$, 表示删除边 (u, v) . 数据保证该操作前该边存在.

③ ? , 表示询问图中的连通块数.

思路

用 `std::map` $last[]$ 记录每条边最后一次出现的时间区间. 对操作①, 令 $last[\{u,v\}]=i$, 其中 $i\in[1,m]$ $(m\geq 1)$ 为操作的编号. 用线段树维护每条边存在的时间区间, 用可撤销并查集支持回退操作.

初始时图中连通块的个数 $component=n$. 并查集合并时令 $component--$, 回退时令 $component++$.

注意特判 $m=0$ 的情况.

代码

```cpp
struct RevocableDSU {
    int n;
    vector<int> fa;
    vector<int> siz;
    vector<pair<int&, int>> hisSiz;
    vector<pair<int&, int>> hisFa;
    int component;

    RevocableDSU() {}
    RevocableDSU(int _n) :n(_n), component(n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        siz = vector<int>(n + 1, 1);
    }

    int find(int x) {  // 无路径压缩
        while (x != fa[x]) x = fa[x];
        return x;
    }

    bool merge(int x, int y) {  // 按秩合并, 返回合并是否成功
        if ((x = find(x)) == (y = find(y))) return false;

        component--;
        if (siz[x] < siz[y]) swap(x, y);
        hisSiz.push_back({ siz[x], siz[x] });
        siz[x] += siz[y];
        hisFa.push_back({ fa[y], fa[y] });
        fa[y] = x;
        return true;
    }

    int getSize(int x) {
        return siz[find(x)];
    }

    bool check(int x, int y) {
        return find(x) == find(y);
    }

    int getVersion() {  // 当前版本的版本号
        return hisFa.size();
    }

    void rollback(int ver) {  // 回退到第ver个版本
```

```
        while (hisFa.size() > ver) {
            component++;
            hisFa.back().first = hisFa.back().second;
            hisFa.pop_back();
            hisSiz.back().first = hisSiz.back().second;
            hisSiz.pop_back();
        }
    }
};

struct SegmentTreeDivide {
    int n, m;  // 节点数、询问数
    RevocableDSU dsu;
    struct Node {
        vector<pair<int, int>> edges;  // 该时刻存在的边
    };
    vector<Node> SegT;
    vector<int> ans;

    SegmentTreeDivide(int _n, int _m) :n(_n), m(_m) {
        dsu = RevocableDSU(n);
        SegT.resize(m + 5 << 2);
        ans.resize(m + 1);
    }

    void insert(int u, int l, int r, int L, int R, pair<int, int> edge) {
        if (L <= l && r <= R) {
            SegT[u].edges.push_back(edge);
            return;
        }

        int mid = l + r >> 1;
        if (L <= mid) insert(u << 1, l, mid, L, R, edge);
        if (R > mid) insert(u << 1 | 1, mid + 1, r, L, R, edge);
    }

    void insert(int L, int R, pair<int, int> edge) {
        insert(1, 1, m, L, R, edge);
    }

    void query(int rt, int l, int r) {
        int ver = dsu.getVersion();

        for (auto [u, v] : SegT[rt].edges)
            dsu.merge(u, v);

        if (l == r) ans[l] = dsu.component;
        else {
            int mid = l + r >> 1;
            query(rt << 1, l, mid), query(rt << 1 | 1, mid + 1, r);
        }

        dsu.rollback(ver);
    }

    void query() {
        query(1, 1, m);
    }
```

```
};

void solve() {
    int n, m; cin >> n >> m;

    if (!m) return;

    SegmentTreeDivide solver(n, m);
    vector<int> ques;
    map<pair<int, int>, int> last;  // que, idx
    for (int i = 1; i <= m; i++) {
        char op; cin >> op;

        if (op == '?') ques.push_back(i);
        else {
            int u, v; cin >> u >> v;
            if (u > v) swap(u, v);

            if (op == '+') last[{ u, v }] = i;
            else {
                solver.insert(last[{ u, v }], i - 1, { u, v });
                last[{ u, v }] = -1;
            }
        }
    }

    for (auto [que, idx] : last)
        if (idx >= 1) solver.insert(idx, m, que);

    solver.query();
    for (auto idx : ques)
        cout << solver.ans[idx] << endl;
}

int main() {
    solve();
}
```