ACM Template of Hartley

Hartley

2020 年 9 月 25 日

Contents

1 图论 **1**

1.1 存图方式 1
1.2 并查集 2
1.3 可持久化并查集 3
1.4 可撤销并查集 6
1.5 最小生成树 7
1.5.1 prim：$O(n^2)$ 7
1.5.2 prim 堆优化：$O(mlogn)$ 8
1.5.3 kruskal：$O(mlogm)$ 9
1.5.4 boruvka：$O((n+m)logn)$ 10
1.6 曼哈顿最小生成树：$O(nlogn+mlogm)$ 11
1.7 最小瓶颈生成树：$O(n+m)$ 15
1.8 次小生成树 17
1.8.1 非严格次小生成树 17
1.8.2 严格次小生成树 21
1.9 kruskal 重构树 26
1.10 最小斯坦纳树 27
1.11 最短路 30
1.11.1 Floyd：$O(n^3)$ 30
1.11.2 dijkstra：$O(n^2)$ 30
1.11.3 dijkstra 堆优化：$O((n+m)logm)$ 31
1.11.4 bellman_ford：$O(nm)$ 32

1.11.5 spfa：$O(nm)$ . . . 33
1.12 k 短路 . . . 35
1.12.1 A* 算法：复杂度玄学 . . . 35
1.12.2 可持久化可并堆优化：$O(T(dijkstra)+nlogn+klogn)$ . . . 37
1.13 树的直径 . . . 41
1.13.1 两次 dfs/bfs . . . 41
1.13.2 树形 dp . . . 42
1.14 图的直径 . . . 43
1.15 最小直径生成树 . . . 43
1.16 最近公共祖先 . . . 47
1.16.1 倍增求 LCA . . . 47
1.16.2 st 表求 LCA . . . 48
1.16.3 离线求 LCA：$O(n+m+q)$ . . . 49
1.17 树链剖分 . . . 51
1.17.1 重链剖分 . . . 51
1.17.2 长链剖分 . . . 53
1.18 DSU On Tree：$O(nlogn)$ . . . 57
1.19 树的重心 . . . 59
1.20 树分治 . . . 60
1.20.1 点分治：$O(nlogn+logn*T(solve))$ . . . 60
1.20.2 边分治：$O(nlogn+logn*T(solve))$ . . . 62
1.20.3 点分树 . . . 66
1.21 网络流 . . . 73
1.21.1 最大流 . . . 73
1.21.2 最小费最大流 . . . 75
1.21.3 最大权闭合图 . . . 80
1.21.4 最大密度子图 . . . 81
1.21.5 平面图网络流 . . . 85
1.21.6 上下界网络流 . . . 86
1.22 无向图全局最小割：$O(n^3)$ . . . 91
1.23 二分图 . . . 92
1.23.1 二分图的判定：$O(n+m)$ . . . 92
1.23.2 二分图最大匹配：$O(nm)$ . . . 93
1.23.3 二分图最大权匹配：$O(n^3)$ . . . 94

1.23.4 二分图多重最大匹配：$O(m\sqrt{n})$ . . . 98
1.23.5 二分图多重最大权匹配：$O(m\sqrt{n})$ . . . 99
1.24 一般图 . . . 99
1.24.1 一般图匹配：$O(n(nlogn+m))$ . . . 99
1.24.2 一般图最大权匹配：$O(n(nlogn+m))$ . . . 102
1.25 换根 dp . . . 110
1.26 树哈希 . . . 111
1.26.1 有根树哈希 . . . 111
1.26.2 无根树哈希 . . . 111
1.27 字典树 Trie . . . 113
1.28 tarjan 算法 . . . 115
1.28.1 有向图的强连通分量 . . . 115
1.28.2 无向图的点双连通分量 . . . 117
1.28.3 无向图的边双连通分量 . . . 118
1.29 kosaraju 算法 . . . 119
1.30 支配树 . . . 121
1.31 虚树 . . . 124
1.32 最小树形图：朱刘算法 . . . 127
1.32.1 定根最小树形图 . . . 127
1.32.2 不定根最小树形图 . . . 130
1.33 2-SAT . . . 131
1.34 差分约束系统 . . . 131
1.35 同余最短路 . . . 131
1.36 拓扑排序：$O(n+m)$ . . . 133
1.36.1 bfs 版本：$O(n+m)$ . . . 133
1.36.2 dfs 版本：$O(n+m)$ . . . 134
1.37 Bron–Kerbosch 算法：$O(3^{\frac{n}{3}})$ . . . 135
1.37.1 只求最大团 . . . 135
1.37.2 求所有极大团 . . . 136
1.38 图的生成树计数问题：矩阵树定理 . . . 137
1.38.1 无向图 . . . 138
1.38.2 有向图 . . . 138
1.38.3 求矩阵行列式：$O(n^3)$ . . . 139
1.38.4 最小生成树计数：$O(n^3)$ . . . 140

1.39 欧拉图 . . . 143
1.39.1 性质 . . . 143
1.39.2 无向图 . . . 143
1.39.3 有向图 . . . 145
1.40 哈密顿图 . . . 146
1.40.1 性质 . . . 146
1.40.2 无向图 . . . 146
1.40.3 竞赛图 . . . 148
1.41 平面图 . . . 150
1.41.1 平面图定义 . . . 150
1.41.2 欧拉公式 . . . 150
1.41.3 平面图的判断 . . . 151
1.41.4 对偶图 . . . 151
1.41.5 点着色 . . . 151
1.42 弦图 . . . 152
1.42.1 最大势算法 MCS：$O(n+m)$ . . . 152
1.42.2 弦图的极大团：$O(n+m)$ . . . 155
1.42.3 弦图的色数/团数：$O(n+m)$ . . . 156
1.42.4 弦图的最大独立集/最小团覆盖：$O(n+m)$ . . . 156
1.43 最小环 . . . 156
1.43.1 朴素 dijkstra：$O(m*n^2)$ . . . 157
1.43.2 堆优化 dijkstra：$O(m*(n+m)logm)$ . . . 158
1.43.3 Floyd：$O(n^3)$ . . . 159
1.44 Prufer 序列 . . . 162
1.44.1 构造 Prufer 序列：$O(n)$ . . . 163
1.44.2 Prufer 序列重建树：$O(n)$ . . . 163
1.45 图的着色 . . . 164
1.45.1 点着色 . . . 164
1.45.2 边着色 . . . 165
1.45.3 色多项式 . . . 167
1.46 LGV 引理 . . . 167
1.47 树上修改问题 . . . 169
1.47.1 单点向外辐射 . . . 169
1.48 QTREE - 一周目 . . . 170

1.48.1 QTREE1 . . . 170
1.48.2 QTREE2 . . . 174
1.48.3 QTREE3 . . . 177
1.48.4 QTREE4 . . . 179
1.48.5 QTREE5 . . . 184
1.48.6 QTREE6 . . . 188
1.48.7 QTREE7 . . . 193

2 数据结构 200
2.1 树状数组 . . . 200
2.1.1 区间修改 + 区间求和 . . . 200
2.1.2 区间最值：分治思想 . . . 201
2.2 主席树 . . . 203
2.2.1 主席树维护静态树上第 k 大 . . . 205
2.3 归并树 . . . 208
2.4 Link Cut Tree . . . 210
2.5 线段树 . . . 214

3 动态规划 216
3.1 最长上升子序列 LIS：$O(nlogn)$ . . . 216

4 数论 217
4.1 快速幂 . . . 217
4.2 求组合数 . . . 217
4.3 求卡特兰数 . . . 218
4.4 矩阵快速幂 . . . 219
4.5 基础公式 . . . 220
4.5.1 多边形面积公式 . . . 220
4.5.2 平方和公式 . . . 220
4.5.3 阶乘分解 . . . 220
4.5.4 算术基本定理拓展 . . . 220
4.5.5 多项式性质 . . . 220

5 杂项 221
5.1 三分：$O(2log_3^n)$ . . . 221

5.2 RMQ 问题：st 表 221
5.3 K-约瑟夫变换 221

6 其它问题 222
6.1 IDE 配置 222
6.1.1 头文件 222
6.1.2 手动加栈 223
6.2 STL 相关 223
6.2.1 数据类型范围 223
6.2.2 基础 STL 的使用 223
6.2.3 cin 加速 224
6.2.4 卡常读入优化 224
6.2.5 交互题写法 225
6.3 Tip 226

1 图论

1.1 存图方式

1 邻接表

```
vector<pii> e[MAXN];
```

2 邻接矩阵

```
int e[MAXN][MAXN];
```

3 自定义结构体

```
struct edge {
    int u, v, w;
} e[MAXM * 2];
int cnt = 0;
```

4 前向星

当需要对一条无向边打标记时，可以从 2 开始存。形如 e[i].f = e[i∧1].f = 0。

```
struct edge {
    int to, next, w;
} e[MAXM * 2]; //双倍内存
int cnt = 0;
int head[MAXN];
void add_edge(int u, int v, int w = 0) { //从1开始存
    e[++cnt].to = v;
    e[cnt].next = head[u];
    e[cnt].w = w;
    head[u] = cnt;
}
void init(int n) {
    cnt = 0;
    fill(head, head + n + 5, 0);
}
```

1.2 并查集

1 路径压缩

```
int pre[MAXN];
int find(int x) {
    return x == pre[x] ? x : pre[x] = find(pre[x]);
}
void init(int n) {
    for(int i = 1; i <= n; i++)
        pre[i] = i;
}
```

2 启发式合并

按秩合并：对两个“大小”不一样的集合，把小的合并到大的上。

启发式合并：把按秩合并的“大小”具体化为高度。

```
const int MAXN = 10005;
int pre[MAXN], dep[MAXN];
void init(int n) {
    for(int i = 1; i <= n; i++) {
        pre[i] = i;
        dep[i] = 1;
    }
}
int find(int x) {
    return x == pre[x] ? x : find(pre[x]);
}
bool Union(int u, int v) {
    int r1 = find(u), r2 = find(v);
    if(r1 == r2)
        return false;
    if(dep[r1] > dep[r2])
        swap(r1, r2);
    pre[r1] = r2;
    if(dep[r1] == dep[r2])
        dep[r2]++;
```

```
    return true;
}
```

1.3 可持久化并查集

可持久化并查集的历史版本需要主席树来维护每一个版本中每一个点的父节点（并查集中的 $fa[x]$ ）。

模板题：

有 m 次操作。操作分为 3 种：

- 【1 a b】合并 a,b 所在集合；
- 【2 k】回到第 k 次操作（执行三种操作中的任意一种都记为一次操作）之后的状态；
- 【3 a b】询问 a,b 是否属于同一集合，如果是则输出 1，否则输出 0。

```
int fa[MAXN << 5];
int dep[MAXN << 5];
//---------------------------
int rt[MAXN], cnt = 0; // rt[i] 表示第i次操作的根节点
struct node {
    int L, R;
} tree[MAXN << 5];
#define lson tree[rt].L
#define rson tree[rt].R
// 建无操作的初始树
void build(int &rt, int L, int R) {
    rt = ++cnt;
    if(L == R) {
        fa[rt] = L; // 这里的rt代表[L,L]
        dep[rt] = 1;
        return;
    }
    int mid = (L + R) >> 1;
    build(lson, L, mid);
    build(rson, mid + 1, R);
}
// 从ori版本更新过来,把pos点的父节点改为Fa
void update(int &rt, int ori, int L, int R, int pos, int Fa) {
    rt = ++cnt;
```

```
25      if(L == R) {
26          fa[rt] = Fa;
27          dep[rt] = dep[ori];
28          return;
29      }
30      int Lson = tree[ori].L;
31      int Rson = tree[ori].R;
32      lson = Lson;
33      rson = Rson;
34      int mid = (L + R) >> 1;
35      if(pos <= mid)
36          update(lson, Lson, L, mid, pos, Fa);
37      else
38          update(rson, Rson, mid + 1, R, pos, Fa);
39  }
40  // 以pos为根的并查集深度+1
41  void add(int rt, int L, int R, int pos) {
42      if(L == R) {
43          dep[rt]++;
44          return;
45      }
46      int mid = (L + R) >> 1;
47      if(pos <= mid)
48          add(lson, L, mid, pos);
49      else
50          add(rson, mid + 1, R, pos);
51  }
52  // 查询在某个版本中pos点的标号
53  int query(int rt, int L, int R, int pos) {
54      if(L == R)
55          return rt;
56      int mid = (L + R) >> 1;
57      if(pos <= mid)
58          return query(lson, L, mid, pos);
59      else
60          return query(rson, mid + 1, R, pos);
61  }
62  // 查询某个版本中pos点的并查集根节点
63  int find_fa(int n, int rt, int pos) {
```

```
64      int x = query(rt, 1, n, pos);
65      if(pos == fa[x])
66          return pos;
67      return find_fa(n, rt, fa[x]);
68  }
69  int main() {
70      int n, m;
71      scanf("%d%d", &n, &m);
72      build(rt[0], 1, n);
73      for(int i = 1; i <= m; i++) {
74          int op;
75          scanf("%d", &op);
76          if(op == 1) {
77              int a, b;
78              scanf("%d%d", &a, &b);
79              rt[i] = rt[i - 1]; // 继承上个版本
80              int r1 = find_fa(n, rt[i], a);
81              int r2 = find_fa(n, rt[i], b);
82              if(r1 == r2)
83                  continue;
84              if(dep[r1] > dep[r2]) // 深度小的合并到深度大的上
85                  swap(r1, r2);
86              update(rt[i], rt[i - 1], 1, n, r1, r2);
87              if(dep[r1] == dep[r2]) // 深度相同,合并后深度会+1
88                  add(rt[i], 1, n, r2);
89          } else if(op == 2) {
90              int k;
91              scanf("%d", &k);
92              rt[i] = rt[k]; // 继承第k个版本
93          } else {
94              int a, b;
95              scanf("%d%d", &a, &b);
96              rt[i] = rt[i - 1]; // 继承上个版本
97              int r1 = find_fa(n, rt[i], a);
98              int r2 = find_fa(n, rt[i], b);
99              if(r1 == r2)
100                 printf("1\n");
101             else
102                 printf("0\n");
```

```
        }
    }
}
```

1.4 可撤销并查集

可撤销并查集就是把每次合并前的状态存入栈中，通过还原栈顶的状态即可实现撤销上一次的合并操作。因此合并时不能用路径压缩，要用启发式合并。

根据题目需求可以在 *node* 中加入新的维护信息，并在合并/撤销时维护。

```
struct node {
    int x, y, depx, depy;
    // 可添加其它维护信息
};
struct UFS {
    int fa[MAXN], dep[MAXN];
    stack<node>stk;
    void init(int n) {
        for(int i = 0; i <= n; i++) {
            fa[i] = i;
            dep[i] = 1;
        }
        while(!stk.empty())
            stk.pop();
    }
    int Find(int x) { // 非路径压缩
        return x == fa[x] ? x : Find(fa[x]);
    }
    bool Union(int x, int y) {
        x = Find(x);
        y = Find(y);
        if(x == y)
            return false;
        stk.push({x, y, dep[x], dep[y]});
        // 启发式合并
        if(dep[x] < dep[y])
            swap(x, y);
        fa[y] = x;
```

```
        if(dep[x] == dep[y])
            dep[x]++;
        return true;
    }
    void Undo() { // 撤销上一次合并
        node tmp = stk.top();
        stk.pop();
        fa[tmp.x] = tmp.x;
        fa[tmp.y] = tmp.y;
        dep[tmp.x] = tmp.depx;
        dep[tmp.y] = tmp.depy;
    }
} ufs;
```

1.5 最小生成树

MST 的性质：

在存在多棵最小生成树的图中，每棵最小生成树的边权值序列排序后是唯一的。

1.5.1 prim：$O(n^2)$

```
int g[MAXN][MAXN]; // 原图
int dis[MAXN];
int vis[MAXN];
int prim(int n) {
    for(int i = 0; i < n + 5; i++) {
        dis[i] = INF;
        vis[i] = 0;
    }
    int ans = 0;
    vis[1] = 1; // 从1开始
    for(int i = 1; i <= n; i++)
        dis[i] = g[1][i];
    for(int i = 0; i < n - 1; i++) { // 找到n-1条边
        pii tmp = mp(INF, -1);
        for(int j = 1; j <= n; j++) // 找未访问的最近节点
            if(!vis[j] && dis[j] < tmp.fi)
                tmp = mp(dis[j], j);
```

```
        int v = tmp.se;
        vis[v] = 1; // 用v更新
        for(int j = 1; j <= n; j++)
            if(!vis[j] && dis[j] > g[v][j])
                dis[j] = g[v][j];
        ans += tmp.fi;
    }
    return ans;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        for(int j = 0; j < n + 5; j++)
            g[i][j] = INF;
        g[i][i] = 0;
    }
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    init(n);
    for(int i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        w = min(w, g[u][v]); // 重边取最小值
        g[u][v] = g[v][u] = w;
    }
    printf("%d\n", prim(n));
}
```

1.5.2 prim 堆优化：$O(mlogn)$

```
int dis[MAXN];
int vis[MAXN];
int prim(int n, int m) {
    fill(dis, dis + n + 5, INF);
    fill(vis, vis + n + 5, 0);
    priority_queue<pii, vector<pii>, greater<pii> >q;
    dis[1] = 0; // 从1开始
    q.push(mp(0, 1));
```

```
    int sum = 0, cnt = 0;
    while(!q.empty() && cnt < n) {
        int d = q.top().fi, u = q.top().se;
        q.pop();
        if(vis[u])
            continue;
        cnt++;
        sum += d;
        vis[u] = 1; // 用v更新
        for(auto i : e[u]) {
            int v = i.fi, w = i.se;
            if(w < dis[v]) {
                dis[v] = w;
                q.push(mp(w, v));
            }
        }
    }
    return sum;
}
```

1.5.3 kruskal: $O(mlogm)$

```
struct edge {
    int u, v, w;
} e[MAXM];
//-------------------------------------------
// 并查集
//-------------------------------------------
int kruskal(int n, int m) {
    for(int i = 1; i <= n; i++)
        pre[i] = i;
    auto cmp = [&](edge a, edge b) {
        return a.w < b.w;
    };
    sort(e + 1, e + m + 1, cmp);
    int sum = 0, cnt = 0;
    for(int i = 1; i <= m; i++) {
        int r1 = find(e[i].u);
        int r2 = find(e[i].v);
```

```
        if(r1 == r2)
            continue;
        pre[r1] = r2;
        sum += e[i].w;
        if(++cnt == n - 1)
            break;
    }
    return sum;
}
```

1.5.4 boruvka：$O((n+m)logn)$

每次找出到所有点 x 的距离最小的边，使用这些边进行两两合并。

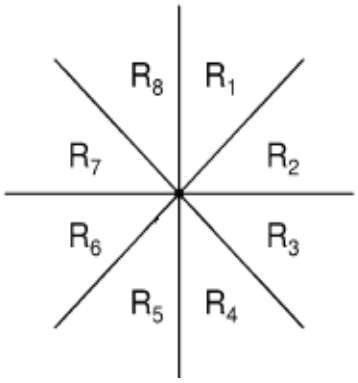
核心思想在于两两合并，可结合字典树求给定点权，边权为点权的异或值，的最小生成树。

```
struct edge {
    int u, v, w, f = 0;
} e[MAXM];
//-----------------------------------------
// 并查集
//-----------------------------------------
int minn[MAXN];
void check(int u, int id) { // 判断是否更优
    if(!minn[u]) {
        minn[u] = id;
        return;
    }
    int x = minn[u];
    if(e[id].w < e[x].w || e[id].w == e[x].w && id < x)
        minn[u] = id;
}
int boruvka(int n, int m) {
    for(int i = 1; i <= n; i++) // 并查集初始化
        pre[i] = i;
    int sum = 0;
    int update = 1; // 是否有更新
    while(update) { // 最多更新logn次
        update = 0;
```

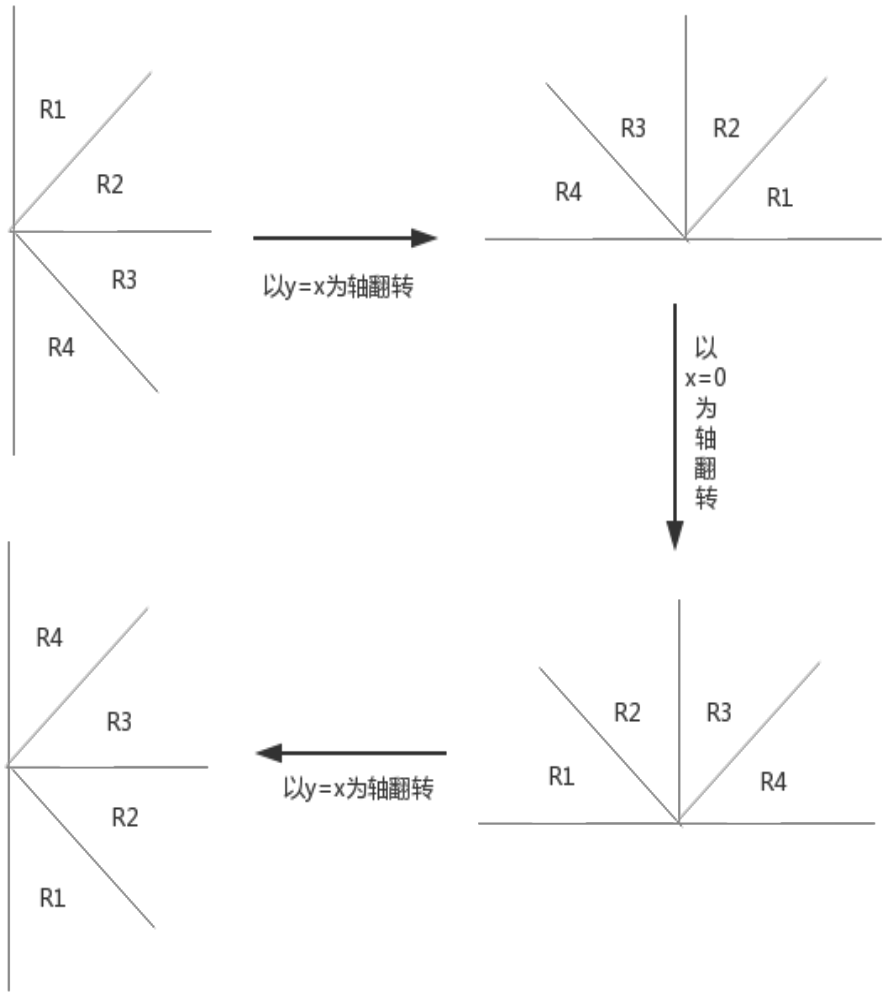
```
        // 找出离每个点最近的且编号最小的边
        fill(minn, minn + n + 5, 0);
        for(int i = 1; i <= m; i++) {
            int r1 = find(e[i].u);
            int r2 = find(e[i].v);
            if(e[i].f || r1 == r2) // 已用的和联通块内的不考虑
                continue;
            check(r1, i);
            check(r2, i);
        }
        for(int i = 1; i <= n; i++) { // 用找到的边更新
            int id = minn[i];
            if(id != 0 && !e[id].f) {
                update = 1;
                e[id].f = 1;
                int r1 = find(e[id].u);
                int r2 = find(e[id].v);
                pre[r1] = r2;
                sum += e[id].w;
            }
        }
    }
    return sum;
}
```

1.6 曼哈顿最小生成树：$O(nlogn + mlogm)$

首先，一个点把平面分成了 8 个部分：



对每个点（a，b），只考虑连接右侧 R1，R2，R3，R4 四个各自区域内的最近点。
对 R1 内的点（x，y），在满足 x >= a && y - x >= b - a 的点中取 x + y 最小的点。
对 R2 内的点（x，y），在满足 y >= b && y - x <= b - a 的点中取 x + y 最小的点。
对 R3 内的点（x，y），在满足 y <= b && x + y >= a + b 的点中取 x - y 最小的点。
对 R4 内的点（x，y），在满足 x >= a && x + y <= a + b 的点中取 x - y 最小的点。
第一个条件预处理 x 和 y 的所在位置，第二个条件用排序，用树状数组维护最近的点。
但是一个个写太麻烦了，可以用坐标旋转来优化代码量：



```
struct point {
    int x, y;
    int id, idx, idy;
} p[MAXN], q[MAXN];
int get_dis(point a, point b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}
bool cmp(point a, point b) { // 对R1内的点排序
    if(a.y - a.x != b.y - b.x)
        return a.y - a.x > b.y - b.x;
    return a.x > b.x;
}
//-------------------------------------------------
struct BIT {
    int n;
    pii a[MAXN * 2];
    void init(int _n) {
        n = _n;
        for(int i = 0; i < n + 5; i++)
            a[i] = {INF, 0};
    }
    int lowbit(int x) {
        return x & (-x);
    }
    pii ask(int x) { // 查询最小值
        pii ans = mp(INF, 0);
        for(; x >= 1; x -= lowbit(x))
            ans = min(ans, a[x]);
        return ans;
    }
    void update(int x, pii val) { // 更新最小值
        for(; x <= n; x += lowbit(x))
            a[x] = min(a[x], val);
    }
    pii ask_back(int x) {
        return ask(n - x + 1);
    }
    void update_back(int x, pii val) {
        update(n - x + 1, val);
```

```
    }
} tree;
//-----------------------------------------------------
// kruskal
//-----------------------------------------------------
int main() {
    // ...
    vi xy;
    for(int i = 1; i <= n; i++) {
        scanf("%d%d", &p[i].x, &p[i].y);
        p[i].id = i; // 点标号
        xy.pb(p[i].x);
        xy.pb(p[i].y);
        q[i] = p[i]; // q存初始坐标
    }
    sort(xy.begin(), xy.end()); // 给x,y排序,标记位置
    for(int i = 1; i <= n; i++) {
        p[i].idx = lower_bound(xy.begin(), xy.end(), p[i].x) - xy.begin() + 1;
        p[i].idy = lower_bound(xy.begin(), xy.end(), p[i].y) - xy.begin() + 1;
    }
    for(int R = 1; R <= 4; R++) {
        if(R == 2 || R == 4) // 以y=x为轴翻转
            for(int i = 1; i <= n; i++) {
                swap(p[i].x, p[i].y);
                swap(p[i].idx, p[i].idy);
            }
        if(R == 3) // 以x=0为轴翻转
            for(int i = 1; i <= n; i++) {
                p[i].x = -p[i].x;
                p[i].idx = 2 * n + 1 - p[i].idx;
            }
        tree.init(n * 2);
        sort(p + 1, p + n + 1, cmp);
        for(int i = 1; i <= n; i++) {
            pii tmp = tree.ask_back(p[i].idx);
            if(tmp.se != 0) {
                int u = p[i].id, v = tmp.se;
                add_edge(u, v, get_dis(q[u], q[v])); // 可能的边
            }
```

```
            tree.update_back(p[i].idx, {p[i].x + p[i].y, p[i].id});
        }
    }
    kruskal();
    // ...
}
```

1.7 最小瓶颈生成树：$O(n+m)$

最小瓶颈生成树是使树上最大边权值最小的生成树。

最小生成树一定是最小瓶颈生成树，最小瓶颈生成树不一定是最小生成树，所以最小瓶颈生成树的限制更松。

二分瓶颈值，check 判断图的联通性。图若连通，则瓶颈不超过 mid，只需考虑不超过 mid 的边；图若不连通，则瓶颈超过 mid，缩点后考虑大于 mid 的边。

```
struct edge {
    int u, v, w;
} e1[MAXM * 2]; // 用于删边
int cnt = 0;
vi e2[MAXN]; // 用于tarjan
//-------------------------------------------
// 路径压缩并查集
//-------------------------------------------
int dfn[MAXN], tim; // 点的遍历序号
int low[MAXN]; // 环的最小序号
int stk[MAXN], top = 0; // 手写栈
int ins[MAXN]; //是否在栈内
int col[MAXN], cnum = 0; // 染色
void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    stk[++top] = u;//进栈
    ins[u] = 1;
    for(auto v : e2[u]) {
        if(!dfn[v]) { // 未访问
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if(ins[v])  // v还在栈内
            low[u] = min(low[u], dfn[v]);
```

```
    }
    if(dfn[u] == low[u]) { // 是环的开头
        cnum++;
        while(stk[top + 1] != u) {
            col[stk[top]] = cnum; // 染色
            ins[stk[top--]] = 0; // 出栈
        }
    }
}
void tarjan_init(int n) {
    tim = top = cnum = 0;
    for(int i = 1; i <= n; i++)
        dfn[i] = low[i] = ins[i] = col[i] = 0;
}
//-----------------------------------------
int neck(int n, int m) {
    int L = 0, R = INF;
    while(L < R) {
        int mid = (L + R) / 2;
        int tot = n; // 连通分量的数量
        ufs_init(n); // 并查集初始化
        for(int i = 1; i <= m; i++)
            if(e1[i].w <= mid) {
                int r1 = find(e1[i].u), r2 = find(e1[i].v);
                if(r1 != r2) {
                    pre[r1] = r2;
                    tot--;
                }
            }
        if(tot == 1) { // 图是连通的
            R = mid;
            int cntm = 0;
            for(int i = 1; i <= m; i++) { // 去掉大于mid的边
                if(e1[i].w > mid)
                    continue;
                e1[++cntm].u = e1[i].u;
                e1[cntm].v = e1[i].v;
                e1[cntm].w = e1[i].w;
            }
```

```
            m = cntm;
        } else { // 图不连通
            L = mid + 1;
            for(int i = 1; i <= n; i++) // 邻接表初始化
                e2[i].clear();
            for(int i = 1; i <= m; i++) {
                int u = e1[i].u, v = e1[i].v, w = e1[i].w;
                if(w <= mid)
                    e2[u].pb(v);
            }
            tarjan_init(n); // tarjan初始化
            for(int i = 1; i <= n; i++) // 染色缩点
                if(!dfn[i])
                    tarjan(i);
            int cntm = 0;
            for(int i = 1; i <= m; i++) { // 更新现有边
                int u = col[e1[i].u];
                int v = col[e1[i].v];
                if(u == v)
                    continue;
                e1[++cntm] = {u, v, e1[i].w};
            }
            n = cnum;
            m = cntm;
        }
    }
    return L;
}
```

1.8 次小生成树

1.8.1 非严格次小生成树

1 prim：$O(n^2)$

先求一遍最小生成树，在求的过程中标记一个相邻节点（这些边覆盖了整棵树）。

然后枚举所有的点对 (u, v)，用非树边 $< u, v >$ 替换 MST 上 u 到 v 的最大边。

```
int g[MAXN][MAXN]; // 原图
```

```cpp
int maxx[MAXN][MAXN]; // MST上两点间最大边权
int dis[MAXN];
int vis[MAXN];
int pre[MAXN]; // 标记拓展当前点的节点
int prim(int n) {
    int ans = 0;
    vis[1] = 1;
    for(int i = 1; i <= n; i++) {
        dis[i] = g[1][i];
        pre[i] = 1;
    }
    for(int i = 0; i < n - 1; i++) {
        pii tmp = mp(INF, -1);
        for(int j = 1; j <= n; j++)
            if(!vis[j] && dis[j] < tmp.fi)
                tmp = mp(dis[j], j);
        int v = tmp.se;
        vis[v] = 1;
        for(int j = 1; j <= n; j++) {
            if(vis[j] && v != j) {
                int maxw = max(maxx[j][pre[v]], dis[v]);
                maxx[v][j] = maxx[j][v] = maxw;
            }
            if(!vis[j] && dis[j] > g[v][j]) {
                dis[j] = g[v][j];
                pre[j] = v; // j由v拓展
            }
        }
        ans += tmp.fi;
    }
    return ans;
}
int prim_2(int n, int sum) {
    int ans = INF;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++) {
            if(i == j || pre[i] == j || pre[j] == i)
                continue;
            // 枚举两点i,j,用非树边替换树边
```

```
            ans = min(ans, sum - maxx[i][j] + g[i][j]);
        }
    return ans;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        for(int j = 0; j < n + 5; j++) {
            g[i][j] = INF;
            maxx[i][j] = 0;
        }
        g[i][i] = vis[i] = 0;
        dis[i] = INF;
    }
}
int main() {
    // ...
    init(n);
    // ...
    int min1 = prim(n);
    int min2 = prim_2(n, min1);
    // ...
}
```

2 kruskal：$O(mlogm)$

先求一遍最小生成树，在求的过程中标记树边。

遍历所有非树边 $< u, v >$，用 LCA 求 MST 上 u 到 v 的最大边权。

```
// 路径压缩并查集
//----------------------------------------
struct edge {
    int u, v, w, f;
} e[MAXM]; // 图边
vector<pii>te[MAXN]; // 树边
int kruskal(int n, int m) {
    auto cmp = [&](edge a, edge b) {
        return a.w < b.w;
    };
    sort(e + 1, e + m + 1, cmp);
```

```
12      int sum = 0, cnt = 0;
13      for(int i = 1; i <= m; i++) {
14          int r1 = find(e[i].u);
15          int r2 = find(e[i].v);
16          if(r1 == r2)
17              continue;
18          pre[r1] = r2;
19          sum += e[i].w;
20          e[i].f = 1; // 标记树边
21          te[e[i].u].pb({e[i].v, e[i].w}); // 存下树边
22          te[e[i].v].pb({e[i].u, e[i].w});
23          if(++cnt == n - 1)
24              break;
25      }
26      return sum;
27  }
28  //----------------------------------------
29  // 倍增求LCA
30  //----------------------------------------
31  int kruskal_2(int n, int m, int sum) {
32      int ans = INF;
33      dfs(1, 0); // LCA预处理
34      for(int i = 1; i <= m; i++) { // 遍历非树边
35          if(e[i].f)
36              continue;
37          int LCA = lca(e[i].u, e[i].v); // 查找u到v的路径上的最大边权
38          ans = min(ans, sum - LCA + e[i].w);
39      }
40      return ans;
41  }
42  void init(int n, int m) {
43      for(int i = 0; i < n + 5; i++) {
44          pre[i] = i;
45          for(int j = 0; j < 25; j++)
46              dis[i][j] = 0;
47          te[i].clear();
48      }
49      for(int i = 0; i < m + 5; i++)
50          e[i].f = 0;
```

```
}
int main() {
    // ...
    init(n, m);
    int min1 = kruskal(n, m);
    int min2 = kruskal_2(n, m, min1);
    // ...
}
```

1.8.2 严格次小生成树

求严格次小生成树的关键是要求在 MST 中，u 到 v 的路径上的最大值和严格次大值。需要对求 LCA 的代码进行修改。

```
int dep[MAXN], f[MAXN][25];
ll maxd[MAXN][25][3]; // [1]存严格次大值,[2]存最大值
void change(ll *a, ll *b) { // 用b数组更新a数组
    if(a[2] != b[2])
        a[0] = b[2];
    else
        a[0] = b[1];
    sort(a, a + 3);
}
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for(int i = 1; (1 << i) <= dep[u]; i++) {
        f[u][i] = f[f[u][i - 1]][i - 1];
        change(maxd[u][i], maxd[u][i - 1]);
        change(maxd[u][i], maxd[f[u][i - 1]][i - 1]);
    }
    for(int i = head[u]; i; i = e[i].next) {
        int v = e[i].to;
        if(v == fa || e[i].f == 0)
            continue;
        maxd[v][0][2] = e[i].w;
        dfs(v, u);
    }
```

```
}
ll get_max[3]; // 存lca(x,y)的结果
void lca(int x, int y) { // 求路径上的最大值和严格次大值
    get_max[0] = get_max[1] = get_max[2] = 0;
    if(dep[x] < dep[y])
        swap(x, y);
    for(int i = 20; i >= 0; i--) {
        if(dep[f[x][i]] >= dep[y]) {
            change(get_max, maxd[x][i]);
            x = f[x][i];
        }
        if(x == y)
            return;
    }
    for(int i = 20; i >= 0; i--)
        if(f[x][i] != f[y][i]) {
            change(get_max, maxd[x][i]);
            change(get_max, maxd[y][i]);
            x = f[x][i];
            y = f[y][i];
        }
    change(get_max, maxd[x][0]);
    change(get_max, maxd[y][0]);
}
```

1 prim 堆优化：$O((n+m)logn)$

先求一遍最小生成树，在求的过程中标记树边。

遍历所有非树边 $< u, v >$，用 LCA 求 MST 上 u 到 v 的严格次大值。

```
ll dis[MAXN];
int vis[MAXN];
int pre_eid[MAXN]; // 拓展当前节点的边的编号
#define pli pair<ll,int>
ll prim(int n) {
    priority_queue<pli, vector<pli>, greater<pli> >q;
    dis[1] = 0;
    q.push(mp(0, 1));
    ll sum = 0;
```

```
    int cnt = 0;
    while(!q.empty() && cnt < n) {
        ll d = q.top().fi;
        int u = q.top().se;
        q.pop();
        if(vis[u])
            continue;
        vis[u] = 1;
        cnt++;
        sum += d;
        e[pre_eid[u]].f = e[pre_eid[u] ^ 1].f = 1; // 标记树边
        for(int i = head[u]; i; i = e[i].next)
            if(e[i].w < dis[e[i].to]) {
                dis[e[i].to] = e[i].w;
                q.push(mp(e[i].w, e[i].to));
                pre_eid[e[i].to] = i; // 标记拓展边
            }
    }
    return sum;
}
//----------------------------------------
// LCA倍增求严格次大值
//----------------------------------------
ll prim_2(int n, ll sum) {
    dfs(1, 0); // LCA初始化
    ll ans = LLINF;
    for(int u = 1; u <= n; u++) { // 遍历非树边
        for(int i = head[u]; i; i = e[i].next) {
            if(e[i].f)
                continue;
            int v = e[i].to;
            lca(u, v);
            ll tmp = sum - get_max[2] + e[i].w;
            if(tmp == sum) // 严格小于,不能等于
                tmp = sum - get_max[1] + e[i].w;
            ans = min(ans, tmp);
        }
    }
    return ans;
```

```
}
void init(int n) {
    cnt = 1; // 注意前向星从2开始存!
    for(int i = 0; i < n + 5; i++) {
        dis[i] = LLINF;
        head[i] = vis[i] = 0;
        for(int j = 0; j < 25; j++)
            for(int k = 0; k < 3; k++)
                maxd[i][j][k] = 0;
    }
}
int main() {
    // ...
    init(n);
    ll min1 = prim(n);
    ll min2 = prim_2(n, min1);
    // ...
}
```

2 kruskal：$O(m(logn+logm))$

先求一遍最小生成树，在求的过程中标记树边。

遍历所有非树边 $<u,v>$，用 LCA 求 MST 上 u 到 v 的严格次大值。

```
// 路径压缩并查集
//-----------------------------------------
struct edge {
    int u, v, f;
    ll w;
} e[MAXM]; // 图边
#define pil pair<int,ll>
vector<pil>te[MAXN]; // 树边
ll kruskal(int n, int m) {
    auto cmp = [&](edge a, edge b) {
        return a.w < b.w;
    };
    sort(e + 1, e + m + 1, cmp);
    ll sum = 0;
    int cnt = 0;
```

```
    for(int i = 1; i <= m; i++) {
        int r1 = find(e[i].u);
        int r2 = find(e[i].v);
        if(r1 == r2)
            continue;
        pre[r1] = r2;
        sum += e[i].w;
        e[i].f = 1; // 标记树边
        te[e[i].u].pb({e[i].v, e[i].w}); // 存下树边
        te[e[i].v].pb({e[i].u, e[i].w});
        if(++cnt == n - 1)
            break;
    }
    return sum;
}
//----------------------------------------
// LCA倍增求严格次大值
//----------------------------------------
ll kruskal_2(int n, int m, ll sum) {
    dfs(1, 0); // LCA初始化
    ll ans = LLINF;
    for(int i = 1; i <= m; i++) { // 遍历非树边
        if(e[i].f)
            continue;
        lca(e[i].u, e[i].v);
        ll tmp = sum - get_max[2] + e[i].w;
        if(tmp == sum)
            tmp = sum - get_max[1] + e[i].w;
        ans = min(ans, tmp);
    }
    return ans;
}
void init(int n, int m) {
    for(int i = 0; i < n + 5; i++) {
        pre[i] = i;
        for(int j = 0; j < 25; j++)
            for(int k = 0; k < 3; k++)
                maxd[i][j][k] = 0;
        te[i].clear();
```

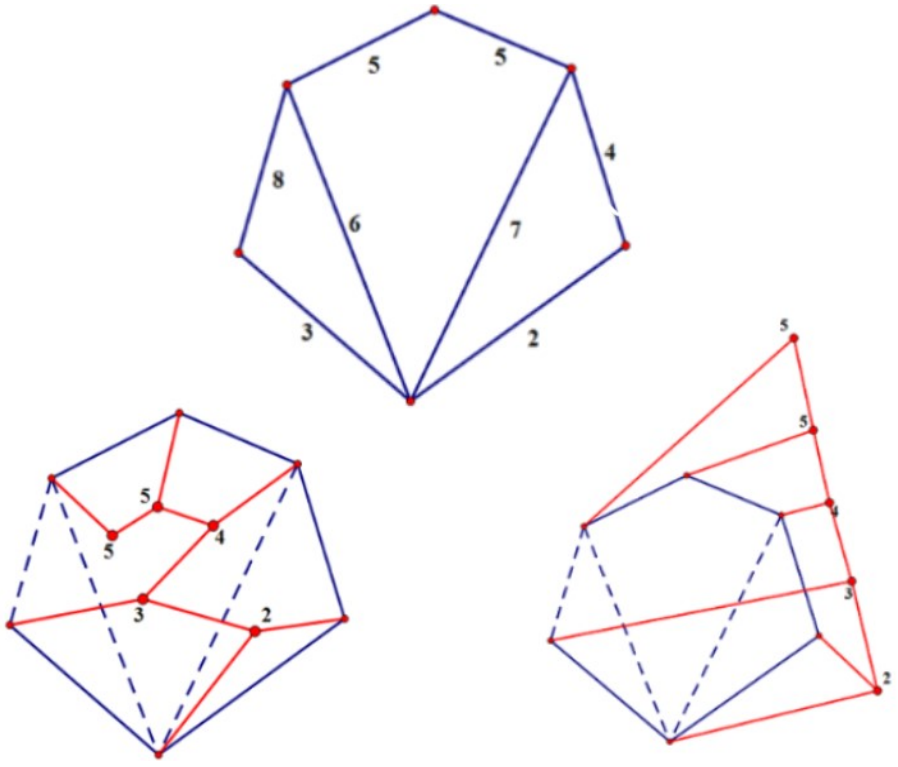
```
    }
    for(int i = 0; i < m + 5; i++)
        e[i].f = 0;
}
int main() {
    // ...
    init(n, m);
    ll min1 = kruskal(n, m);
    ll min2 = kruskal_2(n, m, min1);
    // ...
}
```

1.9 kruskal 重构树

原树中的点为重构树中的叶节点，原树中 n-1 条边变成重构树中新的 n-1 个点。

性质：

1. 原树中两点路径上的边权最大值为重构树上两点 LCA 的权值
2. 按最小生成树建立的话，重构树是一个大根堆



```cpp
const int MAXN = !!!; // 注意MAXN开2倍
//---------------------------------------------
// 结构体e1
// 邻接表e2
// 并查集
//---------------------------------------------
int val[MAXN]; // 存重构树的点权
void ex_kruskal(int n, int m) {
    ufs.init(2 * n); // 并查集初始化
    auto cmp = [&](edge a, edge b) {
        return a.w < b.w;
    };
    sort(e1 + 1, e1 + m + 1, cmp);
    int id = n; // 新点的编号
    for(int i = 1; i <= m; i++) {
        int r1 = find(e1[i].u);
        int r2 = find(e1[i].v);
        if(r1 == r2)
            continue;
        pre[r1] = pre[r2] = ++id;
        val[id] = e[i].w; // 点权为边权
        e2[id].pb(r1); // 存重构树的树边
        e2[r1].pb(id);
        e2[id].pb(r2);
        e2[r2].pb(id);
        if(id == 2 * n - 1)
            break;
    }
}
```

1.10 最小斯坦纳树

定义：

给定包含 n 个点和 m 条带权边的连通图 G，其中有 k 个关键点，最小斯坦纳树是一棵包含 k 个关键点在内的总边权最小的生成树。

思路：

用 $dp(u,S)$ 表示以 u 为根的一棵树，包含集合 S 中所有点的最小边权值和。

有两种状态转移：

$dp(u,T)+dp(u,S-T)\to dp(u,S)\quad(T\subset S)$

$dp(u,S)+e(u,v,w)\to dp(v,S)$

对第一种转移，枚举所有集合 S 的所有子集和树的根结点，复杂度为 $O(n\times 3^k)$ 。

对第二种转移，枚举所有集合 S 并跑最短路，复杂度为 $O(T(\text{最短路})\times 2^k)$ 。

这里用堆优化的 dijkstra 实现，总时间复杂度为 $O(n\times 3^k+(n+m)logm\times 2^k)$ 。

```
vector<pii> e[MAXN];
int key[15]; // k<=10
int dp[MAXN][1055];
// ------------------------------------------------------
int preu[MAXN][1055];
int pres[MAXN][1055];
void dfs(int u, int s) { // 回溯更新路径
    if(dp[u][s] == INF)
        return;
    printf("u: %d s: %d dp: %d\n", u, s, dp[u][s]);
    if(pres[u][s]) {
        int subs = pres[u][s];
        dfs(u, subs);
        dfs(u, s ^ subs);
        return;
    }
    dfs(preu[u][s], s);
}
// ------------------------------------------------------
int dis[MAXN], vis[MAXN];
priority_queue<pii, vector<pii>, greater<pii> >q;
void dijkstra(int n, int s) {
    fill(dis, dis + n + 5, INF);
    fill(vis, vis + n + 5, 0);
    while(!q.empty()) {
        int u = q.top().se;
        q.pop();
        if(vis[u])
            continue;
        vis[u] = 1;
        for(auto i : e[u]) {
```

```
            int v = i.fi, w = i.se;
            if(dp[v][s] > dp[u][s] + w) {
                dp[v][s] = dp[u][s] + w;
                q.push({dp[v][s], v});
                // pres[v][s] = 0;
                // preu[v][s] = u;
            }
        }
    }
}
void init(int n, int k) {
    for(int i = 0; i < n + 5; i++) {
        e[i].clear();
        for(int s = 0; s < (1 << k) + 5; s++) {
            dp[i][s] = INF;
            // preu[i][s] = pres[i][s] = 0;
        }
    }
}
int main() {
    int n, m, k;
    scanf("%d%d%d", &n, &m, &k);
    init(n, k);
    while(m--) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        e[u].pb({v, w});
        e[v].pb({u, w});
    }
    for(int i = 1; i <= k; i++) {
        scanf("%d", key + i);
        dp[key[i]][1 << (i - 1)] = 0;
    }
    for(int s = 1; s < (1 << k); s++) { // 枚举集合
        for(int u = 1; u <= n; u++) { // 枚举根节点
            for(int subs = s & (s - 1); subs; subs = s & (subs - 1)) // 枚举子
                集
                if(dp[u][s] > dp[u][subs] + dp[u][s ^ subs]) {
                    dp[u][s] = dp[u][subs] + dp[u][s ^ subs];
```

```
                    // preu[u][s] = 0;
                    // pres[u][s] = subs;
                }
            if(dp[u][s] != INF)
                q.push({dp[u][s], u});
        }
        dijkstra(n, s);
    }
    // dfs(key[1], (1 << k) - 1);
    printf("%d\n", dp[key[1]][(1 << k) - 1]);
}
```

1.11 最短路

1.11.1 Floyd：$O(n^3)$

```
int dis[MAXN][MAXN];
void floyd(int n) {
    // 可以处理负边,无法处理负环
    for(int k = 1; k <= n; k++)
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
}
```

1.11.2 dijkstra：$O(n^2)$

dijkstra 用于求单源最短路径，无法处理负边。

从起点开始向外拓展，每次找到离已访问节点最近的节点，更新 dis 数组。

```
int g[MAXN][MAXN]; // 原图
int dis[MAXN];
int vis[MAXN];
void dijkstra(int n, int s) {
    for(int i = 0; i < n + 5; i++) {
        dis[i] = INF;
        vis[i] = 0;
    }
```

```
    for(int i = 1; i <= n; i++) // 从s开始
        dis[i] = g[s][i];
    for(int i = 0; i < n - 1; i++) {
        pii tmp = mp(INF, -1);
        for(int j = 1; j <= n; j++) // 找未访问的最近节点
            if(!vis[j] && dis[j] < tmp.fi)
                tmp = mp(dis[j], j);
        if(tmp.se == -1)
            return;
        int v = tmp.se;
        vis[v] = 1;
        for(int j = 1; j <= n; j++) // 用v更新
            if(!vis[j] && dis[v] + g[v][j] < dis[j])
                dis[j] = dis[v] + g[v][j];
    }
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        for(int j = 0; j < n + 5; j++)
            g[i][j] = INF;
        g[i][i] = 0;
    }
}
```

1.11.3 dijkstra 堆优化：$O((n+m)logm)$

堆优化是指在每次找离已访问节点最近的节点时，用优先队列处理。

```
int dis[MAXN];
int vis[MAXN];
priority_queue<pii, vector<pii>, greater<pii> >q;
void dijkstra(int n, int s) {
    fill(dis, dis + n + 5, INF);
    fill(vis, vis + n + 5, 0);
    dis[s] = 0; // 从s开始
    q.push(mp(0, s));
    while(!q.empty()) {
        int x = q.top().se;
```

```
        q.pop();
        if(vis[x])
            continue;
        vis[x] = 1;
        for(auto i : e[x]) { // 用x更新
            int v = i.fi, w = i.se;
            if(dis[v] > dis[x] + w) {
                dis[v] = dis[x] + w;
                q.push(mp(dis[v], v));
            }
        }
    }
}
```

1.11.4 bellman_ford：$O(nm)$

bellman_ford 用于求单源最短路，可以处理负边，不能处理负环，能到达负环的源点不存在最短路（无限小）。

算法每次遍历所有边对 dis 进行更新，如果没有负环，最多更新 n-1 次。如果第 n 次还能更新，则说明存在负环。

bellman_ford 可以打印负环路径，在算法中需要标记拓展当前点的节点。

```
// 结构体存边
//----------------------------------------
int dis[MAXN];
int pre[MAXN]; // 存更新当前点的节点
int bellman_ford(int n, int s) {
    fill(dis, dis + n + 5, INF);
    fill(pre, pre + n + 5, 0);
    dis[s] = 0;
    for(int i = 0; i < n - 1; i++) {
        int update = 0;
        for(int j = 1; j <= cnt; j++) { // 遍历所有边
            int u = e[j].u, v = e[j].v, w = e[j].w;
            if(dis[u] + w < dis[v]) {
                dis[v] = dis[u] + w;
                pre[v] = u;
                update = 1;
```

```
            }
        }
        if(!update) // 没有更新,结束
            break;
    }
    for(int j = 1; j <= cnt; j++) {
        int u = e[j].u, v = e[j].v, w = e[j].w;
        if(dis[u] + w < dis[v]) // 还能更新,有负环
            return u;
    }
    return 0;
}
int vis[MAXN];
vi get_path(int n, int s) { // 打印负环路径
    vi ans;
    int x = bellman_ford(n, s);
    if(!x) // 无负环
        return ans;
    fill(vis, vis + n + 5, 0);
    while(!vis[x]) { // 保证x回到负环上
        vis[x] = 1;
        x = pre[x];
    }
    int now = x;
    ans.pb(now); // x在ans中会出现2次
    do { // 逆向遍历负环
        now = pre[now];
        ans.pb(now);
    } while(now != x);
    reverse(ans.begin(), ans.end());
    return ans;
}
```

1.11.5 spfa: $O(nm)$

spfa 用于求单源最短路，可以处理负边，可以判断负环。

算法每次取出队列的顶部用于更新，任意时刻队列中都不会有两个相同的点，如果没有负环，每个点最多入队 n 次，否则说明有负环。

```cpp
int vis[MAXN]; // 是否在队列中
int dis[MAXN];
int qnum[MAXN]; // 入队次数
bool spfa(int n, int s) {
    for(int i = 0; i < n + 5; i++) {
        dis[i] = INF;
        vis[i] = qnum[i] = 0;
    }
    queue<int>q;
    q.push(s);
    vis[s] = 1;
    dis[s] = 0;
    qnum[s] = 1;
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for(auto i : e[u]) {
            int v = i.fi, w = i.se;
            if(dis[u] + w < dis[v]) {
                dis[v] = dis[u] + w;
                if(!vis[v]) {
                    q.push(v);
                    vis[v] = 1;
                    if(++qnum[v] > n) // 存在负环
                        return false;
                }
            }
        }
    }
    return true;
}
```

1.12 k 短路

1.12.1 A* 算法：复杂度玄学

A* 算法定义了一个对当前状态 x 的估价函数 $f(x) = g(x) + h(x)$，其中 $g(x)$ 为从 s 到达 x 已经走的距离，$h(x)$ 为从 x 到达 t 的最短路。每次取出 $f(x)$ 最优的状态，扩展其所有子状态，可以用优先队列来维护。当我们访问到一个结点第 k 次时，对应的状态的 $g(x)$ 就是从 s 到该结点的第 k 短路。

当图的形态是一个 n 元环的时候，算法复杂度为 $O(knlogn)$。

```
namespace k_path {
int n;
vector<pii>e1[MAXN], e2[MAXN];
int dis[MAXN], vis[MAXN];
void dijkstra(int s) { // 倒跑最短路
    priority_queue<pii, vector<pii>, greater<pii> >q;
    dis[s] = 0;
    q.push(mp(0, s));
    while(!q.empty()) {
        int u = q.top().se;
        q.pop();
        if(vis[u])
            continue;
        vis[u] = 1;
        for(auto tmp : e1[u]) {
            int v = tmp.fi, w = tmp.se;
            if(dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                q.push(mp(dis[v], v));
            }
        }
    }
}
struct node {
    int u, d;
    vi path;
    bool operator < (const node& x) const { // 最小字典序要求
        int f1 = d + dis[u];
        int f2 = x.d + dis[x.u];
```

```
30          if(f1 != f2) // 按已走的路+到终点期望的路排序
31              return f1 > f2;
32          return path > x.path;
33      }
34  };
35  int tim[MAXN]; // 每个点被访问的次数
36  vi ans; // 存k短路的路径
37  int Astar(int s, int t, int k) { // 返回k短路的距离
38      dijkstra(t);
39      priority_queue<node>q;
40      q.push({s, 0, vi{s}});
41      while(!q.empty()) {
42          node tmp = q.top();
43          q.pop();
44          int u = tmp.u, pred = tmp.d;
45          tim[u]++;
46          if(u == t && tim[u] == k) { // 第k次到t
47              ans = tmp.path;
48              return pred;
49          }
50          for(auto nxt : e2[u]) {
51              int v = nxt.fi, w = nxt.se;
52              int f = 1;
53              for(auto i : tmp.path)
54                  if(i == v)
55                      f = 0;
56              if(!f) // 不能走已经走过的点
57                  continue;
58              vi path = tmp.path;
59              path.pb(v);
60              q.push({v, pred + w, path});
61          }
62      }
63      return -1;
64  }
65  void add_edge(int u, int v, int w) {
66      e2[u].pb({v, w});
67      e1[v].pb({u, w}); // 存反向边
68  }
```

```
void init(int _n) {
    n = _n;
    for(int i = 0; i < n + 5; i++) {
        e1[i].clear();
        e2[i].clear();
        ans.clear();
        dis[i] = INF;
        vis[i] = tim[i] = 0;
    }
}
}
int main() {
    int n, m, k, s, t;
    scanf("%d%d%d%d%d", &n, &m, &k, &s, &t);
    k_path::init(n);
    for(int i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        k_path::add_edge(u, v, w); // 有向边
    }
    if(k_path::Astar(s, t, k) == -1) {
        printf("No");
        return 0;
    }
    vi ans = k_path::ans;
    for(int i = 0; i < SZ(ans); i++)
        printf(i == 0 ? "%d" : "-%d", ans[i]);
}
```

1.12.2 可持久化可并堆优化：$O(T(dijkstra) + nlogn + klogn)$

不是很懂，不会求具体路径，先存个板子 QAQ

```
namespace k_path {
struct Tree {
    Tree *ls, *rs;
    int dist;
    int val;
```

```
    int back;
    Tree(int d = 0, int v = 0, int b = 0): dist(d), val(v), back(b) {
        ls = rs = NULL;
    }
} *root[MAXN];
struct Left_Side_Tree {
    Tree* merge(Tree *x, Tree *y) {
        if(!x || !y)
            return x ? x : y;
        if(x->val > y->val)
            swap(x, y);
        Tree *New = new Tree;
        *New = *x;
        New->rs = merge(New->rs, y);
        if(!New->ls || New->ls->dist < New->rs->dist)
            swap(New->ls, New->rs);
        if(New->rs)
            New->dist = New->rs->dist + 1;
        return New;
    }
    Tree* push(Tree *x, int v, int b) {
        Tree *New = new Tree(0, v, b);
        return merge(x, New);
    }
} heap;
struct edge {
    int u, v, w;
};
vector<edge>ze, fe; // 存边和反向边
vi zeid[MAXN], feid[MAXN]; // 存边的序号
void add_edge(int u, int v, int w) {
    ze.pb({u, v, w});
    zeid[u].pb(SZ(ze) - 1);
    fe.pb({v, u, w});
    feid[v].pb(SZ(fe) - 1);
}
int fa[MAXN], fa_eid[MAXN];
int dis[MAXN], vis[MAXN];
void dijkstra(int s) {
```

```
45      priority_queue<pii, vector<pii>, greater<pii> >q;
46      dis[s] = 0;
47      fa_eid[s] = -1;
48      q.push(mp(0, s));
49      while(!q.empty()) {
50          int u = q.top().se;
51          q.pop();
52          if(vis[u])
53              continue;
54          vis[u] = 1;
55          for(auto id : feid[u]) {
56              edge e = fe[id];
57              int v = e.v, w = e.w;
58              if(dis[v] > dis[u] + w) {
59                  dis[v] = dis[u] + w;
60                  fa[v] = u;
61                  fa_eid[v] = id;
62                  q.push({dis[v], v});
63              }
64          }
65      }
66  }
67  void dfs(int u) {
68      if(fa[u])
69          root[u] = root[fa[u]];
70      for(auto id : zeid[u]) {
71          if(id == fa_eid[u])
72              continue;
73          edge e = ze[id];
74          int v = e.v;
75          root[u] = heap.push(root[u], e.w + dis[v] - dis[u], v);
76      }
77      for(auto id : feid[u]) {
78          edge e = fe[id];
79          int v = e.v;
80          if(id == fa_eid[v])
81              dfs(v);
82      }
83  }
```

```
#define piT pair<int,Tree*>
int solve(int s, int t, int k) {
    dijkstra(t);
    if(dis[s] == INF)
        return -1;
    if(--k == 0)
        return dis[s];
    dfs(t);
    priority_queue<piT, vector<piT>, greater<piT> >q;
    if(root[s])
        q.push({dis[s] + root[s]->val, root[s]});
    while(!q.empty()) {
        piT Now = q.top();
        q.pop();
        int pred = Now.fi;
        Tree *rt = Now.se;
        if(--k == 0)
            return pred;
        Tree *next = root[rt->back];
        Tree *ls = rt->ls;
        Tree *rs = rt->rs;
        if(next)
            q.push({pred + next->val, next});
        if(ls)
            q.push({pred + ls->val - rt->val, ls});
        if(rs)
            q.push({pred + rs->val - rt->val, rs});
    }
    return -1;
}
void init(int n) {
    ze.clear();
    fe.clear();
    for(int i = 0; i < n + 5; i++) {
        dis[i] = INF;
        vis[i] = fa[i] = fa_eid[i] = 0;
        zeid[i].clear();
        feid[i].clear();
        root[i] = NULL;
```

```
    }
}
}
int main() {
    int n, m, k, s, t;
    scanf("%d%d%d%d%d", &n, &m, &k, &s, &t);
    k_path::init(n);
    for(int i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        k_path::add_edge(u, v, w);
    }
    printf("%d\n", k_path::solve(s, t, k));
}
```

1.13 树的直径

性质：树的所有直径拥有相同的中点，中点可能是某个顶点，也可能在某条边内部（此时所有的直径必然经过这条边）。

1.13.1 两次 dfs/bfs

从任意一点出发，找离它最远的点 x ，再从 x 点出发，找离它最远的点 y 。x 到 y 就是树的一条直径。可以通过两个端点找到直径的中点。

```
vi e[MAXN];
int dis[MAXN];
void dfs(int u, int fa, int d) {
    dis[u] = d;
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs(v, u, d + 1);
    }
}
int getDia(int n) {
    fill(dis, dis + n + 5, 0);
    dfs(1, 0, 0);
```

```
    int x = 1;
    for(int i = 1; i <= n; i++)
        if(dis[i] > dis[x])
            x = i;
    fill(dis, dis + n + 5, 0);
    dfs(x, 0, 0);
    int ans = 0;
    for(int i = 1; i <= n; i++)
        ans = max(ans, dis[i]);
    return ans;
}
```

1.13.2 树形 dp

用 $dis[u][0]$ 和 $dis[u][1]$ 分别表示以 u 为根的子树中，离 u 最远的叶子节点 a 和次远的叶子节点 b 离 u 的距离。树的直径即为 $max\{dis[u][0]+dis[u][1]\}$ 。缺点是找不到直径的端点和中点。

```
vi e[MAXN];
int ans = 0;
int dis[MAXN][2];
void dfs(int u, int fa) {
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs(v, u);
        if(dis[u][0] < dis[v][1] + 1)
            dis[u][0] = dis[v][1] + 1;
        sort(dis[u], dis[u] + 2);
    }
    ans = max(ans, dis[u][0] + dis[u][1]);
}
int getDia(int n) {
    for(int i = 0; i < n + 5; i++)
        dis[i][0] = dis[i][1] = 0;
    ans = 0;
    dfs(1, 0);
    return ans;
}
```

1.14 图的直径

这里的图只考虑无向图。

图的直径是指任意两点间最短路的最大值，做两次 bfs 找到两个端点即可，与求树的直径类似。

1.15 最小直径生成树

最小直径生成树，指的是无向图的所有生成树中直径最小的生成树。

求解直径最小生成树，首先需要找到图的绝对中心，图的绝对中心可以存在于一条边上或某个结点上，该中心到所有点的最短距离的最大值最小。

根据图的绝对中心的定义可以知道，到绝对中心距离最远的结点至少有两个。

求图的绝对中心：

1. 用 Floyd 等算法求出任意两点间的最短路。

2. 在 $rk[u]$ 中存 $1-n$，按 u 到其它点的最短路的升序排序。

3. 图的绝对中心可能在某个点上。遍历所有点，用距离当前点最远的那个结点来更新。

4. 图的绝对中心可能在某条边上。对一条边 $<u,v>$，另取两点 $p1,p2$ 且满足 $dis[u][p2]<dis[u][p1]$，若 $dis[v][p2]>dis[v][p1]$，则可以用 $p2-u-v-p1$ 来更新。

求最小直径生成树：

根据图的绝对中心的定义，容易得知图的绝对中心是最小直径生成树的直径的中点。

得到无向图的绝对中心后，以绝对中心为起点，生成一个最短路径树，就是最小直径生成树。

```
#define pdi pair<double,int>
struct SPFA {
    vector<pii>e[MAXN];
    double dis[MAXN];
    int vis[MAXN];
    int pre[MAXN];
    priority_queue<pdi, vector<pdi>, greater<pdi> >q;
    double sum = 0; // 生成树总边权
    int cnt = 0; // 已选的点数
    void expand(int u) {
        cnt++;
        sum += dis[u];
        vis[u] = 1;
        for(auto i : e[u]) {
            int v = i.fi;
```

```
            double w = dis[u] + i.se;
            if(w < dis[v]) {
                dis[v] = w;
                pre[v] = u;
                q.push({w, v});
            }
        }
    }
    double solve(int n,  int s1, int s2 = 0) {
        if(s1 != 0)
            expand(s1);
        if(s2 != 0)
            expand(s2);
        while(!q.empty() && cnt < n) {
            double d = q.top().fi;
            int u = q.top().se;
            q.pop();
            if(vis[u])
                continue;
            expand(u);
        }
        return sum;
    }
    void add_edge(int u, int v, int w) {
        e[u].pb({v, w});
        e[v].pb({u, w});
    }
    void init(int n) {
        for(int i = 0; i < n + 5; i++) {
            e[i].clear();
            dis[i] = 1e18;
            vis[i] = pre[i] = 0;
        }
        cnt = sum = 0;
    }
} spfa;
int dis[MAXN][MAXN];
void floyd(int n) {
    for(int k = 1; k <= n; k++)
```

```
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
}
int tmp[MAXN];
int rk[MAXN][MAXN]; // 按距离升序存点
bool cmp(int a, int b) {
    return tmp[a] < tmp[b];
}
struct edge {
    int u, v, w;
} e[MAXM];
void solve(int n, int m) {
    floyd(n);
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            rk[i][j] = j;
            tmp[j] = dis[i][j];
        }
        sort(rk[i] + 1, rk[i] + n + 1, cmp);
    }
    int maxd = INF; // 绝对中心到最远结点的距离的2倍
    pii center; // 绝对中心
    for(int i = 1; i <= n; i++) { // 存在于某个结点上
        int x = dis[i][rk[i][n]] * 2;
        if(x < maxd) {
            maxd = x;
            center = {i, 0};
        }
    }
    for(int i = 1; i <= m; i++) { // 存在于某条边上
        int u = e[i].u, v = e[i].v, w = e[i].w;
        int p1 = rk[u][n];
        for(int j = n - 1; j >= 1; j--) {
            int p2 = rk[u][j];
            if(dis[v][p1] < dis[v][p2]) {
                int x = dis[u][p2] + dis[v][p1] + w;
                if(x < maxd) {
                    maxd = x;
```

```
                    center = {i, p2};
                }
                p1 = p2;
            }
        }
    }
    if(center.se == 0) { // 从u开始生成一个最短路径树
        int u = center.fi;
        spfa.dis[u] = 0;
        spfa.solve(n, u);
    } else { // 从<u,v>这条边开始生成一个最短路径树
        int id = center.fi;
        int p2 = center.se;
        int u = e[id].u, v = e[id].v, w = e[id].w;
        spfa.dis[u] = 0.5 * maxd - dis[u][p2]; // dis必须取真实值
        spfa.dis[v] = 1.0 * w - spfa.dis[u];
        spfa.pre[v] = u;
        spfa.solve(n, u, v);
    }
    for(int i = 1; i <= n; i++) // 输出生成树
        if(spfa.pre[i] != 0)
            printf("%d %d\n", i, spfa.pre[i]);
}
void init(int n) {
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            dis[i][j] = i == j ? 0 : INF;
    spfa.init(n);
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    init(n);
    for(int i = 1; i <= m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        e[i] = {u, v, w};
        int x = min(w, dis[u][v]);
        dis[u][v] = dis[v][u] = x;
```

```
        spfa.add_edge(u, v, w);
    }
    solve(n, m);
}
```

1.16 最近公共祖先

1.16.1 倍增求 LCA

预处理 $O(nlogn)$，查询 $O(logn)$。可以查询路径上边权和、最大边权、最小边权、严格次大边权等。

```
vector<pii>e[MAXN];
int dep[MAXN], f[MAXN][25];
int dis[MAXN][25];
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for(int i = 1; (1 << i) <= dep[u]; i++) {
        f[u][i] = f[f[u][i - 1]][i - 1];
        dis[u][i] = dis[u][i - 1] + dis[f[u][i - 1]][i - 1];
    }
    for(auto i : e[u]) {
        int v = i.fi, w = i.se;
        if(v == fa)
            continue;
        dis[v][0] = w;
        dfs(v, u);
    }
}
int lca(int x, int y) { // 这里求的是路径边权和
    int ans = 0;
    if(dep[x] < dep[y])
        swap(x, y);
    for(int i = 20; i >= 0; i--) {
        if(dep[f[x][i]] >= dep[y]) {
            ans += dis[x][i];
            x = f[x][i];
```

```
        }
        if(x == y)
            return ans;
    }
    for(int i = 20; i >= 0; i--)
        if(f[x][i] != f[y][i]) {
            ans += dis[x][i] + dis[y][i];
            x = f[x][i];
            y = f[y][i];
        }
    ans += dis[x][0] + dis[y][0];
    return ans;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        e[i].clear();
        dep[i] = 0;
        for(int j = 0; j < 25; j++)
            f[i][j] = dis[i][j] = 0;
    }
}
```

1.16.2 st 表求 LCA

预处理 $O(nlogn)$，常数查询。可以查询路径上边权和。

按 dfs 序把节点存入数组，总共存入 $2n-1$ 个数。两个节点第一次出现的区间内深度最小的点即为 LCA 。

```
pii a[MAXN * 2];
pii dp[MAXN * 2][25];
void init_rmq(int n) {
    for(int i = 1; i <= n; i++)
        dp[i][0] = a[i];
    for(int j = 1; (1 << j) <= n; j++)
        for(int i = 1; i + (1 << j) - 1 <= n; i++)
            dp[i][j] = min(dp[i][j - 1], dp[i + (1 << j - 1)][j - 1]);
}
int pos[MAXN]; // 存节点第一次出现的位置
```

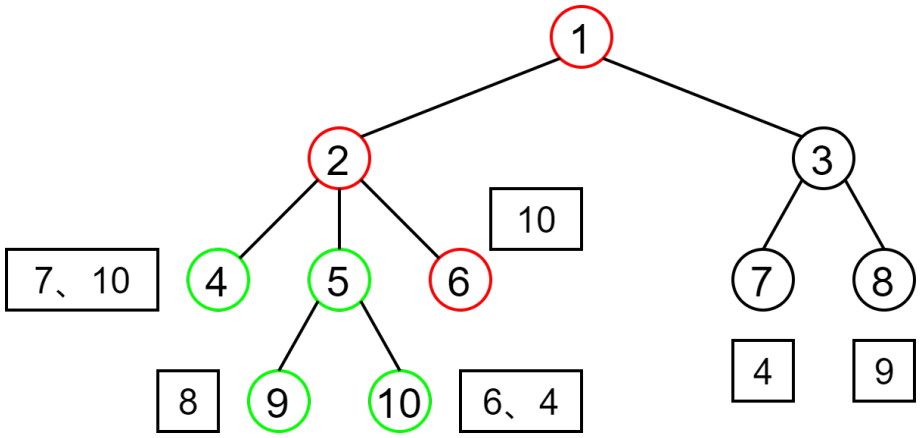
```
int dis[MAXN];
int Log[MAXN * 2];
int lca(int x, int y) { // 这里求的是路径边权和
    if(!x || !y)
        return INF;
    int L = min(pos[x], pos[y]);
    int R = max(pos[x], pos[y]);
    int k = Log[R - L + 1];
    int LCA = min(dp[L][k], dp[R - (1 << k) + 1][k]).se;
    return dis[x] + dis[y] - 2 * dis[LCA];
}
int dep[MAXN], len = 0;
void dfs(int u, int fa) {
    pos[u] = ++len;
    dep[u] = dep[fa] + 1;
    a[len] = mp(dep[u], u);
    for(auto i : e[u]) {
        int v = i.fi, w = i.se;
        if(v == fa)
            continue;
        dis[v] = dis[u] + w;
        dfs(v, u);
        a[++len] = mp(dep[u], u);
    }
}
void lca_init(int rt) {
    len = 0;
    dfs(rt, 0);
    init_rmq(len);
    Log[1] = 0; // 预处理Log,加快查询
    for(int i = 2; i < len + 5; i++)
        Log[i] = Log[i / 2] + 1;
}
```

1.16.3 离线求 LCA：$O(n+m+q)$

把所有的查询压入到两个点 u,v 上，在 dfs 树的时候遍历 u 的查询列表，如果 v 已经回溯了，那么 LCA 就是 v 到根的路径中最深的未回溯节点，查找这个点可以用并查集维护。

如下图，绿色为已回溯结点，红色为未回溯结点。当前遍历到 6 号结点，发现 10 号点已

回溯，LCA 就是 $\{1,2\}$ 中的 2 。




```
// 并查集
//------------------------------------------------
vi e[MAXN];
vector<pii> ask[MAXN]; // ask[u]存u与其它点的查询<v,id>
int vis[MAXN];
int ans[MAXQ];
void tarjan(int u, int fa) {
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        tarjan(v, u);
        pre[v] = u; // v已经回溯,u还未回溯
    }
    for(int i = 0; i < SZ(ask[u]); i++) {
        int v = ask[u][i].fi;
        int id = ask[u][i].se;
        if(vis[v])
            ans[id] = find(v); // 最深的未回溯节点
    }
    vis[u] = 1; // 标记回溯
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        pre[i] = i;
        e[i].clear();
```

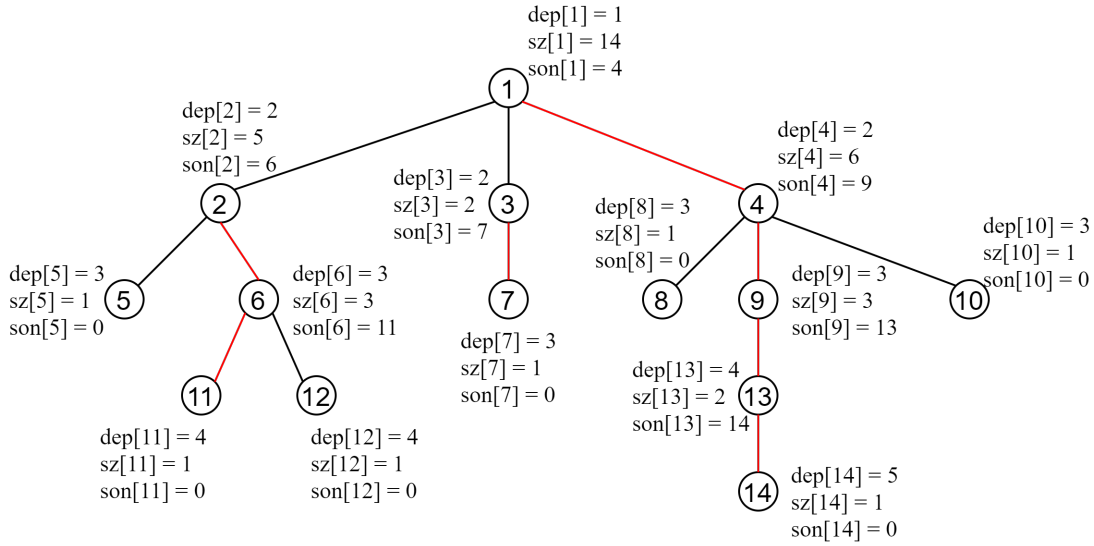
```
26          vis[i] = ans[i] = 0;
27          ask[i].clear();
28      }
29  }
30  int main() {
31      // ...
32      init(n);
33      // ...
34      for(int i = 1; i <= q; i++) {
35          int u, v;
36          scanf("%d%d", &u, &v);
37          ask[u].pb(mp(v, i));
38          ask[v].pb(mp(u, i));
39      }
40      tarjan(rt, 0);
41      for(int i = 1; i <= q; i++)
42          printf("%d\n", ans[i]);
43  }
```

1.17 树链剖分

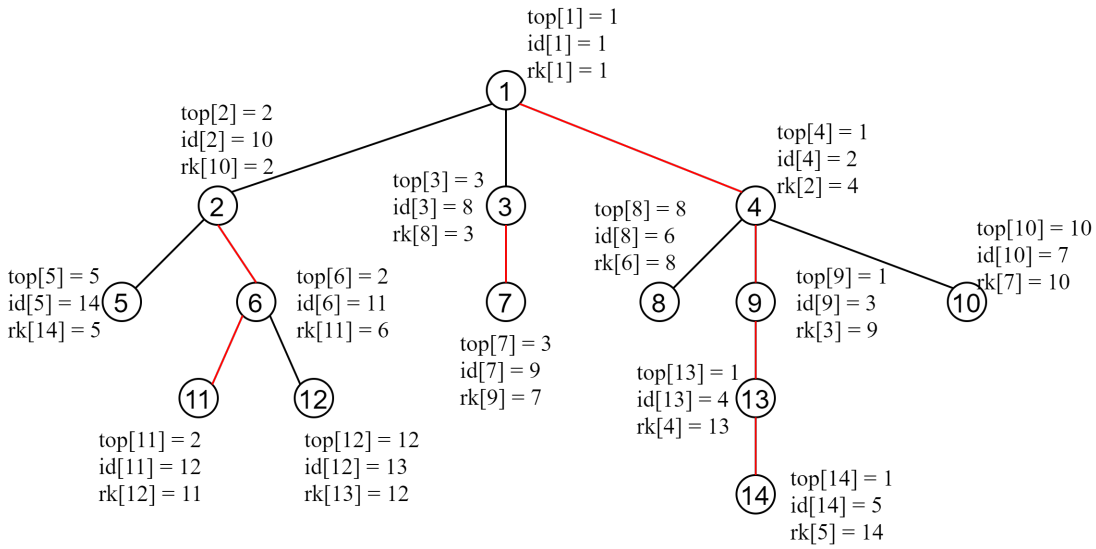
1.17.1 重链剖分

性质：一个节点到根的路径上的轻边不会超过 $logn$ 条。

第一遍 dfs ，得到每个点的 dep，sz，son 。



第二遍 dfs ，得到每个点的 top，id，rk 。



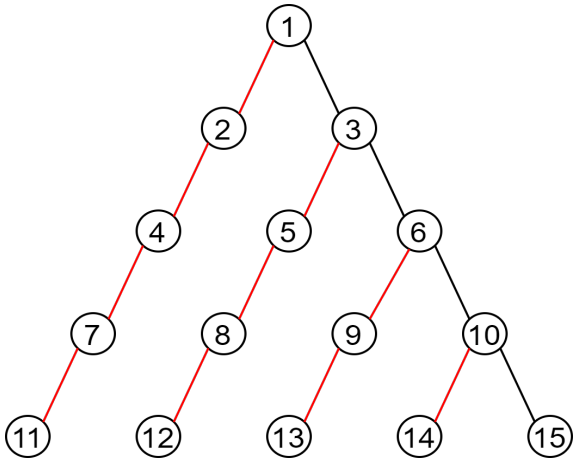

```cpp
int dep[MAXN], sz[MAXN];
int son[MAXN]; // 存重儿子
void dfs(int u, int fa) { // 预处理
    dep[u] = dep[fa] + 1;
    sz[u] = 1;
    for(auto v : e[u]) {
```

```
        if(v == fa)
            continue;
        dfs(v, u);
        sz[u] += sz[v];
        if(sz[v] > sz[son[u]])
            son[u] = v;
    }
}
int top[MAXN]; // 存所在重链的顶部节点
int id[MAXN]; // 点标号->dfs序号
int rk[MAXN]; // dfs序号->点标号
int cnt = 0;
void dfs2(int u, int fa, int Top) { // 标记dfs序
    top[u] = Top;
    id[u] = ++cnt;
    rk[cnt] = u;
    if(!son[u]) // u是叶节点
        return;
    dfs2(son[u], u, Top); // 先走重儿子
    for(auto v : e[u])
        if(v != fa && v != son[u])
            dfs2(v, u, v);
}
int main() {
    // ...
    dfs(root, 0);
    dfs2(root, 0, root);
    // ...
}
```

1.17.2 长链剖分

性质：

1. 一个节点到根的路径上的短边不会超过 $O(\sqrt{n})$ 条（如下图）。
2. 一个节点的 k 级祖先所在的长链长大于等于 k 。



长链剖分可以 $O(nlog^n)$ 预处理，$O(1)$ 查询任意点的 k 级祖先：
每条长链的顶点 x 存向上 $len[x]$ 个祖先和向下 $len[x]$ 个长链上的点
令 $w = log_2^k$ ，先跳到 2^w 级祖先 tmp ，还要向上走 $k' = k - 2^w$ 步
$\because len[tmp]\ 2^w > k'$
$\therefore$ 剩下的 k' 步可以由 tmp 所在长链的顶点的向上或向下数组直接得到

```
vi e[MAXN];
int len[MAXN]; // 从当前点向下的最长链的长度
int son[MAXN];
int dep[MAXN];
int f[MAXN][25];
void dfs(int u, int fa) {
    len[u] = 0;
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for(int i = 1; (1 << i) <= dep[u]; i++)
        f[u][i] = f[f[u][i - 1]][i - 1];
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs(v, u);
        if(len[v] + 1 > len[u]) {
            len[u] = len[v] + 1;
            son[u] = v;
        }
```

```
    }
}
vi up[MAXN], down[MAXN];
int top[MAXN];
void dfs2(int u, int fa) {
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs2(v, u);
    }
    if(u != son[fa]) { // u是所在长链的顶点
        int x = u;
        for(int i = 0; i <= len[u] && x != 0; i++) {
            up[u].pb(x);
            x = f[x][0];
        }
        x = u;
        for(int i = 0; i <= len[u] && x != 0; i++) {
            down[u].pb(x);
            top[x] = u; // 标记top节点
            x = son[x];
        }
    }
}
int Log[MAXN];
int find_k(int u, int k) { // 求u的k级祖先
    if(k == 0)
        return u;
    int w = Log[k];
    u = f[u][w];
    k = k - (1 << w);
    if(dep[u] - dep[top[u]] >= k) // 没超过top[u]
        return down[top[u]][dep[u] - dep[top[u]] - k];
    return up[top[u]][k - dep[u] + dep[top[u]]];
}
void init(int n) {
    Log[1] = 0;
    for(int i = 2; i < n + 5; i++)
        Log[i] = Log[i / 2] + 1;
```

```
59      for(int i = 1; i <= n; i++) {
60          e[i].clear();
61          up[i].clear();
62          down[i].clear();
63          len[i] = son[i] = dep[i] = 0;
64      }
65  }
```

树由所有长链组成，当需要维护长度相关的信息时，可以离线处理。

先 dfs 长儿子 $son[u]$，直接继承长儿子的信息；再 dfs 短儿子 v，用短儿子的信息 $O(len[v])$ 更新信息。总的复杂度为 $O(n)$ 。

如示例图中点的 dfs 序为：

tmp	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
代表的点	1	2	4	7	11	3	5	8	12	6	9	13	10	14	15

其中 tmp[6] - tmp[9] 为一条长链，由 3、5、8、12 组成。那么 tmp[9] 既可以表示距点 12 长度为 0 的点，也可以表示距点 8 长度为 1 的点、距点 5 长度为 2 的点、距点 3 长度为 3 的点，这就是对长儿子信息的继承。

当点 u 对子节点的 dfs 全部完成后，$tmp[u][i](0 \le i \le len[u]-1)$ 表示的就是 u 的子树中到 u 距离为 i 的所有点的信息，可以对答案进行更新。

```
1   int len[MAXN]; // 从当前点向下的最长链的长度
2   int son[MAXN];
3   void dfs(int u, int fa) {
4       len[u] = 1;
5       for(auto v : e[u]) {
6           if(v == fa)
7               continue;
8           dfs(v, u);
9           if(len[v] + 1 > len[u]) {
10              len[u] = len[v] + 1;
11              son[u] = v;
12          }
13      }
14  }
15  int tmp[MAXN], *p = tmp + 1, *dp[MAXN];
16  void dfs2(int u, int fa) {
```

```
    if(u != son[fa]) { // u是长链顶点
        dp[u] = p;
        p += len[u];
    } else
        dp[u] = dp[fa] + 1;
    dp[u][0] = 1;
    if(son[u] != 0)
        dfs2(son[u], u); // 长儿子直接继承
    for(auto v : e[u]) {
        if(v == fa || v == son[u])
            continue;
        dfs2(v, u);
        for(int j = 0; j < len[v]; j++) // 用短儿子更新
            dp[u][j + 1] += dp[v][j];
    }
    // 这里的dp[u][i], 0 <= i <= len[u] - 1
    // 表示u的子树中到u距离为i的点数
}
```

1.18 DSU On Tree：$O(nlogn)$

对于节点 u ：

1. 递归所有的轻儿子 v ，每次递归结束后消除轻儿子的影响
2. 递归重儿子 $son[u]$ ，递归结束后不消除重儿子的影响
3. 更新 u 自己的贡献
4. 更新所有轻儿子 v 的贡献
5. 得到该节点的答案

模板题：

给出一个树，每个结点都有一种颜色。求出每个结点的子树中出现次数最多的颜色的编号和。

```
// 邻接表
// 重链剖分
// DSU On Tree----------------------------------------
int col[MAXN], num[MAXN];
ll ans[MAXN], tmp = 0; // 最多颜色的编号和
int maxx = 0; // 最多颜色的数量
```

```cpp
void check(int col) { // 更新maxx和tmp
    if(num[col] > maxx) {
        maxx = num[col];
        tmp = col;
    } else if(num[col] == maxx)
        tmp += 1ll * col;
}
void update(int u, int fa, int f) {
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        update(v, u, f);
    }
    num[col[u]] += f;
    check(col[u]);
}
void dsu(int u, int fa) {
    for(auto v : e[u]) {
        if(v == fa || v == son[u])
            continue;
        // 递归轻儿子
        dsu(v, u);
        // 去除轻儿子的影响
        update(v, u, -1);
        tmp = maxx = 0;
    }
    // 递归重儿子
    if(son[u] != 0)
        dsu(son[u], u);

    // 更新自己的贡献
    num[col[u]]++;
    check(col[u]);
    // 更新轻儿子的贡献
    for(auto v : e[u]) {
        if(v == fa || v == son[u])
            continue;
        update(v, u, 1);
    }
```

```
    // 得到该结点的答案
    ans[u] = tmp;
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 1; i <= n; i++)
        scanf("%d", col + i);
    for(int i = 0; i < n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].pb(v);
        e[v].pb(u);
    }
    dfs(1, 0);
    dsu(1, 0);
    for(int i = 1; i <= n; i++)
        printf("%lld ", ans[i]);
}
```

1.19 树的重心

重心的性质:

1. 最大子树的 sz 最小
2. 到树中其它点的距离（边权为 1）和最小
3. 除了树只有一个点的情况，树的重心必然在重链上
4. 一棵树最多有 2 个重心，且相邻

根据性质 3，对当前节点 u ，可以用 u 的重儿子子树的重心 p ，在重链上向上爬找到 u 子树的重心。

求所有子树的重心：$O(n)$

```
// 邻接表
// 重链剖分
//-------------------------------------------------
int zx[MAXN][2];
int check(int u, int rt) { // 返回最大子树的大小
    return max(SZ[rt] - SZ[u], SZ[son[u]]);
```

```
}
void get_zx(int u, int fa) {
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        get_zx(v, u);
    }
    int p = zx[son[u]][0];
    if(!p) { // 叶子节点
        zx[u][0] = u;
        return;
    }
    while(SZ[u] - SZ[p] > SZ[p] && p != u) // p向上爬
        p = f[p];
    zx[u][0] = p;
    if(p != u && check(p, u) == check(f[p], u)) // 考虑第2个重心
        zx[u][1] = f[p];
}
```

1.20 树分治

在树上寻找一些满足条件的路径，或求路径的最值，或统计路径的数量。

可能会卡常，能记忆化就记忆化，尽量减小常数！

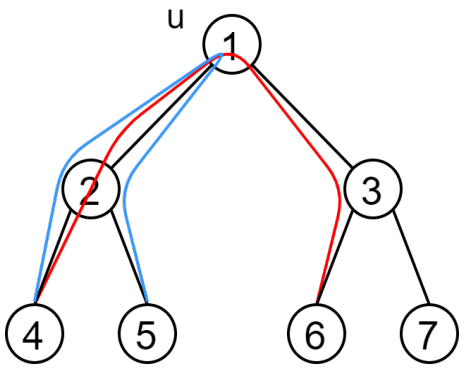
1.20.1 点分治：$O(nlogn + logn * T(solve))$

容斥写法：

divide 中对每个点 u ：

1. 计算所有经过 u 点的路径数（红 + 蓝）。2. 容斥，去掉不合法的路径（蓝）。对这些路径，两个端点一定同时存在于某个子节点 v 的子树内。3. 递归子节点。

（对每个点 u 的 solve 的复杂度不超过 $xlogx, x = sz[u]$ ）



非容斥写法：

依次遍历当前点 u 的子节点 v 所在的子树，逐渐更新信息和答案（类似树形 dp）。根据情况判断，可能需要对子节点正反扫两遍。

```
vector<pii>e[MAXN];
int sz[MAXN];
int vis[MAXN];
int mima = INF; // 最大子树的最小值
int root;
int sumsz; // 当前树的大小
void getrt(int u, int fa) { // 求重心
    sz[u] = 1;
    int maxson = 0;
    for(auto i : e[u]) {
        int v = i.fi;
        if(v == fa || vis[v])
            continue;
        getrt(v, u);
        sz[u] += sz[v];
        maxson = max(maxson, sz[v]);
    }
    maxson = max(maxson, sumsz - sz[u]);
    if(mima > maxson) {
        mima = maxson;
        root = u;
    }
}
// 算贡献----------------------
```

```cpp
void solve1() {}
void solve2() {}
//----------------------------
void divide(int u, int totsz) {
    solve1(); // 当前节点的贡献
    vis[u] = 1;
    for(auto i : e[u]) {
        int v = i.fi;
        int w = i.se;
        if(vis[v])
            continue;
        solve2(); // 容斥
        //分治子树
        mima = INF;
        sumsz = sz[v] > sz[u] ? totsz - sz[u] : sz[v];
        getrt(v, 0);
        divide(root, sz[v]);
    }
}
int main() {
    // ...
    sumsz = n;
    mima = INF;
    getrt(1, 0);
    divide(root, sumsz);
    // ...
}
```

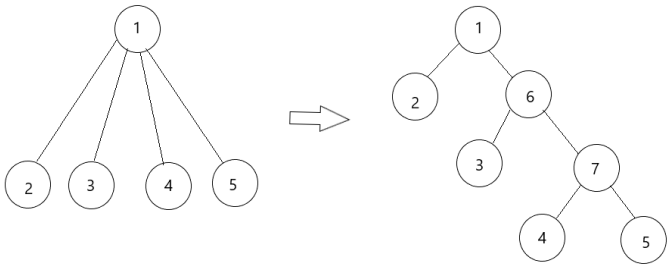
1.20.2 边分治：$O(nlogn + logn * T(solve))$

与点分治找重心类似，边分治需要找重心边。

如果朴素的寻找重心边，复杂度会被菊花图卡成 $O(n^2)$，因此需要将多叉树转化为二叉树。

根据题目需求，可以在 rebuild 中给新边和新点赋予边权和点权。

二叉树中 LCA 的第一个非新点的祖先，是原来多叉树中的 LCA。



边分治每次都将联通块分为两部分，就不用容斥了。

```
const int MAXN = ; // 开2倍!
const int MAXM = ;
// 前向星---------------------------------------
struct edge { // 存二叉树的边
    int to, next, w;
} e[MAXM * 2];
int cnt = 1; // 从2开始存!
int head[MAXN];
void add_edge(int u, int v, int w = 0) {
    e[++cnt].to = v;
    e[cnt].next = head[u];
    e[cnt].w = w;
    head[u] = cnt;
}
// 边分治---------------------------------------
int sz[MAXN];
int mima = INF;
pii root; // 重心边的起点和边标号,{u,id}
int vis[MAXM * 2];
void getrt(int u, int fa, int sum) { // 求重心边
    sz[u] = 1;
    for(int i = head[u]; i; i = e[i].next) {
        int v = e[i].to;
        if(v == fa || vis[i])
            continue;
        getrt(v, u, sum);
        sz[u] += sz[v];
        int max_sz = max(sz[v], sum - sz[v]);
```

```
        if(max_sz < mima) {
            mima = max_sz;
            root = {u, i};
        }
    }
}
//-----------------------------------------------
int dis[MAXN];
int q1[MAXN], cnt1 = 0; // 存u的子树信息
int q2[MAXN], cnt2 = 0; // 存v的子树信息
void dfs(int u, int fa, int f) {
    if(f == 1 && u <= n) { // 不考虑新增点
        // ...
    }
    if(f == 0 && u <= n) {
        // ...
    }
    for(int i = head[u]; i; i = e[i].next) {
        int v = e[i].to, w = e[i].w;
        if(v == fa || vis[i])
            continue;
        dis[v] = dis[u] + w;
        dfs(v, u, f);
    }
}
void solve() { // 计算贡献
    int u = root.fi, id = root.se;
    int v = e[id].to, w = e[id].w;
    dis[u] = dis[v] = 0;
    cnt1 = cnt2 = 0;
    dfs(u, v, 1); // 得到子树信息
    dfs(v, u, 0);
    if(!cnt1 || !cnt2) // 防止MLE或RE
        return;
    // ...
}
//-----------------------------------------------
void divide(int u, int sum) {
    mima = INF;
```

```
68      getrt(u, 0, sum);
69      if(mima == INF)
70          return;
71      int id = root.se;
72      int v = e[id].to;
73      vis[id] = vis[id ^ 1] = 1;
74      solve();
75      divide(u, sum - sz[v]);
76      divide(v, sz[v]);
77  }
78  //-----------------------------------------------
79  vector<pii>e1[MAXN]; // 存原图中的边
80  int New; // 新增节点
81  void rebuild(int u, int fa) { // 建二叉树
82      int tmp = 0; // 计数
83      int last = 0; // 可连节点
84      for(auto i : e1[u]) {
85          int v = i.fi, w = i.se;
86          if(v == fa)
87              continue;
88          tmp++;
89          if(tmp == 1) { // 第一条边
90              add_edge(u, v, w);
91              add_edge(v, u, w);
92              last = u;
93          } else if(tmp == SZ(e1[u]) - (u != 1)) { // 最后一条边
94              add_edge(last, v, w);
95              add_edge(v, last, w);
96          } else {
97              New++;
98              add_edge(last, New, 0); // 新边
99              add_edge(New, last, 0);
100             add_edge(New, v, w); // 原边
101             add_edge(v, New, w);
102             last = New;
103         }
104     }
105     for(auto i : e1[u]) {
106         int v = i.fi;
```

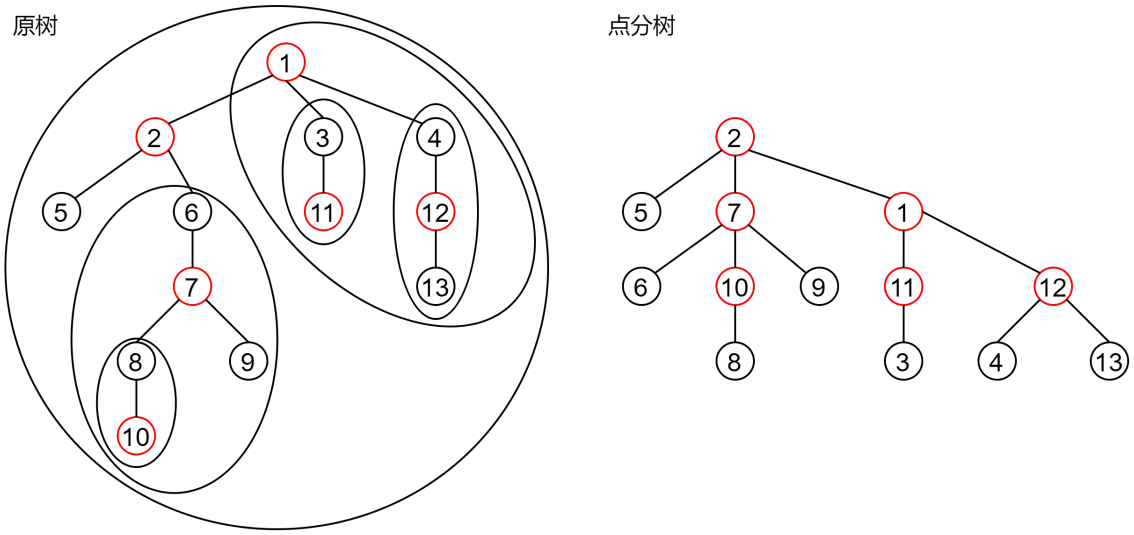
```
        if(v == fa)
            continue;
        rebuild(v, u);
    }
}
//-----------------------------------------------
void init(int n) {
    cnt = 1; // 前向星从2开始存!
    New = n;
    for(int i = 0; i < n + 5; i++) {
        e1[i].clear();
        head[i] = 0;
        vis[i] = vis[i + n] = 0;
    }
}
int main() {
    // ...
    init(n);
    // ...
    rebuild(1, 0);
    divide(1, New);
    // 注意单个点对答案的贡献
}
```

1.20.3 点分树

通过点分治每次找重心的方式来对原树进行重构。

将每次找到的重心与上一层的重心连接，这样就可以形成一棵 $logn$ 层的树。

（由于点分树常数较大，求 LCA 时应使用 ST 表）



常见的维护方法：

对于点分树上的每个点 u，维护两个数据结构 $S1$ 和 $S2$。$S1$ 存储点分树上 u 的子树对 u 的贡献，$S2$ 存储点分树上 u 的子树对 u 的点分树上父节点 fa 的贡献，用来容斥。

对 u 进行修改时，在点分树上从 u 开始向上爬树一直到根节点，在爬树过程中对所有经过节点 p 的 $S1$ 和 $S2$ 进行修改。

对 u 进行查询时，在点分树上从 u 开始向上爬树一直到根节点，在爬树过程中对所有经过节点 p，把 p 的 $S1$ 的贡献加入答案，把 p 的 $S2$ 的贡献删去。

初始化可视为进行 n 次修改。

模板题 1：P2056 [ZJOI2007] 捉迷藏

在一棵有 n 个结点的树，初始时所有结点都是黑色的。你需要实现以下两种操作：第 1 种操作是反转一个节点的颜色（白变黑，黑变白）；第 2 种操作是询问树上最远的两个黑点的距离。

用 $maxd[u]$ 存储点 u 的子树中的黑点到 $fa[u]$ 的最大距离，用 $dif[u]$ 存储点 u 的每个儿子 v 的 $maxd[v]$，用 ans 存储每个节点 u 的 $dif[u]$ 中的最大值与次大值之和。ans 中的最大值即为答案。

所以需要一种能快速加入，删除，查询最大值、次大值的数据结构。可以用 multiset 维护，但是常数较大，可能会 TLE。

我们可以用两个大根堆来维护，分别存加入的数和删除的数，如下所示。

```
1 struct Heap {
```

```
    priority_queue<int>A, B;
    void add(int x) {
        A.push(x);
    }
    void del(int x) {
        A.top() == x ? A.pop() : B.push(x);
    }
    int max1() { // 查询最大值
        while(!A.empty() && !B.empty() && A.top() == B.top()) {
            A.pop();
            B.pop();
        }
        return A.empty() ? -INF : A.top();
    }
    int max2() { // 查询次大值
        int x = max1();
        if(x == -INF)
            return x;
        A.pop();
        int y = max1();
        A.push(x);
        return y;
    }
};
```

模板题 2：BZOJ-3730 点分树 | 震波

在一棵 n 个节点的树上，每个点都有一个点权 val。有两种操作，第 1 种操作是给出 u 和 k，查询离 u 的距离 $\leq k$ 的所有点的总权值和；第 2 种操作是给出 u 和 y，把点 u 的权值修改为 y。

用树状数组维护贡献，树状数组的下标为原树上两点间的距离，需要动态开空间。

复杂度：修改 $O(lognlogn)$，查询 $O(lognlogn)$

```
vi e1[MAXN]; // 原树的边
vi e2[MAXN]; // 点分树的边
// 点分治---------------------------------
int sz[MAXN];
int vis[MAXN];
int mima = INF; // 最大子树的最小值
```

```cpp
int root;
int sumsz; // 当前树的大小
void getrt(int u, int fa) { // 求重心
    sz[u] = 1;
    int maxson = 0;
    for(auto v : e1[u]) {
        if(v == fa || vis[v])
            continue;
        getrt(v, u);
        sz[u] += sz[v];
        maxson = max(maxson, sz[v]);
    }
    maxson = max(maxson, sumsz - sz[u]);
    if(mima > maxson) {
        mima = maxson;
        root = u;
    }
}
void divide(int u, int totsz) {
    vis[u] = 1;
    for(auto v : e1[u]) {
        if(vis[v])
            continue;
        mima = INF;
        sumsz = sz[v] > sz[u] ? totsz - sz[u] : sz[v];
        getrt(v, 0);
        e2[u].pb(root); // 新边
        e2[root].pb(u); // 新边
        divide(root, sz[v]);
    }
}
int build(int n) { // 建立点分树
    sumsz = n;
    mima = INF;
    getrt(1, 0);
    int rt = root;
    divide(root, sumsz);
    return rt;
}
```

```
// LCA-----------------------------------
pii a[MAXN * 2];
pii dp[MAXN * 2][25];
void init_rmq(int n) {
    for(int i = 1; i <= n; i++)
        dp[i][0] = a[i];
    for(int j = 1; (1 << j) <= n; j++)
        for(int i = 1; i + (1 << j) - 1 <= n; i++)
            dp[i][j] = min(dp[i][j - 1], dp[i + (1 << j - 1)][j - 1]);
}
int pos[MAXN]; // 节点第一次出现的位置
int dep[MAXN], len = 0;
void dfs1(int u, int fa) {
    pos[u] = ++len;
    dep[u] = dep[fa] + 1;
    a[len] = mp(dep[u], u);
    for(auto v : e1[u]) {
        if(v == fa)
            continue;
        dfs1(v, u);
        a[++len] = mp(dep[u], u);
    }
}
int lca(int x, int y) { // 求原树上两点的距离
    int L = min(pos[x], pos[y]);
    int R = max(pos[x], pos[y]);
    int k = log(R - L + 1) / log(2);
    int LCA = min(dp[L][k], dp[R - (1 << k) + 1][k]).se;
    return dep[x] + dep[y] - 2 * dep[LCA];
}
void lca_init() { // 建立ST表
    len = 0;
    dfs1(1, 0);
    init_rmq(len);
}
// BIT-----------------------------------
vi S1[MAXN]; // u的子树对u的贡献
vi S2[MAXN]; // u的子树对fa[u]的贡献
int lowbit(int x) {
```

```
85      return x & (-x);
86  }
87  // 注意树状数组的下标表示两点距离,可能为0,因此要右移一位
88  void update(int u, vi a[], int p, int x) {
89      p++; // !!!
90      for(; p < SZ(a[u]); p += lowbit(p))
91          a[u][p] += x;
92  }
93  int get_sum(int u, vi a[], int p) {
94      p++; // !!!
95      ll ans = 0;
96      for(; p >= 1; p -= lowbit(p))
97          ans += a[u][p];
98      return ans;
99  }
100 int fa[MAXN]; // 点分树上父节点,用于爬树
101 int maxd[MAXN];
102 void dfs2(int u, int Fa) {
103     fa[u] = Fa;
104     for(auto v : e2[u]) {
105         if(v == Fa)
106             continue;
107         dfs2(v, u);
108         maxd[u] = max(maxd[u], maxd[v] + lca(u, v));
109     }
110     maxd[u] += lca(u, Fa);
111     S1[u].resize(maxd[u] + 5); // 动态分配空间
112     S2[u].resize(maxd[u] + 5); // 防止 MLE
113 }
114 //----------------------------------------
115 int val[MAXN];
116 void change(int u, int x) { // 修改操作
117     int p = u;
118     while(p != 0) {
119         int d = lca(p, u);
120         update(p, S1, d, x - val[u]);
121         if(fa[p] != 0) {
122             d = lca(fa[p], u);
123             update(p, S2, d, x - val[u]);
```

```
        }
        p = fa[p];
    }
    val[u] = x;
}
int check(int u, int k) { // 查询操作
    int ans = 0;
    int p = u; // 当前节点
    int last = 0; // 上一个访问的节点
    while(p != 0) {
        int d = lca(p, u);
        if(d <= k) {
            // 注意树状数组的范围,防止RE
            int R = min(SZ(S1[p]) - 2, k - d);
            ans += get_sum(p, S1, R);
            if(last != 0) {
                R = min(SZ(S2[last]) - 2, k - d);
                ans -= get_sum(last, S2, R);
            }
        }
        last = p;
        p = fa[p];
    }
    return ans;
}
// 只有一组样例,没有写总的初始化
int main() {
    // ...
    lca_init(); // LCA init
    int rt = build(n); // 建点分树
    dfs2(rt, 0); // BIT init
    for(int i = 1; i <= n; i++) // 更新初始点权
        change(i, tmp[i]);
    // ...
}
```

1.21 网络流

1.21.1 最大流

最大流最小割定理：最大流 = 最小割

1 Edmonds-Karp 算法：$O(nm^2)$

```
int g[MAXN][MAXN];
int pre[MAXN];
int flow[MAXN];
int bfs(int n, int s, int e) {
    fill(pre, pre + n + 5, -1);
    fill(flow, flow + n + 5, 0);
    flow[s] = INF;
    queue<int>q;
    q.push(s);
    while(!q.empty()) {
        int x = q.front();
        q.pop();
        if(x == e) // 每次找一条
            break;
        for(int i = 1; i <= n; i++)
            if(g[x][i] > 0 && pre[i] == -1) {
                pre[i] = x;
                flow[i] = min(flow[x], g[x][i]);
                q.push(i);
            }
    }
    return flow[e];
}
int EK(int n, int s, int e) {
    int ans = 0;
    while(true) {
        int tmp = bfs(n, s, e);
        if(tmp == 0)
            break;
        ans += tmp;
        for(int i = e; i != s;) {
```

```
            int last = pre[i];
            g[last][i] -= tmp;
            g[i][last] += tmp;
            i = last;
        }
    }
    return ans;
}
int main() {
    int n, m;
    while(~scanf("%d%d", &m, &n)) {
        for(int i = 0; i < n + 5; i++)
            for(int j = 0; j < n + 5; j++)
                g[i][j] = 0;
        for(int i = 0; i < m; i++) {
            int u, v, w;
            scanf("%d%d%d", &u, &v, &w);
            g[u][v] += w;
        }
        printf("%d\n", EK(n, 1, n));
    }
}
```

2 Dinic 算法：$O(n^2m)$

二分图中复杂度下降为 $O(m\sqrt{n})$ 。

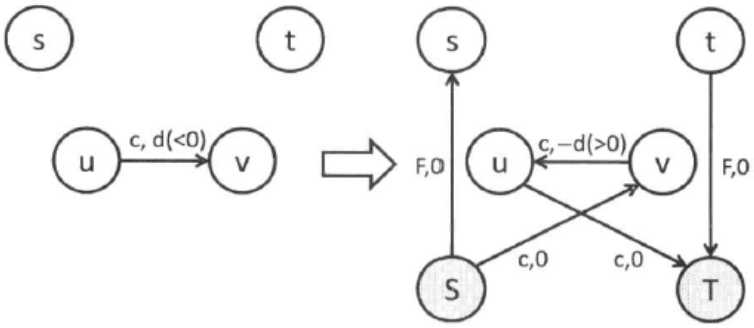
```
namespace maxflow {
struct Edge {
    int v, rev, f;
};
int n, s, t;
int cur[MAXM], dep[MAXN], gap[MAXN];
int flow;
vector<Edge> G[MAXN];
void add_edge(int u, int v, int f) {
    G[u].push_back({v, SZ(G[v]), f});
    G[v].push_back({u, SZ(G[u]) - 1, 0});
}
```

```
int dfs(int u, int lim) {
    if(u == t)
        return lim;
    int num = 0, f;
    for(int &i = cur[u], v; i < SZ(G[u]); ++i) {
        if(dep[v = G[u][i].v] == dep[u] - 1 && (f = G[u][i].f))
            if(G[u][i].f -= (f = dfs(v, std::min(lim - num, f))),
                    G[v][G[u][i].rev].f += f, (num += f) == lim)
                return num;
    }
    if(!--gap[dep[u]++])
        dep[s] = n + 1;
    return ++gap[dep[u]], cur[u] = 0, num;
}
void init(int _n) {
    n = _n;
    for(int i = 0; i < n; ++i)
        G[i].clear();
}
void solve(int _s, int _t) {
    s = _s, t = _t, flow = 0;
    for(int i = 0; i <= n; ++i)
        cur[i] = dep[i] = gap[i] = 0;
    for(gap[0] = n; dep[s] <= n; flow += dfs(s, INF));
}
}
int main() {
    // 这个板子0点不能用,下标必须从1开始
    maxflow::init(n + 5);
    // ...
    maxflow::add_edge(u, v, f);
    //...
    maxflow::solve(s, t);
    printf("%d\n", maxflow::flow);
}
```

1.21.2 最小费最大流

对于费用为负的边：

把负权边改为如下建边。（c 表示容量，d 表示费用，F 表示无限大，S 和 T 为两个新建结点）



$mf = mf + \sum_{\text{负权边 e}} c(e)$
$mc = mc + \sum_{\text{负权边 e}} c(e) * d(e)$

1 类 Dinic 算法：$O(nmf)$

```
namespace MCMF {
int n, tot;
int mf, mc, sum;
vi G[MAXN];
int cap[MAXM * 2], cost[MAXM * 2], edge[MAXM * 2];
void add(int u, int v, int Cap, int Cost) {
    G[u].push_back(++tot);
    edge[tot] = v;
    cap[tot] = Cap;
    cost[tot] = Cost;
}
void add_edge(int u, int v, int Cap, int Cost) {
    add(u, v, Cap, Cost);
    add(v, u, 0, -Cost);
}
int dis[MAXN];
bool augment(int s, int t) {
    priority_queue<pii, vector<pii>, greater<pii> > q;
    fill(dis, dis + n + 1, INF);
    q.push({ dis[t] = 0, t });
    while(!q.empty()) {
```

```
        pii x = q.top();
        q.pop();
        if(dis[x.se] != x.fi)
            continue;
        int &u = x.se, dt, v;
        for(int it : G[u])
            if(cap[it ^ 1] && (dt = dis[u] - cost[it]) < dis[v = edge[it]])
                q.push({ dis[v] = dt, v });
    }
    sum += dis[s];
    for(int i = 0; i <= n; ++i)
        for(int it : G[i])
            cost[it] += dis[edge[it]] - dis[i];
    return dis[s] != INF;
}
bool vis[MAXN];
int dfs(int u, int t, int limit) {
    if(!limit)
        return 0;
    if(u == t) {
        // 每次增加的费用是递增的
        mc += limit * sum;
        return limit;
    }
    int fee = 0, v;
    vis[u] = true;
    for(auto it : G[u]) {
        if(cost[it] || !cap[it] || vis[v = edge[it]])
            continue;
        int water = dfs(v, t, min(limit - fee, cap[it]));
        cap[it] -= water;
        cap[it ^ 1] += water;
        fee += water;
        if(fee == limit)
            break;
    }
    if(fee == limit)
        vis[u] = false;
    return fee;
```

```
}
void init(int x) {
    n = x, tot = 1;
    for(int i = 0; i <= x; ++i)
        G[i].clear();
}
void solve(int s, int t) {
    int res;
    mf = mc = sum = 0;
    do {
        do {
            fill(vis, vis + n + 1, 0);
        } while(mf += (res = dfs(s, t, INF)), res > 0);
    } while(augment(s, t));
}
}
int main() {
    int n, m, s, t;
    scanf("%d%d%d%d", &n, &m, &s, &t);
    MCMF::init(n);
    for(int i = 0; i < m; i++) {
        int u, v, c, w;
        scanf("%d%d%d%d", &u, &v, &c, &w);
        MCMF::add_edge(u, v, c, w);
    }
    MCMF::solve(s, t);
    printf("%d %d\n", MCMF::mf, MCMF::mc);
}
```

2 ZKW 算法：$O(nmf)$

```
namespace ZKW {
struct edge {
    int from, to, cap, cost;
} w[MAXM * 2];
vector<int> E[MAXN];
bool vis[MAXN];
int cnt, cost, flow, price;
```

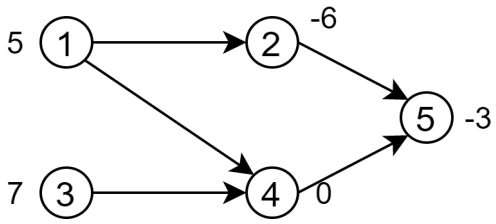
```
8  void init() {
9      cnt = 0;
10     memset(E, 0, sizeof E);
11     memset(vis, 0, sizeof vis);
12 }
13 void add_edge(int u, int v, int cap, int cost) {//双向边
14     w[cnt] = (edge) {
15         u, v, cap, cost
16     };
17     E[u].push_back(cnt++);
18     w[cnt] = (edge) {
19         v, u, 0, -cost
20     };
21     E[v].push_back(cnt++);
22 }
23 int aug(int u, int a, int T) {
24     if(u == T)
25         return cost += a * price, a;
26     vis[u] = 1;
27     int f = a;
28     for(int i = 0; i < SZ(E[u]); i++) {
29         edge e = w[E[u][i]];
30         if(!e.cost && e.cap && !vis[e.to]) {
31             int d = aug(e.to, min(f, e.cap), T);
32             w[E[u][i]].cap -= d;
33             w[E[u][i] ^ 1].cap += d;
34             if(!(f -= d))
35                 return a;
36         }
37     }
38     return a - f;
39 }
40 pii calc(int S, int T) {
41     flow = cost = price = 0;
42     for(;;) {
43         for(;;) {
44             memset(vis, 0, sizeof(vis));
45             int f = aug(S, INF, T);
46             if(!f)
```

```
47                  break;
48              flow += f;
49          }
50          int d = INF;
51          for(int u = 0; u < MAXN; u++)
52              if(vis[u])
53                  for(int i = 0; i < SZ(E[u]); i++) {
54                      edge e = w[E[u][i]];
55                      if(e.cap && !vis[e.to])
56                          d = min(d, e.cost);
57                  }
58          if(d == INF)
59              return make_pair(flow, cost);
60          for(int u = 0; u < MAXN; u++)
61              if(vis[u])
62                  for(int i = 0; i < SZ(E[u]); i++)
63                      w[E[u][i]].cost -= d, w[E[u][i] ^ 1].cost += d;
64          price += d;
65      }
66  }
67  }
68  int main() {
69      // 该板子无法求负边
70      int n, m, s, t;
71      scanf("%d%d%d%d", &n, &m, &s, &t);
72      ZKW::init();
73      for(int i = 0; i < m; i++) {
74          int u, v, cap, cost;
75          scanf("%d%d%d%d", &u, &v, &cap, &cost);
76          ZKW::add_edge(u, v, cap, cost);
77      }
78      pii ans = ZKW::calc(s, t);
79      printf("%d %d\n", ans.fi, ans.se);
80  }
```

1.21.3 最大权闭合图

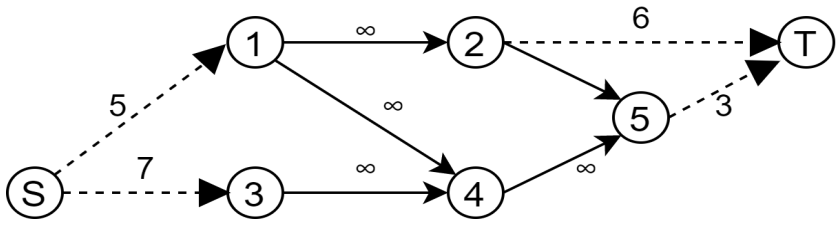
一个有向图 G 的闭合图是 G 的一个点集，满足点集内所有点的出边仍指向点集内的点。给每个点分配一个点权，最大权闭合图即为点权之和最大的闭合图。

下图中由 9 个闭合图：$\emptyset$、$\{3,4,5\}$、$\{4,5\}$、$\{5\}$、$\{2,4,5\}$、$\{2,5\}$、$\{2,3,4,5\}$、$\{1,2,4,5\}$、$\{1,2,3,4,5\}$，其中权最大的闭合图是 $\{3,4,5\}$ ，权和为 4 。



重新建图：把原图中所有边的边权置为 $+\infty$ ，由 S 向所有点权为正的点连边（边权为点权），由所有点权为负的点向 T 连边（边权为点权的绝对值）。

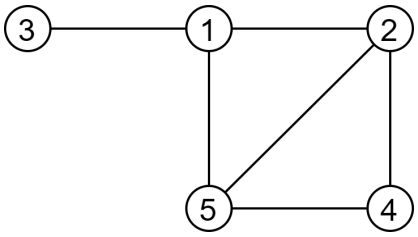
在新图上跑最大流，此时最小割（最大流）具有唯一性，最大权 $=\sum_{i\text{为正权点}} val(i) - maxflow$ 。



1.21.4 最大密度子图

子图的密度定义为子图的边数除以子图的点数，即 $\frac{E}{V}$ 。

最大密度子图即为密度最大的子图，下图中的最大密度子图为 $\{1,2,4,5\}$ ，密度为 $\frac{5}{4}$ 。

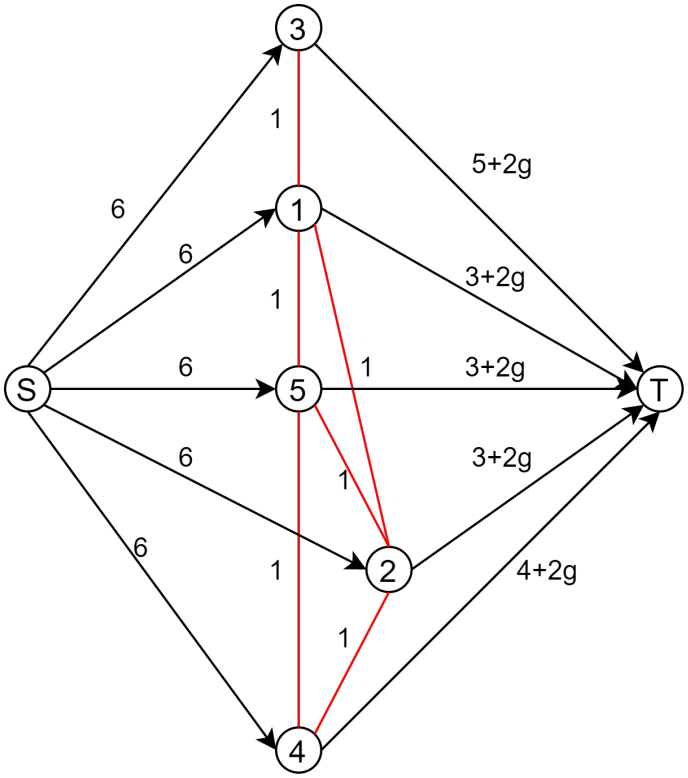


二分答案，每次检查时重新建图跑最大流，复杂度为 $O(nmlogn)$ 。

重新建图：原图中的无向边（红边）保留，流量为 1。由源点向原图中的点连一条边，流量为原图中的总边数 m ；由原图中的点 u 向汇点连一条边，流量为 $m - deg[u] + 2g$ 。

如果 $\frac{nm-maxflow}{2} > 0$ ，说明可以更大。

二分结束后对残余网络 dfs ，能走到的点就是最大密度子图中的点。




```cpp
const double eps = 1e-8;
int n, m;
vi e[MAXN];
int deg[MAXN];
namespace maxflow { // double版最大流
#define INF 1e9
struct Edge {
    int v, rev;
    double f;
};
int n, s, t;
int cur[MAXM], dep[MAXN], gap[MAXN];
double flow;
vector<Edge> G[MAXN];
void add_edge(int u, int v, double f) {
    G[u].push_back({v, SZ(G[v]), f});
    G[v].push_back({u, SZ(G[u]) - 1, 0});
}
```

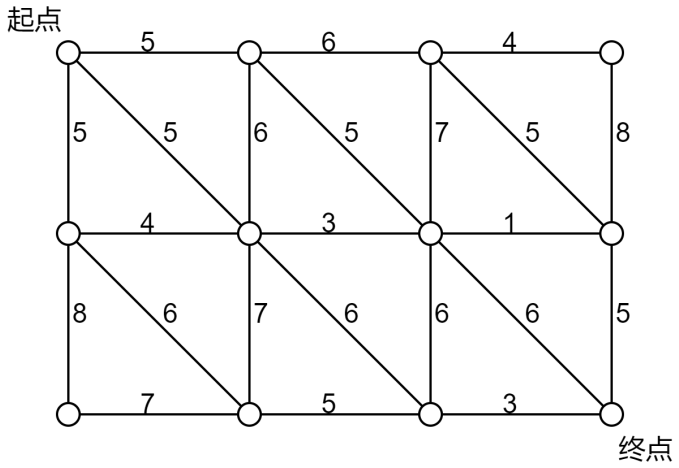
```
double dfs(int u, double lim) {
    if(u == t)
        return lim;
    double num = 0, f;
    for(int &i = cur[u], v; i < SZ(G[u]); ++i) {
        if(dep[v = G[u][i].v] == dep[u] - 1 && (f = G[u][i].f))
            if(G[u][i].f -= (f = dfs(v, std::min(lim - num, f))),
                    G[v][G[u][i].rev].f += f, abs((num += f) - lim) < eps)
                return num;
    }
    if(!--gap[dep[u]++])
        dep[s] = n + 1;
    return ++gap[dep[u]], cur[u] = 0, num;
}
void solve(int _s, int _t) {
    s = _s, t = _t, flow = 0;
    for(int i = 0; i <= n; ++i)
        cur[i] = dep[i] = gap[i] = 0;
    for(gap[0] = n; dep[s] <= n; flow += dfs(s, INF));
}
int vis[MAXN];
vi ans; // 存最大密度子图
void dfs2(int u) { // 遍历残余网络
    vis[u] = 1;
    for(auto i : G[u]) {
        int v = i.v;
        double w = i.f;
        if(!vis[v] && w >= eps) {
            ans.pb(v);
            dfs2(v);
        }
    }
}
void init(int _n) {
    n = _n;
    for(int i = 0; i < n; ++i) {
        G[i].clear();
        vis[i] = 0;
    }
```

```
    ans.clear();
}
}
double check(double g) {
    maxflow::init(n + 2); // 建新图
    for(int u = 1; u <= n; u++)
        for(auto v : e[u])
            maxflow::add_edge(u, v, 1);
    int s = n + 1, t = n + 2;
    for(int i = 1; i <= n; i++) {
        maxflow::add_edge(s, i, m);
        maxflow::add_edge(i, t, m - deg[i] + 2 * g);
    }
    maxflow::solve(s, t);
    return (1.0 * n * m - maxflow::flow) / 2.0;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        e[i].clear();
        deg[i] = 0;
    }
}
int main() {
    while(~scanf("%d%d", &n, &m)) {
        if(m == 0) {
            printf("1\n1\n");
            continue;
        }
        init(n);
        for(int i = 0; i < m; i++) {
            int u, v;
            scanf("%d%d", &u, &v);
            e[u].pb(v), e[v].pb(u);
            deg[u]++, deg[v]++;
        }
        double L = 0, R = m;
        double prec = 1.0 / n / n; // 最小误差
        while(R - L >= prec) {
            double mid = (L + R) / 2;
```

```
            if(check(mid) >= eps)
                L = mid;
            else
                R = mid;
        }
        check(L); // 再跑一遍,否则有误差
        maxflow::dfs2(n + 1);
        vi ans = maxflow::ans;
        sort(ans.begin(), ans.end()); // 升序输出
        printf("%d\n", SZ(ans));
        for(auto i : ans)
            printf("%d\n", i);
    }
}
```

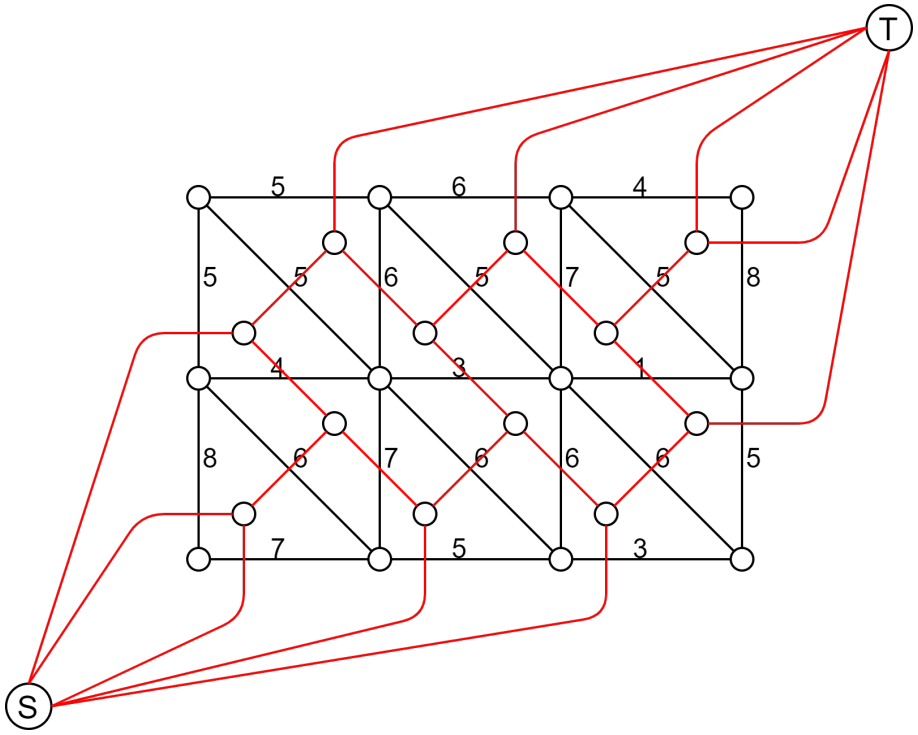
1.21.5 平面图网络流

如下图，通过删去一些边使得起点无法到达终点，删边的代价为边权，求最小的代价。



最小的代价明显是起点到终点的最大流（最小割），但可以转化为对偶图降低复杂度。

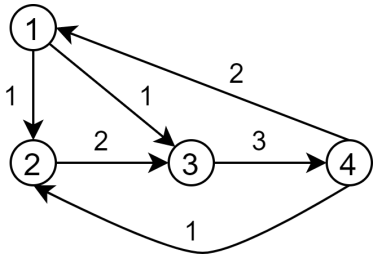
平面图最大流 = 平面图最小割 = 对偶图最短路



1.21.6 上下界网络流

1 无源汇上下界可行流

无源汇上下界可行流，也就是循环流，如下图。



新建源点 S 和汇点 T 。

对每条边 i 的流量范围 $a_i \leq flow \leq b_i$ ，假设每条边已经流了 b_i ，设其为初始流量。每条边在新图中的流量就改为 $b_i - a_i$ 。

令一个点 u 的初始流入量减去初始流出量为 M 。

若 $M = 0$ ，则流量平衡。

若 $M < 0$ ，表明流出量过大，新建一条从 u 到 T ，流量为 $-M$ 的附加边。

若 $M > 0$，表明流入量过大，新建一条从 S 到 u，流量为 M 的附加边。

在新图上跑最大流，如果所有附加边满流，说明存在可行流，否则不存在。

每条边在可行流中的流量为（流量下界 + 它在新图中的流量）。

注意：循环流的可行流没有具体的大小概念，与跑最大流（$solve(S,T)$）得到的 $flow$ 无关！

模板题：

给定一个流图，每条边的流量存在上下界。判断是否存在可行流，若存在则输出每条边的流量。

```
namespace maxflow {
struct Edge {
    int v, rev, f, id;
};
int n, s, t;
int cur[MAXM], dep[MAXN], gap[MAXN];
int flow;
vector<Edge> G[MAXN];
void add_edge(int u, int v, int f, int id = 0) {
    G[u].push_back({v, SZ(G[v]), f, 0});
    G[v].push_back({u, SZ(G[u]) - 1, 0, id}); // 反向边打标记
}
int dfs(int u, int lim) {
    if(u == t)
        return lim;
    int num = 0, f;
    for(int &i = cur[u], v; i < SZ(G[u]); ++i) {
        if(dep[v = G[u][i].v] == dep[u] - 1 && (f = G[u][i].f))
            if(G[u][i].f -= (f = dfs(v, std::min(lim - num, f))),
                    G[v][G[u][i].rev].f += f, (num += f) == lim)
                return num;
    }
    if(!--gap[dep[u]++])
        dep[s] = n + 1;
    return ++gap[dep[u]], cur[u] = 0, num;
}
void init(int _n) {
    n = _n;
    for(int i = 0; i < n; ++i)
```

```
        G[i].clear();
}
void solve(int _s, int _t) {
    s = _s, t = _t, flow = 0;
    for(int i = 0; i <= n; ++i)
        cur[i] = dep[i] = gap[i] = 0;
    for(gap[0] = n; dep[s] <= n; flow += dfs(s, INF));
}
}
int deg[MAXN], ans[MAXM];
void init(int n, int m) {
    fill(deg, deg + n + 5, 0);
    fill(ans, ans + m + 5, 0);
}
int main() {
    int t;
    scanf("%d", &t);
    while(t--) {
        int n, m;
        scanf("%d%d", &n, &m);
        init(n, m);
        maxflow::init(n + 2);
        int S = n + 1, T = n + 2;
        for(int i = 1; i <= m; i++) {
            int u, v, L, R;
            scanf("%d%d%d%d", &u, &v, &L, &R);
            maxflow::add_edge(u, v, R - L, i);
            deg[u] -= L, deg[v] += L;
            ans[i] += L;
        }
        int sum = 0; // 附加边的总流量
        for(int i = 1; i <= n; i++)
            if(deg[i] < 0)
                maxflow::add_edge(i, T, -deg[i]);
            else {
                maxflow::add_edge(S, i, deg[i]);
                sum += deg[i];
            }
        maxflow::solve(S, T);
```

```
        if(maxflow::flow != sum) // 不存在可行流
            printf("NO\n");
        else {
            printf("YES\n");
            for(int u = 1; u <= n; u++)
                for(auto i : maxflow::G[u]) {
                    int f = i.f, id = i.id;
                    ans[id] += f;
                }
            for(int i = 1; i <= m; i++)
                printf("%d\n", ans[i]);
        }
    }
}
```

2 有源汇上下界可行流

有源汇上下界可行流，要求使除了源点和汇点外的所有点流量平衡。

在原图上增加一条 t 到 s 的 $[0,+\infty]$ 的边，就转化为了无源汇上下界可行流问题。

由于有了源点和汇点，这里的可行流存在具体的大小，表示源点到汇点的总流量。

可行流的大小等于 t 到 s 的边中实际流过的流量，与 $solve(S,T)$ 得到的 $flow$ 无关。

```
maxflow::add_edge(t, s, INF); // 形成循环流
int sum = 0;
int S = t + 1, T = S + 1;
for(int i = 1; i <= t; i++)
    if(deg[i] < 0)
        maxflow::add_edge(i, T, -deg[i]);
    else {
        maxflow::add_edge(S, i, deg[i]);
        sum += deg[i];
    }
maxflow::solve(S, T);
if(maxflow::flow != sum) // 不存在可行流
    printf("-1\n");
else {
    int ans = 0; // 可行流的大小
    for(auto i : maxflow::G[s])
```

```
        if(i.v == t)
            ans = i.f;
    printf("%d\n", ans);
}
```

3 有源汇上下界最大流

假设原图中的源点和汇点为 s 和 t，先判断是否存在有源汇上下界可行流（注意判断中新建的源点和汇点为 S 和 T，不要搞混）。

在判断存在可行流后，在残余网络上再跑一次 s 到 t 的最大流，有源汇上下界最大流 $= maxflow :: flow$ 。

如果要知道每条边的具体流量，就给边打标记。

```
maxflow::solve(S, T); // 判断是否存在可行流
if(maxflow::flow != sum) // 不存在可行流
    printf("No\n");
else { // 存在可行流
    printf("Yes\n");
    maxflow::solve(s, t);
    printf("%d\n", maxflow::flow); // 最大流
}
```

4 有源汇上下界最小流

在判断存在可行流后，得到可行流的大小为 tmp 。

在残余网络上把 t 到 s 的边删除，再跑一次 t 到 s 的最大流，有源汇上下界最小流 $= tmp - maxflow :: flow$ 。

```
int tmp = 0; // 可行流的大小
for(auto i : maxflow::G[s])
    if(i.v == t)
        tmp = i.f;
for(auto& i : maxflow::G[t]) // 删除t到s的边
    if(i.v == s)
        i.f = 0;
maxflow::solve(t, s);
printf("%d\n", tmp - maxflow::flow);
```

5 最小费上下界可行流

按可行流建图，跑费用流，总的费用为每条边的下界费用 $+ MCMF :: mc$ 。

1.22 无向图全局最小割：$O(n^3)$

Stoer-Wagner 算法流程：

step1：在图中找出任意 $s-t$ 最小割

step2：把 t 合并到 s ，重复 step1 直到图中只剩 1 个点。

```
int dis[MAXN][MAXN];
int del[MAXN]; // 点是否被合并
int vis[MAXN]; // 点是否已被选
int sum[MAXN]; // 已选点到该点的总边权
pii Mincut(int n, int x) { // 任意找一个s-t最小割
    for(int i = 0; i < n + 5; i++)
        vis[i] = sum[i] = 0;
    vi tmp;
    for(int i = 1; i <= n - x + 1; i++) {
        int x = 0;
        for(int j = 1; j <= n; j++) // 找邻边总权最小的点
            if(!del[j] && !vis[j] && (sum[j] > sum[x] || !x))
                x = j;
        vis[x] = 1;
        tmp.pb(x);
        for(int j = 1; j <= n; j++) // 更新到其它点的总边权
            if(!del[j] && !vis[j])
                sum[j] += dis[x][j];
    }
    int s = tmp[SZ(tmp) - 2];
    int t = tmp[SZ(tmp) - 1];
    return {s, t};
}
int sw(int n) {
    int ans = INF;
    for(int i = 1; i < n; i++) {
        pii tmp = Mincut(n, i);
        int s = tmp.fi, t = tmp.se;
        ans = min(ans, sum[t]); // sum[t]=s-t的最小割
```

```
        del[t] = 1; // t合并到s
        for(int j = 1; j <= n; j++) {
            dis[s][j] += dis[t][j];
            dis[j][s] += dis[j][t];
        }
    }
    return ans;
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for(int i = 1; i <= m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        dis[u][v] += w;
        dis[v][u] += w;
    }
    printf("%d\n", sw(n));
}
```

1.23 二分图

性质：

1. 二分图中不存在长度为奇数的环
2. 最大匹配 = 最大流（因为二分图若有环，一定是偶环）
3. | 最大匹配 |=| 最小顶点覆盖 |

1.23.1 二分图的判定：$O(n+m)$

dfs 或 bfs 遍历图，若发现奇环，则不是二分图。

```
vi e[MAXN];
int col[MAXN], f = 1;
void dfs(int u, int now) {
    col[u] = now;
    for(auto v : e[u]) {
        if(col[v] == -1)
            dfs(v, !now);
```

```
        if(col[v] == now)
            f = 0;
    }
}
void init(int n) {
    f = 1;
    for(int i = 0; i < n + 5; i++) {
        col[i] = -1;
        e[i].clear();
    }
}
int main() {
    // ...
    init(n);
    for(int i = 1; i <= n; i++)
        if(col[i] == -1)
            dfs(i, 0);
    // ...
}
```

1.23.2 二分图最大匹配：$O(nm)$

匈牙利算法流程：

从任意一个未匹配点 u 开始，遍历 u 的相邻点 v 。如果 v 未匹配，则匹配成功。如果点 v 已匹配，试试看能否让 $link[v]$ 去匹配别的点，如果可以，那么 u 和 v 也匹配成功。

只需遍历二分图一侧的点，让这些点都去尝试匹配即可。

```
vi e[MAXN];
int link[MAXN]; // 匹配到的点
int vis[MAXN];
bool dfs(int u) {
    for(auto v : e[u])
        if(!vis[v]) {
            vis[v] = 1;
            if(link[v] == -1 || dfs(link[v])) {
                link[v] = u;
                link[u] = v;
                return true;
```

```
            }
        }
    return false;
}
int hungary(int n1, int n2) {
    fill(link, link + n1 + n2 + 5, -1);
    int ans = 0; // 匹配数量
    for(int i = 1; i <= n1; i++) {
        fill(vis, vis + n1 + n2 + 5, 0);
        if(dfs(i))
            ans++;
    }
    return ans;
}
int main() {
    int n1, n2, m;
    scanf("%d%d%d", &n1, &n2, &m);
    while(m--) {
        int u, v;
        scanf("%d%d", &u, &v);
        v += n1;
        e[u].pb(v);
    }
    printf("%d\n", hungary(n1, n2));
}
```

1.23.3 二分图最大权匹配：$O(n^3)$

最大权匹配的定义：在匹配对数最大的情况下总边权最大的方案。

求最小权匹配：将边权取负，dis 初始化为负无穷，跑 KM。

KM 算法：

1. 每个点分配一个期望值 wx 或 wy。
2. 对当前点找一条增广路径（可行的匹配方案），找到就考虑下一个点。
3. 找不到增广路就调整各点的期望值。
4. 重复 2、3 直到找到增广路。

1 dfs 版本：$O(n^4)$

```
int dis[MAXN][MAXN], slack[MAXN];
int visx[MAXN], linkx[MAXN], wx[MAXN];
int visy[MAXN], linky[MAXN], wy[MAXN];
bool dfs(int u, int cnty) { //匈牙利算法找增广路径
    visx[u] = 1;
    for(int v = 1; v <= cnty; v++) {
        if(visy[v])
            continue;
        int x = wx[u] + wy[v] - dis[u][v];
        if(x == 0) {
            visy[v] = 1;
            if(!linky[v] || dfs(linky[v], cnty)) {
                linkx[u] = v;
                linky[v] = u;
                return true;
            }
        } else
            slack[v] = min(slack[v], x);
    }
    return false;
}
int KM(int cntx, int cnty) {
    fill(wx, wx + cntx + 1, 0);
    fill(wy, wy + cnty + 1, 0);
    for(int i = 1; i <= cntx; i++)
        for(int j = 1; j <= cnty; j++)
            wx[i] = max(wx[i], dis[i][j]);
    fill(linkx, linkx + cntx + 1, 0);
    fill(linky, linky + cnty + 1, 0);
    for(int i = 1; i <= cntx; i++) {
        fill(slack, slack + cnty + 1, INF);
        while(true) {
            fill(visx, visx + cntx + 1, 0);
            fill(visy, visy + cnty + 1, 0);
            if(dfs(i, cnty))
                break;
            int minz = INF;
            for(int j = 1; j <= cnty; j++)
                if(!visy[j])
```

```
                    minz = min(minz, slack[j]);
            for(int j = 1; j <= cntx; j++)
                if(visx[j])
                    wx[j] -= minz;
            for(int j = 1; j <= cnty; j++)
                if(visy[j])
                    wy[j] += minz;
            if(minz == INF) // 无法匹配
                break;
        }
    }
    int ans = 0;
    for(int i = 1; i <= cntx; i++)
        ans += dis[i][linkx[i]];
    return ans;
}
void init(int cntx, int cnty) {
    for(int i = 1; i <= cntx; i++)
        fill(dis[i], dis[i] + cnty + 1, 0);
}
```

2 bfs 版本：$O(n^3)$

bfs 版本必须保证存在完美匹配，如果不存在完美匹配，需要进行一些预处理。

令左右两侧的点数为 $n1$ 和 $n2$（$n1 \geq n2$），在右侧增加一些虚点使两边点数相同，对 dis 的初始化可视为添加了一些边权为负无穷的虚边，这样必然存在完美匹配。

在得到完美匹配后，检查所有的匹配，若匹配边不是虚边，则为一对成功的匹配。

```
int dis[MAXN][MAXN], slack[MAXN];
int visx[MAXN], linkx[MAXN], wx[MAXN];
int visy[MAXN], linky[MAXN], wy[MAXN];
int pre[MAXN];
void bfs(int n, int u) {
    for(int i = 0; i < n + 5; i++) {
        visy[i] = pre[i] = 0;
        slack[i] = INF;
    }
    int y = 0;
```

```
    linky[y] = u; // 暂时存放
    while(true) {
        visy[y] = 1;
        int x = linky[y]; // 给x找匹配
        int minz = INF;
        int nexty = 0; // 找下一个y
        for(int i = 1; i <= n; i++) {
            if(visy[i])
                continue;
            int tmp = wx[x] + wy[i] - dis[x][i];
            if(tmp < slack[i]) {
                slack[i] = tmp;
                pre[i] = y;
            }
            if(slack[i] < minz) {
                minz = slack[i];
                nexty = i;
            }
        }
        for(int i = 0; i <= n; i++) // 注意从0开始
            if(visy[i]) {
                wx[linky[i]] -= minz;
                wy[i] += minz;
            } else
                slack[i] -= minz;
        y = nexty;
        if(!linky[y])
            break;
    }
    while(y) {
        int x = linky[pre[y]];
        linkx[x] = y;
        linky[y] = x;
        y = pre[y];
    }
}
int KM(int nx, int ny) { // nx>=ny
    for(int i = 0; i < nx + 5; i++) {
        wx[i] = wy[i] = 0;
```

```
        linkx[i] = linky[i] = 0;
    }
    for(int i = 1; i <= nx; i++)
        bfs(nx, i);
    int ans = 0;
    for(int i = 1; i <= ny; i++)
        if(dis[linky[i]][i] != -INF) // 不是虚边,匹配成功
            ans += dis[linky[i]][i];
    return ans;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++)
        fill(dis[i], dis[i] + n + 5, -INF);
}
int main() {
    int n1, n2, m;
    scanf("%d%d%d", &n1, &n2, &m);
    int f = 0;
    if(n1 < n2) {
        swap(n1, n2);
        f = 1;
    }
    // n1>=n2
    init(n1);
    while(m--) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        if(f)
            swap(u, v);
        dis[u][v] = max(dis[u][v], w);
    }
    printf("%d\n", KM(n1, n2));
}
```

1.23.4 二分图多重最大匹配：$O(m\sqrt{n})$

设每个点最多和 L_i 条边相连，建立源点 S 和汇点 T。

S 向左侧所有点 i 连一条流量为 L_i 的边，右侧所有点 j 向 T 连一条流量为 L_j 的边，原图中的边依然存在且流量为 1，求最大流。

1.23.5 二分图多重最大权匹配：$O(m\sqrt{n})$

设每个点最多和 L_i 条边相连，建立源点 S 和汇点 T。

S 向左侧所有点 i 连一条流量为 L_i，费用为 0 的边，右侧所有点 j 向 T 连一条流量为 L_j，费用为 0 的边，原图中的边依然，流量为 1 ，费用为边权值，求费用流。

1.24 一般图

性质：

1. 对于连通图，| 最大匹配 |+| 最小边覆盖 |=|V|
2. | 最大独立集 |+| 最小顶点覆盖 |=|V|

1.24.1 一般图匹配：$O(n(nlogn+m))$

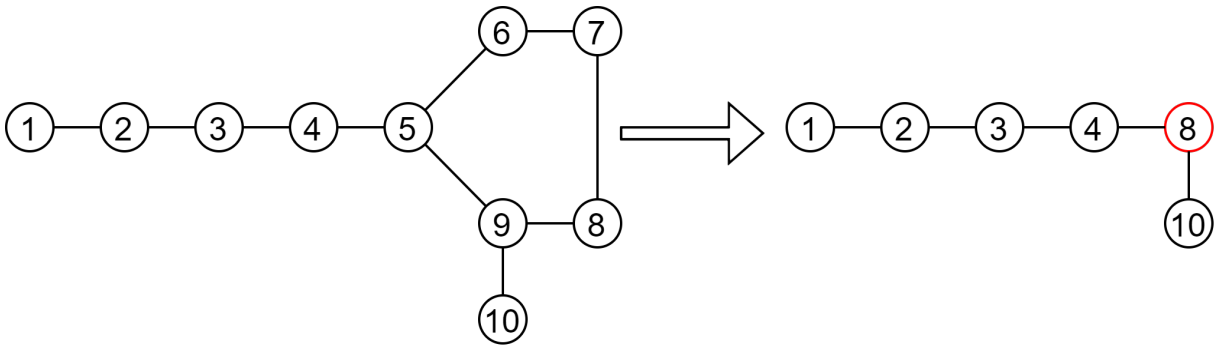
带花树算法流程：

找到未匹配的点 s ，标记为 A 类点，从 s 开始 bfs 。

若搜索到一个花内的点或是遇到偶环，不影响求解。

若搜索到一个未标记点 v ，若 v 有匹配，把 v 设为 B 类点，把它的匹配点 $match[v]$ 设为 A 类点，再把 $match[v]$ 加入队列；若 v 没有匹配，说明找到了一条增广路，原路返回展开带花树。

若搜索到一个已标记点 v ，且形成了奇环，求当前点 u 和 v 的最近公共花祖先，用并查集缩环。




```
vi e[MAXN];
//------------------------------
int fa[MAXN];
int Find(int x) {
```

```
5       return x == fa[x] ? x : fa[x] = Find(fa[x]);
6   }
7   //------------------------------
8   int col[MAXN], pre[MAXN], match[MAXN];
9   int tim = 0, dfn[MAXN];
10  int lca(int u, int v) {  // 最近公共花祖先
11      tim++;
12      u = Find(u);
13      v = Find(v);
14      while(true) { // 轮流向前跳
15          if(u) {
16              if(dfn[u] == tim)
17                  return u;
18              dfn[u] = tim;
19              u = Find(pre[match[u]]);
20          }
21          swap(u, v);
22      }
23  }
24  queue<int>q;
25  void blossom(int x, int y, int LCA) {  // 缩环
26      while(Find(x) != LCA) {
27          pre[x] = y;
28          y = match[x];
29          if(col[y] == 2) {  // 修改为A类点
30              col[y] = 1;
31              q.push(y);
32          }
33          if(Find(x) == x)  // 并查集连向LCA
34              fa[x] = LCA;
35          if(Find(y) == y)
36              fa[y] = LCA;
37          x = pre[y];
38      }
39  }
40  bool Aug(int s, int n) {
41      for(int i = 1; i <= n; i++) {
42          fa[i] = i;
43          col[i] = pre[i] = 0;
```

```
    }
    while(!q.empty())
        q.pop();
    //--------------------------------------
    q.push(s);
    col[s] = 1;  // A类点
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        for(auto v : e[u]) {
            // 花内或偶环不影响求解
            if(Find(u) == Find(v) || col[v] == 2)
                continue;
            if(!col[v]) {  // 未标记点
                pre[v] = u;
                if(!match[v]) {  // 未匹配
                    for(int x = v, tmp; x; x = tmp) {
                        tmp = match[pre[x]];
                        match[x] = pre[x]; // 更新匹配
                        match[pre[x]] = x; // 更新匹配
                    }
                    return true;
                }
                // 有匹配
                col[v] = 2;  // B类点
                col[match[v]] = 1;  // 匹配点为A类点
                q.push(match[v]);
            } else {  // 已标记点, 形成奇环
                int LCA = lca(u, v);
                blossom(u, v, LCA);
                blossom(v, u, LCA);
            }
        }
    }
    return false;
}
void init(int n) {
    tim = 0;
    for(int i = 0; i < n; i++) {
```

```
        e[i].clear();
        match[i] = dfn[i] = 0;
    }
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    init(MAXN);
    for(int i = 0; i < m; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].pb(v);
        e[v].pb(u);
    }
    int ans = 0;
    for(int i = 1; i <= n; i++)
        if(!match[i])
            ans += Aug(i, n);
    printf("%d\n", ans);
    for(int i = 1; i <= n; i++)
        printf("%d ", match[i]);
}
```

1.24.2 一般图最大权匹配：$O(n(nlogn+m))$

也是带花树算法来做，但这个我是真的不会，只能套板子了。

```
const int MAXN = 1e3 + 5; // 至少开两倍
int n_x;
int st[MAXN];
vi flower[MAXN];
int flo_from[MAXN][MAXN];
int lab[MAXN];
int match[MAXN];
int S[MAXN];
int slack[MAXN];
deque<int>q;
int dfn[MAXN];
```

```
12  int pa[MAXN];
13  int tim = 0;
14  struct Edge {
15      int u, v, w;
16  } g[MAXN][MAXN];
17  int dist(Edge e) {
18      return lab[e.u] + lab[e.v] - e.w * 2;
19  }
20  void q_push(int n, int x) {
21      if(x <= n)
22          return q.pb(x);
23      for(auto i : flower[x])
24          q_push(n, i);
25  }
26  int get_lca(int u, int v) {
27      for(tim++; u || v; swap(u, v)) {
28          if(!u)
29              continue;
30          if(dfn[u] == tim)
31              return u;
32          dfn[u] = tim;
33          u = st[match[u]];
34          if(u)
35              u = st[pa[u]];
36      }
37      return 0;
38  }
39  int get_pr(int b, int xr) {
40      int pr = find(flower[b].begin(), flower[b].end(), xr) - flower[b].begin();
41      if(pr % 2) {
42          reverse(flower[b].begin() + 1, flower[b].end());
43          return SZ(flower[b]) - pr;
44      } else
45          return pr;
46  }
47  void set_match(int n, int u, int v) {
48      match[u] = g[u][v].v;
49      if(u <= n)
50          return;
```

```
51      Edge e = g[u][v];
52      int xr = flo_from[u][e.u];
53      int pr = get_pr(u, xr);
54      for(int i = 0; i < pr; i++)
55          set_match(n, flower[u][i], flower[u][i ^ 1]);
56      set_match(n, xr, v);
57      rotate(flower[u].begin(), flower[u].begin() + pr, flower[u].end());
58  }
59  void Aug(int n, int u, int v) {
60      int xnv = st[match[u]];
61      set_match(n, u, v);
62      if(!xnv)
63          return;
64      set_match(n, xnv, st[pa[xnv]]);
65      Aug(n, st[pa[xnv]], xnv);
66  }
67  void set_st(int n, int x, int b) {
68      st[x] = b;
69      if(x <= n)
70          return;
71      for(auto i : flower[x])
72          set_st(n, i, b);
73  }
74  void update_slack(int u, int x) {
75      if(!slack[x] || dist(g[u][x]) < dist(g[slack[x]][x]))
76          slack[x] = u;
77  }
78  void set_slack(int n, int x) {
79      slack[x] = 0;
80      for(int u = 1; u <= n; ++u)
81          if(g[u][x].w > 0 && st[u] != x && S[st[u]] == 0)
82              update_slack(u, x);
83  }
84  void add_blossom(int n, int u, int LCA, int v) {
85      int b = n + 1;
86      while(b <= n_x && st[b])
87          b++;
88      n_x = max(n_x, b);
89      lab[b] = S[b] = 0;
```

```
90      match[b] = match[LCA];
91      flower[b].clear();
92      flower[b].pb(LCA);
93      for(int x = u, y; x != LCA; x = st[pa[y]]) {
94          flower[b].pb(x);
95          y = st[match[x]];
96          flower[b].pb(y);
97          q_push(n, y);
98      }
99      reverse(flower[b].begin() + 1, flower[b].end());
100     for(int x = v, y; x != LCA; x = st[pa[y]]) {
101         flower[b].pb(x);
102         y = st[match[x]];
103         flower[b].pb(y);
104         q_push(n, y);
105     }
106     set_st(n, b, b);
107     for(int i = 1; i <= n_x; i++)
108         g[b][i].w = g[i][b].w = 0;
109
110     for(int i = 1; i <= n; i++)
111         flo_from[b][i] = 0;
112
113     for(auto xs : flower[b]) {
114         for(int x = 1; x <= n_x; ++x)
115             if(g[b][x].w == 0 || dist(g[xs][x]) < dist(g[b][x])) {
116                 g[b][x] = g[xs][x];
117                 g[x][b] = g[x][xs];
118             }
119         for(int x = 1; x <= n; ++x)
120             if(flo_from[xs][x])
121                 flo_from[b][x] = xs;
122     }
123     set_slack(n, b);
124 }
125 void expand_blossom(int n, int b) {
126     for(auto i : flower[b])
127         set_st(n, i, i);
128     int xr = flo_from[b][g[b][pa[b]].u];
```

```
129     int pr = get_pr(b, xr);
130     for(int i = 0; i < pr; i += 2) {
131         int xs = flower[b][i];
132         int xns = flower[b][i + 1];
133         pa[xs] = xns;
134         S[xs] = 1;
135         S[xns] = 0;
136         slack[xs] = 0;
137         set_slack(n, xns);
138         q_push(n, xns);
139     }
140     S[xr] = 1;
141     pa[xr] = pa[b];
142     for(int i = pr + 1; i < SZ(flower[b]); i++) {
143         int xs = flower[b][i];
144         S[xs] = -1;
145         set_slack(n, xs);
146     }
147     st[b] = 0;
148 }
149 bool on_found_Edge(int n, const Edge &e) {
150     int u = st[e.u], v = st[e.v];
151     if(S[v] == -1) {
152         pa[v] = e.u;
153         S[v] = 1;
154         int nu = st[match[v]];
155         slack[v] = slack[nu] = 0;
156         S[nu] = 0;
157         q_push(n, nu);
158     } else if(S[v] == 0) {
159         int LCA = get_lca(u, v);
160         if(!LCA) {
161             Aug(n, u, v);
162             Aug(n, v, u);
163             return true;
164         } else
165             add_blossom(n, u, LCA, v);
166     }
167     return false;
```

```
}
bool matching(int n) {
    for(int i = 0; i < n_x + 5; i++) {
        S[i] = -1;
        slack[i] = 0;
    }
    q.clear();
    for(int i = 1; i <= n_x; i++)
        if(st[i] == i && !match[i]) {
            S[i] = pa[i] = 0;
            q_push(n, i);
        }
    if(q.empty())
        return false;

    while(true) {
        while(!q.empty()) {
            int u = q.front();
            q.pop_front();
            if(S[st[u]] == 1)
                continue;
            for(int v = 1; v <= n; v++)
                if(g[u][v].w > 0 && st[u] != st[v]) {
                    if(dist(g[u][v]) == 0 && on_found_Edge(n, g[u][v]))
                        return true;
                    else if(dist(g[u][v]) != 0)
                        update_slack(u, st[v]);
                }
        }
        int d = INF;
        for(int b = n + 1; b <= n_x; ++b)
            if(st[b] == b && S[b] == 1)
                d = min(d, lab[b] / 2);
        for(int x = 1; x <= n_x; ++x)
            if(st[x] == x && slack[x]) {
                if(S[x] == -1)
                    d = min(d, dist(g[slack[x]][x]));
                else if(S[x] == 0)
                    d = min(d, dist(g[slack[x]][x]) / 2);
```

```
            }
        for(int u = 1; u <= n; ++u) {
            if(S[st[u]] == 0) {
                if(lab[u] <= d)
                    return false;
                lab[u] -= d;
            } else if(S[st[u]] == 1)
                lab[u] += d;
        }
        for(int b = n + 1; b <= n_x; ++b)
            if(st[b] == b) {
                if(S[st[b]] == 0)
                    lab[b] += d * 2;
                else if(S[st[b]] == 1)
                    lab[b] -= d * 2;
            }
        q.clear();
        for(int x = 1; x <= n_x; ++x)
            if(st[x] == x && slack[x] && st[slack[x]] != x && !dist(g[slack[x
                ]][x]))
                if(on_found_Edge(n, g[slack[x]][x]))
                    return true;
        for(int b = n + 1; b <= n_x; ++b)
            if(st[b] == b && S[b] == 1 && lab[b] == 0)
                expand_blossom(n, b);
    }
    return false;
}
pair<ll, int> solve(int n) {
    ll sum = 0;
    int cnt = 0, maxw = 0;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            maxw = max(maxw, g[i][j].w);
    for(int i = 1; i <= n; i++)
        lab[i] = maxw;

    while(matching(n))
        cnt++;
```

```
245
246     for(int i = 1; i <= n; i++)
247         if(match[i] && match[i] < i)
248             sum += (ll)g[i][match[i]].w;
249
250     return mp(sum, cnt);
251 }
252 void init(int n) {
253     n_x = n;
254     tim = 0;
255     for(int i = 0; i < n + 5; i++) {
256         for(int j = 0; j < n + 5; j++) {
257             g[i][j] = Edge{i, j, 0};
258             flo_from[i][j] = 0;
259         }
260         flo_from[i][i] = i;
261         match[i] = 0;
262         st[i] = i;
263         flower[i].clear();
264     }
265 }
266 int main() {
267     int n, m;
268     scanf("%d%d", &n, &m);
269     init(n);
270     for(int i = 0; i < m; i++) {
271         int u, v, w;
272         scanf("%d%d%d", &u, &v, &w);
273         w = max(w, g[u][v].w);
274         g[u][v].w = g[v][u].w = w;
275     }
276     printf("%lld\n", solve(n).fi);
277     for(int i = 1; i <= n; i++)
278         printf("%d ", match[i]);
279 }
```

1.25 换根 dp

做两遍 dfs，第一次求出所有点的子树的贡献；第二次求出以每个点为根时，整棵树的贡献，一般通过当前点和父节点的信息进行转移。

模板题：

给一棵无根树，每个点的贡献为点到根的距离 $+1$，求整棵树的最大贡献。

```
vi e[MAXN];
int sz[MAXN];
ll dp[MAXN];
void dfs(int u, int fa) {
    sz[u] = 1;
    dp[u] = 0;
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs(v, u);
        sz[u] += sz[v];
        dp[u] += dp[v];
    }
    dp[u] += (ll)sz[u];
}
void dfs2(int u, int fa) {
    if(fa != 0) {
        ll tmp = dp[fa] - sz[u] - dp[u];
        dp[u] += tmp + sz[fa] - sz[u];
        sz[u] = sz[fa];
    }
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs2(v, u);
    }
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n - 1; i++) {
        int u, v;
```

```
        scanf("%d%d", &u, &v);
        e[u].pb(v);
        e[v].pb(u);
    }
    dfs(1, 0);
    dfs2(1, 0);
    ll ans = 0;
    for(int i = 1; i <= n; i++)
        ans = max(ans, dp[i]);
    printf("%lld\n", ans);
}
```

1.26 树哈希

1.26.1 有根树哈希

判断有根树同构：通过根的哈希值判断。

```
const int mod = 1e9 + 7;
const int base = 311;
int sz[MAXN], Hash[MAXN];
void gethash1(int u, int fa) { // 得到每个点的子树的哈希值
    sz[u] = 1;
    Hash[u] = 0;
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        gethash1(v, u);
        Hash[u] = (Hash[u] + (ll)sz[v] * Hash[v] % mod) % mod;
        sz[u] += sz[v];
    }
    Hash[u] ^= (ll)sz[u] * base % mod;
}
```

1.26.2 无根树哈希

判断无根树同构：通过比较以重心为根的哈希值判断。

1 先求重心，以重心为根求哈希值

```
vi zx; // 存重心
int mima, sumsz;
void getzx(int u, int fa) { // 求重心
    sz[u] = 1;
    int maxson = 0;
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        getzx(v, u);
        sz[u] += sz[v];
        maxson = max(maxson, sz[v]);
    }
    maxson = max(maxson, sumsz - sz[u]);
    if(mima == maxson)
        zx.pb(u);
    if(mima > maxson) {
        mima = maxson;
        zx.clear();
        zx.pb(u);
    }
}
void init(int x) { // x为整棵树的大小
    sumsz = x;
    mima = INF;
    zx.clear();
}
int main() {
    init(n);
    getzx(1, 0);
    vi tmp;
    for(int i = 0; i < SZ(zx); i++) {
        int rt = zx[i];
        gethash(rt, 0); // 以重心为根哈希
        tmp.pb(Hash[rt]);
    }
    sort(tmp.begin(), tmp.end());
    // 同构树得到的tmp相同
```

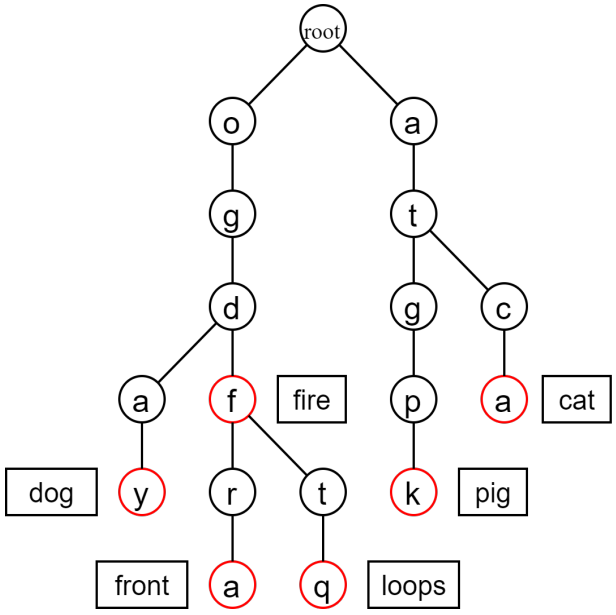
```
}
```

2 换根 dp 求所有点为根的哈希值，再求重心

```
void gethash2(int u, int fa, int n) { // 得到所有点为根时的哈希
    int fh = Hash[fa];
    fh ^= 1ll * n * base % mod;
    fh = (fh - 1ll * Hash[u] * sz[u] % mod + mod) % mod;
    fh ^= 1ll * (n - sz[u]) * base % mod;
    Hash[u] ^= 1ll * sz[u] * base % mod;
    Hash[u] = (Hash[u] + 1ll * (n - sz[u]) * fh % mod) % mod;
    Hash[u] ^= 1ll * n * base % mod;
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        gethash2(v, u, n);
    }
}
int main() {
    // ...
    gethash1(1, 0);
    gethash2(1, 0, n);
    // ...
}
```

1.27 字典树 Trie

模板题：
假设存在一种新语言为 X 语，提供一些英语单词和 X 语单词的对应关系。
现在有一些查询，每个查询给一个 X 语单词，问对应的英语单词是什么。
插入 $O(len)$ ，查找 $O(len)$，len 为最长单词的长度，也是字典树的高度。



映射关系:

ogday --- dog

atca --- cat

atgpk --- pig

ogdfra --- front

ogdftq --- loops

ogdf --- fire

```
int cnt = 0; // 0为根节点
int trie[MAXN][26];
string ans[MAXN];
void insert(string s, string t) {
    int p = 0;
    for(int i = 0; i < SZ(s); i++) {
        int x = s[i] - 'a';
        if(!trie[p][x])
            trie[p][x] = ++cnt; // 增加结点
        p = trie[p][x];
    }
    ans[p] = t;
}
string find(string s) {
    int p = 0;
    for(int i = 0; i < SZ(s); i++) {
        int x = s[i] - 'a';
        if(!trie[p][x])
            return "No";
        p = trie[p][x];
    }
    if(ans[p] == "")
```

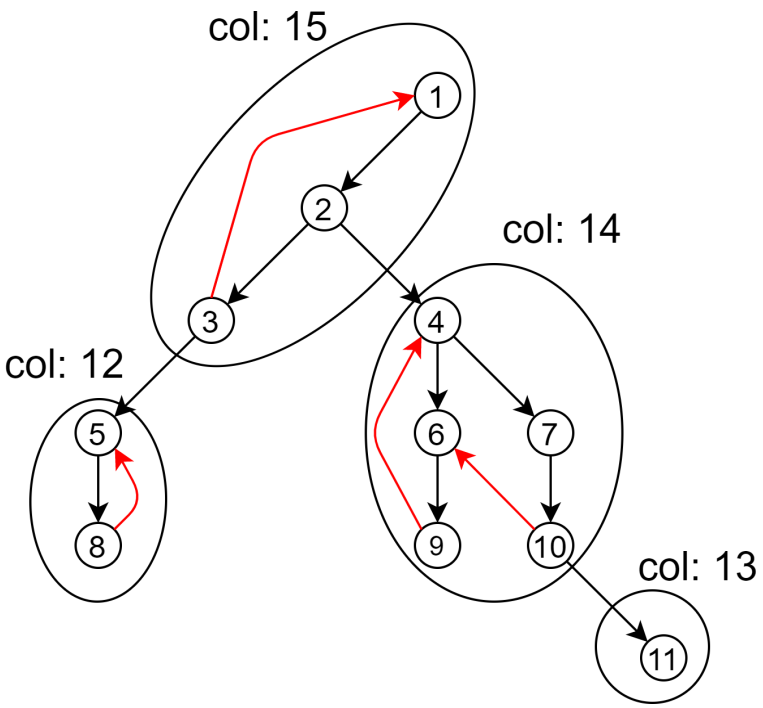
```
23          return "No";
24      return ans[p];
25  }
```

1.28 tarjan 算法

在 tarjan 算法中，把一张图内的边分为四种：
树边：dfs 树上的边。
回边：连向树上祖先结点（不包括父节点）的边。
前向边：连向后代（不包括子节点）的边。
横跨边：连向非祖先且非后代的边。

1.28.1 有向图的强连通分量

定义：任意两点都可互达的极大子图
性质：得到的 SCC 的标号序列满足逆拓扑序



```cpp
const int MAXN = ; // MAXN开2倍
vi e[MAXN];
int tim = 0; // 时间戳
int dfn[MAXN]; // 点的遍历序号
int low[MAXN]; // 回边/横跨边能抵达的最小时间戳
int col[MAXN], cnum; // 染色
int stk[MAXN], top = 0; // 手写栈
int ins[MAXN]; // 是否在栈内
void tarjan(int u) {
    low[u] = dfn[u] = ++tim;
    stk[++top] = u; // 进栈
    ins[u] = 1;
    for(auto v : e[u])
        if(!dfn[v]) { // 树边
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if(ins[v])  // 回边/前向边/横跨边
            low[u] = min(low[u], dfn[v]);
    if(dfn[u] == low[u]) { // u是SCC的开头
        cnum++; // SCC缩成一个新点
        while(stk[top + 1] != u) {
            col[stk[top]] = cnum; // 染色
            ins[stk[top--]] = 0; // 出栈
        }
    }
}
void init(int n) {
    tim = top = 0;
    cnum = n;
    for(int i = 0; i < n + 5; i++) {
        dfn[i] = low[i] = ins[i] = col[i] = 0;
        e[i].clear();
    }
}
int main() {
    // ...
    init(n);
    for(int i = 1; i <= n; i++)
        if(!dfn[i])
```

```
            tarjan(i);
    // ...
}
```

1.28.2 无向图的点双连通分量

定义：不存在割点的极大子图

性质：若点双中存在奇环，则连通分量内所有点至少在一个奇环上。

```
vi e[MAXN];
int tim = 0, dfn[MAXN], low[MAXN];
int stk[MAXN], top = 0;
void tarjan(int u) {
    stk[++top] = u; // 进栈
    low[u] = dfn[u] = ++tim;
    for(auto v : e[u]) {
        if(!dfn[v]) { // 树边
            tarjan(v);
            low[u] = min(low[u], low[v]);
            if(low[v] >= dfn[u]) { // u是割点或根
                vi tmp; // 点双
                int f = 1;
                while(f) { // v子树弹出
                    tmp.pb(stk[top]);
                    if(stk[top] == v)
                        f = 0;
                    stk[top--] = 0;
                }
                tmp.pb(u);
                // tmp中存着当前点双的所有点
            }
        } else // 回边
            low[u] = min(low[u], dfn[v]);
    }
    // if(e[u].empty()) 说明u是孤立点,也是一个点双
}
void init(int n) {
    tim = top = 0;
```

```
    for(int i = 0; i < n + 5; i++) {
        stk[i] = dfn[i] = low[i] = 0;
        e[i].clear();
    }
}
int main() {
    //...
    init(n);
    for(int i = 1; i <= n; i++)
        if(!dfn[i])
            tarjan(i);
    //...
}
```

1.28.3 无向图的边双连通分量

定义：不存在割边的极大子图

性质：边双连通分量缩点后会形成一棵树

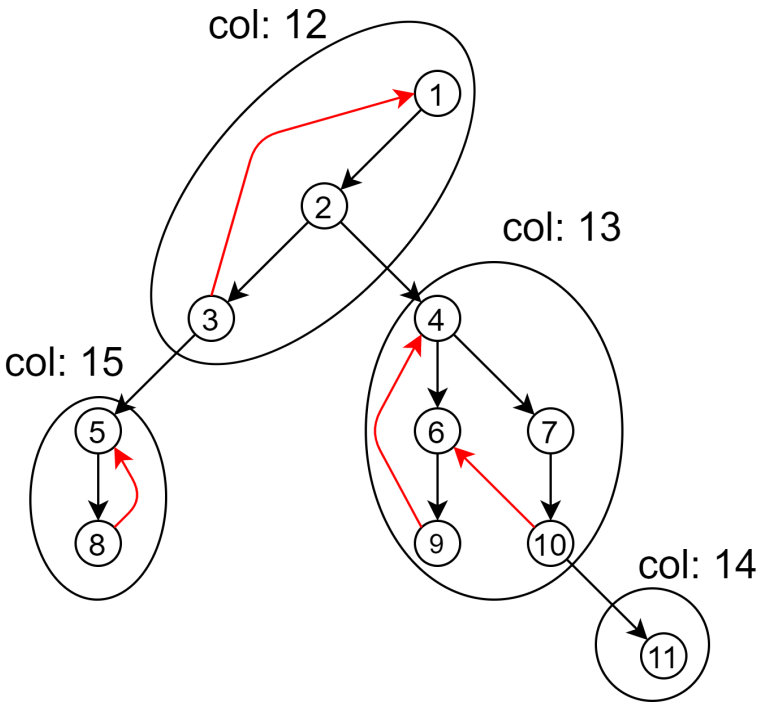
```
const int MAXN = ; // MAXN开两倍
//前向星,从2开始
//---------------------------------
int tim = 0, dfn[MAXN], low[MAXN];
int cut[MAXM * 2]; // 标记割边
vector<pii>cute; // 存割边
void tarjan(int u, int fe) {
    low[u] = dfn[u] = ++tim;
    for(int i = head[u]; i; i = e[i].next) {
        int v = e[i].to;
        if(!dfn[v]) { // 树边
            tarjan(v, i ^ 1);
            low[u] = min(low[u], low[v]);
            if(low[v] == dfn[v]) { // 标记割边
                cut[i] = cut[i ^ 1] = 1;
                cute.pb({u, v});
            }
        } else if(i != fe) // 排除fa->u的边
            low[u] = min(low[u], dfn[v]);
```

```
    }
}
int cnum, col[MAXN];
void paint(int u, int cnow) { // 染色缩点
    col[u] = cnow;
    for(int i = head[u]; i; i = e[i].next) {
        int v = e[i].to;
        if(cut[i] || col[v])
            continue;
        paint(v, cnow);
    }
}
void init(int n, int m) {
    cnt = 1; // 前向星从2开始
    tim = 0;
    cnum = n;
    cute.clear();
    for(int i = 0; i < n + 5; i++)
        head[i] = dfn[i] = low[i] = col[i] = 0;
    for(int i = 0; i < m * 2 + 5; i++)
        cut[i] = 0;
}
int main() {
    // ...
    init(n, m);
    for(int  i = 1; i <= n; i++)
        if(!dfn[i])
            tarjan(i, 0);
    for(int i = 1; i <= n; i++) // 缩点
        if(!col[i])
            paint(i, ++cnum);
    // ...
}
```

1.29 kosaraju 算法

kosaraju 算法用于求强连通分量。

性质：得到的 SCC 的标号序列满足拓扑序




```cpp
const int MAXN = ; // MAXN开2倍
vector<pii> e[MAXN];
int vis[MAXN];
int lat[MAXN], top = 0; // 后序遍历
void dfs1(int u) {
    vis[u] = 1;
    for(auto i : e[u]) {
        int v = i.fi, f = i.se;
        if(vis[v] || f < 0) // 跑原边
            continue;
        dfs1(v);
    }
    lat[++top] = u;
}
int col[MAXN], cnum;
void dfs2(int u) {
    col[u] = cnum;
    for(auto i : e[u]) {
        int v = i.fi, f = i.se;
```

```
        if(col[v] || f > 0) // 跑反向边
            continue;
        dfs2(v);
    }
}
void kosaraju(int n) {
    top = 0;
    for(int i = 1; i <= n; i++)
        if(!vis[i])
            dfs1(i);
    for(int i = top; i >= 1; i--)
        if(!col[lat[i]]) {
            cnum++;
            dfs2(lat[i]);
        }
}
void init(int n) {
    cnum = n;
    for(int i = 0; i < n + 5; i++)
        col[i] = vis[i] = 0;
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    init(n);
    while(m--) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].pb({v, 1}); // 原边
        e[v].pb({u, -1}); // 反向边
    }
    kosaraju(n);
    // ...
}
```

1.30 支配树

流图：从一个点 $root$ 出发能抵达所有点的有向图

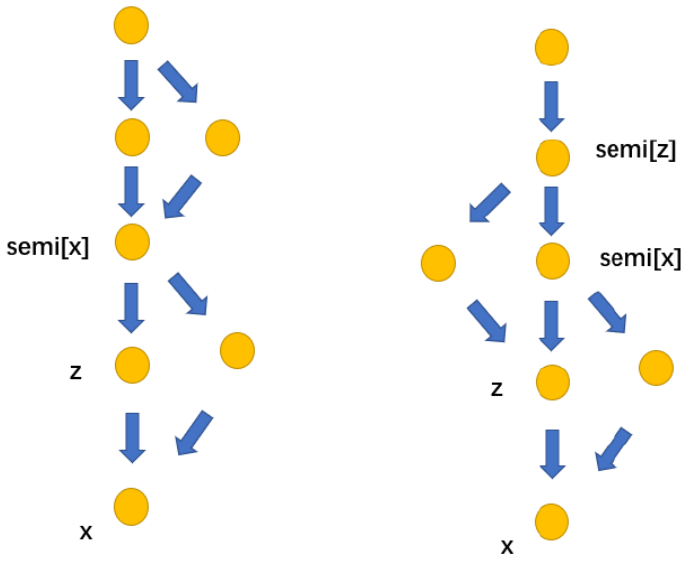
必经点：在流图中，从 $root$ 到某点 u 必须经过的点，被称为 u 的必经点

最近必经点：dfn 最大的必经点，记为 idom

支配树：流图的支配树是一棵有向树，从 $root$ 出发能抵达所有点。对每条边 (u,v)，满足 u 是 v 的最近必经点。从 $root$ 到 u 经过的所有点构成 u 的必经点集合。

半必经点：对点 y，若 x 能通过一些 $dfn > dfn[y]$ 的点（不包括 x 和 y）抵达 y，且 x 是满足条件的点中 dfn 最小的，则 x 是 y 的半必经点。半必经点唯一，且不一定属于必经点。

Lengauer_Tarjan 算法用于求支配树，复杂度为 $O((n+m)logn)$。




```
1  struct edge {
2      vi e[MAXN];
3      void add_edge(int u, int v) {
4          e[u].pb(v);
5      }
6  } a, b, c, d; // a为原图,b为反向图,c临时存放,d为支配树
7  int dfn[MAXN];
8  int id[MAXN]; // dfn的反数组
9  int fa[MAXN];
10 int tim = 0;
11 void get_dfn(int u) { // 得到dfn
12     dfn[u] = ++tim;
13     id[tim] = u;
14     for(auto v : a.e[u]) {
15         if(dfn[v])
```

```
            continue;
        fa[v] = u;
        get_dfn(v);
    }
}
int idom[MAXN]; // 最近必经点
int sdom[MAXN]; // 半必经点
int val[MAXN]; // 祖先链中最小的dfn
int pre[MAXN];
int find(int x) { // 计算x的sdom
    if(x == pre[x])
        return x;
    int root = find(pre[x]);
    if(dfn[sdom[val[pre[x]]]] < dfn[sdom[val[x]]])
        val[x] = val[pre[x]];
    return pre[x] = root;
}
void lengauer_tarjan(int root) {
    get_dfn(root);
    for(int i = tim; i > 1; i--) { // 计算sdom
        int u = id[i];
        for(auto v : b.e[u]) { // 反向边v->u
            if(!dfn[v]) // 不在dfs树上
                continue;
            find(v);
            int a = val[v];
            if(dfn[sdom[a]] < dfn[sdom[u]]) // 更新
                sdom[u] = sdom[a];
        }
        c.add_edge(sdom[u], u);
        pre[u] = fa[u];
        u = fa[u];
        for(auto v : c.e[u]) {
            find(v);
            if(sdom[val[v]] == u)
                idom[v] = u;
            else
                idom[v] = val[v];
        }
```

```
    }
    for(int i = 2; i <= tim; i++) { // 修正
        int u = id[i];
        if(idom[u] != sdom[u])
            idom[u] = idom[idom[u]];
    }
}
void init(int n) {
    tim = 0;
    for(int i = 0; i < n + 5; i++) {
        sdom[i] = pre[i] = val[i] = i;
        a.e[i].clear();
        b.e[i].clear();
        c.e[i].clear();
        d.e[i].clear();
        idom[i] = dfn[i] = id[i] = fa[i] = 0;
    }
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    init(n);
    while(m--) {
        int u, v;
        scanf("%d%d", &u, &v);
        a.add_edge(u, v); // 原边
        b.add_edge(v, u); // 反向边
    }
    int root = 1; // 流图的根节点
    lengauer_tarjan(root);
    for(int i = 1; i <= n; i++) // 建支配树
        d.add_edge(idom[i], i);
    // ...
}
```

1.31 虚树

只处理原树的某些关键点和部分关联的 LCA ，忽略其它点。

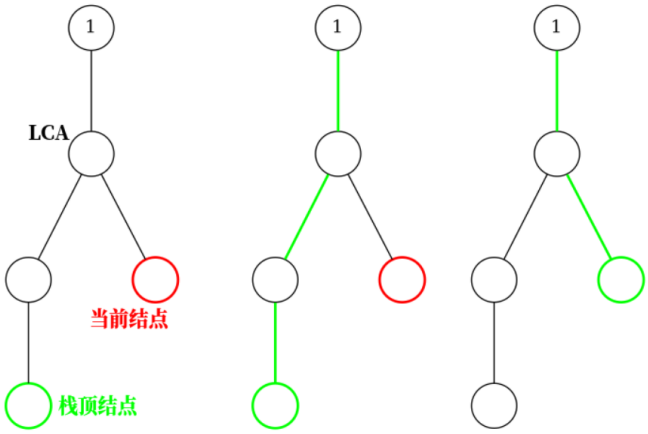
将关键点按 dfs 序排序，用单调栈处理。

首先，将根节点入栈。

按 dfs 序从小到大添加关键点。

若当前点 u 和栈顶点 v 的 LCA = v，说明 u 和 v 在一条链上，将 u 入栈；

若 LCA v，如下左图，绿色的链表示栈内当前存的链，需要把中间的图变成右边的图。



那么把原栈内的节点不停弹栈并连边，直到次大节点的 dfs 序小于等于 LCA 的 dfs 序为止，如果此时栈首不是 LCA，要让 LCA 入栈，最后再把当前节点入栈。

假设原树中有 k 个关键结点，建虚树的复杂度为 $O(klogk)$ 。

```
vector<pii>e[MAXN];
int k, h[MAXN]; // 存关键点
int dfn[MAXN], tim = 0; // dfs序时间戳
int dep[MAXN], f[MAXN][25], dis[MAXN][25];
void dfs(int u, int fa) { // 倍增求LCA
    dfn[u] = ++tim;
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for(int i = 1; (1 << i) <= dep[u]; i++) {
        f[u][i] = f[f[u][i - 1]][i - 1];
        dis[u][i] = min(dis[u][i - 1], dis[f[u][i - 1]][i - 1]);
    }
    for(auto i : e[u]) {
        int v = i.fi;
        if(v == fa)
            continue;
        dis[v][0] = i.se;
```

```
18          dfs(v, u);
19      }
20  }
21  pii lca(int x, int y) { // 返回LCA和虚树边权
22      int ans = INF; // 这里的虚树边权为原树路径上最小边权
23      if(dep[x] < dep[y])
24          swap(x, y);
25      for(int i = 20; i >= 0; i--) {
26          if(dep[f[x][i]] >= dep[y]) {
27              ans = min(ans, dis[x][i]);
28              x = f[x][i];
29          }
30          if(x == y)
31              return mp(x, ans);
32      }
33      for(int i = 20; i >= 0; i--)
34          if(f[x][i] != f[y][i]) {
35              ans = min(ans, min(dis[x][i], dis[y][i]));
36              x = f[x][i];
37              y = f[y][i];
38          }
39      ans = min(ans, min(dis[x][0], dis[y][0]));
40      return mp(f[x][0], ans);
41  }
42  bool cmp(int x, int y) {
43      return dfn[x] < dfn[y];
44  }
45  vector<pii> e2[MAXN]; // 虚树边
46  int stk[MAXN], top = 0; // 单调栈
47  void build(int k) {
48      sort(h + 1, h + k + 1, cmp); // 关键点排序
49      stk[top = 1] = 1; // 根节点入栈
50      e2[1].clear(); // 首次入栈清空邻接表
51      for(int i = 1; i <= k; i++) {
52          int u = h[i];
53          int LCA = lca(u, stk[top]).fi;
54          if(LCA != stk[top]) {
55              // u不在栈所存的链上
56              while(dfn[LCA] < dfn[stk[top - 1]]) {
```

```
                // 次大节点的dfs序大于LCA的dfs序
                int x = stk[top - 1], y = stk[top];
                e2[x].pb({y, lca(x, y).se});
                top--;
            }
            if(dfn[LCA] > dfn[stk[top - 1]]) {
                // LCA首次入栈
                e2[LCA].clear();
                e2[LCA].pb({stk[top], lca(LCA, stk[top]).se});
                stk[top] = LCA;
            } else {
                // LCA就是次大节点，弹出栈顶
                e2[LCA].pb({stk[top], lca(LCA, stk[top]).se});
                top--;
            }
        }
        // u首次入栈
        e2[u].clear();
        stk[++top] = u;
    }
    for(int i = 1; i < top; i++) { // 栈内链连接
        int u = stk[i], v = stk[i + 1];
        e2[u].pb({v, lca(u, v).se});
    }
}
int main() {
    // ...
    dfs(1, 0);
    build(k);
    // ...
}
```

1.32 最小树形图：朱刘算法

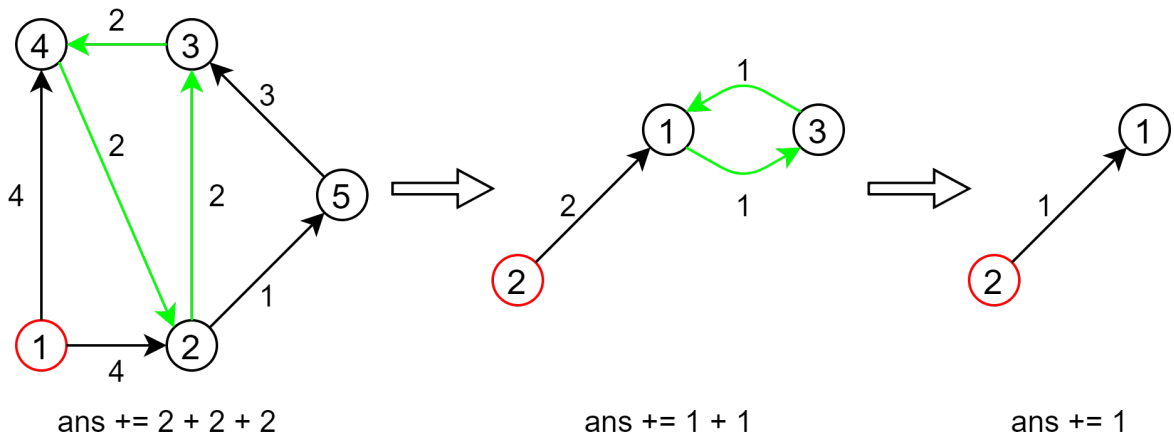
1.32.1 定根最小树形图

给定一张流图和源点 r ，以 r 为根的最小权叶向生成树称为最小树形图。

朱刘算法流程：

1. 对 r 以外的点 x ，找出每个点的入边中边权最小的边，存在 $pre[x]$ 和 $in[x]$ 中。把所

有点的 in 加入答案，如果这些边不构成环，算法结束。2. 如果这些边构成环，把环缩点，对原图中 v 在环内的边 (u,v,w)，$w = w - in[v]$ 。3. 重复 1、2 直到算法结束。




```cpp
struct edge {
    int u, v, w;
} e[MAXM];
int pre[MAXN], in[MAXN]; // 边权最小的入边的u和w
int vis[MAXN];
int id[MAXN];
int zhu_liu(int n, int m, int root) {
    int ans = 0;
    while(true) {
        fill(in + 1, in + n + 1, INF);
        in[root] = 0;
        for(int i = 1; i <= m; i++) { // 找到最小入边
            int u = e[i].u, v = e[i].v;
            if(u != v && e[i].w < in[v]) {
                in[v] = e[i].w;
                pre[v] = u;
            }
        }
        int tim = 0;
        for(int i = 1; i <= n; i++)
            vis[i] = id[i] = 0;
        for(int i = 1; i <= n; i++) {
```

```
            if(in[i] == INF) // i是孤立点,无解
                return -1;
            ans += in[i];
            int v = i;
            while(vis[v] != i && v != root && !id[v]) { // 找环的开头
                vis[v] = i;
                v = pre[v];
            }
            if(v != root && !id[v]) { // 标记环上的点
                id[v] = ++tim;
                for(int u = pre[v]; u != v; u = pre[u])
                    id[u] = tim;
            }
        }
        if(!tim) // 无环
            break;
        for(int i = 1; i <= n; i++) // 标记剩余点
            if(!id[i])
                id[i] = ++tim;
        for(int i = 1; i <= m; i++) { // 缩点
            int u = e[i].u, v = e[i].v;
            e[i].u = id[u];
            e[i].v = id[v];
            if(id[u] != id[v])
                e[i].w -= in[v];
        }
        root = id[root];
        n = tim;
    }
    return ans;
}
void init(int n) {
    fill(pre, pre + n + 5, 0);
}
int main() {
    int n, m, r;
    scanf("%d%d%d", &n, &m, &r);
    init(n);
    for(int i = 1; i <= m; i++)
```

```
62          scanf("%d%d%d", &e[i].u, &e[i].v, &e[i].w);
63      printf("%d", zhu_liu(n, m, r));
64  }
```

1.32.2 不定根最小树形图

把原图中所有的边权扩大 $n+1$ 倍，记 $sum=\sum w_i+n$ 。

建超级源点 $S=n+1$ ，S 向所有点 i 连一条边权为 $sum+i$ 的边，以 S 为根跑朱刘算法。

令 $tmp=zhu\_liu(n+1,m+n,S)$ ，若 $tmp>sum*2||tmp==-1$ ，说明无解。

最小树形图的总边权 $ans=(tmp-sum)/(n+1)$ ，根节点 $rt=(tmp-sum)\%(n+1)$ 。（此时的根节点是所有最小树形图中根节点编号最小的）

```
1   int main() {
2       // ...
3       scanf("%d%d", &n, &m);
4       init(n);
5       ll sum = n;
6       for(int i = 1; i <= m; i++) {
7           int u, v;
8           ll w;
9           scanf("%d%d%lld", &u, &v, &w);
10          w *= (n + 1);
11          e[i] = {u, v, w};
12          sum += w;
13      }
14      int S = n + 1;
15      for(int i = 1; i <= n; i++)
16          e[m + i] = {S, i, sum + i};
17      ll tmp = zhu_liu(n + 1, m + n, S);
18      if(tmp > sum * 2 || tmp == -1) // 无解
19          printf("No\n");
20      else {
21          ll ans = (tmp - sum) / (n + 1); // 总边权
22          int rt = (tmp - sum) % (n + 1); // 根节点
23          // ...
24      }
25  }
```

1.33 2-SAT

对于节点 x ，用 x 表示 x=0 ，用 x+n 表示 x=1 ，根据变量关系建图。

1 判断是否有解：

求图的 SCC ，若 x 和 x+n 位于同一个 SCC 中则无解，否则一定有解。

2 求一组具体的解：

可以通过拓扑序判断：如果 x 的拓扑序在 $\neg x$ 之后，取 x 为真。

如果用 tarjan 求 SCC,由于得到 SCC 的标号满足逆拓扑序,$ans[i] = col[i] < col[i+n]?0 : 1$ 。

1.34 差分约束系统

注意节点之间的隐藏条件！

① A - B >= C 相当于：B 连向 A，权为 C 的边

② A - B <= C 相当于：B 连向 A，权为 C 的边

③ A - B = C 可转化为 ①+②

④ A - B < C 如果在整数域上，可转化为 A - B <= C - 1

1. 判断是否有解：

新建源点 S ，S 向每个点连一条边权为 0 的边。

用形如②的不等式建图，bellman_ford 或 spfa 求最短路，若存在负环则无解，否则有解。

2. 求一组具体解：

跑完最短路后，取 $x_i = dis[i]$ 即为一组解。

3. 求两个变量的最大差值：

用形如②的不等式建图，两点的最短路即为最大差值

此时若存在负环（无最短路），表示最大差值不存在（无限小）；若两点不连通，表示最大差值为无限大。

4. 求两个变量的最小差值：

用形如①的不等式建图，两点的最长路即为最小差值

此时若存在正环，表示最小差值为无限大；若两点不连通，表示最小差值不存在（无限小）。

1.35 同余最短路

问题形式：

给定 n 个整数，求这 n 个整数能拼凑出哪些其他整数或求这 n 个整数不能拼凑出的最小整数。（n 个整数可以多次取）

解决方法：

令 n 个整数按升序排列依次为 $a_1, a_2, \cdots, a_n$ 。

在图中构造 $0, 1, \cdots, (a_1-2), (a_1-1)$ 这 a_1 个点，代表 $\bmod a_1$ 后的余数。对于每个点 u ，可以用 $a_2, \cdots, a_n$ 建 $n-1$ 条有向边，即从 u 到 $(u+a_i) \bmod a_1$ 连一条权为 a_i 的边。

跑最短路，得到的 dis_i 就代表用这 n 个整数能组成的 $\bmod a_1 = i$ 的最小的整数。

模板题：

给出 $n, L, R, a_1, a_2, \cdots, a_n$ ，求有多少个 $b \in [L, R]$ 满足 $\sum_{i=1}^{n} a_i x_i = b$ 存在非负整数解。

```
#define pil pair<int,ll>
#define pli pair<ll,int>
vector<pil>e[MAXN];
ll dis[MAXN];
int vis[MAXN];
priority_queue<pli, vector<pli>, greater<pli> >q;
void dijkstra(int n, int s) {
    fill(dis, dis + n + 5, LLINF);
    fill(vis, vis + n + 5, 0);
    dis[s] = 0; // 从s开始
    q.push(mp(0, s));
    while(!q.empty()) {
        int x = q.top().se;
        q.pop();
        if(vis[x])
            continue;
        vis[x] = 1;
        for(auto i : e[x]) { // 用x更新
            int v = i.fi;
            ll w = i.se;
            if(dis[v] > dis[x] + w) {
                dis[v] = dis[x] + w;
                q.push(mp(dis[v], v));
            }
        }
    }
}
```

```cpp
int a[MAXN];
int main() {
    int n;
    ll L, R;
    scanf("%d%lld%lld", &n, &L, &R);
    for(int i = 1; i <= n; i++)
        scanf("%d", a + i);
    sort(a + 1, a + n + 1);
    for(int u = 0; u < a[1]; u++)
        for(int i = 2; i <= n; i++) {
            int  v = (u + a[i]) % a[1];
            e[u].pb({v, a[i]});
        }
    int s = a[1];
    e[s].pb({0, 0});
    dijkstra(a[1] + 1, s);
    ll ans = 0;
    for(int i = 0; i < a[1]; i++) {
        if(dis[i] <= R)
            ans += (R - dis[i]) / a[1] + 1;
        if(dis[i] <= L - 1)
            ans -= (L - 1 - dis[i]) / a[1] + 1;
    }
    printf("%lld\n", ans);
}
```

1.36 拓扑排序：$O(n+m)$

Kahn 算法流程：

每次取出一个入读为 0 的点 u，加入拓扑序列，在图中把 u 和相关的边删去，重复这一操作直到图中无点。

如果没有入读为 0 的点了，但图中仍有点剩余，说明有环。

1.36.1 bfs 版本：$O(n+m)$

```cpp
vi e[MAXN];
int in[MAXN];
bool topo(int n, vi& ans) {
```

```
    queue<int>q;
    for(int i = 1; i <= n; i++)
        if(!in[i])
            q.push(i);
    while(!q.empty()) {
        int u = q.front();
        ans.pb(u);
        q.pop();
        for(auto v : e[u])
            if(--in[v] == 0)
                q.push(v);
    }
    if(SZ(ans) != n)
        return false;
    return true;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        in[i] = 0;
        e[i].clear();
    }
}
```

1.36.2 dfs 版本：$O(n+m)$

```
vi e[MAXN];
int in[MAXN], vis[MAXN];
vi ans;
void dfs(int u) {
    vis[u] = 1;
    ans.pb(u);
    for(auto v : e[u])
        if(--in[v] == 0)
            dfs(v);
}
bool topo(int n) {
    for(int u = 1; u <= n; u++)
        if(!in[u] && !vis[u])
            dfs(u);
```

```
    if(SZ(ans) != n)
        return false;
    return true;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        in[i] = vis[i] = 0;
        e[i].clear();
    }
    ans.clear();
}
```

1.37 Bron-Kerbosch 算法：$O(3^{\frac{n}{3}})$

Bron-Kerbosch 算法用于求无向图的最大团或最大独立集。

定义：最大团就是点数最多的完全子图

性质：原图的最大独立集 = 补图的最大团

1.37.1 只求最大团

```
int path[MAXN][MAXN];
int vis[MAXN]; // 当前极大团中的点
int siz[MAXN]; // 所在极大团的大小
int res, group[MAXN]; // 最大团
bool dfs(int n, int u, int num) {
    for(int i = u + 1; i <= n; i++) {
        if(siz[i] + num <= res) // 剪枝
            return false;
        if(path[u][i]) {
            int j = 0;
            for(; j < num; j++)
                if(!path[i][vis[j]])
                    break;
            if(j == num) { // 皆与i相连
                vis[num] = i;
                if(dfs(n, i, num + 1))
                    return true;
            }
```

```
        }
    }
    if(num > res) {
        for(int i = 0; i < num; i++)
            group[i] = vis[i];
        res = num;
        return true;
    }
    return false;
}
void bron_kerbosch(int n) {
    res = 0;
    for(int i = n; i > 0; i--) {
        vis[0] = i;
        dfs(n, i, 1);
        siz[i] = res;
    }
}
void init() {
    memset(path, 0, sizeof path);
}
```

1.37.2 求所有极大团

```
bool path[MAXN][MAXN]; //表示结点之间的连接
int du[MAXN]; // 节点度数
int some[MAXN][MAXN]; // 待加入的点
int none[MAXN][MAXN]; // 已搜过的点
int all[MAXN][MAXN]; // 当前极大团中的点
int ans;
void dfs(int dep, int an, int sn, int nn) {
    if(!sn && !nn) // 是一个极大团
        ans = max(ans, an);
    int u = some[dep][0];  // 选取Pivot结点
    for(int j = 0; j < an; ++j)
        all[dep + 1][j] = all[dep][j];
    for(int i = 0; i < sn; ++i) {
        int v = some[dep][i];
        if(path[u][v]) // 相邻节点不考虑
```

```
            continue;
        all[dep + 1][an] = v; // 更新极大团
        int tsn = 0, tnn = 0;
        for(int j = 0; j < sn; ++j) // 更新待选点
            if(path[v][some[dep][j]])
                some[dep + 1][tsn++] = some[dep][j];
        for(int j = 0; j < nn; ++j) // 更新已搜点
            if(path[v][none[dep][j]])
                none[dep + 1][tnn++] = none[dep][j];
        dfs(dep + 1, an + 1, tsn, tnn);
        some[dep][i] = 0; // 去除v
        none[dep][nn++] = v; // 加入v
    }
}
void bron_kerbosch(int n) {
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            du[i] += path[i][j];
    for(int i = 0; i < n; i++)
        some[0][i] = i + 1;
    auto cmp = [&](int x, int y) {
        return du[x] > du[y];
    };
    sort(some[0], some[0] + n, cmp); // pivot选择优化
    dfs(0, 0, n, 0);
}
void init() {
    memset(path, 0, sizeof path);
    memset(du, 0, sizeof du);
    ans = 0;
}
```

1.38 图的生成树计数问题：矩阵树定理

这里的图，包括有向图和无向图，允许重边和自环存在。

1.38.1 无向图

定义度数矩阵 $D(G)$：$D_{ij}(G)=\begin{cases}deg(i) & i=j\\ 0 & i\neq j\end{cases}$

设 $\#e(i,j)$ 为点 i 与点 j 相连的边数，定义邻接矩阵 $A(G)$：$A_{ij}(G)=\begin{cases}0 & i=j\\ \#e(i,j) & i\neq j\end{cases}$

定义 Laplace 矩阵 $L(G)$：$L(G)=D(G)-A(G)$

记图 G 的所有生成树个数为 $t(G)$

定理 1 ：

对于任意的 i，都有：$t(G)=\det L(G)\begin{pmatrix}1,2,\cdots,i-1,i+1,\cdots,n\\ 1,2,\cdots,i-1,i+1,\cdots,n\end{pmatrix}$

其中 $L(G)\begin{pmatrix}1,2,\cdots,i-1,i+1,\cdots,n\\ 1,2,\cdots,i-1,i+1,\cdots,n\end{pmatrix}$ 表示矩阵 $L(G)$ 去掉第 i 行和第 i 列的得到的子矩阵。也就是说，无向图的 Laplace 矩阵的所有 $n-1$ 阶主子式相等。

定理 2 ：

设 $\lambda_1,\lambda_2,\cdots,\lambda_{n-1}$ 为 $L(G)$ 的 $n-1$ 个非零特征值，那么有 $t(G)=\frac{1}{n}\lambda_1\lambda_2\cdots\lambda_{n-1}$

1.38.2 有向图

定义出度矩阵 $D^{out}(G)$ 和入度矩阵 $D^{in}(G)$：

$D^{out}_{ij}(G)=\begin{cases}deg^{out}(i) & i=j\\ 0 & i\neq j\end{cases}$ $D^{in}_{ij}(G)=\begin{cases}deg^{in}(i) & i=j\\ 0 & i\neq j\end{cases}$

设 $\#e(i,j)$ 为点 i 指向点 j 的有向边数，定义邻接矩阵 $A(G)$：$A_{ij}(G)=\begin{cases}0 & i=j\\ \#e(i,j) & i\neq j\end{cases}$

定义出度 Laplace 矩阵 $L^{out}(G)$：$L^{out}(G)=D^{out}(G)-A(G)$

定义入度 Laplace 矩阵 $L^{in}(G)$：$L^{in}(G)=D^{in}(G)-A(G)$

记图 G 的以 r 为根的所有根向树形图个数为 $t^{root}(G,r)$。

记图 G 的以 r 为根的所有叶向树形图个数为 $t^{leaf}(G,r)$。

定理 3 ：

对于任意的 k，都有：$t^{root}(G,k)=\det L^{out}(G)\begin{pmatrix}1,2,\cdots,k-1,k+1,\cdots,n\\ 1,2,\cdots,k-1,k+1,\cdots,n\end{pmatrix}$

如果要统计一张图所有的根向树形图，只要枚举所有的 k，求和即可。

定理 4 ：

对于任意的 k，都有：$t^{leaf}(G,k)=\det L^{in}(G)\binom{1,2,\cdots,k-1,k+1,\cdots,n}{1,2,\cdots,k-1,k+1,\cdots,n}$

如果要统计一张图所有的叶向树形图，只要枚举所有的 k，求和即可。

定理 5（BEST 定理） ：

设 G 是有向欧拉图，那么 G 的不同欧拉回路总数 $ec(G)$ 为：

$ec(G)=t^{root}(G,k)\prod_{v\in V}(\deg(v)-1)!$

$deg(i)$ 表示点 i 的出度或入度，两者相同。

对欧拉图 G 的任意两个节点 k,k'，都有 $t^{root}(G,k)=t^{root}(G,k')$，且欧拉图 G 的所有节点的入度和出度相等。

如果要求指定起点 s 的欧拉回路数量，答案还要乘上 $deg(s)$。

1.38.3 求矩阵行列式：$O(n^3)$

mod 意义下，求上三角矩阵只能用辗转相除法，复杂度为 $O(n^3+n^2logP)$

若题目有高精度要求，需要套大数板子。

```
struct Matrix {
    int n, m;
    ll x[MAXN + 5][MAXN + 5];
    Matrix(int _n, int _m): n(_n), m(_m) {
        init(n, m);
    }
    void init(int n, int m) {
        for(int i = 0; i < n + 5; i++)
            for(int j = 0; j < m + 5; j++)
                x[i][j] = 0;
    }
    ll det(int n, int del) { // del为主子式去掉的那行/列
        ll ans = 1;
        for(int i = 1; i <= n; i++) {
            if(i == del)
                continue;
            for(int j = i + 1; j <= n; j++) {
                if(j == del)
                    continue;
                while(x[j][i]) {
```

```
                    ll y = x[i][i] / x[j][i];
                    for(int k = 1; k <= n; k++) {
                        x[i][k] = (x[i][k] - y * x[j][k] % mod + mod) % mod;
                        swap(x[i][k], x[j][k]);
                    }
                    ans = mod - ans; // 取负
                }
            }
            ans = ans * x[i][i] % mod ;
        }
        return ans;
    }
} D(MAXN, MAXN), A(MAXN, MAXN), L(MAXN, MAXN);
void Matrix_sub(int n) { // L=D-A
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            L.x[i][j] = (D.x[i][j] - A.x[i][j] + mod) % mod;
}
```

1.38.4 最小生成树计数：$O(n^3)$

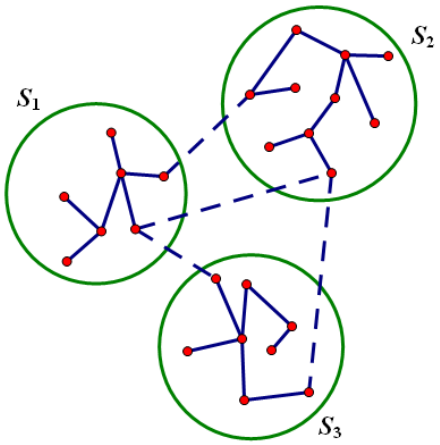
最小生成树性质 ：

若 A, B 同为最小生成树，则 A, B 的排序后的边权序列是相同的。

若 A, B 同为最小生成树，从小到大枚举边权 w_i，把相同权值的边全部加入后，图的联通性相同。

最小生成树计数方法 ：

每次从小到大在图中加入相同边权的边，计算每个新联通块的生成树方案数，与答案相乘，再把新的联通块缩点。




```cpp
int f1[MAXN], f2[MAXN];
int find(int x, int f[]) {
    return f[x] == x ? x : f[x] = find(f[x], f);
}
//-----------------------------------------
struct edge {
    int u, v, w;
} e[MAXM];
vector<edge> e2[MAXN]; // 存=当前边权的边
int id[MAXN], cnt = 0; // 离散化
int matrix_tree(int n) {
    int ans = 1;
    for(int i = 1; i <= n; i++) {
        if(e2[i].empty())
            continue;
        cnt = 0;
        for(auto j : e2[i]) {
            int u = find(j.u, f1);
            int v = find(j.v, f1);
            if(!id[u])
                id[u] = ++cnt;
            if(!id[v])
                id[v] = ++cnt;
            D.x[id[u]][id[u]]++;
            D.x[id[v]][id[v]]++;
```

```
            A.x[id[u]][id[v]]++;
            A.x[id[v]][id[u]]++;
        }
        for(int j = 1; j <= cnt; j++)
            for(int k = 1; k <= cnt; k++)
                L.x[j][k] = ((D.x[j][k] - A.x[j][k]) % mod + mod) % mod;
        ans = 1ll * ans * L.det(cnt - 1) % mod;
        D.init(cnt, cnt);
        A.init(cnt, cnt);
    }
    for(int i = 1; i <= n; i++) {
        id[i] = 0;
        e2[i].clear();
        f1[i] = find(i, f2);
    }
    return ans;
}
int solve(int n, int m) {
    for(int i = 1; i <= n; i++)
        f1[i] = f2[i] = i;
    auto cmp = [&](edge a, edge b) {
        return a.w < b.w;
    };
    sort(e + 1, e + m + 1, cmp);
    int ans = 1;
    for(int i = 1, j; i <= m; i = j + 1) {
        j = i;
        while(j + 1 <= m && e[i].w == e[j + 1].w)
            j++;
        // [i, j]的边权相同
        for(int k = i; k <= j; k++) { // 连通块合并
            int u = find(e[k].u, f1);
            int v = find(e[k].v, f1);
            if(u != v)
                f2[find(u, f2)] = find(v, f2);
        }
        for(int k = i; k <= j; k++) { // 每个连通块的边存入并查集的根
            int u = find(e[k].u, f1);
            int v = find(e[k].v, f1);
```

```
            if(u != v)
                e2[find(u, f2)].pb(e[k]);
        }
        ans = 1ll * ans * matrix_tree(n) % mod;
    }
    return ans % mod;
}
int main() {
    // ...
    int ans = solve(n, m);
    for(int i = 2; i <= n; i++)
        if(f1[1] != f1[i]) // 不连通
            ans = 0;
    // ...
}
```

1.39 欧拉图

1.39.1 性质

欧拉图中所有顶点的度数都是偶数。

若 G 是欧拉图，则它为若干个边不重的圈的并。

若 G 是半欧拉图，则它为若干个边不重的圈和一条简单路径的并。

1.39.2 无向图

1 判断存在性

G 是欧拉图（欧拉回路）当且仅当 G 是连通的且没有奇度顶点。

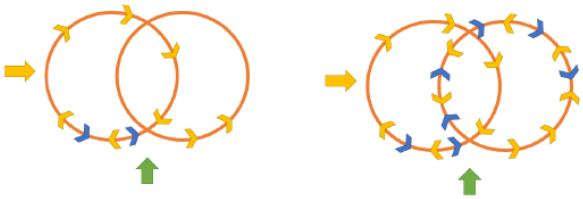
G 是半欧拉图（欧拉通路）当且仅当 G 中恰有 0 个或 2 个奇度顶点。

2 求欧拉回路/通路：$O(n+m)$

Hierholzer 算法：

求回路从任意点开始 dfs ，求通路需要从奇度顶点开始 dfs 。

dfs 搜索到没有分支可以前进时回溯记录路径，对走过的边标记不能再走。



若要求字典序最小，在每轮 dfs 选点前 sort 一下就行。

```
struct edge {
    int to, next, w, f;
} e[MAXM * 2];
int cnt = 1;
int head[MAXN];
void add_edge(int u, int v, int w = 0) {
    e[++cnt].to = v;
    e[cnt].next = head[u];
    e[cnt].w = w;
    e[cnt].f = 1;
    head[u] = cnt;
}
stack<int>path; // 存路径
void dfs(int u) {
    for(int &i = head[u]; i; i = e[i].next) { // 防止有环导致多次遍历
        if(!e[i].f)
            continue;
        e[i].f = e[i ^ 1].f = 0; // 标记不能再走
        int v = e[i].to;
        dfs(v);
        // printf("u: %d v: %d\n", u, v);
        path.push(u);
    }
}
int deg[MAXN];
void init(int n) {
    cnt = 1; // 注意前向星从2开始存
    for(int i = 0; i < n + 5; i++)
        head[i] = deg[i] = 0;
    while(!path.empty())
```

```
        path.pop();
}
```

1.39.3 有向图

1 判断存在性

G 是欧拉图（欧拉回路）当且仅当 G 的所有点属于一个强连通分量且每个点的入度与出度相同。

G 是半欧拉图（欧拉通路）当且仅当 G 的所有顶点属于一个强连通分量且满足以下 3 个条件：

- 最多只有一个顶点的出度与入度差为 1。
- 最多只有一个顶点的入度与出度差为 1。
- 所有其他顶点的入度和出度相同。

2 求欧拉回路/通路：$O(n+m)$

依然用 Hierholzer 算法，求回路从任意点开始 dfs ，求通路从出度比入度大 1 的那个点开始 dfs 。

```
struct edge {
    int to, next, w, f;
} e[MAXM * 2];
int cnt = 0;
int head[MAXN];
void add_edge(int u, int v, int w = 0) {
    e[++cnt].to = v;
    e[cnt].next = head[u];
    e[cnt].w = w;
    e[cnt].f = 1;
    head[u] = cnt;
}
stack<int>path; // 存路径
void dfs(int u) {
    for(int &i = head[u]; i; i = e[i].next) { // 防止有环导致多次遍历
        if(!e[i].f)
            continue;
        e[i].f = 0; // 标记不能再走
        int v = e[i].to;
```

```
        dfs(v);
        printf("u: %d v: %d\n", u, v);
        path.push(u);
    }
}
int in[MAXN], out[MAXN];
void init(int n) {
    cnt = 0;
    for(int i = 0; i < n + 5; i++)
        head[i] = in[i] = out[i] = 0;
    while(!path.empty())
        path.pop();
}
```

1.40 哈密顿图

1.40.1 性质

设 $G =< V, E >$ 是哈密顿图，则对于 V 的任意非空真子集 V_1，均有 $p(G - V_1) \leq |V_1|$。其中 $p(x)$ 为 x 的连通分支数。

设 $G =< V, E >$ 是半哈密顿图，则对于 V 的任意非空真子集 V_1，均有 $p(G-V_1) \leq |V_1|+1$。其中 $p(x)$ 为 x 的连通分支数。

完全图 $K_{2k+1}(k \geq 1)$ 中含 k 条边不重复的哈密顿回路，且这 k 条哈密顿回路覆盖了完全图的所有边。

完全图 $K_{2k}(k \geq 2)$ 中含 $k-1$ 条边不重复的哈密顿回路，从这 K_{2k} 中删除这 $k-1$ 条边不重复的哈密顿回路后所得图含 k 条互不相邻的边。

1.40.2 无向图

1 充分条件

设 G 是 $n(n \geq 2)$ 的无向简单图，若对于 G 中任意不相邻的顶点 v_i, v_j，均有 $d(v_i)+d(v_j) \geq n-1$，则 G 中存在哈密顿通路。

推论 1：设 G 是 $n(n \geq 3)$ 的无向简单图，若对于 G 中任意不相邻的顶点 v_i, v_j，均有 $d(v_i) + d(v_j) \geq n$，则 G 中存在哈密顿回路，从而 G 为哈密顿图。

推论 2：设 G 是 $n(n \geq 3)$ 的无向简单图，若对于 G 中任意顶点 v_i，均有 $d(v_i) \geq \frac{n}{2}$，则 G 中存在哈密顿回路，从而 G 为哈密顿图。

2 已知存在，求一条哈密顿回路：$O(n^2)$

```
int g[MAXN][MAXN]; // 原图
int vis[MAXN];
vi path; // 哈密顿回路
int cnt = 0;
void expand(int n, int &t) {
    while(true) {
        int i = 1;
        for(; i <= n; i++)
            if(!vis[i] && g[t][i]) {
                vis[i] = 1;
                path[cnt++] = i;
                t = i;
                break;
            }
        if(i > n)
            return;
    }
}
void Hamilton(int n) {
    int s = 1, t = 0;
    for(int i = 1; i <= n; i++)
        if(g[s][i])
            t = i;
    vis[s] = vis[t] = 1;
    path[cnt++] = s;
    path[cnt++] = t;
    while(true) {
        expand(n, t);
        reverse(path.begin(), path.begin() + cnt);
        swap(s, t);
        expand(n, t);
        if(!g[s][t]) {
            for(int i = 2; i < cnt - 2; i++) {
                if(g[s][path[i]] && g[path[i - 1]][t]) {
                    reverse(path.begin() + i, path.begin() + cnt);
                    break;
                }
```

```
            }
        }
        if(cnt == n)
            return;
        for(int i = 1; i <= n; i++) {
            if(vis[i])
                continue;
            int f = 0;
            for(int j = 0; j < cnt; j++)
                if(g[i][path[j]]) {
                    vis[i] = 1;
                    reverse(path.begin(), path.begin() + j);
                    reverse(path.begin() + j, path.begin() + cnt);
                    path[cnt++] = i;
                    s = path[0];
                    t = path[cnt - 1];
                    f = 1;
                    break;
                }
            if(f)
                break;
        }
    }
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        vis[i] = 0;
        for(int j = 0; j < n + 5; j++)
            g[i][j] = 0;
    }
    path.clear();
    path.resize(n + 1); // 动态分配空间
    cnt = 0;
}
```

1.40.3 竞赛图

定义：给无向完全图的每条边加上方向就得到一张竞赛图。

1 充分条件

设 D 为 $n(n \geq 2)$ 阶竞赛图，则 D 具有哈密顿通路。

若 D 含 $n(n \geq 2)$ 阶竞赛图作为子图，则 D 具有哈密顿通路。

强连通的竞赛图 D 具有哈密顿回路。

若 D 含 $n(n \geq 2)$ 阶强连通的竞赛图作为子图，则 D 具有哈密顿回路。

2 已知存在，求一条哈密顿通路：$O(n^2)$

```
int g[MAXN][MAXN];
vi Hamilton(int n) {
    vi ans;
    ans.pb(1);
    for(int i = 2; i <= n; i++) {
        if(g[ans.back()][i]) { // 直接拓展
            ans.pb(i);
            continue;
        }
        int pre = SZ(ans);
        for(int j = 0; j < SZ(ans) - 1; j++)
            if(g[ans[j]][i] && g[i][ans[j + 1]]) { // 中部插入
                ans.insert(ans.begin() + j + 1, i);
                break;
            }
        if(pre == SZ(ans)) // 如果不能插入,那一定在开头
            ans.insert(ans.begin(), i);
    }
    return ans;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++)
        for(int j = 0; j < n + 5; j++)
            g[i][j] = 0;
}
```

1.41 平面图

1.41.1 平面图定义

设 G 是平面图，由 G 的边将 G 所在的平面划分成若干个区域，每个区域称为 G 的一个面，其中面积无限的面称为无限面或外部面，面积有限的称为有限面或内部面。包围每个面的所有边组成的回路称为该面的边界，边界的长度称为该面的次数，面 R 的次数记为 $deg(R)$。

简单图：任何两点间至多有一条边，无自环。

平面图中所有面的次数之和等于边数 m 的 2 倍，即 $\sum_{i=1}^{r} deg(R_i) = 2m$。

若在简单平面图 G 的任意不相邻顶点间添加边，所得图为非平面图，称 G 为极大平面图。

若 G 为 $n(n \geq 3)$ 阶简单的连通平面图，G 为极大平面图当且仅当 G 的每个面的次数均为 3。

1.41.2 欧拉公式

若连通平面图 $G =< V, E >$ 中共有个 n 顶点、m 条边和 r 个面，则有 $n - m + r = 2$。

对于有 $p(p \geq 1)$ 个连通分支的平面图 G，有 $n - m + r = p + 1$。

设 G 是连通的平面图，且 G 每个面的次数至少为 $l(l \geq 3)$，则有 $m \leq \frac{l}{l-2}(n-2)$。

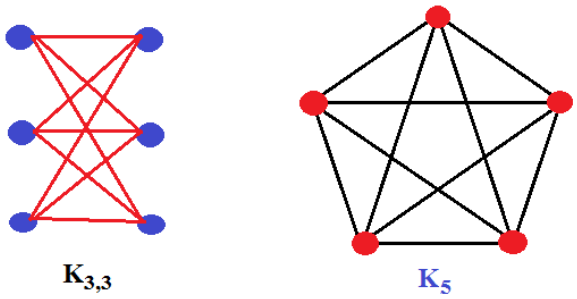
对于有 $p(p \geq 1)$ 个连通分支的平面图 G，有 $m \leq \frac{l}{l-2}(n-p-1)$。

设 G 是 $n(n \geq 3)$ 阶 m 条边的连通平面简单图，则 $m \leq 3n - 6$。

极大连通平面简单图的边数 $m = 3n - 6$。

若连通平面简单图 G 不以 K_3 为子图，即 G 是每个面由 4 条或 4 条以上的边围成的连通平面图，则 $m \leq 2n - 4$。

$K_{3,3}$ 和 K_5 是非平面图。



在平面简单图 G 中至少有一个顶点 v_0 满足 $d(v_0) \leq 5$。

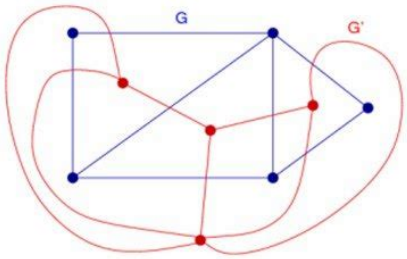
1.41.3 平面图的判断

若两个图 G_1 和 G_2 同构，或者通过反复插入或删除度数为 2 的顶点后同构，则称 G_1 和 G_2 是 2 度顶点内同构的。

库拉托夫斯基定理：一个图是平面图的充分必要条件是它不含与 $K_{3,3}$ 或 K_5 在 2 度顶点内同构的子图。

1.41.4 对偶图

定义如下图，G 和 G^* 互为对偶图。



对偶图性质 ：

G 中的自环对应 G^* 中的桥，G 中的桥对应 G^* 中的自环。

若 G 与 G^* 同构，则称是自对偶图。

对于 G 和 G^*，有 $n = r^*, m = m^*, r = n^*$，且 $d(v_i^*) = deg(R_i)$，即对偶图中点的度数和原图中对应面的次数相同。

G 是连通平面图当且仅当 G^{**} 同构于 G。

同构的图的对偶图不一定是同构的。

1.41.5 点着色

定义 ：

使图 G 是 $k-$ 可着色的数 k 的最小值称为 G 的色数，记作 $\chi(G)$。如果 $\chi(G) = k$，则称 G 是 $k-$ 色的。

性质 ：

如果图 G 的顶点的最大度数为 $\Delta(G)$，则 $\chi(G) \leq 1 + \Delta(G)$，取等于的情况只有奇回路或完全图。

四色定理：任何平面图都是 4- 可着色的。

1.42 弦图

定义 ：

导出子图（诱导子图）：点集为原图点集子集，边集为所有满足两个端点均在选定点集中的图。

团：完全子图。

团数：最大团的点数，记为 $\omega(G)$ 。

色数：最小染色的颜色数，记为 $\chi(G)$ 。

最大独立集：最大的点集使得点集中任意两点都没有边直接相连。该集合的大小记为 $\alpha(G)$ 。

最小团覆盖：用最少的团覆盖所有的点。使用团的数量记为 $\kappa(G)$ 。

弦：连接环中不相邻两点的边。

弦图：任意长度大于 3 的环都有一个弦的图称为弦图。

相邻点集：与点 x 相邻的点集记为 $N(x)$ 。

Lemma 1 ： 团数 $\omega(G)=\chi(G)$ 色数

Lemma 2 ： 最大独立集数 $\alpha(G)=\kappa(G)$ 最小团覆盖数

Lemma 3 ： 弦图的任意导出子图一定是弦图。

Lemma 4 ： 弦图的任意导出子图一定不可能是一个点数大于 3 的环。

点割集：对于图上的两点 u,v ，定义这两点间的点割集为满足删除这一集合后，u,v 两点之间不连通。如果关于 u,v 两点间的一个点割集的任意子集都不是点割集，则称这个点割集为极小点割集。

Lemma 5 ： 图关于 u,v 的极小点割集将原图分成了若干个连通块，设包含 u 的连通块为 V_1 ，包含 v 的连通块为 V_2 ，则对于极小点割集上的任意一点 a ，$N(a)$ 一定包含 V_1 和 V_2 中的点。

Lemma 6 ： 弦图上任意两点间的极小点割集的导出子图一定为一个团。

单纯点：若点集 $\{x\}+N(x)$ 的导出子图为一个团，则称点 x 为单纯点。

Lemma 7 ： 任何一个弦图都至少有一个单纯点，不是完全图的弦图至少有两个不相邻的单纯点。

完美消除序列：令 $n=|V|$ ，完美消除序列 $v_1,v_2,\cdots,v_n$ 为 $1,2,\cdots,n$ 的一个排列，满足 v_i 在 $\{v_i,v_{i+1},\cdots,v_n\}$ 的导出子图中为单纯点。

Lemma 8 ： 一个无向图是弦图当且仅当其有一个完全消除序列。

1.42.1 最大势算法 MCS：$O(n+m)$

从后向前填入完美消除序列。

用 $adj(x)$ 表示 x 相邻的已入序列的节点数量，每次选择 adj 值最大的未入序列节点填入序列。

用链表维护 adj 最大的未入序列节点。

如果原图是弦图，此时求出的就是完美消除序列；如果原图不是弦图，此时求出的一定不是完美消除序列，所以可以通过判断求出的序列是否是完美消除序列来判断原图是否是弦图。

判断方法 $check()$ ：

根据完美消除序列的定义,设 v_i 在 $v_i, v_{i+1}, \cdots, v_n$ 中相邻的点从小到大为 $\{v_{c_1}, v_{c_2}, \cdots, v_{c_k}\}$ ，则只需判断 v_{c_1} 与其他点是否直接连通即可，复杂度也是 $O(n+m)$ 。

```
vi e[MAXN];
int q[MAXN]; // 完美消除序列
int adj[MAXN]; // 相邻的已入队节点的数量
int pre[MAXN * 2], nex[MAXN * 2], p; // 链表
void push(int x) {
    int L = p + adj[x], R = nex[L];
    nex[L] = x;
    nex[x] = R;
    pre[x] = L;
    pre[R] = x;
}
void del(int x) {
    int L = pre[x], R = nex[x];
    nex[L] = R;
    pre[R] = L;
}
int pos[MAXN]; // 点在序列中的位置
void MCS(int n) {
    p = n + 1; // 移动下标
    for(int i = 1; i <= n; i++)
        push(i);
    int maxx = 0; // 未入队节点的最大的adj值
    for(int i = n; i >= 1; i--, maxx++) {
        while(!nex[p + maxx])
            maxx--;
        int u = nex[p + maxx];
        del(u);
        q[i] = u;
        pos[u] = i;
```

```
        for(auto v : e[u]) {
            if(pos[v])
                continue;
            del(v);
            adj[v]++;
            push(v);
        }
    }
}
int g[MAXN][MAXN]; // 两点是否连通
bool check(int n) { // 判断是否是完美消除序列
    for(int i = 1; i <= n; i++) {
        int u = q[i];
        vi tmp;
        for(auto v : e[u])
            if(pos[u] < pos[v]) {
                tmp.pb(v);
                if(pos[tmp[0]] > pos[v])
                    swap(tmp[0], tmp.back());
            }
        for(int j = 1; j < SZ(tmp); j++)
            if(!g[tmp[0]][tmp[j]])
                return false;
    }
    return true;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        e[i].clear();
        adj[i] = 0;
        pre[i] = pre[i + n] = 0;
        nex[i] = nex[i + n] = 0;
        pos[i] = 0;
        for(int j = 0; j < n + 5; j++)
            g[i][j] = 0;
    }
}
```

1.42.2 弦图的极大团：$O(n+m)$

重新定义 $N(x)$ 为满足与 x 直接有边相连的，且在完美消除序列中在 x 的位置之后的点的序列。则当 x 为某个极大团中在完美消除序列中位置最靠前的点，该极大团为 $\{x\}+N(x)$ 。

弦图最多有 n 个极大团。求出弦图的每个极大团后，可以判断每个 $\{x\}+N(x)$ 是否为极大团。

设 $A=\{x\}+N(x), B=\{y\}+N(y)$ ，若 $A \subsetneqq B$ ，则 A 不是极大团。此时在完美消除序列上显然有 y 在 x 前。

设 $Next(x)$ 表示 $N(x)$ 中在完美消除序列上最靠前的点，$y*$ 表示所有满足 $A \subseteq B$ 的 y 中的最靠后的点。此时必然有 $Next(y*)=x$ ，否则 $y*$ 不是最靠后的。

$A \subsetneqq B$ 当且仅当 $|A|+1 \leq |B|$ 。

问题转化为判断是否存在 y ，满足 $Next(y)=x$ 且 $|N(x)|+1 \leq |N(y)|$ ，复杂度为 $O(n+m)$ 。

```
int vis[MAXN];
int Next[MAXN];
int Num[MAXN]; // Num(x)表示N(x)的大小
int cal(int n) {
    for(int i = 1; i <= n; i++) {
        int u = q[i];
        vi tmp;
        for(auto v : e[u])
            if(pos[u] < pos[v]) {
                tmp.pb(v);
                if(pos[tmp[0]] > pos[v])
                    swap(tmp[0], tmp.back());
            }
        if(!tmp.empty()) // 可能为空
            Next[u] = tmp[0];
        Num[u] = SZ(tmp);
    }
    int ans = 0; // 极大团数量
    for(int i = 1; i <= n; i++) {
        int u = q[i];
        if(!vis[u]) // u是极大团的最靠前的点
            ans++;
        if(Num[Next[u]] + 1 <= Num[u])
```

```
            vis[Next[u]] = 1;
    }
    return ans;
}
```

1.42.3 弦图的色数/团数：$O(n+m)$

$\max\limits_{i=1}^{n}\{adj(i)+1\}$ 即为弦图的色数/团数。

```
int ans = 0;
for(int i = 1; i <= n; i++)
    ans = max(ans, adj[i] + 1);
```

1.42.4 弦图的最大独立集/最小团覆盖：$O(n+m)$

从前往后遍历完美消除序列，若当前点 u 不与 ans 中的任何点相邻，则把 u 加入 ans 。

```
vi ans, vis(n + 1, 0);
for(int i = 1; i <= n; i++) {
    int u = q[i];
    int f = 1;
    for(auto v : e[u])
        if(vis[v])
            f = 0;
    if(f) {
        ans.pb(u);
        vis[u] = 1;
    }
}
```

1.43 最小环

这里的环的点数满足 $n \geq 3$。

暴力思想：枚举所有边 $e(u,v)$，计算在删去 e 后 u 到 v 的最短路，最短路加上 e 的边权就是一个环的贡献。以下两种方法都是这种思路，根据最短路算法选择的不同，时间复杂度会有区别。

1.43.1 朴素 dijkstra：$O(m*n^2)$

```
int g[MAXN][MAXN]; // 原图
int dis[MAXN];
int vis[MAXN];
void dijkstra(int n, int s) {
    for(int i = 0; i < n + 5; i++) {
        dis[i] = INF;
        vis[i] = 0;
    }
    for(int i = 1; i <= n; i++)
        dis[i] = g[s][i];
    for(int i = 0; i < n - 1; i++) {
        pii tmp = mp(INF, -1);
        for(int j = 1; j <= n; j++)
            if(!vis[j] && dis[j] < tmp.fi)
                tmp = mp(dis[j], j);
        if(tmp.se == -1)
            return;
        int v = tmp.se;
        vis[v] = 1;
        for(int j = 1; j <= n; j++)
            if(!vis[j] && dis[v] + g[v][j] < dis[j])
                dis[j] = dis[v] + g[v][j];
    }
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        for(int j = 0; j < n + 5; j++)
            g[i][j] = INF;
        g[i][i] = 0;
    }
}
int main() {
    // ...
    init(n);
    vector<pii>e;
    // ...
    int ans = INF;
```

```
    for(auto i : e) { // 枚举所有边
        int u = i.fi, v = i.se;
        int tmp = g[u][v];
        g[u][v] = g[v][u] = INF; // 暂时删去
        dijkstra(n, u);
        ans = min(ans, dis[v] + tmp);
        g[u][v] = g[v][u] = tmp; // 还原
    }
    // ...
}
```

1.43.2 堆优化 dijkstra：$O(m*(n+m)logm)$

```
vector<pii> e[MAXN]; // 删去重边后的图
int dis[MAXN];
int vis[MAXN];
priority_queue<pii, vector<pii>, greater<pii> >q;
void dijkstra(int n, int s, int t) {
    fill(dis, dis + n + 5, INF);
    fill(vis, vis + n + 5, 0);
    dis[s] = 0;
    q.push(mp(0, s));
    while(!q.empty()) {
        int x = q.top().se;
        q.pop();
        if(vis[x])
            continue;
        vis[x] = 1;
        for(auto i : e[x]) {
            int v = i.fi, w = i.se;
            if(x == s && v == t || x == t && v == s) // 删去当前边
                continue;
            if(dis[v] > dis[x] + w) {
                dis[v] = dis[x] + w;
                q.push(mp(dis[v], v));
            }
        }
    }
}
```

```cpp
int g[MAXN][MAXN]; // 用于去除重边
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        for(int j = 0; j < n + 5; j++)
            g[i][j] = INF;
        g[i][i] = 0;
    }
}
int main() {
    // ...
    init(n);
    // ...
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++) {
            if(i == j || g[i][j] == INF)
                continue;
            e[i].pb(mp(j, g[i][j])); // 有效边
        }
    int ans = INF;
    for(int u = 1; u <= n; u++) // 枚举所有边
        for(auto i : e[u]) {
            int v = i.fi, w = i.se;
            dijkstra(n, u, v);
            ans = min(ans, dis[v] + w);
        }
    // ...
}
```

1.43.3 Floyd：$O(n^3)$

注意到 Floyd 算法有一个性质：在最外层循环到点 k 时（尚未开始第 k 次循环），最短路数组 dis 中，$dis(u,v)$ 表示的是从 u 到 v 且仅使用编号在 $[1,k)$ 区间中的点更新后的最短路。

设一个环上有 3 个点 i j k，且 k 是环上编号最大的点，那么在遍历到第 k 层循环时可以知道环的总边权是 $dis(i,j)+g(i,k)+g(k,j)$。

```cpp
int g[MAXN][MAXN]; // 原图
int dis[MAXN][MAXN];
int floyd(int n) {
```

```
    for(int i = 1; i <= n; i++) // 拷贝原图
        for(int j = 1; j <= n; j++)
            dis[i][j] = g[i][j];
    int ans = INF;
    for(int k = 1; k <= n; k++) {
        for(int i = 1; i < k; i++) // i->j, i->k->j 成环
            for(int j = i + 1; j < k; j++)
                if(g[i][k] != INF && g[k][j] != INF) // 防止见祖宗
                    ans = min(ans, dis[i][j] + g[i][k] + g[k][j]);
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    }
    return ans;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        for(int j = 0; j < n + 5; j++)
            g[i][j] = INF;
        g[i][i] = 0;
    }
}
int main() {
    // ...
    init(n);
    // ...
    int ans = floyd(n);
    // ...
}
```

如果要查找最小环路径，在更新路径时进行一些标记即可。

```
int g[MAXN][MAXN]; // 原图
int dis[MAXN][MAXN];
int con[MAXN][MAXN]; // con(i,j) 表示连接 i,j 的中介
int path[MAXN], cnt = 0; // 存储路径
void check(int L, int R) {
    int mid = con[L][R];
    if(mid == 0) {
```

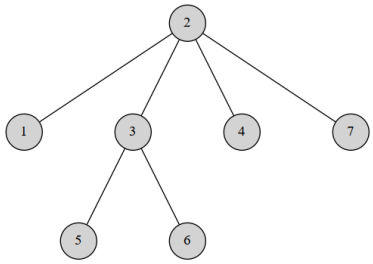
```
        path[++cnt] = L;
        return;
    }
    check(L, mid);
    check(mid, R);
}
void floyd(int n) {
    for(int i = 1; i <= n; i++) // 拷贝原图
        for(int j = 1; j <= n; j++)
            dis[i][j] = g[i][j];
    int ans = INF;
    for(int k = 1; k <= n; k++) {
        for(int i = 1; i < k; i++) // i->j, i->k->j 成环
            for(int j = i + 1; j < k; j++)
                if(g[i][k] != INF && g[k][j] != INF) // 防止见祖宗
                    if(ans > dis[i][j] + g[i][k] + g[k][j]) {
                        ans = dis[i][j] + g[i][k] + g[k][j];
                        cnt = 0; // 更新路径
                        check(i, j);
                        path[++cnt] = j;
                        path[++cnt] = k;
                    }
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                if(dis[i][j] > dis[i][k] + dis[k][j]) {
                    dis[i][j] = dis[i][k] + dis[k][j];
                    con[i][j] = k; // 标记中介
                }
    }
    if(ans == INF) { // 无环
        printf("No solution.\n");
        return;
    }
    for(int i = 1; i <= cnt; i++) // 路径打印
        printf("%d ", path[i]);
}
```

1.44 Prufer 序列

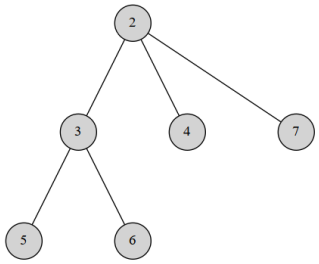
Prufer 序列的定义 ：

Prufer 序列可以将一个有 $n(n \geq 3)$ 个结点的树用 $n-2$ 个整数 $x_i(1 \leq x_i \leq n)$ 表示，它相当于完全图的生成树与数列之间的双射。

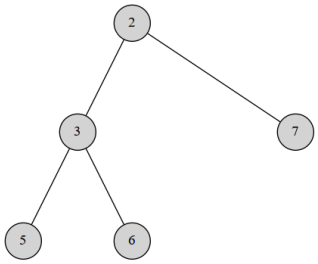
Prufer 序列的建立过程：每次选择一个编号最小的叶结点并删掉它，然后在序列中记录下它连接到的那个结点。重复 $n-2$ 次后就只剩下两个结点，算法结束。



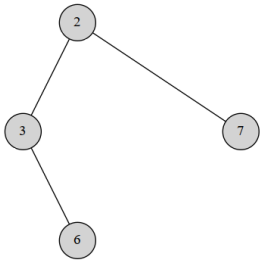
Prufer Code:



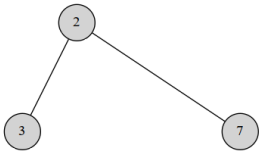
Prufer Code: 2



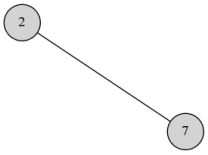
Prufer Code: 2 2



Prufer Code: 2 2 3



Prufer Code: 2 2 3 3



Prufer Code: 2 2 3 3 2

Prufer 序列的性质 ：

1. 在构造完 Prufer 序列后原树中会剩下两个结点，其中一个一定是编号最大的点 n。
2. 每个结点在序列中出现的次数是其度数减 1。（没有出现的就是叶结点）

Prufer 序列的应用 ：

注意：以下公式在题目中大部分情况都需要特判！

1. 证明 Cayley 公式：完全图 K_n 有 n^{n-2} 棵生成树。
2. 对于给定度数为 $d_1 \cdots d_n$ 的一棵无根树，共有 $\dfrac{(n-2)!}{\prod_{i=1}^{n}(d_i-1)!}$ 种情况。
3. 对于一个 n 个点 m 条边的无向图，图中已存在 k 个联通块。通过添加 $k-1$ 条边使整个图连通的方案数为 $n^{k-2} \cdot \prod_{i=1}^{k} sz(i)$，其中 $sz(i)$ 表示第 i 个联通块的大小。
4. 一个森林内部节点的度数平方和等于 2 *（长度为 1 的路径数 + 长度为 2 的路径数）

1.44.1 构造 Prufer 序列：$O(n)$

```
vi e[MAXN];
int deg[MAXN], f[MAXN];
void dfs(int u, int fa) {
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs(v, u);
        f[v] = u;
        deg[u]++, deg[v]++;
    }
}
vi prufer(int n) {
    dfs(n, 0); // 以n为根dfs
    int p = 0;
    while(deg[p] != 1)
        p++;
    vi ans;
    int Leaf = p++;
    for(int i = 0; i < n - 2; i++) {
        int fa = f[Leaf];
        ans.pb(fa);
        if(--deg[fa] == 1 && fa < p)
            Leaf = fa;
        else {
            while(deg[p] != 1)
                p++;
            Leaf = p++;
        }
    }
    return ans; // 返回prufer序列
}
```

1.44.2 Prufer 序列重建树：$O(n)$

```
vi e[MAXN];
int deg[MAXN];
void prufer_decode(vi code) { // 参数为prufer序列
```

```
    int n = SZ(code) + 2;
    fill(deg + 1, deg + n + 1, 1); // 还原deg
    for(auto i : code)
        deg[i]++;
    int p = 0;
    while(deg[p] != 1)
        p++;
    int Leaf = p++;
    for(auto fa : code) {
        e[Leaf].pb(fa); // fa -> Leaf
        e[fa].pb(Leaf);
        if(--deg[fa] == 1 && fa < p)
            Leaf = fa;
        else {
            while(deg[p] != 1)
                p++;
            Leaf = p++;
        }
    }
    e[Leaf].pb(n); // n -> Leaf
    e[n].pb(Leaf);
}
```

1.45 图的着色

1.45.1 点着色

这里讨论的是无向无环图。

对无向图顶点着色，且相邻顶点不能同色。若 G 是 k- 可着色的，但不是 $(k-1)$- 可着色的，则称 k 是 G 的色数，记为 $\chi'(G)$ 。

对任意图 G ，有 $\chi(G) \leq \Delta(G)+1$ ，其中 $\Delta(G)$ 为最大度。

Brooks 定理 ：

设连通图不是完全图也不是奇圈，则 $\chi(G) \leq \Delta(G)$ 。

1.45.2 边着色

对无向图的边着色，要求相邻的边涂不同种颜色。若 G 是 k- 边可着色的，但不是 $(k-1)$- 边可着色的，则称 k 是 G 的边色数，记为 $\chi'(G)$ 。

Vizing 定理 ：

若 G 是简单图（无重边无自环），则 $\Delta(G) \leq \chi'(G) \leq \Delta(G)+1$ 。

若 G 是二分图，则 $\chi'(G) = \Delta(G)$ 。

当 $n(n>1)$ 为奇数时，$\chi'(K_n) = n$; 当 n 为偶数时，$\chi'(K_n) = n-1$ 。

二分图 Vizing 定理的构造性证明 ：

按照顺序在二分图中加边。

我们在尝试加入边 (x,y) 的时候，我们尝试寻找对于 x 和 y 的编号最小的尚未被使用过的颜色，假设分别为 lx 和 ly 。

如果 $lx = ly$ 此时我们可以直接将这条边的颜色设置为 lx 。

否则假设 $lx < ly$, 我们可以尝试将节点 y 连出去的颜色为 lx 的边的颜色修改为 ly 。

修改的过程可以被近似的看成是一条从 y 出发，依次经过颜色为 $lx, ly, lx, ly, \cdots$ 的边的有限唯一增广路。

因为增广路有限所以我们可以将增广路上所有的边反色，即原来颜色为 lx 的修改为 ly ，原来颜色为 ly 的修改为 lx 。

根据二分图的性质，节点 x 不可能为增广路节点，否则与最小未使用颜色为 lx 矛盾。

所以我们可以在增广之后直接将连接 x 和 y 的边的颜色设为 lx 。

总构造时间复杂度为 $O(nm)$ 。

```
// 前向星从2开始
//------------------------------------------------------------
int col[MAXN][MAXN]; // col[u][c]表示点u的颜色为c的边的id
void change(int u, int L1, int L2) {
    if(!col[u][L1]) { //无法增广
        col[u][L2] = 0;
        return;
    }
    int i = col[u][L1];
    int v = e[i].to;
    change(v, L2, L1); // 继续增广
    e[i].w = e[i ^ 1].w = L2; // 修改颜色L1->L2
```

```
13      col[u][L2] = i;
14      col[v][L2] = i ^ 1;
15  }
16  void paint(int n) {
17      for(int u = 1; u <= n; u++) {
18          for(int i = head[u]; i; i = e[i].next) {
19              if(e[i].w > 0) // 已染色
20                  continue;
21              int v = e[i].to;
22              int Lx = INF, Ly = INF;
23              for(int i = 1; i <= n; i++) { // 找最小的未使用颜色
24                  if(!col[u][i])
25                      Lx = min(Lx, i);
26                  if(!col[v][i])
27                      Ly = min(Ly, i);
28              }
29              if(Lx < Ly)
30                  change(v, Lx, Ly);
31              if(Lx > Ly)
32                  change(u, Ly, Lx);
33              int L = min(Lx, Ly);
34              e[i].w = e[i ^ 1].w = L; // 染色
35              col[u][L] = i;
36              col[v][L] = i ^ 1;
37          }
38      }
39  }
40  int deg[MAXN];
41  void init(int n) {
42      cnt = 1; // 前向星从2开始存!
43      for(int i = 0; i < n + 5; i++) {
44          head[i] = deg[i] = 0;
45          for(int j = 0; j < n + 5; j++)
46              col[i][j] = 0;
47      }
48  }
```

1.45.3 色多项式

这里的着色指的是点着色。

用 $f(G,k)$ 表示 G 的不同 k 着色方案的总数。

完全图：$f(K_n,k)=A_k^n=k(k-1)\cdots(k-n+1)$

零图：$f(N_n,k)=k^n$

树：$f(G,k)=k(k-1)^{n-1}$

定理 1 ：

对于无向无环图 G ，删去 G 上的一条边 $e(u,v)$ 得到图 G' ，把 G' 上的 u 点和 v 点合并得到 G'' ，则：$f(G',k)=f(G,k)+f(G'',k)$ 。

定理 2 ：

设 V_1 是 G 的点割集，且 $G[V_1]$ 是 G 的 $|V_1|$ 阶完全子图，$G-V_1$ 有 $p(p\geq 2)$ 个连通分支，则：$f(G,k)=\dfrac{\prod_{i=1}^{p}f(H_i,k)}{f(G[V_1],k)^{p-1}}$ ，其中 $H_i=G[V_1\cup V(G_i)]$ 。

1.46 LGV 引理

LGV 引理仅适用于有向无环图，用于处理有向无环图上的不相交路径计数等问题。

1 定义：

$\omega(P)$ 表示 P 这条路径上所有边的边权之积。

$e(u,v)$ 表示 u 到 v 的每一条路径 P 的 $\omega(P)$ 之和，即 $e(u,v)=\sum\limits_{P}\omega(P)$ 。

起点集合 A ，是有向无环图上的一个子集，大小为 n 。

终点集合 B ，是有向无环图上的一个子集，大小为 n 。

对于一组 $A\to B$ 的不相交路径 $S(S_1,\cdots,S_n)$，即对于任何 $i\neq j$ ，S_i 和 S_j 没有公共顶点。

S_i 表示一条从 A_i 到 B_{σ_i} 的路径。

$\sigma_1,\cdots,\sigma_n$ 是一个排列。

$N(\sigma)$ 表示排列 σ 的逆序对个数。

2 引理：

$$M=\begin{bmatrix} e(A_1,B_1) & e(A_1,B_2) & \cdots & e(A_1,B_n) \\ e(A_2,B_1) & e(A_2,B_2) & \cdots & e(A_2,B_n) \\ \vdots & \vdots & \ddots & \vdots \\ e(A_n,B_1) & e(A_n,B_2) & \cdots & e(A_n,B_n) \end{bmatrix}$$

$$\det(M)=\sum_{S:A\to B}(-1)^{N(\sigma(S))}\prod_{i=1}^{n}\omega S_i$$

其中 $\sum_{S:A\to B}$ 表示满足上文要求的 $A\to B$ 的每一组不相交的路径 S。

当 n 个起点和终点恰好只有一个对应关系使得路径不相交时，$N(\sigma)=0$；再把所有边的边权设为 1，使 $\omega(S_i)=1$ 。此时的 $det(M)$ 即为不相交的路径组 S 的数量。

复杂度：O(求 n 对点对之间的路径数 $+n^3$)

3 例题：

1. hdu5852 Intersection is not allowed!

题意：

有一个 $n\times n$ 的棋盘，一个棋子从 (x,y) 只能走到 $(x,y+1)$ 或 $(x+1,y)$ ，有 k 个棋子，一开始第 i 个棋子放在 $(1,a_i)$ ，最终要到 (n,b_i) ，路径要两两不相交，求方案数对 10^9+7 取模。$1\le n\le 10^5$, $1\le k\le 100$ ， 保证 $1\le a_1<a_2<\cdots<a_n\le n$ ， $1\le b_1<b_2<\cdots<b_n\le n$ 。

题解：

如果路径不相交就一定是 a_i 到 b_i ，逆序对个数为 0。

从 $(1,a_i)$ 到 (n,b_j) 的路径数量相当于从 $n-1+b_j-a_i$ 步中选 $n-1$ 步向下走，即 $e(A_i,B_j)=C_{n-1+b_j-a_i}^{n-1}$ 。

2. Monotonic Matrix

题意：

求满足如下条件的 $n\times m$ 矩阵 A 的数量：

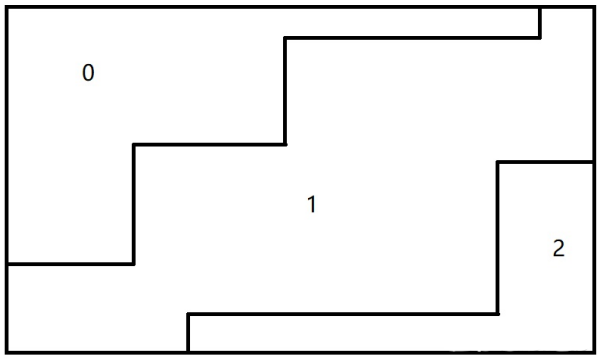
$\forall 1\ i\ n, 1\ j\ m, A_{i,j}\in\{0,1,2\}$

$\forall 1\ i<n, 1\ j\ m, A_{i,j}\ A_{i+1,j}$

$\forall 1\ i\ n, 1\ j<m, A_{i,j}\ A_{i,j+1}$

题解：

转化一下题意，发现是要用两条分界线把矩阵分为 0、1、2 三块区域，而且由于题目中的限制条件，分界线一定是从左下走到右上，那么就转化成了不相交路径问题。



3. 网格图

题意：

给定 $n \times m$ 的网格图，网格中有 C 个格子是特殊点，现在要找两条从 $(1,1)$ 到 (n,m) 的路径，要求两条路径经过的特殊点个数之和不能超过 D ，并且两条路径不能在除起点与终点之外格子相交，求路径数。

$2 \leq n,m \leq 10^5, 0 \leq D \leq C \leq min\{200, n \times m - 2\}$ 。

题解：

由于特殊点很少，我们将特殊点按照 $x+y$ 排序，那么一条路径上的特殊点编号递增，就可以 DP 了。

设 $f(i,j)$ 表示到了第 i 个特殊点，经过了 j 个特殊点的方案数；$g(i,j)$ 表示从特殊点 i 走到特殊点 j 且在之间没有经过其它特殊点的方案数；$h(i,j)$ 表示从特殊点 i 走到特殊点 j 随便走的方案数，因此：

$h(i,j) = C_{x_j - x_i + y_j - y_i - 2}^{x_j - x_i - 1}$

$g(i,j) = h(i,j) - \sum_{k=i+1}^{j-1} g(i,k) * h(k,j)$

$f(i,j) = \sum_{k=1}^{i-1} f(k,j-1) * g(k,i)$

把起点和终点也看做特殊点，枚举两条路径可能经过的特殊点数量，统计答案，复杂度 $O(C^3)$ 。

1.47 树上修改问题

1.47.1 单点向外辐射

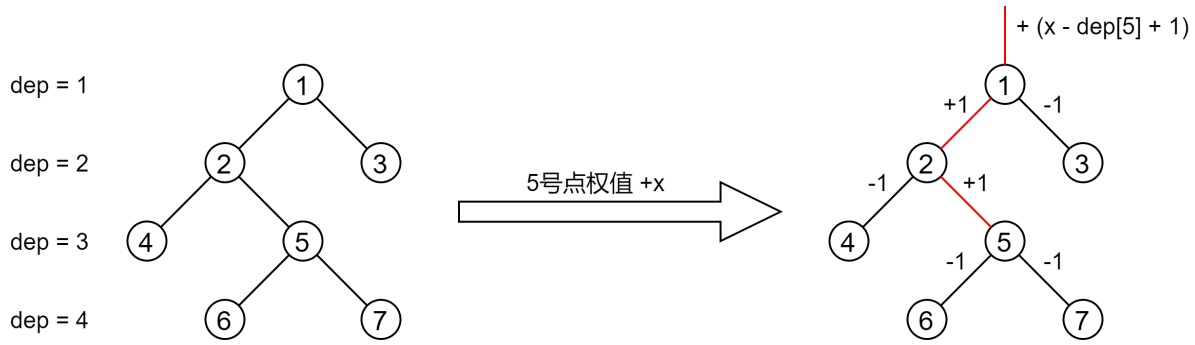
修改操作 ：

对某点 u 的权值增加 x，并且对其它所有点 v 的权值增加 $x - dis(u, v)$。

解决方法 ：

把点的权值转化为点到根节点的路径权值和，如下图。

重链剖分后，可用树状数组维护信息。



1.48 QTREE - 一周目

1.48.1 QTREE1

题意：

给一棵树，有两种操作：修改某条边的边权；查询两点路径上的最大边权。

思路：

重链剖分，用线段树维护边权。

```
struct edge {
    int v, w, id;
};
vector<edge>e[MAXN];
int dep[MAXN], sz[MAXN];
int son[MAXN]; // 存重儿子
int Fa[MAXN];
int e_v[MAXN];
int e_w[MAXN];
void dfs(int u, int fa) { // 预处理
    dep[u] = dep[fa] + 1;
    sz[u] = 1;
```

```
13      Fa[u] = fa;
14      for(auto i : e[u]) {
15          int v = i.v;
16          if(v == fa)
17              continue;
18          dfs(v, u);
19          sz[u] += sz[v];
20          if(sz[v] > sz[son[u]])
21              son[u] = v;
22          e_v[i.id] = v;
23          e_w[i.id] = i.w;
24      }
25  }
26  int top[MAXN]; // 存所在重链的顶部节点
27  int id[MAXN]; // 点标号->dfs序号
28  int rk[MAXN]; // dfs序号->点标号
29  int cnt = 0;
30  void dfs2(int u, int fa, int Top) { // 标记dfs序
31      top[u] = Top;
32      id[u] = ++cnt;
33      rk[cnt] = u;
34      if(!son[u]) // u是叶节点
35          return;
36      dfs2(son[u], u, Top); // 先走重儿子
37      for(auto i : e[u]) {
38          int v = i.v;
39          if(v != fa && v != son[u])
40              dfs2(v, u, v);
41      }
42  }
43  #define lson rt<<1
44  #define rson rt<<1|1
45  #define mid (Tree[rt].L+Tree[rt].R)/2
46  struct node {
47      int L, R;
48      int maxx;
49  } Tree[MAXN << 2];
50  void push_up(int rt) {
51      Tree[rt].maxx = max(Tree[lson].maxx, Tree[rson].maxx);
```

```
}
void build(int rt, int L, int R) {
    Tree[rt].L = L;
    Tree[rt].R = R;
    Tree[rt].maxx = 0;
    if(L == R)
        return;
    if(L <= mid)
        build(lson, L, mid);
    if(mid + 1 <= R)
        build(rson, mid + 1, R);
    push_up(rt);
}
void update(int rt, int p, int x) {
    if(Tree[rt].L == p && Tree[rt].R == p) {
        Tree[rt].maxx = x;
        return;
    }
    if(p <= mid)
        update(lson, p, x);
    else
        update(rson, p, x);
    push_up(rt);
}
int get_max(int rt, int L, int R) {
    if(L == Tree[rt].L && Tree[rt].R == R)
        return Tree[rt].maxx;
    int ans = 0;
    if(R <= mid)
        return get_max(lson, L, R);
    if(mid + 1 <= L)
        return get_max(rson, L, R);
    return max(get_max(lson, L, mid), get_max(rson, mid + 1, R));
}
void init(int n) {
    cnt = 0;
    for(int i = 0; i < n + 5; i++) {
        e[i].clear();
        dep[i] = sz[i] = son[i] = Fa[i] = 0;
```

```
        e_v[i] = e_w[i] = 0;
        top[i] = id[i] = rk[i] = 0;
    }
}
int main() {
    cin.tie(0);
    cin.sync_with_stdio(0);
    int t;
    cin >> t;
    while(t--) {
        int n;
        cin >> n;
        init(n);
        for(int i = 1; i <= n - 1; i++) {
            int u, v, w;
            cin >> u >> v >> w;
            e[u].pb({v, w, i});
            e[v].pb({u, w, i});
        }
        dfs(1, 0);
        dfs2(1, 0, 1);
        build(1, 1, n);
        for(int i = 1; i <= n - 1; i++) {
            int p = id[e_v[i]];
            update(1, p, e_w[i]);
        }
        string s;
        while(cin >> s && s != "DONE") {
            if(s == "QUERY") {
                int x, y;
                cin >> x >> y;
                int ans = 0;
                while(top[x] != top[y]) {
                    int tx = top[x], ty = top[y];
                    if(dep[tx] < dep[ty]) {
                        swap(x, y);
                        swap(tx, ty);
                    }
                    ans = max(ans, get_max(1, id[tx], id[x]));
```

```
                    x = Fa[tx];
                }
                if(dep[x] < dep[y])
                    swap(x, y);
                if(x != y)
                    ans = max(ans, get_max(1, id[y] + 1, id[x]));
                cout << ans << '\n';
            } else if(s == "CHANGE") {
                int e, w;
                cin >> e >> w;
                int p = id[e_v[e]];
                update(1, p, w);
            }
        }
    }
}
```

1.48.2 QTREE2

题意：

给一棵树，有两种操作：查询两点路径上的边权和；查询 u 到 v 的路径上第 k 个点的编号。

思路：

判断第 k 个点是谁的祖先节点，然后直接跳。

```
vector<pii>e[MAXN];
int dep[MAXN], f[MAXN][25];
int dis[MAXN][25];
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for(int i = 1; (1 << i) <= dep[u]; i++) {
        f[u][i] = f[f[u][i - 1]][i - 1];
        dis[u][i] = dis[u][i - 1] + dis[f[u][i - 1]][i - 1];
    }
    for(auto i : e[u]) {
        int v = i.fi, w = i.se;
        if(v == fa)
```

```
            continue;
        dis[v][0] = w;
        dfs(v, u);
    }
}
pii lca(int x, int y) {
    int sum = 0;
    if(dep[x] < dep[y])
        swap(x, y);
    for(int i = 20; i >= 0; i--) {
        if(dep[f[x][i]] >= dep[y]) {
            sum += dis[x][i];
            x = f[x][i];
        }
        if(x == y)
            return {x, sum};
    }
    for(int i = 20; i >= 0; i--)
        if(f[x][i] != f[y][i]) {
            sum += dis[x][i] + dis[y][i];
            x = f[x][i];
            y = f[y][i];
        }
    sum += dis[x][0] + dis[y][0];
    return {f[x][0], sum};
}
int kth_fa(int u, int k) {
    for(int i = 20; i >= 0; i--)
        if(k & (1 << i))
            u = f[u][i];
    return u;
}
void init(int n) {
    for(int i = 0; i < n + 5; i++) {
        e[i].clear();
        dep[i] = 0;
        for(int j = 0; j < 25; j++)
            f[i][j] = dis[i][j] = 0;
    }
```

```
}
int main() {
    cin.tie(0);
    cin.sync_with_stdio(0);
    int t;
    cin >> t;
    while(t--) {
        int n;
        cin >> n;
        init(n);
        for(int i = 1; i <= n - 1; i++) {
            int u, v, w;
            cin >> u >> v >> w;
            e[u].pb({v, w});
            e[v].pb({u, w});
        }
        dfs(1, 0);
        string s;
        while(cin >> s && s != "DONE") {
            if(s == "DIST") {
                int x, y;
                cin >> x >> y;
                cout << lca(x, y).se << '\n';
            } else if(s == "KTH") {
                int x, y, k;
                cin >> x >> y >> k;
                int LCA = lca(x, y).fi;
                int cntx = dep[x] - dep[LCA] + 1;
                if(k <= cntx)
                    cout << kth_fa(x, k - 1) << '\n';
                else {
                    int cnt = dep[x] + dep[y] - 2 * dep[LCA] + 1;
                    cout << kth_fa(y, cnt - k) << '\n';
                }
            }
        }
    }
}
```

1.48.3 QTREE3

题意：

给一棵树，每个点有颜色，初始都是白色。有两种操作：改变某点颜色（白变黑，黑变白）；查询 1 到 u 的路径上第 1 个黑点的编号，没有则输出 -1 。

思路：

重链剖分，每条重链的顶部结点维护重链上的黑点，向根节点跳的过程中更新答案。

```
vi e[MAXN];
int dep[MAXN], sz[MAXN];
int son[MAXN]; // 存重儿子
int Fa[MAXN];
void dfs(int u, int fa) { // 预处理
    dep[u] = dep[fa] + 1;
    sz[u] = 1;
    Fa[u] = fa;
    for(auto v : e[u]) {
        if(v == fa)
            continue;
        dfs(v, u);
        sz[u] += sz[v];
        if(sz[v] > sz[son[u]])
            son[u] = v;
    }
}
int top[MAXN]; // 存所在重链的顶部节点
int id[MAXN]; // 点标号->dfs序号
int rk[MAXN]; // dfs序号->点标号
int cnt = 0;
void dfs2(int u, int fa, int Top) { // 标记dfs序
    top[u] = Top;
    id[u] = ++cnt;
    rk[cnt] = u;
    if(!son[u]) // u是叶节点
        return;
    dfs2(son[u], u, Top); // 先走重儿子
    for(auto v : e[u])
        if(v != fa && v != son[u])
```

```
            dfs2(v, u, v);
}
int col[MAXN];
set<int>black[MAXN];
int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    for(int i = 0; i < n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].pb(v);
        e[v].pb(u);
    }
    dfs(1, 0);
    dfs2(1, 0, 1);
    while(q--) {
        int op, u;
        scanf("%d%d", &op, &u);
        if(op == 0) {
            col[u] ^= 1;
            if(col[u] == 1)
                black[top[u]].insert(id[u]);
            else
                black[top[u]].erase(id[u]);
        } else {
            int ans = -1;
            while(u) {
                if(!black[top[u]].empty()) {
                    int tmp = *black[top[u]].begin();
                    if(tmp <= id[u])
                        ans = tmp;
                }
                u = Fa[top[u]];
            }
            if(ans != -1)
                ans = rk[ans];
            printf("%d\n", ans);
        }
    }
```

```
70 }
```

1.48.4 QTREE4

题意：

给一棵树，每个点有颜色，初始都是黑色。有两种操作：改变某点颜色（白变黑，黑变白）；输出树上两个黑点间的距离最大值（两个点可以相同），没有黑点则输出“They have disappeared.”。

思路：

动态点分治，详见点分树模块。

```
1  vector<pii>e[MAXN];
2  // 点分治--------------------------------------------
3  int sz[MAXN];
4  int vis[MAXN];
5  int mima = INF; // 最大子树的最小值
6  int root;
7  int sumsz; // 当前树的大小
8  void getrt(int u, int fa) { // 求重心
9      sz[u] = 1;
10     int maxson = 0;
11     for(auto i : e[u]) {
12         int v = i.fi;
13         if(v == fa || vis[v])
14             continue;
15         getrt(v, u);
16         sz[u] += sz[v];
17         maxson = max(maxson, sz[v]);
18     }
19     maxson = max(maxson, sumsz - sz[u]);
20     if(mima > maxson) {
21         mima = maxson;
22         root = u;
23     }
24 }
25 int Fa[MAXN];
26 void divide(int u, int totsz) {
27     vis[u] = 1;
```

```
28      for(auto i : e[u]) {
29          int v = i.fi;
30          if(vis[v])
31              continue;
32          mima = INF;
33          sumsz = sz[v] > sz[u] ? totsz - sz[u] : sz[v];
34          getrt(v, 0);
35          Fa[root] = u;
36          divide(root, sz[v]);
37      }
38  }
39  int build(int n) { // 建立点分树
40      sumsz = n;
41      mima = INF;
42      getrt(1, 0);
43      int rt = root;
44      divide(root, sumsz);
45      return rt;
46  }
47  // LCA-----------------------------------------------
48  pii a[MAXN * 2];
49  pii dp[MAXN * 2][25];
50  void init_rmq(int n) {
51      for(int i = 1; i <= n; i++)
52          dp[i][0] = a[i];
53      for(int j = 1; (1 << j) <= n; j++)
54          for(int i = 1; i + (1 << j) - 1 <= n; i++)
55              dp[i][j] = min(dp[i][j - 1], dp[i + (1 << j - 1)][j - 1]);
56  }
57  int pos[MAXN]; // 存节点第一次出现的位置
58  int dis[MAXN];
59  int Log[MAXN * 2];
60  int lca(int x, int y) { // 这里求的是路径边权和
61      if(!x || !y)
62          return INF;
63      int L = min(pos[x], pos[y]);
64      int R = max(pos[x], pos[y]);
65      int k = Log[R - L + 1];
66      int LCA = min(dp[L][k], dp[R - (1 << k) + 1][k]).se;
```

```
67      return dis[x] + dis[y] - 2 * dis[LCA];
68  }
69  int dep[MAXN], len = 0;
70  void dfs(int u, int fa) {
71      pos[u] = ++len;
72      dep[u] = dep[fa] + 1;
73      a[len] = mp(dep[u], u);
74      for(auto i : e[u]) {
75          int v = i.fi, w = i.se;
76          if(v == fa)
77              continue;
78          dis[v] = dis[u] + w;
79          dfs(v, u);
80          a[++len] = mp(dep[u], u);
81      }
82  }
83  void lca_init(int rt) {
84      len = 0;
85      dfs(rt, 0);
86      init_rmq(len);
87      Log[1] = 0; // 预处理Log,加快查询
88      for(int i = 2; i < len + 5; i++)
89          Log[i] = Log[i / 2] + 1;
90  }
91  //------------------------------------------------------
92  struct Heap {
93      priority_queue<int>A, B;
94      void add(int x) {
95          if(abs(x) >= INF)
96              return;
97          A.push(x);
98      }
99      void del(int x) {
100         if(A.empty() || abs(x) >= INF)
101             return;
102         A.top() == x ? A.pop() : B.push(x);
103     }
104     int max1() { // 查询最大值
105         while(!A.empty() && !B.empty() && A.top() == B.top()) {
```

```
106             A.pop();
107             B.pop();
108         }
109         return A.empty() ? -INF : A.top();
110     }
111     int max2() { // 查询次大值
112         int x = max1();
113         if(x == -INF)
114             return x;
115         A.pop();
116         int y = max1();
117         A.push(x);
118         return y;
119     }
120 } maxd[MAXN], dif[MAXN], ans;
121 int col[MAXN], cnt_1 = 0;
122 void change(int u) {
123     col[u] ^= 1;
124     ans.del(dif[u].max1() + dif[u].max2());
125     if(col[u] == 1) {
126         cnt_1++;
127         dif[u].add(0);
128     }
129     if(col[u] == 0) {
130         cnt_1--;
131         dif[u].del(0);
132     }
133     ans.add(dif[u].max1() + dif[u].max2());
134     int p = u;
135     while(Fa[p] != 0) {
136         int d = lca(u, Fa[p]);
137         ans.del(dif[Fa[p]].max1() + dif[Fa[p]].max2());
138         dif[Fa[p]].del(maxd[p].max1());
139         if(col[u] == 1)
140             maxd[p].add(d);
141         if(col[u] == 0)
142             maxd[p].del(d);
143         dif[Fa[p]].add(maxd[p].max1());
144         ans.add(dif[Fa[p]].max1() + dif[Fa[p]].max2());
```

```
        p = Fa[p];
    }
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n - 1; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        e[u].pb({v, w});
        e[v].pb({u, w});
    }
    lca_init(1);
    int rt = build(n);
    for(int i = 1; i <= n; i++)
        change(i);
    int q;
    scanf("%d", &q);
    while(q--) {
        char op = getchar();
        while(op != 'C' && op != 'A')
            op = getchar();
        if(op == 'A') {
            if(cnt_1 == 0)
                printf("They have disappeared.\n");
            else if(cnt_1 == 1)
                printf("0\n");
            else
                printf("%d\n", max(0, ans.max1()));
        } else if(op == 'C') {
            int u;
            scanf("%d", &u);
            change(u);
        }
    }
}
```

1.48.5 QTREE5

题意：

给一棵树，每个点有颜色，初始都是黑色。有两种操作：改变某点颜色（白变黑，黑变白）；输出 u 到最近的白点的距离，没有白点则输出”-1”。

思路：

动态点分治模板。

```
vi e[MAXN];
// 点分治----------------------------------------
int sz[MAXN];
int vis[MAXN];
int mima = INF; // 最大子树的最小值
int root;
int sumsz; // 当前树的大小
void getrt(int u, int fa) { // 求重心
    sz[u] = 1;
    int maxson = 0;
    for(auto v : e[u]) {
        if(v == fa || vis[v])
            continue;
        getrt(v, u);
        sz[u] += sz[v];
        maxson = max(maxson, sz[v]);
    }
    maxson = max(maxson, sumsz - sz[u]);
    if(mima > maxson) {
        mima = maxson;
        root = u;
    }
}
int Fa[MAXN];
void divide(int u, int totsz) {
    vis[u] = 1;
    for(auto v : e[u]) {
        if(vis[v])
            continue;
        mima = INF;
```

```
31          sumsz = sz[v] > sz[u] ? totsz - sz[u] : sz[v];
32          getrt(v, 0);
33          Fa[root] = u;
34          divide(root, sz[v]);
35      }
36  }
37  int build(int n) { // 建立点分树
38      sumsz = n;
39      mima = INF;
40      getrt(1, 0);
41      int rt = root;
42      divide(root, sumsz);
43      return rt;
44  }
45  // LCA-------------------------------------------
46  pii a[MAXN * 2];
47  pii dp[MAXN * 2][25];
48  void init_rmq(int n) {
49      for(int i = 1; i <= n; i++)
50          dp[i][0] = a[i];
51      for(int j = 1; (1 << j) <= n; j++)
52          for(int i = 1; i + (1 << j) - 1 <= n; i++)
53              dp[i][j] = min(dp[i][j - 1], dp[i + (1 << j - 1)][j - 1]);
54  }
55  int pos[MAXN]; // 存节点第一次出现的位置
56  int dis[MAXN];
57  int Log[MAXN * 2];
58  int lca(int x, int y) { // 这里求的是路径边权和
59      if(!x || !y)
60          return INF;
61      int L = min(pos[x], pos[y]);
62      int R = max(pos[x], pos[y]);
63      int k = Log[R - L + 1];
64      int LCA = min(dp[L][k], dp[R - (1 << k) + 1][k]).se;
65      return dis[x] + dis[y] - 2 * dis[LCA];
66  }
67  int dep[MAXN], len = 0;
68  void dfs(int u, int fa) {
69      pos[u] = ++len;
```

```
70      dep[u] = dep[fa] + 1;
71      a[len] = mp(dep[u], u);
72      for(auto v : e[u]) {
73          if(v == fa)
74              continue;
75          dis[v] = dis[u] + 1;
76          dfs(v, u);
77          a[++len] = mp(dep[u], u);
78      }
79  }
80  void lca_init(int rt) {
81      len = 0;
82      dfs(rt, 0);
83      init_rmq(len);
84      Log[1] = 0; // 预处理Log,加快查询
85      for(int i = 2; i < len + 5; i++)
86          Log[i] = Log[i / 2] + 1;
87  }
88  //------------------------------------------------
89  multiset<int>S1[MAXN], S2[MAXN];
90  int col[MAXN];
91  void change(int u) {
92      col[u] ^= 1;
93      int p = u;
94      while(p != 0) {
95          int d = lca(p, u);
96          if(col[u] == 1)
97              S1[p].insert(d);
98          else
99              S1[p].erase(S1[p].find(d));
100         if(Fa[p] != 0) {
101             int d = lca(Fa[p], u);
102             if(col[u] == 1)
103                 S2[p].insert(d);
104             else
105                 S2[p].erase(S2[p].find(d));
106         }
107         p = Fa[p];
108     }
```

```
}
int check(int u) {
    multiset<int>ans;
    int p = u;
    int last = 0;
    while(p != 0) {
        int d = lca(p, u);
        if(!S1[p].empty())
            ans.insert(*S1[p].begin() + d);
        if(last != 0 && !S2[last].empty()) {
            auto p = ans.find(*S2[last].begin() + d);
            if(p != ans.end())
                ans.erase(p);
        }
        last = p;
        p = Fa[p];
    }
    if(ans.empty())
        return -1;
    return *ans.begin();
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].pb(v);
        e[v].pb(u);
    }
    lca_init(1);
    build(n);
    int q;
    scanf("%d", &q);
    while(q--) {
        int op, u;
        scanf("%d%d", &op, &u);
        if(op == 0)
            change(u);
```

```
148         else
149             printf("%d\n", check(u));
150     }
151 }
```

1.48.6 QTREE6

题意：

给一棵树，每个点有颜色，初始都是黑色。有两种操作：改变某点颜色（白变黑，黑变白）；输出 u 所在的和 $col[u]$ 相同颜色的联通块的大小。

思路：

用 $Black(u)$ 表示每个子结点子树内的黑色联通块大小，如果自己也是黑色，就再加 1。$White(u)$ 同理，即：

$Black(u) = \sum_{v} Black(v) + (col[u] == 1)$

$White(u) = \sum_{v} White(v) + (col[u] == 0)$

对于查询,从当前点 u 向上找到最远的同色结点 anc ,答案就是 $Black(anc)$ 或 $White(anc)$。

对于修改，以黑变白为例。令 $x1 = Black(u)$ ，把从 $Fa[u]$ 向上到最近的白色结点 anc 的 $Black$ 值减去 $x1$ ，再把 $Black(u)$ 减 1 ；把 $White(u)$ 加 1 ，令 $x2 = White(u)$ ，把从 $Fa[u]$ 向上到最近的黑色结点 anc 的 $White$ 值加上 $x2$ 。

具体实现：重链剖分，用树状数组维护每个点的 $Black$ 和 $White$ 值，用线段树的每个结点维护对应区间内最右侧的黑色结点和白色结点。

```
1  vi e[MAXN];
2  int dep[MAXN], sz[MAXN];
3  int son[MAXN]; // 存重儿子
4  int Fa[MAXN];
5  void dfs(int u, int fa) { // 预处理
6      dep[u] = dep[fa] + 1;
7      sz[u] = 1;
8      Fa[u] = fa;
9      for(auto v : e[u]) {
10         if(v == fa)
11             continue;
12         dfs(v, u);
13         sz[u] += sz[v];
14         if(sz[v] > sz[son[u]])
```

```
15              son[u] = v;
16      }
17  }
18  int top[MAXN]; // 存所在重链的顶部节点
19  int id[MAXN]; // 点标号->dfs序号
20  int rk[MAXN]; // dfs序号->点标号
21  int cnt = 0;
22  void dfs2(int u, int fa, int Top) { // 标记dfs序
23      top[u] = Top;
24      id[u] = ++cnt;
25      rk[cnt] = u;
26      if(!son[u]) // u是叶节点
27          return;
28      dfs2(son[u], u, Top); // 先走重儿子
29      for(auto v : e[u])
30          if(v != fa && v != son[u])
31              dfs2(v, u, v);
32  }
33  // 线段树-----------------------------------------------------------
34  struct Seg_Tree {
35  #define lson rt<<1
36  #define rson rt<<1|1
37  #define mid (Tree[rt].L+Tree[rt].R)/2
38      struct node {
39          int L, R, p[2];
40      } Tree[MAXN << 2];
41      void push_up(int rt) {
42          Tree[rt].p[0] = max(Tree[lson].p[0], Tree[rson].p[0]);
43          Tree[rt].p[1] = max(Tree[lson].p[1], Tree[rson].p[1]);
44      }
45      void build(int rt, int L, int R) {
46          Tree[rt].L = L;
47          Tree[rt].R = R;
48          Tree[rt].p[0] = 0;
49          Tree[rt].p[1] = R;
50          if(L == R)
51              return;
52          build(lson, L, mid);
53          build(rson, mid + 1, R);
```

```
54      }
55      int query(int rt, int L, int R, int col) {
56          if(L <= Tree[rt].L && Tree[rt].R <= R)
57              return Tree[rt].p[col];
58          int ans = 0;
59          if(R >= mid + 1)
60              ans = query(rson, L, R, col);
61          if(ans != 0)
62              return ans;
63          if(L <= mid)
64              ans = query(lson, L, R, col);
65          return ans;
66      }
67      void update(int rt, int p, int col) {
68          if(Tree[rt].L == Tree[rt].R) {
69              Tree[rt].p[col ^ 1] = p;
70              Tree[rt].p[col] = 0;
71              return;
72          }
73          if(p <= mid)
74              update(lson, p, col);
75          else
76              update(rson, p, col);
77          push_up(rt);
78      }
79  } seg;
80  // 树状数组----------------------------------------------------------
81  struct B_I_T {
82      int n;
83      int sum1[MAXN][2], sum2[MAXN][2];
84      int lowbit(int x) {
85          return x & (-x);
86      }
87      void update(int L, int R, int k, int col) {
88          int x = L;
89          while(L <= n) {
90              sum1[L][col] += k;
91              sum2[L][col] += k * x;
92              L += lowbit(L);
```

```
        }
        R++, x = R;
        while(R <= n) {
            sum1[R][col] -= k;
            sum2[R][col] -= k * x;
            R += lowbit(R);
        }
    }
    int get_sum(int L, int R, int col) {
        int ans = 0, x = R;
        while(R > 0) {
            ans += (x + 1) * sum1[R][col] - sum2[R][col];
            R -= lowbit(R);
        }
        L--, x = L;
        while(L > 0) {
            ans -= (x + 1) * sum1[L][col] - sum2[L][col];
            L -= lowbit(L);
        }
        return ans;
    }
    void init(int _n) {
        n = _n;
        for(int i = 1; i <= n; i++)
            update(i, i, sz[rk[i]], 1);
    }
} bit;
//---------------------------------------------------------------------
void skip_update(int u, int col, int x) {
    while(u != 0) {
        int L = seg.query(1, id[top[u]], id[u], col ^ 1);
        if(L != 0) {
            bit.update(L, id[u], x, col);
            break;
        }
        bit.update(id[top[u]], id[u], x, col);
        u = Fa[top[u]];
    }
}
```

```cpp
int col[MAXN];
void change(int u) {
    // bit更新
    // 从父节点到最近的异色结点
    int x1 = bit.get_sum(id[u], id[u], col[u]);
    skip_update(Fa[u], col[u], -x1);
    bit.update(id[u], id[u], -1, col[u]);
    // 从父节点到最近的同色结点
    bit.update(id[u], id[u], 1, col[u] ^ 1);
    int x2 = bit.get_sum(id[u], id[u], col[u] ^ 1);
    skip_update(Fa[u], col[u] ^ 1, x2);
    // 线段树更新
    seg.update(1, id[u], col[u]);
    // 颜色更新
    col[u] ^= 1;
}
int check(int u) {
    int  p = u;
    int Top = u; // 最近的同色祖先结点
    while(p != 0) {
        int L = seg.query(1, id[top[p]], id[p], col[u] ^ 1);
        if(L != 0) { // 有异色点
            if(rk[L] != p)
                Top = rk[L + 1];
            break;
        }
        Top = top[p]; // 没有异色点
        p = Fa[top[p]];
    }
    return bit.get_sum(id[Top], id[Top], col[u]);
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].pb(v);
        e[v].pb(u);
```

```
171     }
172     dfs(1, 0);
173     dfs2(1, 0, 1);
174     for(int i = 1; i <= n; i++)
175         col[i] = 1;
176     seg.build(1, 1, n);
177     bit.init(n);
178     int q;
179     scanf("%d", &q);
180     while(q--) {
181         int op, u;
182         scanf("%d%d", &op, &u);
183         if(op == 0)
184             printf("%d\n", check(u));
185         else
186             change(u);
187     }
188 }
```

1.48.7 QTREE7

题意：

给一棵树，给定每个点初始的颜色和点权。有三种操作：改变某点颜色（白变黑，黑变白）；改变某点的点权；输出 u 所在的和 $col[u]$ 相同颜色的联通块中的点的最大点权。

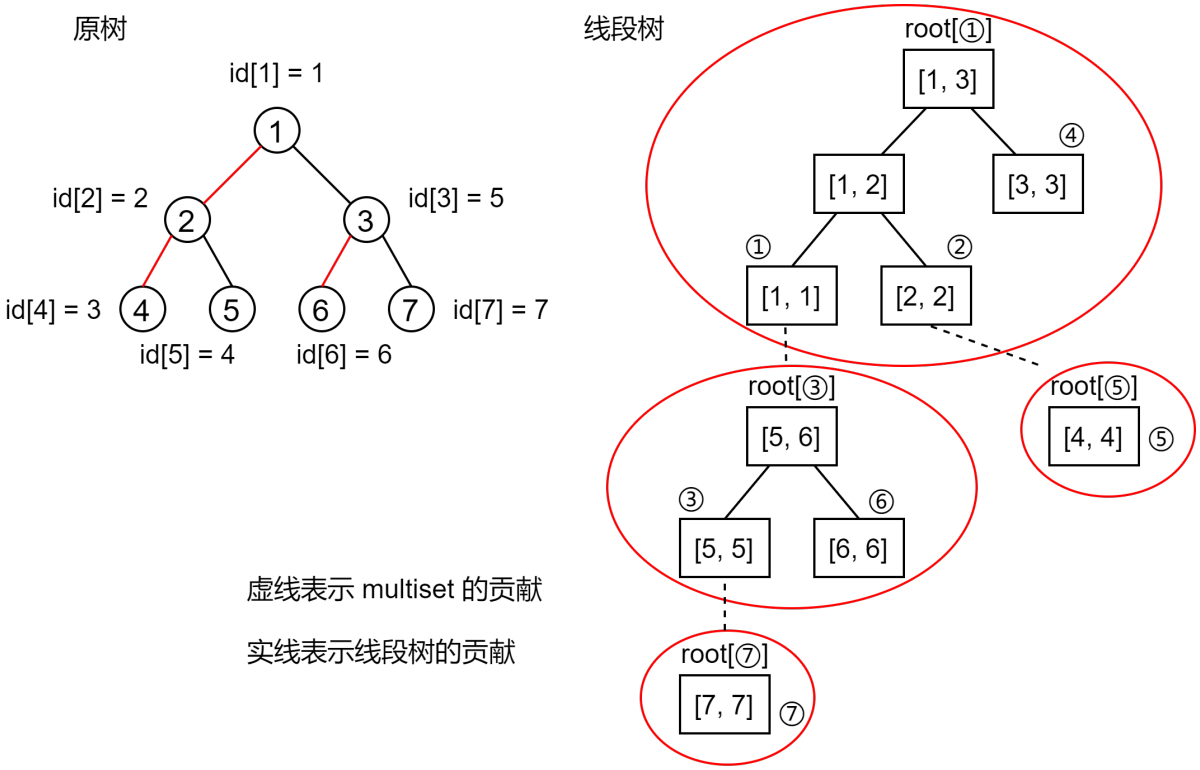
思路：

把原树重链剖分。

每个结点用 maxx 维护轻儿子（一条重链的顶点）对它的贡献，maxx 要支持最大值查询和元素删除，因此用 multiset 实现。

对每条重链用一棵线段树维护，线段树上的每个结点维护 Lp，Lv，Rp，Rv 等信息。

对线段树上的某个代表【L，R】区间的结点，Lp 表示从左往右的最后一个同色结点的位置，Lv 表示【L，Lp】内结点的 maxx 的最大值；Rp 表示从右往左的最后一个同色结点的位置，Rv 表示【Rp，R】内结点的 maxx 的最大值。



对于查询，从 u 向上找最远的同色结点 x 所在的重链。最终找到一个点 p，满足 x 存在于 p 到 $top[p]$ 之间。通过这条重链上维护的信息来查找答案，详见代码。

对于更新颜色，把从 u 向上到根节点的路径一段一段存下来（即存跳跃的两个端点）。把记录的路径上的信息从下往上更新，详见代码。

对于更新点权，与更新颜色类似，详见代码。

```
vi e[MAXN];
int val[MAXN], col[MAXN];
//--------------------------------------------------------
int dep[MAXN], sz[MAXN];
int son[MAXN]; // 存重儿子
int Fa[MAXN];
void dfs(int u, int fa) { // 预处理
    dep[u] = dep[fa] + 1;
    sz[u] = 1;
    Fa[u] = fa;
    for(auto v : e[u]) {
```

```
12          if(v == fa)
13              continue;
14          dfs(v, u);
15          sz[u] += sz[v];
16          if(sz[v] > sz[son[u]])
17              son[u] = v;
18      }
19  }
20  int top[MAXN]; // 存所在重链的顶部节点
21  int id[MAXN]; // 点标号->dfs序号
22  int rk[MAXN]; // dfs序号->点标号
23  int len[MAXN]; // 重链长度
24  int cnt = 0;
25  void dfs2(int u, int fa, int Top) { // 标记dfs序
26      top[u] = Top;
27      len[Top]++;
28      id[u] = ++cnt;
29      rk[cnt] = u;
30      if(!son[u]) // u是叶节点
31          return;
32      dfs2(son[u], u, Top); // 先走重儿子
33      for(auto v : e[u])
34          if(v != fa && v != son[u])
35              dfs2(v, u, v);
36  }
37  // 线段树-----------------------------------------------
38  int root[MAXN]; // 重链Top结点在线段树中的结点编号
39  multiset<int>maxx[MAXN][2]; // 轻儿子对父节点的贡献
40  #define mid (Tree[rt].L+Tree[rt].R)/2
41  struct node {
42      int L, R;
43      int Lp, Lv; // [L, Lp]同色, Lv是其中最大值
44      int Rp, Rv; // [Rp, R]同色, Rv是其中最大值
45      int lson, rson;
46  } Tree[MAXN << 2];
47  int cnt_n = 0;
48  void push_up1(int rt, int u) { // 线段树叶结点更新
49      int x = val[u];
50      if(!maxx[u][col[u]].empty())
```

```
51          x = max(x, *(--maxx[u][col[u]].end()));
52      Tree[rt].Lp = Tree[rt].Rp = id[u];
53      Tree[rt].Lv = Tree[rt].Rv = x;
54  }
55  node push_up2(int id1, int id2) { // 线段树非叶节点更新
56      node a = Tree[id1];
57      node b = Tree[id2];
58      int f = 0;
59      if(col[rk[a.R]] == col[rk[b.L]])
60          f = 1;
61      node ans = {a.L, b.R, a.Lp, a.Lv, b.Rp, b.Rv, id1, id2};
62      if(f && a.Lp == a.R) { // [a.L, a.R]+[b.L, b.Lp]同色
63          ans.Lp = b.Lp;
64          ans.Lv = max(a.Lv, b.Lv);
65      }
66      if(f && b.Rp == b.L) { // [a.Rp, a.R]+[b.L, b.R]同色
67          ans.Rp = a.Rp;
68          ans.Rv = max(a.Rv, b.Rv);
69      }
70      return ans;
71  }
72  void build(int &rt, int L, int R) {
73      rt = ++cnt_n;
74      Tree[rt].L = L;
75      Tree[rt].R = R;
76      if(L == R) {
77          int u = rk[L]; // u是线段树的叶节点
78          for(auto v : e[u]) {
79              if(v == Fa[u] || v == son[u])
80                  continue;
81              // 轻儿子v是新的重链顶点
82              build(root[v], id[v], id[v] + len[v] - 1);
83              maxx[u][col[v]].insert(Tree[root[v]].Lv);
84          }
85          push_up1(rt, u);
86          return;
87      }
88      build(Tree[rt].lson, L, mid);
89      build(Tree[rt].rson, mid + 1, R);
```

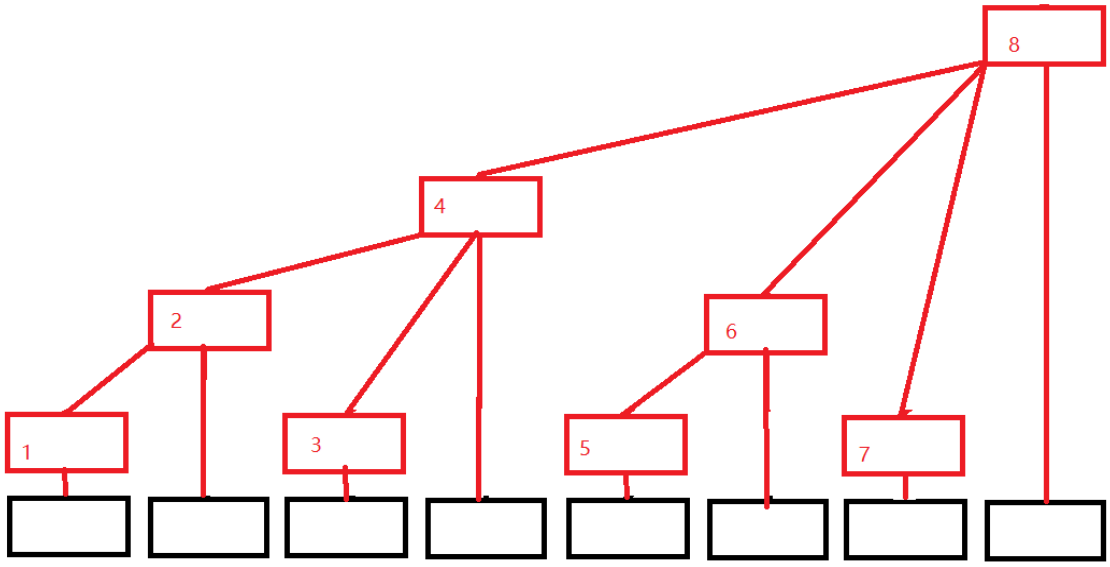
```
90      Tree[rt] = push_up2(Tree[rt].lson, Tree[rt].rson);
91  }
92  //-------------------------------------------------------
93  deque<int> path; // 一段一段存路径
94  void skip_path(int u) { // 得到u到根节点的路径
95      path.clear();
96      while(u != 0) {
97          path.push_front(u);
98          path.push_front(top[u]);
99          u = Fa[top[u]];
100     }
101 }
102 void update(int rt, int i, int f) {
103     if(Tree[rt].L == Tree[rt].R) {
104         int u = rk[Tree[rt].L]; // u是线段树叶节点
105         if(i + 2 < SZ(path)) { // 还有下一段路径
106             int v = path[i + 1]; // v是重链顶点
107             auto p = maxx[u][col[v]].find(Tree[root[v]].Lv);
108             maxx[u][col[v]].erase(p); // 去掉原来的贡献
109             update(root[v], i + 2, f);
110             maxx[u][col[v]].insert(Tree[root[v]].Lv); // 加入新贡献
111         }
112         if(i == SZ(path) - 1 && f) // 改变颜色
113             col[u] ^= 1;
114         push_up1(rt, u); // 更新
115         return;
116     }
117     int u = path[i];
118     if(id[u] <= mid)
119         update(Tree[rt].lson, i, f);
120     else
121         update(Tree[rt].rson, i, f);
122     Tree[rt] = push_up2(Tree[rt].lson, Tree[rt].rson); // 更新
123 }
124 void change_col(int u) {
125     skip_path(u);
126     update(1, 1, 1);
127 }
128 void change_val(int u, int x) {
```

```
129     val[u] = x;
130     skip_path(u);
131     update(1, 1, 0);
132 }
133 //------------------------------------------------------
134 int query(int rt, int u) {
135     if(Tree[rt].L == Tree[rt].R)
136         return Tree[rt].Lv;
137     int lson = Tree[rt].lson;
138     int rson = Tree[rt].rson;
139     int ans = 0;
140     if(id[u] <= mid) {
141         ans = query(lson, u);
142         // 延申到rk[mid]下方,即当前点的右儿子
143         if(Tree[lson].Rp <= id[u] && col[rk[mid]] == col[rk[mid + 1]])
144             ans = max(ans, Tree[rson].Lv);
145     } else {
146         ans = query(rson, u);
147         // 延申到rk[mid+1]上方,即当前点的左儿子
148         if(id[u] <= Tree[rson].Lp && col[rk[mid]] == col[rk[mid + 1]])
149             ans = max(ans, Tree[lson].Rv);
150     }
151     return ans;
152 }
153 int check(int u) {
154     while(u != 0) { // 向上找最远的同色点所在的重链
155         int Top = top[u];
156         if(Tree[root[Top]].Lp < id[u] || col[Fa[Top]] != col[Top] || !Fa[Top])
157             break;
158         u = Fa[u];
159     }
160     int Top = top[u];
161     return query(root[Top], u);
162 }
163 //------------------------------------------------------
164 int main() {
165     int n;
166     scanf("%d", &n);
167     for(int i = 0; i < n - 1; i++) {
```

```
168         int u, v;
169         scanf("%d%d", &u, &v);
170         e[u].pb(v);
171         e[v].pb(u);
172     }
173     for(int i = 1; i <= n; i++)
174         scanf("%d", col + i);
175     for(int i = 1; i <= n; i++)
176         scanf("%d", val + i);
177     dfs(1, 0);
178     dfs2(1, 0, 1);
179     build(root[1], 1, len[1]);
180     int q;
181     scanf("%d", &q);
182     while(q--) {
183         int op;
184         scanf("%d", &op);
185         if(op == 0) {
186             int u;
187             scanf("%d", &u);
188             printf("%d\n", check(u));
189         }
190         if(op == 1) {
191             int u;
192             scanf("%d", &u);
193             change_col(u);
194         }
195         if(op == 2) {
196             int u, x;
197             scanf("%d%d", &u, &x);
198             change_val(u, x);
199         }
200     }
201 }
```

2 数据结构

2.1 树状数组



原数组：$a[1]\cdots a[n]$

差值数组：$d[i]=a[i]-a[i-1]$

差值前缀和：$sum1[x]=\sum_{i=1}^{x} d[i]$ $sum2[x]=\sum_{i=1}^{x} i*d[i]$

前缀和：$\sum_{i=1}^{x} a[i]=(x+1)*sum1[x]-sum2[x]$

2.1.1 区间修改 + 区间求和

```
int a[MAXN];
int sum1[MAXN];
int sum2[MAXN];
int lowbit(int x) { // pow(2, 右侧连续0的长度)
    return x & (-x);
}
void update(int n, int L, int R, int k) {
    int x = L;
    while(L <= n) {
```

```
        sum1[L] += k;
        sum2[L] += k * x;
        L += lowbit(L);
    }
    R++, x = R;
    while(R <= n) {
        sum1[R] -= k;
        sum2[R] -= k * x;
        R += lowbit(R);
    }
}
int get_sum(int L, int R) {
    int ans = 0, x = R;
    while(R > 0) {
        ans += (x + 1) * sum1[R] - sum2[R];
        R -= lowbit(R);
    }
    L--, x = L;
    while(L > 0) {
        ans -= (x + 1) * sum1[L] - sum2[L];
        L -= lowbit(L);
    }
    return ans;
}
void init() {
    memset(sum1, 0, sizeof sum1);
    memset(sum2, 0, sizeof sum2);
}
int main() {
    init();
    for(int i = 1; i <= n; i++) {
        scanf("%d", a + i);
        update(n, i, i, a[i]);
    }
}
```

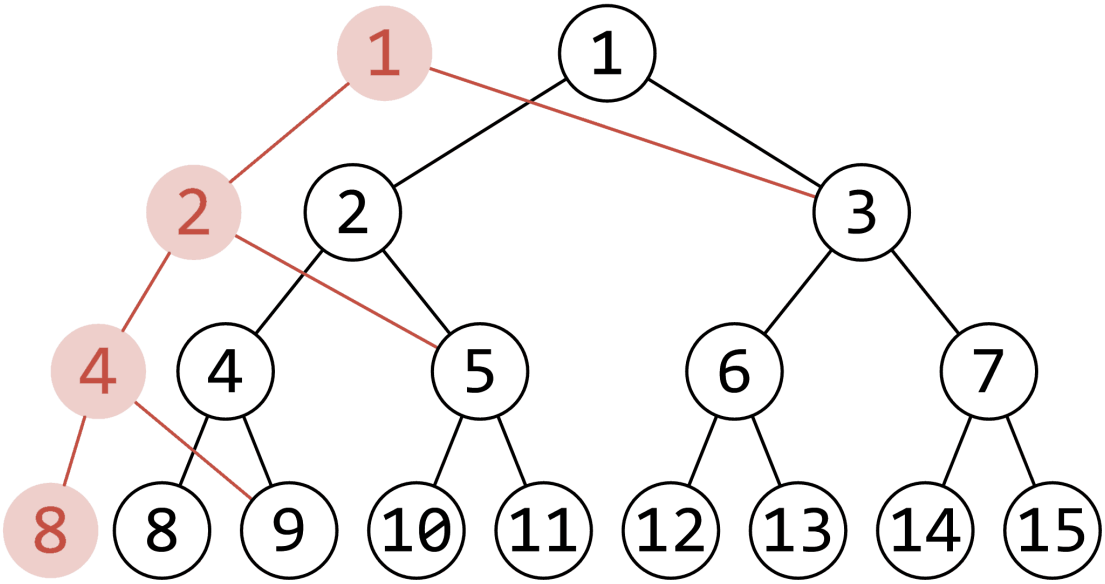
2.1.2 区间最值：分治思想

```
int h[MAXN];
```

```
int maxx[MAXN], minn[MAXN];
int lowbit(int x) {
    return x & (-x);
}
void update(int n, int p, int x) { // lognlogn
    h[p] = x;
    for(; p <= n; p += lowbit(p)) {
        maxx[p] = h[p];
        minn[p] = h[p];
        for(int p2 = 1; p2 < lowbit(p); p2 <<= 1) {
            maxx[p] = max(maxx[p], maxx[p - p2]);
            minn[p] = min(minn[p], minn[p - p2]);
        }
    }
}
int get_max(int L, int R) { // logn
    if(L == R)
        return h[L];
    if(L <= R - lowbit(R))
        return max(get_max(L, R - lowbit(R)), maxx[R]);
    return max(get_max(L, R - 1), h[R]);
}
int get_min(int L, int R) { // logn
    if(L == R)
        return h[L];
    if(L <= R - lowbit(R))
        return min(get_min(L, R - lowbit(R)), minn[R]);
    return min(get_min(L, R - 1), h[R]);
}
int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    for(int i = 1; i <= n; i++) {
        int x;
        scanf("%d", &x);
        update(n, i, x);
    }
    while(q--) {
        int L, R;
```

```
41          scanf("%d%d", &L, &R);
42          printf("%d\n", get_max(L, R) - get_min(L, R));
43      }
44  }
```

2.2 主席树




```
1   const int MAXN = 2e5 + 5;
2   int ls[MAXN << 5], rs[MAXN << 5], sum[MAXN << 5], rt[MAXN];
3   int cnt = 0;
4   int build(int l, int r) {
5       int root = ++cnt;
6       sum[root] = 0;
7       if(l < r) {
8           int mid = (l + r) >> 1;
9           ls[root] = build(l, mid);
10          rs[root] = build(mid + 1, r);
11      }
12      return root;
13  }
14  int update(int ori, int l, int r, int pos) { //覆盖更新ori节点
```

```
15      int now = ++cnt;
16      ls[now] = ls[ori];
17      rs[now] = rs[ori];
18      sum[now] = sum[ori] + 1;//个数+1
19      if(l < r) {
20          int mid = (l + r) >> 1;
21          if(mid >= pos)//判断更新左节点还是右节点
22              ls[now] = update(ls[now], l, mid, pos);
23          else
24              rs[now] = update(rs[now], mid + 1, r, pos);
25      }
26      return now;
27  }
28  int query(int u, int v, int l, int r, int k) {//返回区间第k小的标号
29      if(l == r)
30          return l;
31      int mid = (l + r) >> 1;
32      int x = sum[ls[v]] - sum[ls[u]];
33      if(x >= k)
34          return query(ls[u], ls[v], l, mid, k);
35      else
36          return query(rs[u], rs[v], mid + 1, r, k - x);
37  }
38  int a[MAXN], b[MAXN], ulen;
39  int main() {
40      int n, m;
41      scanf("%d%d", &n, &m);
42      for(int i = 1; i <= n; i++) {
43          scanf("%d", a + i);
44          b[i] = a[i];
45      }
46      sort(b + 1, b + n + 1);
47      ulen = unique(b + 1, b + n + 1) - b - 1;
48      rt[0] = build(1, ulen);
49      for(int i = 1; i <= n; i++) {
50          int pos = lower_bound(b + 1, b + ulen + 1, a[i]) - b;
51          rt[i] = update(rt[i - 1], 1, ulen, pos);
52      }
53      while(m--) {
```

```
        int L, R, k;
        scanf("%d%d%d", &L, &R, &k);
        int ans = query(rt[L - 1], rt[R], 1, ulen, k);
        printf("%d\n", b[ans]);
    }
}
```

2.2.1 主席树维护静态树上第 k 大

给定一棵树，树上每个点有点权。每次询问 u 到 v 的路径上第 k 大的点权。

```
//主席树-----------------------------------------------
int a[MAXN], b[MAXN], ulen;
int ls[MAXN << 5], rs[MAXN << 5], sum[MAXN << 5], rt[MAXN];
int cnt1 = 0;
int build(int l, int r) {
    int root = ++cnt1;
    sum[root] = 0;
    if(l < r) {
        int mid = (l + r) >> 1;
        ls[root] = build(l, mid);
        rs[root] = build(mid + 1, r);
    }
    return root;
}
int update(int ori, int l, int r, int pos) { //覆盖更新ori节点
    int now = ++cnt1;
    ls[now] = ls[ori];
    rs[now] = rs[ori];
    sum[now] = sum[ori] + 1;//个数+1
    if(l < r) {
        int mid = (l + r) >> 1;
        if(mid >= pos)//判断更新左节点还是右节点
            ls[now] = update(ls[now], l, mid, pos);
        else
            rs[now] = update(rs[now], mid + 1, r, pos);
    }
    return now;
```

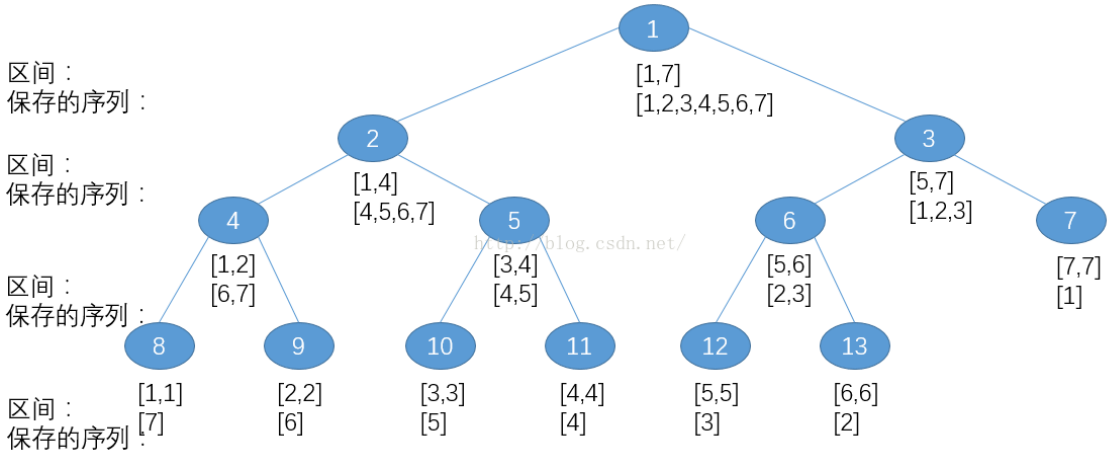
```
}
int query(int u, int v, int LCA, int lca_fa, int l, int r, int k) { //返回区间第k小的标号
    if(l == r)
        return l;
    int mid = (l + r) >> 1;
    int x = sum[ls[u]] + sum[ls[v]] - sum[ls[LCA]] - sum[ls[lca_fa]];
    if(x >= k)
        return query(ls[u], ls[v], ls[LCA], ls[lca_fa], l, mid, k);
    else
        return query(rs[u], rs[v], rs[LCA], rs[lca_fa], mid + 1, r, k - x);
}
//前向星-----------------------------------------------
struct edge {
    int to, next;
} e[MAXM * 2]; //双倍内存
int cnt2 = 0;
int head[MAXN];
void add_edge(int u, int v) { //从1开始存
    e[++cnt2].to = v;
    e[cnt2].next = head[u];
    head[u] = cnt2;
}
//LCA-------------------------------------------------
int dep[MAXN], f[MAXN][25];
void dfs(int u, int fa) {
    //dfs序建主席树
    int pos = lower_bound(b + 1, b + ulen + 1, a[u]) - b;
    rt[u] = update(rt[fa], 1, ulen, pos);
    //处理f数组
    dep[u] = dep[fa] + 1;
    for(int i = 1; (1 << i) <= dep[u]; i++)
        f[u][i] = f[f[u][i - 1]][i - 1];
    for(int i = head[u]; i; i = e[i].next) {
        int v = e[i].to;
        if(v == fa)//去掉双向边
            continue;
        f[v][0] = u;
        dfs(v, u);
```

```
    }
}
int lca(int x, int y) {
    if(dep[x] < dep[y])
        swap(x, y);
    for(int i = 20; i >= 0; i--) {
        if(dep[f[x][i]] >= dep[y])
            x = f[x][i];
        if(x == y)
            return x;
    }
    for(int i = 20; i >= 0; i--)
        if(f[x][i] != f[y][i]) {
            x = f[x][i];
            y = f[y][i];
        }
    return f[x][0];
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for(int i = 1; i <= n; i++) {
        scanf("%d", a + i);
        b[i] = a[i];
    }
    for(int i = 0; i < n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        add_edge(u, v);
        add_edge(v, u);
    }
    sort(b + 1, b + n + 1);
    ulen = unique(b + 1, b + n + 1) - b - 1;
    rt[0] = build(1, ulen);
    dfs(1, 0);
    while(m--) {
        int u, v, k;
        scanf("%d%d%d", &u, &v, &k);
        int LCA = lca(u, v);
```

```
105         int lca_fa = f[LCA][0];
106         int ans = query(rt[u], rt[v], rt[LCA], rt[lca_fa], 1, ulen, k);
107         printf("%d\n", b[ans]);
108     }
109 }
```

2.3 归并树

建树 $O(nlogn)$，查询第 k 小 $O(lognlognlogn)$




```
1   int a[MAXN];
2   int Merge[30][MAXN];
3   void build(int deep, int l, int r) {
4       if(l == r) {
5           Merge[deep][l] = a[l];
6           return;
7       }
8       int mid = (l + r) >> 1;
9       build(deep + 1, l, mid);
10      build(deep + 1, mid + 1, r);
11      for(int i = l, j = mid + 1, k = l; i <= mid || j <= r;) { //归并排序构造当
            前节点
12          if(j > r)
13              Merge[deep][k++] = Merge[deep + 1][i++];
14          else if(i > mid || Merge[deep + 1][i] > Merge[deep + 1][j])
```

```
15              Merge[deep][k++] = Merge[deep + 1][j++];
16          else
17              Merge[deep][k++] = Merge[deep + 1][i++];
18      }
19  }
20  //计算[L,R]交[l,r]中小于x的有多少个数
21  int calc(int deep, int L, int R, int l, int r, int x) {
22      if(l <= L && R <= r)
23          return lower_bound(Merge[deep] + L, Merge[deep] + R + 1, x) - Merge[
                deep] - L;
24      int mid = (L + R) >> 1;
25      int ans = 0;
26      if(l <= mid)
27          ans += calc(deep + 1, L, mid, l, r, x);
28      if(mid < r)
29          ans += calc(deep + 1, mid + 1, R, l, r, x);
30      return ans;
31  }
32  //区间第k小值查询(1,...,k,...,n)
33  int query(int n, int l, int r, int k) {
34      int L = 1, R = n;
35      while(L < R) {//答案为cnt = k - 1的最大的数
36          int mid = (L + R + 1) >> 1;
37          int cnt = calc(0, 1, n, l, r, Merge[0][mid]);
38          if(cnt <= k - 1)
39              L = mid ;
40          else
41              R = mid - 1;
42      }
43      return Merge[0][L];
44  }
45  int main() {
46      int n, m;
47      scanf("%d%d", &n, &m);
48      for(int i = 1; i <= n; i++)
49          scanf("%d", a + i);
50      build(0, 1, n);
51      while(m--) {
52          int l, r, k;
```

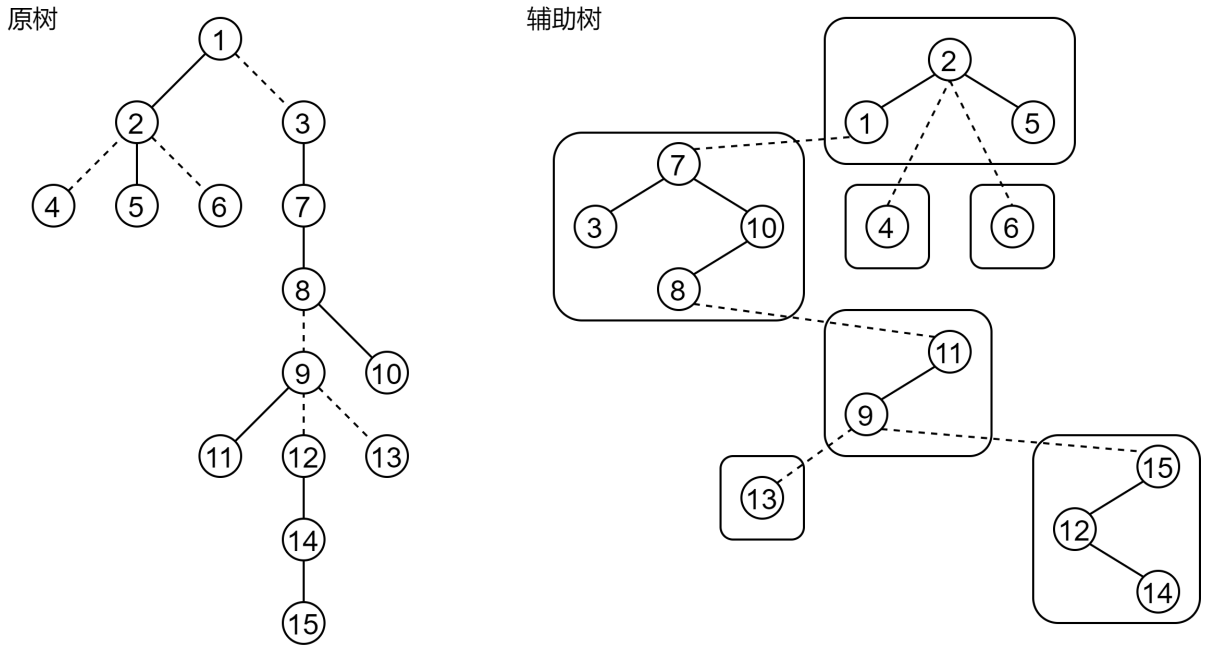
```
53          scanf("%d%d%d", &l, &r, &k);
54          printf("%d\n", query(n, l, r, k));
55      }
56  }
```

2.4 Link Cut Tree

Link Cut Tree 用于解决动态树问题：维护一个森林，支持删边，加边（保证仍是森林），同时维护森林中的一些信息。

LCT 用一棵辅助树来维护原树。辅助树由多个 splay 组成，每个 splay 表示原树上一条自上而下的路径，并且中序遍历 splay 得到的序列和该路径自上而下的序列相同。

每种操作的复杂度可近似看作 $O(logn)$ 。




```
1   #define ls ch[p][0]
2   #define rs ch[p][1]
3   struct LCT {
4       int ch[MAXN][2]; // splay上的左右儿子
5       int Fa[MAXN]; // splay上的父节点
6       int sz[MAXN]; // splay上的子树大小
```

```
7     int tag[MAXN]; // splay的子树翻转标记
8     int get(int x) { // 得到Fa[x]->x的方向
9         return ch[Fa[x]][1] == x;
10    }
11    bool isRoot(int x) { // 判断x是否是一个splay的根
12        return ch[Fa[x]][0] != x && ch[Fa[x]][1] != x;
13    }
14    void push_up(int p) { // 维护相关信息
15        sz[p] = sz[ls] + sz[rs] + 1;
16    }
17    void Reverse(int p) { // 翻转子树
18        swap(ls, rs);
19        tag[p] ^= 1;
20    }
21    void push_down(int p) { // 向下传递标记
22        if(tag[p] != 0) {
23            Reverse(ls);
24            Reverse(rs);
25            tag[p] = 0;
26        }
27    }
28    void Rotate(int x) { // x向上旋转
29        int y = Fa[x], z = Fa[y];
30        int k1 = get(x), k2 = get(y), f = isRoot(y);
31        // y->x的儿子
32        ch[y][k1] = ch[x][!k1];
33        if(ch[x][!k1] != 0)
34            Fa[ch[x][!k1]] = y;
35        // x->y
36        ch[x][!k1] = y;
37        Fa[y] = x;
38        // z->x, 注意y是splay的根时不能改
39        if(!f)
40            ch[z][k2] = x;
41        Fa[x] = z;
42        push_up(y);
43        push_up(x);
44        if(z != 0)
45            push_up(z);
```

```
46      }
47      void Update(int p) { // 从上向下push_down
48          if(!isRoot(p))
49              Update(Fa[p]);
50          push_down(p);
51      }
52      void Splay(int x) { // 使x成为当前splay中的根
53          Update(x);
54          while(!isRoot(x)) {
55              if(!isRoot(Fa[x])) {
56                  if(get(x) == get(Fa[x]))
57                      Rotate(Fa[x]);
58                  else
59                      Rotate(x);
60              }
61              Rotate(x);
62          }
63      }
64      void Access(int x) { // 把x到根节点的路径单独搞成一个splay
65          for(int p = 0; x != 0; p = x, x = Fa[x]) {
66              Splay(x);
67              ch[x][1] = p; // 因为p的深度更大
68              push_up(x);
69          }
70      }
71      void makeRoot(int p) { // 使p成为原树的根和splay中的根
72          Access(p);
73          Splay(p);
74          Reverse(p); // 翻转splay,p变成了最浅的点
75      }
76      void Split(int x, int y) { // 拉出x-y的路径搞成一个splay
77          makeRoot(x); // x变成原树的根
78          Access(y);
79          Splay(y); // y变成splay的根
80      }
81      int findRoot(int p) { // 找p在原树中的树根
82          Access(p);
83          Splay(p);
84          for(; ls != 0; p = ls)
```

```
            push_down(p);
        Splay(p);
        return p;
    }
    bool Link(int x, int y) { // 连接x,y
        makeRoot(x);
        if(findRoot(y) == x) // 已经在一棵树中
            return false;
        Fa[x] = y;
        return true;
    }
    bool Cut(int x, int y) { // 断开x,y
        makeRoot(x);
        // findRoot后x为splay的根,且splay中应只有x,y
        if(findRoot(y) != x || Fa[y] != x || ch[y][0])
            return false;
        ch[x][1] = Fa[y] = 0;
        push_up(x);
        return true;
    }
    void init(int n) {
        for(int i = 1; i <= n; i++) {
            Fa[i] = ch[i][0] = ch[i][1] = tag[i] = 0;
            sz[i] = 1;
        }
    }
    void dfs(int u) { // 用于检查
        push_down(u);
        if(ch[u][0]) {
            dfs(ch[u][0]);
            printf("u: %d L: %d\n", u, ch[u][0]);
        }
        if(ch[u][1]) {
            dfs(ch[u][1]);
            printf("u: %d R: %d\n", u, ch[u][1]);
        }
    }
} lct;
```

2.5 线段树

```
struct node { //线段树节点
    int l, r;
    ll w, f;
} tree[MAXN << 2]; //四倍空间
void build(int k, int nl, int nr) { //构造线段树
    //注意build(1,1,n)会RuntimeError
    tree[k].l = nl;
    tree[k].r = nr;
    if(tree[k].l == tree[k].r) {
        scanf("%lld", &tree[k].w);
        return;
    }
    int m = (nl + nr) / 2;
    build(2 * k, nl, m);
    build(2 * k + 1, m + 1, nr);
    tree[k].w = tree[2 * k].w + tree[2 * k + 1].w;
}
void down(int k) { //懒标记下传
    tree[2 * k].f += tree[k].f;
    tree[2 * k + 1].f += tree[k].f;
    tree[2 * k].w += tree[k].f * (tree[2 * k].r - tree[2 * k].l + 1);
    tree[2 * k + 1].w += tree[k].f * (tree[2 * k + 1].r - tree[2 * k + 1].l +
        1);
    tree[k].f = 0;
}
int ask_point(int k, int x) { //单点查询
    if(tree[k].l == tree[k].r)
        return tree[k].w;
    if(tree[k].f)
        down(k);
    int m = (tree[k].l + tree[k].r) / 2;
    if(x <= m)
        return ask_point(2 * k, x);
    else
        return ask_point(2 * k + 1, x);
}
void point_add(int k, int x, int num) { //单点修改
```

```
37      if(tree[k].l == tree[k].r) {
38          tree[k].w += num;
39          return;
40      }
41      if(tree[k].f)
42          down(k);
43      int m = (tree[k].l + tree[k].r) / 2;
44      if(x <= m)
45          point_add(2 * k, x, num);
46      else
47          point_add(2 * k + 1, x, num);
48      tree[k].w = tree[2 * k].w + tree[2 * k + 1].w;
49  }
50  ll sum(int k, int x, int y) { //区间求和
51      if(x == tree[k].l && y == tree[k].r)
52          return tree[k].w;
53      if(tree[k].f)
54          down(k);
55      int m = (tree[k].l + tree[k].r) >> 1;
56      if(y <= m)
57          return sum(k << 1, x, y);
58      else if(x > m)
59          return sum(k << 1 | 1, x, y);
60      else
61          return sum(k << 1, x, m) + sum(k << 1 | 1, m + 1, y);
62  }
63  void change_interval(int k, int a, int b, int x) { //区间修改
64      if(a <= tree[k].l && b >= tree[k].r) {
65          tree[k].w += (tree[k].r - tree[k].l + 1) * x;
66          tree[k].f += x;
67          return;
68      }
69      if(tree[k].f)
70          down(k);
71      int m = (tree[k].l + tree[k].r) / 2;
72      if(a <= m)
73          change_interval(2 * k, a, b, x);
74      if(b > m)
75          change_interval(2 * k + 1, a, b, x);
```

```
76      tree[k].w = tree[2 * k].w + tree[2 * k + 1].w;
77  }
78  int main() {
79      int n, m;
80      scanf("%d", &n); //n个底层节点
81      scanf("%d", &m); //m个查询
82      build(1, 1, n);
83      while(m--) {
84          int q;
85          scanf("%d", &q);
86          if(q == 1) {
87              int a, b, x;
88              scanf("%d%d%d", &a, &b, &x);
89              change_interval(1, a, b, x);
90          } else {
91              int pos, p2;
92              scanf("%d", &pos);
93              printf("%d\n", ask_point(1, pos));
94          }
95      }
96  }
```

3 动态规划

3.1 最长上升子序列 LIS：$O(nlogn)$

性质：b 数组一定是单调递增的，因此可以二分

```
1   int LIS(vi a) {
2       vi b;
3       for(int i = 0; i < SZ(a); i++) {
4           int p = lower_bound(b.begin(), b.end(), a[i]) - b.begin();
5           if(p == SZ(b))
6               b.pb(a[i]);
7           else
8               b[p] = a[i];
9       }
10      return SZ(b);
```

```
}
```

4 数论

4.1 快速幂

```
ll qpow(ll a, ll b) {
    ll ans = 1;
    while(b) {
        if(b & 1)
            ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
```

4.2 求组合数

```
ll qpow(ll a, ll b) {
    ll res = 1;
    a %= mod;
    while(b) {
        if(b & 1)
            res = res * a % mod;
        b >>= 1;
        a = a * a % mod;
    }
    return res % mod;
}
const int N = 2e5 + 5;
ll fac[N], inv[N];
void init() {
    fac[0] = 1;
    for(int i = 1; i < N; i++)
        fac[i] = fac[i - 1] * i % mod;
    inv[N - 1] = qpow(fac[N - 1], mod - 2);
```

```
    for(int i = N - 2; i >= 0; i--)
        inv[i] = inv[i + 1] * (i + 1) % mod;
}
ll c(ll n, ll m) { // 数据范围很大不够
    if(m > n)
        return 0;
    if(m == 0)
        return 1;
    if(n < N)
        return fac[n] * inv[m] % mod * inv[n - m] % mod;
    ll res = inv[m];
    for(int i = n - m + 1; i <= n; i++)
        res = res * i % mod;
    return res;
}
```

4.3 求卡特兰数

```
ll qpow(ll a, ll b) {
    ll ans = 1;
    while(b) {
        if(b & 1)
            ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
ll Inv[MAXN];
ll inv(ll a) { //mod为素数
    return qpow(a, mod - 2);
}
ll ktl[MAXN];
void init() {
    for(int i = 0; i < MAXN; i++)
        Inv[i] = inv(i);
    ktl[1] = 1;
    for(int i = 2; i < MAXN; i++)
        ktl[i] = 1ll * (ktl[i - 1] * (4 * i - 2) % mod) * Inv[i + 1] % mod;
```

```
}
```

4.4 矩阵快速幂

```
struct Matrix { //矩阵类
    int n, m;
    ll v[maxn][maxn];
    Matrix(int n, int m): n(n), m(m) {
        init();
    }
    void init() {
        memset(v, 0, sizeof(v));
    }
    Matrix(ll a[maxn][maxn], ll x, ll y): n(x), m(y) { //用数组初始化矩阵
        for(int i = 0; i < x; i++)
            for(int j = 0; j < y; j++)
                v[i][j] = a[i][j];
    }
    Matrix operator*(const Matrix B)const { //重载矩阵乘法
        Matrix C(n, B.m);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < B.m; j++)
                for(int k = 0; k < m; k++)
                    C.v[i][j] = (v[i][k] * B.v[k][j] % mod + C.v[i][j] + mod) %
                        mod;
        return C;
    }
};
Matrix unit(maxn, maxn); //单位矩阵
Matrix qpow(Matrix a, ll x) { //矩阵快速幂
    Matrix ret = unit;
    while(x) {
        if(x & 1)
            ret = ret * a;
        a = a * a;
        x >>= 1;
    }
    return ret;
}
```

```
int main() {
    for(int i = 0; i < maxn; i++) //初始化单位矩阵
        unit.v[i][i] = 1;
    //数组一定要开maxn
    ll a[5][5] = {1, ax*by % mod, ay*bx % mod, ax*bx % mod, ay*by % mod,
                  0, ax, 0, 0, ay,
                  0, 0, bx, 0, by,
                  0, ax*by % mod, ay*bx % mod, ax*bx % mod, ay*by % mod,
                  0, 0, 0, 0, 1
                 };
    Matrix left(a, 5, 5);
    ll b[5][5] = {{a0*b0 % mod}, {a0}, {b0}, {a0*b0 % mod}, {1}};
    Matrix right(b, 5, 1);
    right = qpow(left, n - 1) * right; //注意次数
}
```

4.5 基础公式

4.5.1 多边形面积公式

$$S = \frac{1}{2} \times [(x_1y_2 - x_2y_1) + (x_2y_3 - x_3y_2) + \cdots + (x_{n-1}y_n - x_ny_{n-1}) + (x_ny_1 - x_1y_n)]$$

4.5.2 平方和公式

$$\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{n(n+1)(2n+1)}{6}$$
$$= C_{n+2}^3 + C_{n+1}^3 = \frac{1}{4}C_{n+1}^2 = nC_{n+1}^2 - C_{n+1}^3$$

4.5.3 阶乘分解

阶乘 $N!$ 中包含质因子 x 的个数为 $\frac{N}{x} + \frac{N}{x^2} + \frac{N}{x^3} + \cdots$

4.5.4 算术基本定理拓展

N 的约数个数 $= (c_1 + 1) * (c_2 + 1) * \cdots * (c_n + 1)$

N 的约数之和 $= (1 + p_1 + p_1^2 + \cdots p_1^{c_1}) * (1 + p_2 + p_2^2 + \cdots p_2^{c_2}) * \cdots * (1 + p_n + p_n^2 + \cdots + p_n^{c_n})$

4.5.5 多项式性质

实数域不可拆分多项式只有两种：一次多项式和二次的 $b^2 < 4ac$

5 杂项

5.1 三分：$O(2log_3^n)$

```
int L, R;
while(L < R) {
    int mid = (L + R) / 2;
    if(check(mid) > check(mid + 1)) // 凹函数,凸函数相反
        L = mid + 1;
    else
        R = mid;
}
```

5.2 RMQ 问题：st 表

```
int a[MAXN];
int dp[MAXN][25];
void init_rmq(int n) {
    for(int i = 1; i <= n; i++)
        dp[i][0] = a[i];
    for(int j = 1; (1 << j) <= n; j++)
        for(int i = 1; i + (1 << j) - 1 <= n; i++)
            dp[i][j] = min(dp[i][j - 1], dp[i + (1 << j - 1)][j - 1]);
}
int rmq(int L, int R) {
    int k = log(R - L + 1) / log(2);
    return min(dp[L][k], dp[R - (1 << k) + 1][k]);
}
```

5.3 K-约瑟夫变换

对长度为 n 的序列，每次从当前位置开始取第 k 个数，将其移除放到新序列的尾部，然后从下一个位置开始继续循环操作，直到原序列为空。

如【1，2，3，4，5】进行 3-约瑟夫变换，得到【3，1，5，2，4】。

如何求变化序列？

设上一个被取出来的数字是当时的第 pos 个（初始设为 1 ），当前还剩下 cnt 个数字，那么下一个被选出来的数应该是当前剩下的所有数字中的第【$(pos + k - 2)\%cnt + 1$ 】个。

```
vi a(n + 1); // 树状数组
vi c(n + 1); // 变化后的序列
for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j += lowbit(j))
        a[j]++;

int pos = 1;
for(int i = n; i >= 1; i--) {
    pos = (pos + k - 2) % i + 1;
    int L = 1, R = n;
    while(L < R) {
        int mid = (L + R) / 2;
        int sum = 0;
        for(int j = mid; j >= 1; j -= lowbit(j))
            sum += a[j];
        if(sum < pos)
            L = mid + 1;
        else
            R = mid;
    }
    c[n - i + 1] = L;
    for(int j = L; j <= n; j += lowbit(j))
        a[j]--;
}
```

6 其它问题

6.1 IDE 配置

6.1.1 头文件

```
#include<bits/stdc++.h>
#define ll long long
#define INF 0x3f3f3f3f
#define LLINF 0x3f3f3f3f3f3f3f3f
#define pii pair<int,int>
#define vi vector<int>
```

```
#define SZ(x) (int)x.size()
#define pb push_back
#define mp make_pair
#define fi first
#define se second
using namespace std;
```

6.1.2 手动加栈

```
#define _CRT_SECURE_NO_WARNINGS
#pragma GCC optimize(3)
#pragma GCC optimize("Ofast")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx,tune=native")
#pragma comment(linker, "/stack:200000000")
```

6.2 STL 相关

6.2.1 数据类型范围

int 最大值：$2^{31}-1$

unsigned int 最大值：$2^{32}-1$

long long（__int64）最大值：$2^{63}-1$

unsigned long long 最大值：$2^{64}-1$

6.2.2 基础 STL 的使用

multiset ：

erase()：直接 erase(x) 会删除所有的 x，若要删除单个 x 应写作 erase(set.find(x))。

取 set 中的最大元素：int maxx = * – (Set.end())

map 和 unordered_map ：

map：

内部实现是红黑树（非严格平衡二叉搜索树），具有自动排序功能，可以从小到大遍历。

unordered_map：

内部实现是哈希表，内部元素排列是无序的，可以 $O(1)$ 查询，但哈希表的建立比较耗费时间。

atoi 函数 ：

```
string s = "123";
int x = atoi(s.c_str());
```

整型转字符串 ：

```
ostringstream ss;
ss << 123 << "66" << 'a';
string s = ss.str();
```

unique 函数 ：

```
sort(a.begin(), a.end());
auto p = unique(a.begin(), a.end());
a.erase(p, a.end());
```

next_permutation 函数 ：

```
while(next_permutation(s.begin(), s.end())) {} // string
while(next_permutation(a, a + n)) {} // 数组
while(next_permutation(a.begin(), a.end())) {} // vector
```

6.2.3 cin 加速

```
cin.tie(0);
cin.sync_with_stdio(0);
```

6.2.4 卡常读入优化

```
template<typename T>void read(T&x) {
    static char c;
    static int f;
    for(c = getchar(), f = 1; c < '0' || c > '9'; c = getchar())
        if(c == '-')
```

```
            f = -f;
    for(x = 0; c >= '0' && c <= '9'; c = getchar())
        x = x * 10 + (c & 15);
    x *= f;
}
template<typename T>void write(T x) {
    static char q[65];
    int cnt = 0;
    if(x < 0)
        putchar('-'), x = -x;
    q[++cnt] = x % 10, x /= 10;
    while(x)
        q[++cnt] = x % 10, x /= 10;
    while(cnt)
        putchar(q[cnt--] + '0');
}
```

6.2.5 交互题写法

```
int main() {
    int L = -1e9, R = 1e9;
    while(L < R) {
        int mid = (L + R) / 2;
        printf("%d\n", mid);
        fflush(stdout);
        string s;
        cin >> s;
        if(s == "equal")
            return 0;
        if(s == "small")
            L = mid + 1;
        else
            R = mid - 1;
    }
    printf("%d", L);
}
```

6.3 Tip

1. 构造题一定要注意和子结构之间的递推关系！
2. 能用数组解决绝不用 map。
3. 网格图网络流一般把行和列抽象为点。
4. 乘法的常见变形是把乘数拆成几部分。
5. 对网络流中不确定方向的边，可以先指定一个方向，用退流来模拟改变方向。