树上问题

树的直径

```
struct Tree {
    int n;
    vector<vector<int>> ver;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
    }
    void add(int x, int y) {
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    int getlen(int root) { // 获取x所在树的直径
        map<int, int> dep; // map用于优化输入为森林时的深度计算，亦可用vector
        function<void(int, int)> dfs = [&](int x, int fa) -> void {
            for (auto y : ver[x]) {
                if (y == fa) continue;
                dep[y] = dep[x] + 1;
                dfs(y, x);
            }
            if (dep[x] > dep[root]) {
                root = x;
            }
        };
        dfs(root, 0);
        int st = root; // 记录直径端点

        dep.clear();
        dfs(root, 0);
        int ed = root; // 记录直径另一端点

        return dep[root];
    }
};
```

树论大封装（直径+重心+中心）

```
struct Tree {
    int n;
    vector<vector<pair<int, int>>> e;
    vector<int> dep, parent, maxdep, d1, d2, s1, s2, up;
    Tree(int n) {
        this->n = n;
        e.resize(n + 1);
        dep.resize(n + 1);
        parent.resize(n + 1);
        maxdep.resize(n + 1);
```

```
        d1.resize(n + 1);
        d2.resize(n + 1);
        s1.resize(n + 1);
        s2.resize(n + 1);
        up.resize(n + 1);
    }
    void add(int u, int v, int w) {
        e[u].push_back({w, v});
        e[v].push_back({w, u});
    }
    void dfs(int u, int fa) {
        maxdep[u] = dep[u];
        for (auto [w, v] : e[u]) {
            if (v == fa) continue;
            dep[v] = dep[u] + 1;
            parent[v] = u;
            dfs(v, u);
            maxdep[u] = max(maxdep[u], maxdep[v]);
        }
    }

    void dfs1(int u, int fa) {
        for (auto [w, v] : e[u]) {
            if (v == fa) continue;
            dfs1(v, u);
            int x = d1[v] + w;
            if (x > d1[u]) {
                d2[u] = d1[u], s2[u] = s1[u];
                d1[u] = x, s1[u] = v;
            } else if (x > d2[u]) {
                d2[u] = x, s2[u] = v;
            }
        }
    }
    void dfs2(int u, int fa) {
        for (auto [w, v] : e[u]) {
            if (v == fa) continue;
            if (s1[u] == v) {
                up[v] = max(up[u], d2[u]) + w;
            } else {
                up[v] = max(up[u], d1[u]) + w;
            }
            dfs2(v, u);
        }
    }

    int radius, center, diam;
    void getCenter() {
        center = 1; //中心
        for (int i = 1; i <= n; i++) {
            if (max(d1[i], up[i]) < max(d1[center], up[center])) {
                center = i;
            }
```

```cpp
        }
        radius = max(d1[center], up[center]); //距离最远点的距离的最小值
        diam = d1[center] + up[center] + 1; //直径
    }

    int rem; //删除重心后剩余连通块体积的最小值
    int cog; //重心
    vector<bool> vis;
    void getCog() {
        vis.resize(n);
        rem = INT_MAX;
        cog = 1;
        dfsCog(1);
    }
    int dfsCog(int u) {
        vis[u] = true;
        int s = 1, res = 0;
        for (auto [w, v] : e[u]) {
            if (vis[v]) continue;
            int t = dfsCog(v);
            res = max(res, t);
            s += t;
        }
        res = max(res, n - s);
        if (res < rem) {
            rem = res;
            cog = u;
        }
        return s;
    }
};
```

点分治 / 树的重心

重心的定义：删除树上的某一个点，会得到若干棵子树；删除某点后，得到的最大子树最小，这个点称为重心。我们假设某个点是重心，记录此时最大子树的最小值，遍历完所有点后取最大值即可。

> 重心的性质：重心最多可能会有两个，且此时两个重心相邻。

点分治的一般过程是：取重心为新树的根，随后使用 `dfs` 处理当前这棵树，灵活运用 `child` 和 `pre` 两个数组分别计算通过根节点、不通过根节点的路径信息，根据需要进行答案的更新；再对子树分治，寻找子树的重心，……。时间复杂度降至 $\mathcal{O}(N\log N)$ 。

```cpp
int root = 0, MaxTree = 1e18; //分别代表重心下标、最大子树大小
vector<int> vis(n + 1), siz(n + 1);
auto get = [&](auto self, int x, int fa, int n) -> void { // 获取树的重心
    siz[x] = 1;
    int val = 0;
    for (auto [y, w] : ver[x]) {
        if (y == fa || vis[y]) continue;
        self(self, y, x, n);
        siz[x] += siz[y];
        val = max(val, siz[y]);
```

```
    }
    val = max(val, n - siz[x]);
    if (val < MaxTree) {
        MaxTree = val;
        root = x;
    }
};

auto clac = [&](int x) -> void { // 以 x 为新的根，维护询问
    set<int> pre = {0}; // 记录到根节点 x 距离为 i 的路径是否存在
    vector<int> dis(n + 1);
    for (auto [y, w] : ver[x]) {
        if (vis[y]) continue;
        vector<int> child; // 记录 x 的子树节点的深度信息
        auto dfs = [&](auto self, int x, int fa) -> void {
            child.push_back(dis[x]);
            for (auto [y, w] : ver[x]) {
                if (y == fa || vis[y]) continue;
                dis[y] = dis[x] + w;
                self(self, y, x);
            }
        };
        dis[y] = w;
        dfs(dfs, y, x);

        for (auto it : child) {
            for (int i = 1; i <= m; i++) { // 根据询问更新值
                if (q[i] < it || !pre.count(q[i] - it)) continue;
                ans[i] = 1;
            }
        }
        pre.insert(child.begin(), child.end());
    }
};

auto dfz = [&](auto self, int x, int fa) -> void { // 点分治
    vis[x] = 1; // 标记已经被更新过的旧重心，确保只对子树分治
    clac(x);
    for (auto [y, w] : ver[x]) {
        if (y == fa || vis[y]) continue;
        MaxTree = 1e18;
        get(get, y, x, siz[y]);
        self(self, root, x);
    }
};

get(get, 1, 0, n);
dfz(dfz, root, 0);
```

最近公共祖先 LCA

树链剖分解法

预处理时间复杂度 $\mathcal{O}(N)$ ；单次查询 $\mathcal{O}(\log N)$ ，常数较小。

```
struct HLD {
    int n, idx;
    vector<vector<int>> ver;
    vector<int> siz, dep;
    vector<int> top, son, parent;

    HLD(int n) {
        this->n = n;
        ver.resize(n + 1);
        siz.resize(n + 1);
        dep.resize(n + 1);

        top.resize(n + 1);
        son.resize(n + 1);
        parent.resize(n + 1);
    }
    void add(int x, int y) { // 建立双向边
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    void dfs1(int x) {
        siz[x] = 1;
        dep[x] = dep[parent[x]] + 1;
        for (auto y : ver[x]) {
            if (y == parent[x]) continue;
            parent[y] = x;
            dfs1(y);
            siz[x] += siz[y];
            if (siz[y] > siz[son[x]]) {
                son[x] = y;
            }
        }
    }
    void dfs2(int x, int up) {
        top[x] = up;
        if (son[x]) dfs2(son[x], up);
        for (auto y : ver[x]) {
            if (y == parent[x] || y == son[x]) continue;
            dfs2(y, y);
        }
    }
    int lca(int x, int y) {
        while (top[x] != top[y]) {
            if (dep[top[x]] > dep[top[y]]) {
                x = parent[top[x]];
            } else {
                y = parent[top[y]];
```

```
                }
            }
            return dep[x] < dep[y] ? x : y;
        }
        int clac(int x, int y) { // 查询两点间距离
            return dep[x] + dep[y] - 2 * dep[lca(x, y)];
        }
        void work(int root = 1) { // 在此初始化
            dfs1(root);
            dfs2(root, root);
        }
};
```

树上倍增解法

预处理时间复杂度 $\mathcal{O}(N\log N)$ ；单次查询 $\mathcal{O}(\log N)$ ，但是常数比树链剖分解法更大。

封装一：基础封装，针对无权图。

```
struct Tree {
    int n;
    vector<vector<int>> ver, val;
    vector<int> lg, dep;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
        val.resize(n + 1, vector<int>(30));
        lg.resize(n + 1);
        dep.resize(n + 1);
        for (int i = 1; i <= n; i++) { //预处理 log
            lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
        }
    }
    void add(int x, int y) { // 建立双向边
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    void dfs(int x, int fa) {
        val[x][0] = fa; // 储存 x 的父节点
        dep[x] = dep[fa] + 1;
        for (int i = 1; i <= lg[dep[x]]; i++) {
            val[x][i] = val[val[x][i - 1]][i - 1];
        }
        for (auto y : ver[x]) {
            if (y == fa) continue;
            dfs(y, x);
        }
    }
    int lca(int x, int y) {
        if (dep[x] < dep[y]) swap(x, y);
        while (dep[x] > dep[y]) {
            x = val[x][lg[dep[x] - dep[y]] - 1];
        }
```

```
        if (x == y) return x;
        for (int k = lg[dep[x]] - 1; k >= 0; k--) {
            if (val[x][k] == val[y][k]) continue;
            x = val[x][k];
            y = val[y][k];
        }
        return val[x][0];
    }
    int clac(int x, int y) { // 倍增查询两点间距离
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
    void work(int root = 1) { // 在此初始化
        dfs(root, 0);
    }
};
```

封装二：扩展封装，针对有权图，支持"倍增查询两点路径上的最大边权"功能。

```
struct Tree {
    int n;
    vector<vector<int>> val, Max;
    vector<vector<pair<int, int>>> ver;
    vector<int> lg, dep;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
        val.resize(n + 1, vector<int>(30));
        Max.resize(n + 1, vector<int>(30));
        lg.resize(n + 1);
        dep.resize(n + 1);
        for (int i = 1; i <= n; i++) { //预处理 log
            lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
        }
    }
    void add(int x, int y, int w) { // 建立双向边
        ver[x].push_back({y, w});
        ver[y].push_back({x, w});
    }
    void dfs(int x, int fa) {
        val[x][0] = fa;
        dep[x] = dep[fa] + 1;
        for (int i = 1; i <= lg[dep[x]]; i++) {
            val[x][i] = val[val[x][i - 1]][i - 1];
            Max[x][i] = max(Max[x][i - 1], Max[val[x][i - 1]][i - 1]);
        }
        for (auto [y, w] : ver[x]) {
            if (y == fa) continue;
            Max[y][0] = w;
            dfs(y, x);
        }
    }
    int lca(int x, int y) {
        if (dep[x] < dep[y]) swap(x, y);
```

```cpp
        while (dep[x] > dep[y]) {
            x = val[x][lg[dep[x] - dep[y]] - 1];
        }
        if (x == y) return x;
        for (int k = lg[dep[x]] - 1; k >= 0; k--) {
            if (val[x][k] == val[y][k]) continue;
            x = val[x][k];
            y = val[y][k];
        }
        return val[x][0];
    }
    int clac(int x, int y) { // 倍增查询两点间距离
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
    int query(int x, int y) { // 倍增查询两点路径上的最大边权（带权图）
        auto get = [&](int x, int y) -> int {
            int ans = 0;
            if (x == y) return ans;
            for (int i = lg[dep[x]]; i >= 0; i--) {
                if (dep[val[x][i]] > dep[y]) {
                    ans = max(ans, Max[x][i]);
                    x = val[x][i];
                }
            }
            ans = max(ans, Max[x][0]);
            return ans;
        };
        int fa = lca(x, y);
        return max(get(x, fa), get(y, fa));
    }
    void work(int root = 1) { // 在此初始化
        dfs(root, 0);
    }
};
```

树上路径交

计算两条路径的交点数量，直接载入任意 LCA 封装即可。

```cpp
int intersection(int x, int y, int X, int Y) {
    vector<int> t = {lca(x, X), lca(x, Y), lca(y, X), lca(y, Y)};
    sort(t.begin(), t.end());
    int r = lca(x, y), R = lca(X, Y);
    if (dep[t[0]] < min(dep[r], dep[R]) || dep[t[2]] < max(dep[r], dep[R])) {
        return 0;
    }
    return 1 + clac(t[2], t[3]);
}
```

树上启发式合并 (DSU on tree)

$\mathcal{O}(N\log N)$。

```
struct HLD {
    vector<vector<int>> e;
    vector<int> siz, son, cnt;
    vector<LL> ans;
    LL sum, Max;
    int hson;
    HLD(int n) {
        e.resize(n + 1);
        siz.resize(n + 1);
        son.resize(n + 1);
        ans.resize(n + 1);
        cnt.resize(n + 1);
        hson = 0;
        sum = 0;
        Max = 0;
    }
    void add(int u, int v) {
        e[u].push_back(v);
        e[v].push_back(u);
    }
    void dfs1(int u, int fa) {
        siz[u] = 1;
        for (auto v : e[u]) {
            if (v == fa) continue;
            dfs1(v, u);
            siz[u] += siz[v];
            if (siz[v] > siz[son[u]]) son[u] = v;
        }
    }
    void calc(int u, int fa, int val) {
        cnt[color[u]] += val;
        if (cnt[color[u]] > Max) {
            Max = cnt[color[u]];
            sum = color[u];
        } else if (cnt[color[u]] == Max) {
            sum += color[u];
        }
        for (auto v : e[u]) {
            if (v == fa || v == hson) continue;
            calc(v, u, val);
        }
    }
    void dfs2(int u, int fa, int opt) {
        for (auto v : e[u]) {
            if (v == fa || v == son[u]) continue;
            dfs2(v, u, 0);
        }
        if (son[u]) {
            dfs2(son[u], u, 1);
```

```cpp
            hson = son[u]; //记录重链编号，计算的时候跳过
        }
        calc(u, fa, 1);
        hson = 0; //消除的时候所有儿子都清除
        ans[u] = sum;
        if (!opt) {
            calc(u, fa, -1);
            sum = 0;
            Max = 0;
        }
    }
};
```

prufur 序列

对树建立 Prüfer 序列

Prüfer 是这样建立的：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。

显然使用堆可以做到 $O(n\log n)$ 的复杂度

```cpp
// 代码摘自原文，结点是从 0 标号的
vector<vector<int>> adj;

vector<int> pruefer_code() {
  int n = adj.size();
  set<int> leafs;
  vector<int> degree(n);
  vector<bool> killed(n, false);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1) leafs.insert(i);
  }

  vector<int> code(n - 2);
  for (int i = 0; i < n - 2; i++) {
    int leaf = *leafs.begin();
    leafs.erase(leafs.begin());
    killed[leaf] = true;
    int v;
    for (int u : adj[leaf])
      if (!killed[u]) v = u;
    code[i] = v;
    if (--degree[v] == 1) leafs.insert(v);
  }
  return code;
}
```

```python
# 结点是从 0 标号的
adj = [[]]

```

```python

def pruefer_code():
    n = len(adj)
    leafs = set()
    degree = [0] * n
    killed = [False] * n
    for i in range(1, n):
        degree[i] = len(adj[i])
        if degree[i] == 1:
            leafs.intersection(i)
    code = [0] * (n - 2)
    for i in range(1, n - 2):
        leaf = leafs[0]
        leafs.pop()
        killed[leaf] = True
        for u in adj[leaf]:
            if killed[u] == False:
                v = u
        code[i] = v
        if degree[v] == 1:
            degree[v] = degree[v] - 1
            leafs.intersection(v)
    return code
```

Cayley 公式 (Cayley's formula)

完全图 K_n 有 n^{n-2} 棵生成树。

怎么证明？方法很多，但是用 Prüfer 序列证是很简单的。任意一个长度为 $n-2$ 的值域 $[1,n]$ 的整数序列都可以通过 Prüfer 序列双射对应一个生成树，于是方案数就是 n^{n-2}。

图连通方案数

Prüfer 序列可能比你想得还强大。它能创造比 凯莱公式 更通用的公式。比如以下问题：

> 一个 n 个点 m 条边的带标号无向图有 k 个连通块。我们希望添加 $k-1$ 条边使得整个图连通。求方案数。

设 s_i 表示每个连通块的数量。我们对 k 个连通块构造 Prüfer 序列，然后你发现这并不是普通的 Prüfer 序列。因为每个连通块的连接方法很多。不能直接淦就设啊。于是设 d_i 为第 i 个连通块的度数。由于度数之和是边数的两倍，于是 $\sum_{i=1}^{k} d_i = 2k-2$。则对于给定的 d 序列构造 Prüfer 序列的方案数是

$$n^{k-2} \cdot \prod_{i=1}^{k} s_i$$

重链剖分

```cpp
struct HPD_tree
{
    int tree_size;
    bool is_hpd_init = false;
    std::vector<std::vector<std::pair<int, i64>>> adj;
    std::vector<int> Fa, size, hson, top, rank, dfn, depth;
    HPD_tree(int n = 0) {
```

```
        tree_size = n;
        adj.resize(tree_size + 1);
    }
    void add_edge(int u, int v, i64 w = 1) {
        adj[u].push_back({ v,w });
        adj[v].push_back({ u,w });
    }
    void HPD_init() {
        is_hpd_init = true;
        Fa.assign(tree_size + 1, 0);
        size.assign(tree_size + 1, 0);
        hson.assign(tree_size + 1, 0);
        top.assign(tree_size + 1, 0);
        rank.assign(tree_size + 1, 0);
        dfn.assign(tree_size + 1, 0);
        depth.assign(tree_size + 1, 0);
        std::function<void(int, int, int)> dfs1 = [&](int u, int p, int d)->void {
            hson[u] = 0;
            size[hson[u]] = 0;
            size[u] = 1;
            depth[u] = d;
            for (auto [v, w] : adj[u])if (v != p) {
                dfs1(v, u, d + 1);
                size[u] += size[v];
                Fa[v] = u;
                if (size[v] > size[hson[u]]) {
                    hson[u] = v;
                }
            }
            };
        dfs1(1, 0, 0);
        int tot = 0;
        std::function<void(int, int, int)> dfs2 = [&](int u, int p, int t)->void {
            top[u] = t;
            dfn[u] = ++tot;
            rank[tot] = u;
            if (hson[u]) {
                dfs2(hson[u], u, t);
                for (auto [v, w] : adj[u])if (v != p && v != hson[u]) {
                    dfs2(v, u, v);
                }
            }
            };
        dfs2(1, 0, 1);
    }
    int lca(int u, int v) {
        if (!is_hpd_init)HPD_init();
        while (top[u] != top[v]) {
            if (depth[top[u]] > depth[top[v]])
                u = Fa[top[u]];
            else
                v = Fa[top[v]];
        }
```

```cpp
            return depth[u] > depth[v] ? v : u;
        }
        i64 dist(int u, int v) {
            int w = lca(u, v);
            return depth[u] - depth[w] + depth[v] - depth[w] + 1;
        }
        a3 get_diam() {
            i64 cur; int pos;
            std::function<void(int, int, i64)> dfs = [&](int u, int p, i64 d) {
                if (d > cur) {
                    cur = d;
                    pos = u;
                }
                for (auto [v, dis] : adj[u])if (v != p) {
                    dfs(v, u, d + dis);
                }
                };
            cur = 0, pos = 1;
            dfs(pos, 0, cur);
            int u = pos;
            cur = 0;
            dfs(pos, 0, cur);
            int v = pos;
            return { u,v,cur };
        }
    };
```

/END/