

## 树上问题

### 树的直径

```

1  struct Tree {
2      int n;
3      vector<vector<int>> ver;
4      Tree(int n) {
5          this->n = n;
6          ver.resize(n + 1);
7      }
8      void add(int x, int y) {
9          ver[x].push_back(y);
10         ver[y].push_back(x);
11     }
12     int getlen(int root) { // 获取x所在树的直径
13         map<int, int> dep; // map用于优化输入为森林时的深度计算，亦可用vector
14         function<void(int, int)> dfs = [&](int x, int fa) -> void {
15             for (auto y : ver[x]) {
16                 if (y == fa) continue;
17                 dep[y] = dep[x] + 1;
18                 dfs(y, x);
19             }
20             if (dep[x] > dep[root]) {
21                 root = x;
22             }
23         };
24         dfs(root, 0);
25         int st = root; // 记录直径端点
26
27         dep.clear();
28         dfs(root, 0);
29         int ed = root; // 记录直径另一端点
30
31         return dep[root];
32     }
33 };

```

### 树论大封装（直径+重心+中心）

```

1  struct Tree {
2      int n;
3      vector<vector<pair<int, int>>> e;
4      vector<int> dep, parent, maxdep, d1, d2, s1, s2, up;
5      Tree(int n) {
6          this->n = n;
7          e.resize(n + 1);
8          dep.resize(n + 1);
9          parent.resize(n + 1);
10         maxdep.resize(n + 1);

```

```

11     d1.resize(n + 1);
12     d2.resize(n + 1);
13     s1.resize(n + 1);
14     s2.resize(n + 1);
15     up.resize(n + 1);
16 }
17 void add(int u, int v, int w) {
18     e[u].push_back({w, v});
19     e[v].push_back({w, u});
20 }
21 void dfs(int u, int fa) {
22     maxdep[u] = dep[u];
23     for (auto [w, v] : e[u]) {
24         if (v == fa) continue;
25         dep[v] = dep[u] + 1;
26         parent[v] = u;
27         dfs(v, u);
28         maxdep[u] = max(maxdep[u], maxdep[v]);
29     }
30 }
31
32 void dfs1(int u, int fa) {
33     for (auto [w, v] : e[u]) {
34         if (v == fa) continue;
35         dfs1(v, u);
36         int x = d1[v] + w;
37         if (x > d1[u]) {
38             d2[u] = d1[u], s2[u] = s1[u];
39             d1[u] = x, s1[u] = v;
40         } else if (x > d2[u]) {
41             d2[u] = x, s2[u] = v;
42         }
43     }
44 }
45 void dfs2(int u, int fa) {
46     for (auto [w, v] : e[u]) {
47         if (v == fa) continue;
48         if (s1[u] == v) {
49             up[v] = max(up[u], d2[u]) + w;
50         } else {
51             up[v] = max(up[u], d1[u]) + w;
52         }
53         dfs2(v, u);
54     }
55 }
56
57 int radius, center, diam;
58 void getCenter() {
59     center = 1; //中心
60     for (int i = 1; i <= n; i++) {
61         if (max(d1[i], up[i]) < max(d1[center], up[center])) {
62             center = i;
63         }

```

```

64     }
65     radius = max(d1[center], up[center]); //距离最远点的距离的最小值
66     diam = d1[center] + up[center] + 1; //直径
67 }
68
69 int rem; //删除重心后剩余连通块体积的最小值
70 int cog; //重心
71 vector<bool> vis;
72 void getCog() {
73     vis.resize(n);
74     rem = INT_MAX;
75     cog = 1;
76     dfsCog(1);
77 }
78 int dfsCog(int u) {
79     vis[u] = true;
80     int s = 1, res = 0;
81     for (auto [w, v] : e[u]) {
82         if (vis[v]) continue;
83         int t = dfsCog(v);
84         res = max(res, t);
85         s += t;
86     }
87     res = max(res, n - s);
88     if (res < rem) {
89         rem = res;
90         cog = u;
91     }
92     return s;
93 }
94 };

```

## 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  $\mathcal{O}(N \log N)$ 。

```

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

```

```

11     }
12     val = max(val, n - siz[x]);
13     if (val < MaxTree) {
14         MaxTree = val;
15         root = x;
16     }
17 };
18
19 auto clac = [&](int x) -> void { // 以 x 为新的根, 维护询问
20     set<int> pre = {0}; // 记录到根节点 x 距离为 i 的路径是否存在
21     vector<int> dis(n + 1);
22     for (auto [y, w] : ver[x]) {
23         if (vis[y]) continue;
24         vector<int> child; // 记录 x 的子树节点的深度信息
25         auto dfs = [&](auto self, int x, int fa) -> void {
26             child.push_back(dis[x]);
27             for (auto [y, w] : ver[x]) {
28                 if (y == fa || vis[y]) continue;
29                 dis[y] = dis[x] + w;
30                 self(self, y, x);
31             }
32         };
33         dis[y] = w;
34         dfs(dfs, y, x);
35
36         for (auto it : child) {
37             for (int i = 1; i <= m; i++) { // 根据询问更新值
38                 if (q[i] < it || !pre.count(q[i] - it)) continue;
39                 ans[i] = 1;
40             }
41         }
42         pre.insert(child.begin(), child.end());
43     }
44 };
45
46 auto dfz = [&](auto self, int x, int fa) -> void { // 点分治
47     vis[x] = 1; // 标记已经被更新过的旧重心, 确保只对子树分治
48     clac(x);
49     for (auto [y, w] : ver[x]) {
50         if (y == fa || vis[y]) continue;
51         MaxTree = 1e18;
52         get(get, y, x, siz[y]);
53         self(self, root, x);
54     }
55 };
56
57 get(get, 1, 0, n);
58 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。

```

1  struct HLD {
2      int n, idx;
3      vector<vector<int>> ver;
4      vector<int> siz, dep;
5      vector<int> top, son, parent;
6
7      HLD(int n) {
8          this->n = n;
9          ver.resize(n + 1);
10         siz.resize(n + 1);
11         dep.resize(n + 1);
12
13         top.resize(n + 1);
14         son.resize(n + 1);
15         parent.resize(n + 1);
16     }
17     void add(int x, int y) { // 建立双向边
18         ver[x].push_back(y);
19         ver[y].push_back(x);
20     }
21     void dfs1(int x) {
22         siz[x] = 1;
23         dep[x] = dep[parent[x]] + 1;
24         for (auto y : ver[x]) {
25             if (y == parent[x]) continue;
26             parent[y] = x;
27             dfs1(y);
28             siz[x] += siz[y];
29             if (siz[y] > siz[son[x]]) {
30                 son[x] = y;
31             }
32         }
33     }
34     void dfs2(int x, int up) {
35         top[x] = up;
36         if (son[x]) dfs2(son[x], up);
37         for (auto y : ver[x]) {
38             if (y == parent[x] || y == son[x]) continue;
39             dfs2(y, y);
40         }
41     }
42     int lca(int x, int y) {
43         while (top[x] != top[y]) {
44             if (dep[top[x]] > dep[top[y]]) {
45                 x = parent[top[x]];
46             } else {
47                 y = parent[top[y]];

```

```

48     }
49     }
50     return dep[x] < dep[y] ? x : y;
51 }
52 int clac(int x, int y) { // 查询两点间距离
53     return dep[x] + dep[y] - 2 * dep[lca(x, y)];
54 }
55 void work(int root = 1) { // 在此初始化
56     dfs1(root);
57     dfs2(root, root);
58 }
59 };

```

## 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。**

```

1  struct Tree {
2      int n;
3      vector<vector<int>> ver, val;
4      vector<int> lg, dep;
5      Tree(int n) {
6          this->n = n;
7          ver.resize(n + 1);
8          val.resize(n + 1, vector<int>(30));
9          lg.resize(n + 1);
10         dep.resize(n + 1);
11         for (int i = 1; i <= n; i++) { //预处理 log
12             lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
13         }
14     }
15     void add(int x, int y) { // 建立双向边
16         ver[x].push_back(y);
17         ver[y].push_back(x);
18     }
19     void dfs(int x, int fa) {
20         val[x][0] = fa; // 储存 x 的父节点
21         dep[x] = dep[fa] + 1;
22         for (int i = 1; i <= lg[dep[x]]; i++) {
23             val[x][i] = val[val[x][i - 1]][i - 1];
24         }
25         for (auto y : ver[x]) {
26             if (y == fa) continue;
27             dfs(y, x);
28         }
29     }
30     int lca(int x, int y) {
31         if (dep[x] < dep[y]) swap(x, y);
32         while (dep[x] > dep[y]) {
33             x = val[x][lg[dep[x]] - dep[y] - 1];
34         }

```

```

35     if (x == y) return x;
36     for (int k = lg[dep[x]] - 1; k >= 0; k--) {
37         if (val[x][k] == val[y][k]) continue;
38         x = val[x][k];
39         y = val[y][k];
40     }
41     return val[x][0];
42 }
43 int clac(int x, int y) { // 倍增查询两点间距离
44     return dep[x] + dep[y] - 2 * dep[lca(x, y)];
45 }
46 void work(int root = 1) { // 在此初始化
47     dfs(root, 0);
48 }
49 };

```

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。

```

1  struct Tree {
2      int n;
3      vector<vector<int>> val, Max;
4      vector<vector<pair<int, int>>> ver;
5      vector<int> lg, dep;
6      Tree(int n) {
7          this->n = n;
8          ver.resize(n + 1);
9          val.resize(n + 1, vector<int>(30));
10         Max.resize(n + 1, vector<int>(30));
11         lg.resize(n + 1);
12         dep.resize(n + 1);
13         for (int i = 1; i <= n; i++) { //预处理 log
14             lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
15         }
16     }
17     void add(int x, int y, int w) { // 建立双向边
18         ver[x].push_back({y, w});
19         ver[y].push_back({x, w});
20     }
21     void dfs(int x, int fa) {
22         val[x][0] = fa;
23         dep[x] = dep[fa] + 1;
24         for (int i = 1; i <= lg[dep[x]]; i++) {
25             val[x][i] = val[val[x][i - 1]][i - 1];
26             Max[x][i] = max(Max[x][i - 1], Max[val[x][i - 1]][i - 1]);
27         }
28         for (auto [y, w] : ver[x]) {
29             if (y == fa) continue;
30             Max[y][0] = w;
31             dfs(y, x);
32         }
33     }
34     int lca(int x, int y) {
35         if (dep[x] < dep[y]) swap(x, y);

```

```

36     while (dep[x] > dep[y]) {
37         x = val[x][lg[dep[x] - dep[y]] - 1];
38     }
39     if (x == y) return x;
40     for (int k = lg[dep[x]] - 1; k >= 0; k--) {
41         if (val[x][k] == val[y][k]) continue;
42         x = val[x][k];
43         y = val[y][k];
44     }
45     return val[x][0];
46 }
47 int clac(int x, int y) { // 倍增查询两点间距离
48     return dep[x] + dep[y] - 2 * dep[lca(x, y)];
49 }
50 int query(int x, int y) { // 倍增查询两点路径上的最大边权（带权图）
51     auto get = [&](int x, int y) -> int {
52         int ans = 0;
53         if (x == y) return ans;
54         for (int i = lg[dep[x]]; i >= 0; i--) {
55             if (dep[val[x][i]] > dep[y]) {
56                 ans = max(ans, Max[x][i]);
57                 x = val[x][i];
58             }
59         }
60         ans = max(ans, Max[x][0]);
61         return ans;
62     };
63     int fa = lca(x, y);
64     return max(get(x, fa), get(y, fa));
65 }
66 void work(int root = 1) { // 在此初始化
67     dfs(root, 0);
68 }
69 };

```

## 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。

```

1  int intersection(int x, int y, int X, int Y) {
2      vector<int> t = {lca(x, X), lca(x, Y), lca(y, X), lca(y, Y)};
3      sort(t.begin(), t.end());
4      int r = lca(x, y), R = lca(X, Y);
5      if (dep[t[0]] < min(dep[r], dep[R]) || dep[t[2]] < max(dep[r], dep[R])) {
6          return 0;
7      }
8      return 1 + clac(t[2], t[3]);
9  }

```



## 树上启发式合并 (DSU on tree)

$\mathcal{O}(N \log N)$ 。

```

1  struct HLD {
2      vector<vector<int>> e;
3      vector<int> siz, son, cnt;
4      vector<LL> ans;
5      LL sum, Max;
6      int hson;
7      HLD(int n) {
8          e.resize(n + 1);
9          siz.resize(n + 1);
10         son.resize(n + 1);
11         ans.resize(n + 1);
12         cnt.resize(n + 1);
13         hson = 0;
14         sum = 0;
15         Max = 0;
16     }
17     void add(int u, int v) {
18         e[u].push_back(v);
19         e[v].push_back(u);
20     }
21     void dfs1(int u, int fa) {
22         siz[u] = 1;
23         for (auto v : e[u]) {
24             if (v == fa) continue;
25             dfs1(v, u);
26             siz[u] += siz[v];
27             if (siz[v] > siz[son[u]]) son[u] = v;
28         }
29     }
30     void calc(int u, int fa, int val) {
31         cnt[color[u]] += val;
32         if (cnt[color[u]] > Max) {
33             Max = cnt[color[u]];
34             sum = color[u];
35         } else if (cnt[color[u]] == Max) {
36             sum += color[u];
37         }
38         for (auto v : e[u]) {
39             if (v == fa || v == hson) continue;
40             calc(v, u, val);
41         }
42     }
43     void dfs2(int u, int fa, int opt) {
44         for (auto v : e[u]) {
45             if (v == fa || v == son[u]) continue;
46             dfs2(v, u, 0);
47         }
48         if (son[u]) {
49             dfs2(son[u], u, 1);

```

```

50         hson = son[u]; //记录重链编号，计算的时候跳过
51     }
52     calc(u, fa, 1);
53     hson = 0; //消除的时候所有儿子都清除
54     ans[u] = sum;
55     if (!opt) {
56         calc(u, fa, -1);
57         sum = 0;
58         Max = 0;
59     }
60 }
61 };

```

## 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

```

1 // 代码摘自原文，结点是从 0 标号的
2 vector<vector<int>> adj;
3
4 vector<int> pruefer_code() {
5     int n = adj.size();
6     set<int> leafs;
7     vector<int> degree(n);
8     vector<bool> killed(n, false);
9     for (int i = 0; i < n; i++) {
10         degree[i] = adj[i].size();
11         if (degree[i] == 1) leafs.insert(i);
12     }
13
14     vector<int> code(n - 2);
15     for (int i = 0; i < n - 2; i++) {
16         int leaf = *leafs.begin();
17         leafs.erase(leafs.begin());
18         killed[leaf] = true;
19         int v;
20         for (int u : adj[leaf])
21             if (!killed[u]) v = u;
22         code[i] = v;
23         if (--degree[v] == 1) leafs.insert(v);
24     }
25     return code;
26 }

```

```

1 # 结点是从 0 标号的
2 adj = [[]]
3

```

```

4
5 def pruefer_code():
6     n = len(adj)
7     leafs = set()
8     degree = [0] * n
9     killed = [False] * n
10    for i in range(1, n):
11        degree[i] = len(adj[i])
12        if degree[i] == 1:
13            leafs.intersection(i)
14    code = [0] * (n - 2)
15    for i in range(1, n - 2):
16        leaf = leafs[0]
17        leafs.pop()
18        killed[leaf] = True
19        for u in adj[leaf]:
20            if killed[u] == False:
21                v = u
22        code[i] = v
23        if degree[v] == 1:
24            degree[v] = degree[v] - 1
25            leafs.intersection(v)
26    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

```

1 struct HPD_tree
2 {
3     int tree_size;
4     bool is_hpd_init = false;
5     std::vector<std::vector<std::pair<int, i64>>> adj;
6     std::vector<int> Fa, size, hson, top, rank, dfn, depth;
7     HPD_tree(int n = 0) {

```

```

8     tree_size = n;
9     adj.resize(tree_size + 1);
10 }
11 void add_edge(int u, int v, i64 w = 1) {
12     adj[u].push_back({ v,w });
13     adj[v].push_back({ u,w });
14 }
15 void HPD_init() {
16     is_hpd_init = true;
17     Fa.assign(tree_size + 1, 0);
18     size.assign(tree_size + 1, 0);
19     hson.assign(tree_size + 1, 0);
20     top.assign(tree_size + 1, 0);
21     rank.assign(tree_size + 1, 0);
22     dfn.assign(tree_size + 1, 0);
23     depth.assign(tree_size + 1, 0);
24     std::function<void(int, int, int)> dfs1 = [&](int u, int p, int d)->void {
25         hson[u] = 0;
26         size[hson[u]] = 0;
27         size[u] = 1;
28         depth[u] = d;
29         for (auto [v, w] : adj[u]) if (v != p) {
30             dfs1(v, u, d + 1);
31             size[u] += size[v];
32             Fa[v] = u;
33             if (size[v] > size[hson[u]]) {
34                 hson[u] = v;
35             }
36         }
37     };
38     dfs1(1, 0, 0);
39     int tot = 0;
40     std::function<void(int, int, int)> dfs2 = [&](int u, int p, int t)->void {
41         top[u] = t;
42         dfn[u] = ++tot;
43         rank[tot] = u;
44         if (hson[u]) {
45             dfs2(hson[u], u, t);
46             for (auto [v, w] : adj[u]) if (v != p && v != hson[u]) {
47                 dfs2(v, u, v);
48             }
49         }
50     };
51     dfs2(1, 0, 1);
52 }
53 int lca(int u, int v) {
54     if (!is_hpd_init) HPD_init();
55     while (top[u] != top[v]) {
56         if (depth[top[u]] > depth[top[v]])
57             u = Fa[top[u]];
58         else
59             v = Fa[top[v]];
60     }

```

```

61         return depth[u] > depth[v] ? v : u;
62     }
63     i64 dist(int u, int v) {
64         int w = lca(u, v);
65         return depth[u] - depth[w] + depth[v] - depth[w] + 1;
66     }
67     a3 get_diam() {
68         i64 cur; int pos;
69         std::function<void(int, int, i64)> dfs = [&](int u, int p, i64 d) {
70             if (d > cur) {
71                 cur = d;
72                 pos = u;
73             }
74             for (auto [v, dis] : adj[u]) if (v != p) {
75                 dfs(v, u, d + dis);
76             }
77         };
78         cur = 0, pos = 1;
79         dfs(pos, 0, cur);
80         int u = pos;
81         cur = 0;
82         dfs(pos, 0, cur);
83         int v = pos;
84         return { u, v, cur };
85     }
86 };

```

/END/