倍增 LCA 与树链剖分

树上二进制快跳求共同祖先 + 重链剖分把树压成线段

HGNU-ACM 实验室算法组

承接第 5 章线段树：手算倍增表与两步跳 + 重链两遍 DFS + 2 道真题精讲





本章目录

倍增法求最近公共祖先（LCA）

树链剖分（HLD）：把树路径压成连续区间

第 1 节用二进制倍增把树上“找共同祖先”和“算两点距离”降到 $O(\log n)$；第 2 节用轻重链剖分把整棵树拆成 $O(\log n)$ 段连续区间，交给第 5 章学过的线段树维护。

本章目录（续）

两大树上高级结构真题精讲

常见错误、检查清单与练习安排

第 3 节逐题精讲《点的距离》与《树的统计》，完整演示读题、建模、手推样例与分步代码；
第 4 节汇总高频错误、提交检查清单与第 7 章《平衡树 Treap》预告。

承接第 5 章：当数据从一维数组变成“一棵树”

第 5 章回顾 vs 本章新问题

第 5 章我们用树状数组和线段树在 $O(\log n)$ 时间内解决了一维数组的区间求和与最值。
但如果把数据换成一张家谱树或公司汇报树（有根树，共 n 个点、$n-1$ 条边），怎么快速回答下面两个问题？

- 问题 **A**：员工 u 和员工 v 想找一位共同的领导来拍板，而且级别越低越好（离两人最近），这位领导是谁？
- 问题 **B**：如果树上每条边都有路程长度，从点 u 走到点 v 的唯一路径总长度是多少？

本页要点

一维线段树只认识连续下标；面对树形分支结构，我们需要先学会“快速找共同祖先”。

暴力做法怎么写？为什么在 $n = 10^5$ 时会 TLE？

最直觉的“一步一步爬父亲”

先记录每个点到根的深度 `dep[u]` 和直接父节点 `fa[u]`：

1. 如果 `u` 比 `v` 深，就让 `u = fa[u]` 每次往上爬 1 步，直到两人一样深；
2. 然后 `u = fa[u]`、`v = fa[v]` 同步每次爬 1 步，直到 `u == v` 相遇。

- 最坏情况（单链退化）：如果树长成一条从 $1 \to 2 \to \cdots \to 10^5$ 的长竹竿，每次查询最深叶子和根附近节点，单次要爬 $\approx 10^5$ 步。
- 操作次数估算：$q = 10^5$ 次询问 $\times 10^5$ 步 $= 10^{10}$ 次操作，远超 1 秒限制（约 10^8 次），必报 **TLE**！

本页要点

暴力慢在“每次只敢迈 1 步”；要想快，必须学会一次跨 2^k 步的“倍增跳跃”。

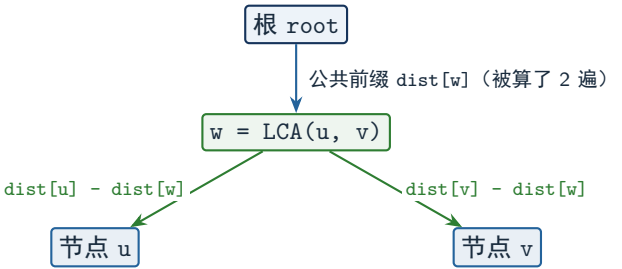
先听懂四个新术语（大白话版）

术语	大白话解释
深度 `dep[u]`	从根节点走到点 u 经过的边数（约定根的深度为 0 或 1）。
祖先（Ancestor）	从点 u 一路往根走，沿途经过的所有点（包含 u 自己和根）。
最近公共祖先 LCA	同时是 u 和 v 的祖先中，深度最大（离两人最近）的那一个节点。
根到点距离 `dist[u]`	从根节点沿树边走到点 u 的边权总和；无权树时 `dist[u]` = `dep[u]`。

本页要点

树上任意两点的简单路径有且仅有一条，且一定在它们的 LCA 处拐弯！

TikZ 图解：树上两点距离为何等于 dist[u]+dist[v]-2*dist[lca]



- 把“从 u 走到 w”与“从 w 走到 v”拼起来：$(\texttt{dist[u]} - \texttt{dist[w]}) + (\texttt{dist[v]} - \texttt{dist[w]})$。
- 化简后就是：`dist(u, v) = dist[u] + dist[v] - 2 * dist[w]`。

本页要点

只要预处理出每个点到根的距离 `dist[]`，求树上距离就完全等价于求 `LCA(u, v)`！

倍增核心思想：为什么跳 $1, 2, 4, 8, \ldots$ 步能拼出任意高度？

生活类比：用 $1, 2, 4, 8$ 公斤的砝码称重

任何正整数都能写成二进制形式，例如要往上爬 13 步：

$$13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$$

不需要一步步爬 13 次，只需依次跳 8 步、4 步、1 步，共 3 次就到了！

- 对于 $n \le 10^5$ 个节点，树高最多 $10^5 < 2^{17} = 131072$。
- 所以每个点最多只需要记 $K = 18$ 个跳跃终点（跳 $2^0, 2^1, \ldots, 2^{17}$ 步到哪），就能覆盖整棵树！

本页要点

用 $O(n \log n)$ 的空间提前记好“2^j 步跳到谁”，把每次询问从 $O(n)$ 压成 $O(\log n)$。

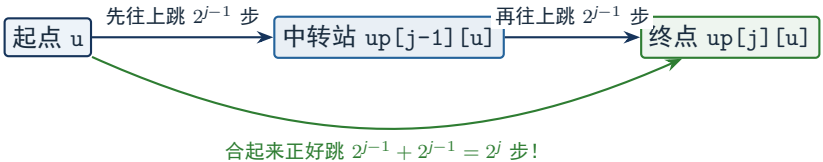
为什么用二进制倍增，而不用三进制或十进制？

跳跃进制	递推关系	为什么不如二进制方便？
二进制（2^j）	`up[j][u] = up[j-1][up[j-1][u]]`	只跳 1 次中转，且用 `(d >> j) & 1` 就能拆位
三进制（3^j）	需跳两次中转，且每位有 $0, 1, 2$ 三种状态	递推变复杂，每位最多要跳 2 次
固定步长（如 100）	先跳 100 步再跳 1 步（分块）	遇到 $n = 10^5$ 仍需 $\sqrt{n} \approx 316$ 步，慢于 17 步

本页要点

二进制既让递推公式最简单（$2^{j-1} + 2^{j-1} = 2^j$），又让每位只有“跳/不跳”两种选择！

TikZ 图解：up[j][u] 的含义与倍增递推公式



- 状态定义：up[j][u] 表示从节点 u 往上跳 2^j 步到达的祖先；若跳出根节点上方，约定为 0。
- 转移方程：up[j][u] = up[j - 1][up[j - 1][u]]（约定 up[j][0] = 0）。

本页要点

计算第 j 层时必须保证所有点的第 j-1 层都算完了，因此外层循环必须是 j！

手算跟踪表：5 个点链式树的 up[j][u] 预处理过程

树结构：根为 1，边为 1-2，2-3，3-4，4-5（一条 5 点链，dep[1]=0..dep[5]=4），取 $K = 3$。

轮次	含义	u=1	u=2	u=3	u=4	u=5
j=0	跳 1 步（直接父亲）	0	1	2	3	4
j=1	up[0][up[0][u]]（2 步）	0	0	up[0][2]=1	up[0][3]=2	up[0][4]=3
j=2	up[1][up[1][u]]（4 步）	0	0	up[1][1]=0	up[1][2]=0	up[1][3]=1
终止	j=3 超出 $K = 3$	预处理结束，耗时 $O(n \log n)$				

本页要点

因为约定 up[*][0] = 0，所以跳出根节点上方会自动停在 0，不会数组越界！

手算跟踪表（二）：6 点分叉树的 up[j][u] 预处理

分叉树：根 1，边 1-2，1-3，2-4，2-5，3-6（dep[1]=0，dep[2..3]=1，dep[4..6]=2），取 $K = 3$。

轮次	u=1	u=2	u=3	u=4	u=5	u=6
j=0（1 步）	0	1	1	2	2	3
j=1（2 步）	0	up[0][1]=0	up[0][1]=0	up[0][2]=1	up[0][2]=1	up[0][3]=1
j=2（4 步）	0	0	0	up[1][1]=0	up[1][1]=0	up[1][1]=0
终止	所有节点最多 2 步到根，因此 j=2 全为 0，查表准备就绪					

本页要点

分叉树上不同分支的节点（如 5 和 6）在 j=1 时就已经共同指向祖先 1！

细节追问：带边权树与 long long 溢出边界

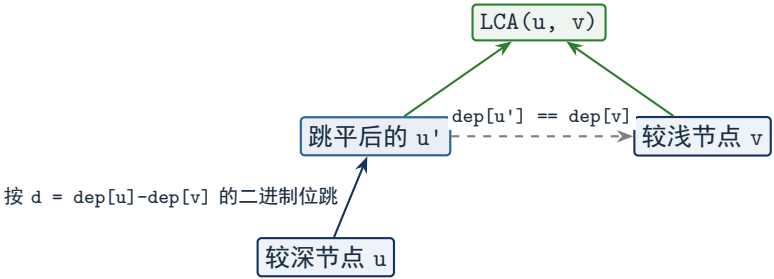
无权树 vs 带边权树，代码只差一行！

- 无权树：每条边长度默认为 1，此时 `dist[u] = dep[u]`，距离不超过 $n \le 10^5$。
- 带边权树：若每条边有边权 $w \le 10^9$，只需把更新改成 `dist[v] = dist[u] + w`。
- 溢出警报：若树是一条 10^5 个点的长链且边权 $w = 10^9$，两端点距离可达 $10^5 \times 10^9 = 10^{14}$，远超 `int` 上限（$\approx 2.1 \times 10^9$）！

本页要点

只要题目出现边权，`dist[]` 数组和返回值一律开 `long long`，养成零失误习惯！

TikZ 图解：求 LCA 第一步——先让深节点跳平

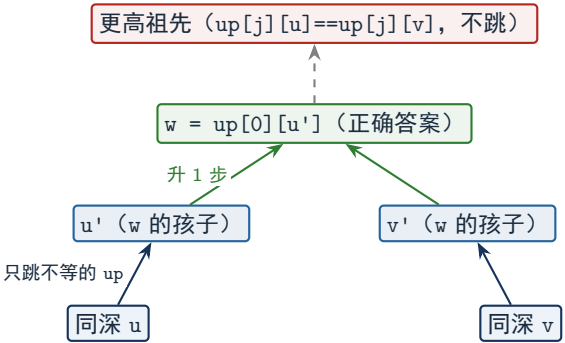


- 永远先保证 dep[u] >= dep[v]（不够就 swap(u, v)），令深度差 d = dep[u] - dep[v]。
- 枚举二进制位 j = 0..K-1：若 (d >> j) & 1 为真，就令 u = up[j][u]。
- 特判：如果跳平后发现 u == v，说明 v 本身就是 u 的祖先，直接返回 u！

本页要点

跳平后必须立刻写 if (u == v) return u;，否则第二步还会强行往上跳错！

TikZ 图解：求 LCA 第二步——从大步到小步同步试探



- 从 `j = K - 1` 倒序试探到 0：只要 `up[j][u] != up[j][v]` 就说明还没撞到公共祖先，放心一起跳；否则不跳。

本页要点

循环结束时，`u` 和 `v` 恰好停在 LCA 正下方一格，返回 `up[0][u]` 即可！

追问：第二步为什么必须从 K-1 倒序跳，不能从 0 正序跳？

举例拆穿“正序跳”的漏洞

假设同深度的 u 和 v 距离它们的最近公共祖先 w 恰好是 5 步（二进制 $5 = 4 + 1$），我们要把它们跳到离 w 差 1 步的位置（即往上跳 4 步）：

- 正确做法（倒序 $j = 3, 2, 1, 0$）：$j = 3$（跳 8 步）相等不跳；$j = 2$（跳 4 步）不等，跳！此时离 w 只差 1 步；之后 $j = 1, 0$ 都会越过或撞上 w，都不跳。最终停在 w 下方 1 步！
- 错误做法（正序 $j = 0, 1, 2, 3$）：$j = 0$（跳 1 步）不等，跳！ $j = 1$（跳 2 步）不等，跳！已经跳了 $1 + 2 = 3$ 步，离 w 还有 2 步，但后面只剩 $j = 2$（跳 4 步，会越过 w 不敢跳），结果被卡在离 w 差 2 步的地方，返回 up[0][u] 就错了！

本页要点

任何非负整数都能从高位到低位唯一贪心凑出（二进制试探），从低位到高位会提前消耗掉小步长！

手算跟踪表：用 6 点树追踪 lca(5，6) 的完整过程

树结构：根 1，边 1-2，1-3，2-4，2-5，3-6。深度 dep[5]=2，dep[6]=2，取 $K = 3$。

阶段	j	u	v	up[j][u]	up[j][v]	决策与原因
跳平深度	-	5	6	-	-	d = 2 − 2 = 0，已同深，且 u != v
倒序同步	2	5	6	0	0	相等（跳 4 步出界），不跳
倒序同步	1	5	6	1	1	相等（正好撞上祖先 1），不跳
倒序同步	0	5	6	2	3	不等！同步跳：u=2，v=3
终止步	-	2	3	1	1	循环结束，返回 up[0][2] = 1

本页要点

每步都问自己“跳 2^j 步后会不会撞到同一个点?”，不会就跳，会就忍住！

为什么倍增 LCA 一定是正确的？（不变量三段论 + 复杂度）

环节	严格证明要点（设目标 $w = \mathrm{LCA}(u, v)$，距同深点距离 $D \geq 1$）
初始化	抬平后若 `u == v` 已返回；否则 u, v 同深且 $u \neq v$，待跳步数 $D - 1 \in [0, 2^K - 1]$。
保持	`j=K-1..0` 维持“仍需跳 r 步到 w 下方 1 格”且 $0 \leq r < 2^{j+1}$。若 $r \geq 2^j$ 则 `up[j][u]!=up[j][v]`，跳后 $r \leftarrow r - 2^j < 2^j$；否则不跳。
终止	`j=0` 结束后 $r = 0$，u, v 停在 w 的两个不同子节点，返回 `up[0][u]=w`。

本页要点

预处理 $O(n \log n)$、单次查询 $O(\log n)$；把 $D - 1$ 从高位到低位拼出即停在 LCA 正下方一格！

倍增 LCA 模板代码（1/2）：非递归建树与预处理

```
int K = 1; while ((1 << K) <= n) ++K;                    // 保证 2^K > n
vector<vector<int>> up(K, vector<int>(n + 1, 0));
vector<int> dep(n + 1, 0), st = {1}; vector<long long> dist(n + 1, 0);
while (!st.empty()) {                                    // 显式栈遍历，防长链爆栈
    int u = st.back(); st.pop_back();
    for (auto [v, w] : g[u]) if (v != up[0][u]) {
        up[0][v] = u; dep[v] = dep[u] + 1; dist[v] = dist[u] + w; st.push_back(v);
    }
}
for (int j = 1; j < K; ++j) for (int u = 1; u <= n; ++u) up[j][u] = up[j-1][up[j
    -1][u]];
```

本页要点

用 vector 显式栈建树，就算退化成 10^5 长链也绝不爆栈 RE！

倍增 LCA 模板代码（2/2）：两步走查询

```
auto lca = [&](int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);                    // 永远让 u 当更深的一方
    int d = dep[u] - dep[v];
    for (int j = 0; j < K; ++j)                         // 1. 按二进制拆分把 u 跳齐同深
        if ((d >> j) & 1) u = up[j][u];
    if (u == v) return u;                               // 特判：v 本身就是 u 的祖先
    for (int j = K - 1; j >= 0; --j)                    // 2. 从高幂往低幂同步试探上跳
        if (up[j][u] != up[j][v]) { u = up[j][u]; v = up[j][v]; }
    return up[0][u];                                    // 停在 LCA 正下方一格，升 1 步
    即答案
};
```

本页要点

口诀：先跳平、判相等、再倒序同步跳、最后返回 up[0][u]！

上机验证：3 组自测用例与反事实追问

用 6 点树（边 1-2，1-3，2-4，2-5，3-6）上机测试以下 3 组询问

1. 普通跨分支：查 (5，6)，预期 LCA = 1，距离 dist = 2 + 2 - 2*0 = 4；
2. 直系祖孙（边界）：查 (4，2)，预期跳平后直接 u == v 返回 LCA = 2，距离 1；
3. 同一节点（最小边界）：查 (3，3)，预期 d = 0，特判返回 LCA = 3，距离 0。

反事实追问（写代码前想清楚）

- 追问 **1**：把 j = K-1..0 改成正序 0..K-1 会怎样？（答：长链会少跳而输出错的祖先！）
- 追问 **2**：把外层 j 和内层 u 互换会怎样？（答：up[2][5] 用到未算的 up[1][3]，全是 0！）

本页要点

造数据时一定要包含“直系祖孙”和“同一点”，专抓漏写 if（u == v）return u; 的 bug！

倍增 LCA 高频易错对照（WA / TLE / RE 定位）

易错点	错误写法	正确写法	评测结果
预处理循环写反	先枚举 u 再枚举 j	先枚举 j=1..K-1 再枚举 u	**WA**
跳平后忘判相等	没写 `if (u == v) return u;`	跳平深度后立即特判返回	**WA**
同步跳正序枚举	`for (j = 0; j < K; ++j)`	必须 `j = K-1` 倒序到 0	**WA**
K 开得太小	$n = 10^5$ 只开 K=15	保证 $2^K > n$（或动态求 K）	**WA**
长链递归建树	直接写递归 `dfs(1, 0)`	用显式栈非递归遍历建树	**RE**（爆栈）

本页要点

看到大样例报 **RE**（段错误），优先检查是不是 10^5 长链导致递归 `dfs` 爆栈！

思维拓展：如果不仅求 LCA，还想求路径上的“最大边权”？

倍增表也能顺带维护路径极值（如次小生成树 / 货车运输）

除了记录“跳 2^j 步到哪个点” `up[j][u]`，再多开一个数组 `mx[j][u]` 表示“从 u 往上跳 2^j 步沿途经过的最大边权”：

- 预处理递推：`mx[j][u] = max(mx[j-1][u], mx[j-1][ up[j-1][u] ]);`
- 查询时每跳一次 `u = up[j][u]`，顺手用 `mx[j][u]` 更新答案！

- 局限性：如果题目要求“动态修改某条边权或某个点权”，倍增表每次改动要重算大量节点（太慢），这时就要请出下一节的树链剖分！

本页要点

静态树上最值可用倍增顺带维护；一旦要求“动态修改”，就必须交给树链剖分 + 线段树！

痛点升级：如果要“修改点权 + 查路径最大值/总和”怎么办？

倍增 LCA 的局限性

上一节的倍增 LCA 擅长静态求公共祖先与树上距离；但如果题目要求：

- 操作 1：把节点 u 的权值修改为 t（动态单点改）；
- 操作 2：查询点 u 到点 v 路径上的最大点权和点权总和。

- 暴力做法：每次从 u 和 v 沿着父亲一步步爬到 LCA，顺便累加权值。单链退化时 $q = 10^5$ 次询问 $\times 10^5$ 步 $= 10^{10}$，必 **TLE**！
- 核心矛盾：第 5 章的线段树能在 $O(\log n)$ 内修改并查询“连续数组区间”，但它不认识分叉的树。

本页要点

树链剖分（HLD）的作用就是当“翻译官”：把一棵树拆成几条链，拼成一维连续数组塞进线段树！

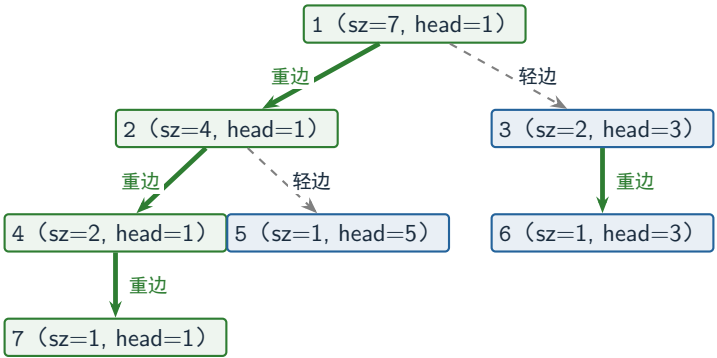
先听懂七个新术语（大白话版）

术语	大白话解释
子树大小 sz[u]	以 u 为根的整棵子树里一共有多少个节点（包含 u 自己）。
重儿子 heavy[u]	u 的所有儿子中，子树最大（sz 最大）的那个儿子（并列任选一个）。
轻儿子	u 除了重儿子以外的其余所有儿子（叶子节点没有儿子，heavy[u]=0）。
重边 / 轻边	连向重儿子的边叫重边；连向轻儿子的边叫轻边。
重链	若干条重边首尾相连形成的一条直链；孤立点也算一条长度为 1 的重链。
链顶 head[u]	节点 u 所在的那条重链中，深度最浅（最靠上）的那个起点。
DFS 序 pos[u]	按特定顺序遍历整棵树时，节点 u 被访问到的新编号（$1 \sim n$）。

本页要点

每个非叶子节点有且仅有 1 个重儿子，因此全树会被完美拆成互不相交的若干条重链！

TikZ 图解：一棵 7 点树的重儿子、重链与链顶



本页要点

图中共有 3 条重链：1–2–4–7（链顶 1）、3–6（链顶 3）、5（链顶 5）。

核心定理：为什么从任意点到根最多只经过 $\leq \log_2 n$ 条重链？

一条轻边带来的“至少减半”魔法

假设 v 是 u 的轻儿子，而 h 是 u 的重儿子：

- 因为 h 是重儿子，所以 $\mathtt{sz[h]} \geq \mathtt{sz[v]}$；
- 又因为 u 至少包含了 h 和 v 两棵子树，所以 $\mathtt{sz[u]} \geq 1 + \mathtt{sz[h]} + \mathtt{sz[v]} > 2 \cdot \mathtt{sz[v]}$；
- 换句话说：每向上跨越一条轻边，所在子树的节点数至少翻倍（向下走则至少减半）！

- 数值直觉：$n = 10^5$ 时，子树大小从 1 每次至少翻倍：$1 \to 2 \to 4 \to \cdots \to 10^5$ 最多翻倍 17 次。
- 所以从树上任意节点到根，最多跨越 $\log_2 n$ 条轻边，也就最多经过 $\log_2 n + 1$ 条重链！

本页要点

轻边少得惊人（$\leq \log_2 n$ 条），这就是树链剖分能达到 $O(\log^2 n)$ 的根本原因！

第一遍 DFS（dfs1）：摸清家底，选出每个点的重儿子

dfs1(u, p) 要完成的 4 件小事

1. 记下直接父亲 `fa[u] = p`，以及深度 `dep[u] = dep[p] + 1`；
2. 初始化自身子树大小 `sz[u] = 1`；
3. 遍历所有儿子 $v \neq p$，递归 `dfs1(v, u)`，然后累加 `sz[u] += sz[v]`；
4. 如果 `sz[v] > sz[heavy[u]]`，就更新 `heavy[u] = v`（打擂台找最大子树）。

本页要点

注意 `sz[0] = 0`，所以第一个儿子 v 一定满足 `sz[v] > sz[0]`，自然成为初始擂主！

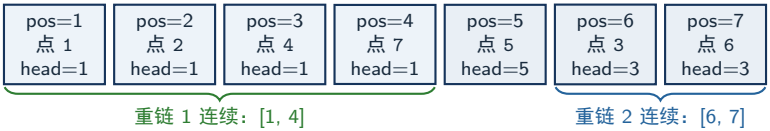
手算跟踪表：7 点树的 dfs1 计算结果（按回溯顺序）

节点 u	fa[u]	dep[u]	儿子们的 sz	sz[u]	heavy[u]
7	4	4	无（叶子）	1	0（返回）
4	2	3	sz[7]=1	2	7
5	2	3	无（叶子）	1	0
2	1	2	sz[4]=2, sz[5]=1	4	4（$2>1$）
6	3	3	无（叶子）	1	0
3	1	2	sz[6]=1	2	6
1	0	1	sz[2]=4, sz[3]=2	7	2（$4>2$）

本页要点

dfs1 自底向上汇总：先算完所有儿子的 sz，再确定谁是重儿子！

第二遍 DFS（dfs2）：先走重儿子，把重链拉成连续数组！



- 进门给 `pos[u]=++timer`，`head[u]=h`；先递归重儿子 `dfs2(heavy[u], h)`，再递归轻儿子 `dfs2(v, v)`。

本页要点

因为“先重后轻”，同一条重链上的所有点被分配到的 `pos[]` 下标一定是连续整数！

手算跟踪表：7 点树的 dfs2 访问轨迹与数组映射

初始调用 dfs2(1, 1)，全局计数器 timer = 0。

步序	调用参数 (u, h)	pos[u]	head[u]	下一步动作
1	dfs2(1, 1)	1	1	优先走重儿子 heavy[1]=2，传链顶 1
2	dfs2(2, 1)	2	1	优先走重儿子 heavy[2]=4，传链顶 1
3	dfs2(4, 1)	3	1	优先走重儿子 heavy[4]=7，传链顶 1
4	dfs2(7, 1)	4	1	无重儿子也无轻儿子，回溯到 2
5	dfs2(5, 5)	5	5	5 是 2 的轻儿子，自立门户 h=5，回溯到 1
6	dfs2(3, 3)	6	3	3 是 1 的轻儿子，自立门户 h=3，走重儿子 6
7	dfs2(6, 3)	7	3	叶子遍历完毕，dfs2 全部终止

本页要点

把每个点的原权值 w[u] 存进 val[pos[u]] = w[u]，就能直接用 val[] 建线段树了！

路径爬跳合并：如何沿着重链把 (u, v) 拼成几段区间？

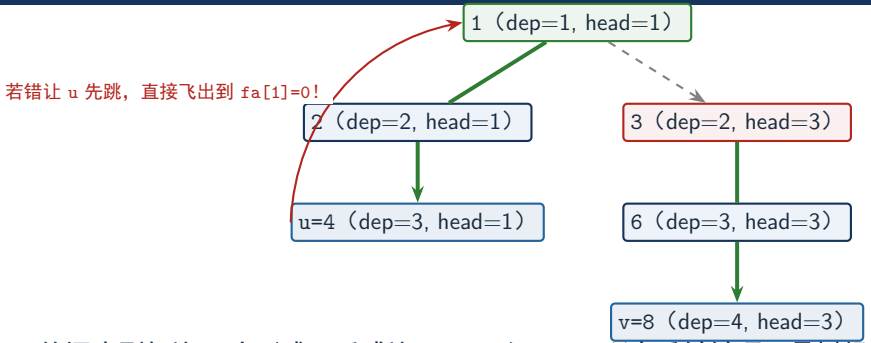
三步爬跳策略（类似“两人分别坐快车去同一条高速路”）

1. 只要两人不在同一条重链（`head[u] != head[v]`）：
   - 比较谁的链顶更深（若 `dep[head[u]] < dep[head[v]]` 就 `swap(u, v)`），永远让链顶更深的 `u` 往上跳；
   - 用线段树查询连续区间 `[pos[head[u]], pos[u]]` 并合并进答案；
   - 让 `u = fa[head[u]]` 跨过轻边，跳到上一条重链继续！
2. 落入同一条重链（`head[u] == head[v]`）：
   - 让更浅的点在前面（若 `dep[u] > dep[v]` 则 `swap(u, v)`），最后用线段树查区间 `[pos[u], pos[v]]` 即完工！

本页要点

因为链顶上方的区间 `pos[head[u]] <= pos[u]`，传给线段树时左端点一定是 `pos[head[u]]`！

TikZ 反例警告：为什么必须比 dep[head[u]]，不能比 dep[u]？



- 假如 u 的深度刚好比 v 大（或写反成比 dep[u]），而 u 所在重链链顶已是树根 1，一旦错让 u 跳向 fa[head[u]]，就会直接跳过 LCA(u, v) 飞出根节点！

本页要点

比链顶深度 dep[head[u]] 才能保证：谁所在链顶更靠下，谁的整条链段就绝不会越过 LCA！

手算跟踪表：在 7 点树上查询路径（5，6）的爬跳轨迹

树上节点 5（pos=5，head=5，dep=3，fa=2）与 6（pos=7，head=3，dep=3，fa=3）。

轮次	u/head	v/head	链顶比较	查询区间	跳向与说明
1	5/5	6/3	dep[5]>dep[3]	[5，5]	u 跳到 fa[5]=2
2	2/1	6/3	dep[1]<dep[3]	[6，7]	交换后跳 6 到 fa[3]=1
终止	2/1	1/1	同链（head=1）	[1，2]	同链查完即返回

- 合并的三段：[5,5]（点 5）、[6,7]（点 3，6）、[1,2]（点 1，2），不重不漏正好覆盖路径 5-2-1-3-6！

本页要点

每次跳出当前重链都会经过一条轻边；每条路径最多跳 $O(\log n)$ 次，每次查线段树 $O(\log n)$！

为什么树链剖分一定是正确的？（不变量三段论 + 复杂度）

论证环节	严格证明要点（设目标路径 $u_0 \leftrightarrow v_0$，公共祖先 $w = \mathrm{LCA}(u_0, v_0)$）
初始化	初始 $u = u_0, v = v_0$，待查询路径正好是 u 到 v 的全部节点。
保持	当 `head[u] != head[v]` 时，选 `dep[head[u]] >= dep[head[v]]` 一侧：因为 w 是 v 的祖先，`dep[w] <= dep[head[v]] <= dep[head[u]]`，故 `[head[u]..u]` 整段完全属于路径且不含 w 之上节点；跳至 `fa[head[u]]` 后剩余待查路径不变量保持。
终止	当 `head[u] == head[v]` 时，u, v 在同链上，浅者即为 w，补查 `[pos[u], pos[v]]` 即覆盖全路径。

- 复杂度：预处理 $O(n)$，单次路径查询/修改 $O(\log^2 n)$（常数极小）。

本页要点

树剖把树上路径无损拆分成 $\leq 2\log_2 n$ 段连续数组区间，每段直接调用线段树！

树链剖分模板代码（1/2）：两遍 DFS 剖分

```
void dfs1(int u, int p) {
    fa[u] = p; dep[u] = dep[p] + 1; sz[u] = 1;
    for (int v : g[u]) if (v != p) {
        dfs1(v, u); sz[u] += sz[v];
        if (sz[v] > sz[heavy[u]]) heavy[u] = v;      // 选子树最大的儿子作重儿子
    }
}
void dfs2(int u, int h) {
    head[u] = h; pos[u] = ++timer;                   // 先重后轻，使重链在 DFS 序上连
    续
    if (heavy[u]) dfs2(heavy[u], h);
    for (int v : g[u]) if (v != fa[u] && v != heavy[u]) dfs2(v, v);
}
```

本页要点

调用 `dfs1(1, 0); dfs2(1, 1);` 后，别忘了把点权按 `val[pos[u]] = w[u]` 重排再建线段树！

树链剖分模板代码（2/2）：路径爬跳查询

```
Info path_query(int u, int v, SegTree &seg) {
    Info res;                                          // 默认 sum=0, mx=INT_MIN
    while (head[u] != head[v]) {                       // 两人不在同一条重链上
        if (dep[head[u]] < dep[head[v]]) swap(u, v); // 永远选链顶更深的一侧往上跳
        res = merge_info(res, seg.query(1, 1, n, pos[head[u]], pos[u]));
        u = fa[head[u]];                               // 顺着轻边跨到上一条重链
    }
    if (dep[u] > dep[v]) swap(u, v);                   // 最后落入同一条重链，浅的在前
    return merge_info(res, seg.query(1, 1, n, pos[u], pos[v]));
}
```

本页要点

如果题目还顺便需要求 LCA(u, v)，跳出 while 循环后深度较浅的点（即此时的 u）就是 LCA！

上机验证：3 组自测用例与反事实追问

上机测试建议（3 组必测边界）

1. 单链树（$n = 5$）：全树只有 1 条重链，while 循环一次都不进，直接查 [pos[u]，pos[v]]；
2. 星形树（根 1 连出 4 叶子）：验证轻边跳法与 head[u] 自立门户是否正确；
3. 全负权小树（$w_i = -100$）：查单点路径（u，u）最大值，确认不会误输出 0。

反事实追问

追问：若 dfs2 改成先循环轻儿子，最后再走重儿子，会发生什么？
回答：重链上的 pos[] 会被轻子树隔断而不连续，线段树查 [pos[head[u]]，pos[u]] 会误算进无关节点，彻底 **WA**！

本页要点

“先重后轻”是重链在 DFS 序上严格连续的唯一保证，顺序绝对不能颠倒！

树链剖分高频易错对照与方法对比

易错点	错误写法	正确写法	评测结果
爬跳比较错深度	`dep[u] < dep[v]`	`dep[head[u]] < dep[head[v]]`	**WA**
线段树传错编号	传原编号 `u` 修改/查询	必须传映射后的 `pos[u]`	**WA**
最大值初值写 0	`int mx = 0;`	设极小值 `INT_MIN`	**WA**
轻儿子传错链顶	`dfs2(v, h)`	轻儿子新开重链 `dfs2(v, v)`	**WA**

- 怎么选工具：只求静态 LCA 或距离选倍增 **LCA**；需要动态改点权/路径权值必须用树链剖分 + 线段树。

本页要点

树剖时刻分清“原节点编号 `u`”与“DFS 序位置 `pos[u]`”两套坐标系！

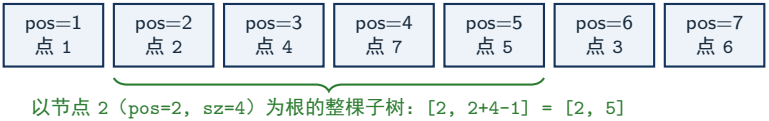
两套坐标系对照表：再也不把 u 和 pos[u] 弄混

数组 / 函数	传入的下标是谁	存的值是谁
`fa[u]`, `dep[u]`, `sz[u]`	原节点编号 `u`	节点 `u` 的父亲、深度、子树大小
`heavy[u]`, `head[u]`	原节点编号 `u`	节点 `u` 的重儿子、所在重链的链顶（都是原编号）
`pos[u]`	原节点编号 `u`	节点 `u` 映射到线段树里的新下标（$1 \sim n$）
`val[i]`（建树初值）	新下标 `i = pos[u]`	节点 `u` 的初始点权 `w[u]`
`seg.update / query`	新下标 `pos[u]`	查询连续区间 `[pos[head[u]], pos[u]]`

本页要点

记住一条铁律：树剖数组用原编号 `u`，线段树接口一律用新下标 `pos[u]`！

额外福利：树链剖分顺带支持“整棵子树”的修改与查询！



- 无论怎么排重儿子，dfs2 在离开点 u 之前，一定会先把 u 的整棵子树全部访问完！
- 所以以 u 为根的子树在 DFS 序上也是连续的一整段：[pos[u]，pos[u] + sz[u] - 1]！
- 只需要在原本的线段树上加一个懒标记，就能同时处理“路径修改/查询”和“子树修改/查询”！

本页要点

一条 DFS 序同时让“重链”连续、也让“子树”连续，这就是树链剖分被称为万金油的原因！

两种极端树形下的树剖表现（单链 vs 星形树）

树形态	重链划分结果	路径查询实际跳几次？
单链退化（10^5 点）	全树只有 1 条重链（`head[u]=1`）	`while` 循环 0 次！直接 1 次线段树区间查询 $O(\log n)$ 搞定！
星形树（根连 10^5 叶子）	重儿子与根同链，其余叶子各自成链	任意两点最多跳 $1 \sim 2$ 次轻边就到根 1！

本页要点

树越“长”，重链越长、跳的次数越少；树越“胖”，轻边虽然多但树高极浅，怎么都快！

倍增 LCA 与树链剖分全面对比：什么时候用谁？

对比维度	倍增 LCA	树链剖分（HLD）
预处理复杂度	时间 $O(n \log n)$，空间 $O(n \log n)$	时间 $O(n)$，空间 $O(n)$
单次求 LCA 复杂度	$O(\log n)$	$O(\log n)$（常数往往比倍增更小）
静态路径距离 / 边极值	非常适合（$O(\log n)$）	可以做（$O(\log^2 n)$）
动态修改点权 / 路径加	不支持	核心强项（配线段树 $O(\log^2 n)$）
代码量	约 25 行	约 $50 \sim 80$ 行（含线段树）

本页要点

只需静态求 LCA/距离选倍增；只要题目出现“修改”二字，果断上树链剖分！

例 1 | [基础] 点的距离（一本通 1552）：先读懂要求

大白话题意

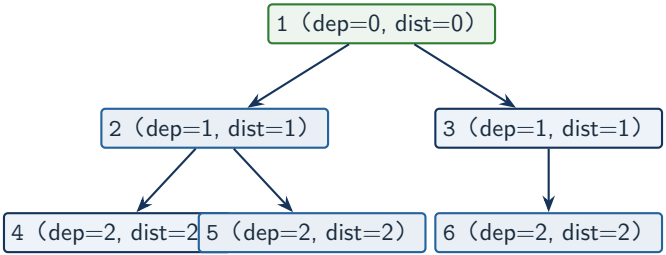
给定一棵 n 个节点的无向无权树（$1 \le n \le 10^5$），共有 Q 次询问，每次询问节点 x 到节点 y 之间的距离（即路径上经过的边数）。

项目	具体要求
输入	第 1 行 n；接下来 $n-1$ 行每行 x, y 表示一条无向边；接着 Q 及 Q 行询问 x, y。
处理	对每次询问 (x, y)，在 $O(\log n)$ 内算出树上唯一简单路径的边数。
输出	共 Q 行，每行一个整数表示对应询问的距离。
样例	$n = 6$，边 1-2，1-3，2-4，2-5，3-6，询问 (2,6) 和 (5,6) $\Rightarrow$ 输出 3 和 4。

本页要点

无权树默认每条边长度为 1，节点到根的距离就等于节点深度 `dep[u]`！

例 1 | 怎么想到的与建模 TikZ 图解



- 为什么不能暴力：$n \le 10^5$ 时若树退化成一条链，每次向上爬父亲会达到 $O(nQ)$ 超时。
- 模型转化：以 1 为根预处理 `up[j][u]` 与 `dist[u]`，每次询问用公式 `dist[u] + dist[v] - 2 * dist[lca(u, v)]` 即可 $O(\log n)$ 出解！

本页要点

用倍增 LCA 把“两段路径拼接”变成“两次查前缀距离再减去两倍公共前缀”。

例 1 | 样例手推跟踪表

对照左图 6 点树（dist[1]=0，dist[2]=dist[3]=1，dist[4]=dist[5]=dist[6]=2），逐步演算样例的两组询问：

询问	跳平深度后	同步跳后停在	lca(u,v)	代入距离公式	输出
(2，6)	6 上跳 1 步到 3	2 与 3 不跳	up[0][2]=1	1 + 2 - 2*0	3
(5，6)	已同深（dep=2）	跳 1 步到 2 与 3	up[0][2]=1	2 + 2 - 2*0	4

- 肉眼核对：路径 $2 \to 1 \to 3 \to 6$ 经过 3 条边；路径 $5 \to 2 \to 1 \to 3 \to 6$ 经过 4 条边，完全吻合！

本页要点

手推样例先核对 lca 是否正确，再核对距离公式，两步分开查错最快！

例 1 ｜ 核心步骤表与相对模板的变化

按顺序计算	纸上得到	核对问题
1. 读边建邻接表	无向图 `g[u]`，每条边权 `w=1`	两条有向边是否都存了？
2. 非递归遍历树	算出 `up[0][v]`, `dep[v]`, `dist[v]`	长链时会不会递归爆栈？
3. 填满倍增表	外层 `j=1..K-1` 算 `up[j][u]`	根节点 1 的 `up[*][1]` 是否全为 0？
4. 回答每次询问	`dist[u]+dist[v]-2*dist[lca]`	`dist` 是否用 `long long` 防扩展带权溢出？

相对倍增 LCA 模板改了哪里？

- 邻接表存成 `pair<int, int>` 并默认 `w = 1`，以后题目改成带权树只需多读一个 `w`；
- 用显式栈 `vector<int> st = {1}` 代替递归 DFS，彻底杜绝 10^5 单链爆栈 RE。

本页要点

把“读边、建表、求 LCA、算公式”拆成四步核对，代码一次写对！

例 1 | 核心代码（1/2）：建树与倍增表预处理

```
int K = 1; while ((1 << K) <= n) ++K;                    // 最小幂次 K 满足 2^K > n
vector<vector<int>> up(K, vector<int>(n + 1, 0));
vector<int> dep(n + 1, 0), st = {1}; vector<long long> dist(n + 1, 0);
while (!st.empty()) {                                     // 非递归遍历建树防爆栈
    int u = st.back(); st.pop_back();
    for (auto [v, w] : g[u]) if (v != up[0][u]) {
        up[0][v] = u; dep[v] = dep[u] + 1; dist[v] = dist[u] + w; st.push_back(v);
    }
}
for (int j = 1; j < K; ++j)                               // 外层枚举 j 递推所有 2^j 级祖先
    for (int u = 1; u <= n; ++u) up[j][u] = up[j - 1][up[j - 1][u]];
```

本页要点

与 code/lca_dist.cpp 一致；注意 dist 开 long long，方便直接拓展带权树。

例 1 | 核心代码（2/2）：lca 函数与回答询问

```
auto lca = [&](int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);                    // 保证 u 深度更大
    int d = dep[u] - dep[v];
    for (int j = 0; j < K; ++j) if ((d >> j) & 1) u = up[j][u]; // 1. 跳齐同深
    if (u == v) return u;                               // 直系祖孙特判
    for (int j = K - 1; j >= 0; --j)                    // 2. 倒序同步上跳
        if (up[j][u] != up[j][v]) { u = up[j][u]; v = up[j][v]; }
    return up[0][u];
};
int q; cin >> q;
while (q--) { int u, v; cin >> u >> v; cout << dist[u] + dist[v] - 2 * dist[lca(u,
    v)] << '\n'; }
```

本页要点

用 '\n' 换行并关闭同步 ios::sync_with_stdio(false)，应对 10^5 次输出不超时。

例 1 | 易错与 3 组自测边界

常见失分点

- 直系祖孙忘特判：问 (2, 4) 时跳平后已 `u == v == 2`，漏写 `if (u == v) return u;` 会多跳一层返回 1（WA）；
- 无向边只加了一边：只写 `g[u].push_back` 会导致从 1 遍历不到某些节点（WA）。

自测场景	测试输入简述	预期输出
最小边界（单点）	$n = 1$，询问 (1, 1)	0
直系祖孙	样例树询问 (2, 4) 和 (1, 5)	1、2
长链压力（$n = 10^5$）	链 1-2-...-100000，询问 (1, 100000)	99999（不爆栈）

本页要点

提交前用 $n = 1$ 和直系祖孙两份小数据自测，再跑一次长链防 RE！

例 2 | [提高] 树的统计（一本通 1560）：先读懂要求

大白话题意

有一棵 n 个节点的树（$1 \le n \le 3 \times 10^4$），每个节点有初始点权 $w_i \in [-30000, 30000]$。共 $q \le 2 \times 10^5$ 次操作：

- `CHANGE u t`：把节点 u 的权值修改为 t；
- `QMAX u v`：询问从 u 到 v 路径上所有节点（含 u, v）的最大点权；
- `QSUM u v`：询问从 u 到 v 路径上所有节点（含 u, v）的点权总和。

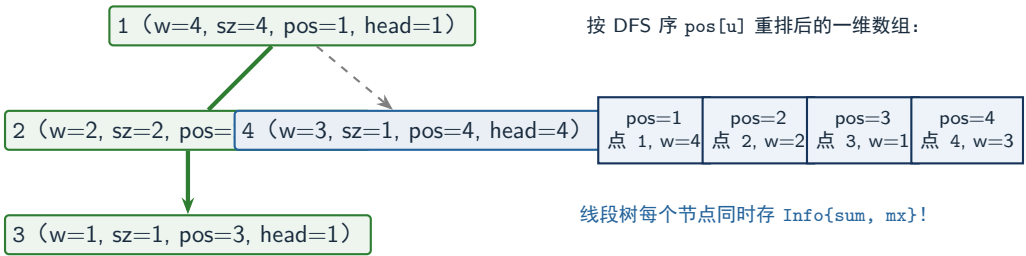
原题样例（$n = 4$，点权 4，2，1，3）

边为 1-2，2-3，4-1。前 4 次询问 `QMAX 3 4` $\to 4$，`QMAX 3 3` $\to 1$，`QMAX 3 2` $\to 2$，`QMAX 2 3` $\to 2$，`QSUM 3 4` $\to 10$。

本页要点

注意审题：点权范围是 $-30000 \sim 30000$，意味着路径最大权值可以是负数！

例 2 | 怎么想到的与建模 TikZ 图解



- 既有单点修改，又有树上路径求和与最值 ⇒ 标准树链剖分 + 线段树模板题！
- 一条线段树同时维护 sum 和 mx（写成 struct Info），比写两棵线段树更短更不易错。

本页要点

建线段树前一定要按 val[hld.pos[i]] = w[i] 重排初始点权！

例 2 | 样例手推跟踪表

重链 1：1-2-3（pos=1,2,3，初始 w=[4,2,1]）；重链 2：4（pos=4，head=4，w=3）。

操作	树剖拆分区间	线段树读取 / 修改	输出
QMAX 3 4	head[4]=4 跳区间 [4,4]，再同链 [1,3]	max(3, max(4,2,1))	4
QSUM 3 4	同上：[4,4] 与 [1,3]	3 + (4+2+1)	10
CHANGE 1 5	改点 1 对应 pos[1]=1	val[1] <- 5	-
QMAX 3 4	[4,4] 与 [1,3]（此时 val[1]=5）	max(3, max(5,2,1))	5
CHANGE 3 6	改点 3 对应 pos[3]=3	val[3] <- 6	-
QSUM 3 4	[4,4] 与 [1,3]（此时 val=[5,2,6,3]）	3 + (5+2+6)	16

本页要点

所有操作最终都变成对数组 val[1..4] 的单点修改和区间查询，与原题输出分毫不差！

例 2 | 核心步骤表与相对模板的变化

按顺序计算	纸上得到	核对问题
1. `dfs1`, `dfs2` 剖分	每个点的 `sz`, `heavy`, `head`, `pos`	`dfs1(1, 0)` 的 `dep[0]` 是否为 0?
2. 重排数组建线段树	`val[pos[i]] = w[i]` 传入 `SegTree`	是否不小心传了原数组 `w`?
3. `CHANGE u t`	单点改线段树位置 `pos[u]`	有没有把 `u` 错写成 `pos[u]`?
4. `QMAX / QSUM u v`	爬跳合并 `Info{sum, mx}`	`Info` 的默认 `mx` 是否为 `INT_MIN`?

相对树剖模板改了哪里？

- 线段树节点合并封装为 `Info merge_info(Info a, Info b)`，一次遍历就能同时算出 `sum` 和 `mx`；
- 注意 $n \le 3 \times 10^4$，直接写递归 `dfs1`，`dfs2` 不会爆栈，代码简洁明了。

本页要点

空 `Info` 必须初始化为 `{sum = 0, mx = INT_MIN}`，这是全负权数据 AC 的关键！

例 2 | 核心代码（1/4）：Info 与线段树建树

```
struct Info { long long sum = 0; int mx = INT_MIN; }; // 空区间默认值：和 0，最大负无穷
Info merge_info(Info a, Info b) { return {a.sum + b.sum, max(a.mx, b.mx)}; }
struct SegTree {
    int n; vector<Info> tr;
    SegTree(int n, const vector<int> &val) : n(n), tr(4 * n + 4) { build(1, 1, n, val); }
    void build(int p, int l, int r, const vector<int> &val) {
        if (l == r) { tr[p] = {val[l], val[l]}; return; }
        int m = (l + r) / 2;
        build(p * 2, l, m, val); build(p * 2 + 1, m + 1, r, val);
        tr[p] = merge_info(tr[p * 2], tr[p * 2 + 1]);
    }
```

本页要点

线段树大小开 `4 * n + 4`，把 `sum` 与 `mx` 打包进 `Info` 一次维护！

例 2 | 核心代码（2/4）：线段树单点修改与区间查询

```
void update(int p, int l, int r, int idx, int v) {
    if (l == r) { tr[p] = {v, v}; return; }
    int m = (l + r) / 2;
    if (idx <= m) update(p * 2, l, m, idx, v); else update(p * 2 + 1, m + 1, r,
 idx, v);
    tr[p] = merge_info(tr[p * 2], tr[p * 2 + 1]);
}
Info query(int p, int l, int r, int ql, int qr) {
    if (ql <= l && r <= qr) return tr[p];
    int m = (l + r) / 2; Info res;
    if (ql <= m) res = merge_info(res, query(p * 2, l, m, ql, qr));
    if (qr > m)  res = merge_info(res, query(p * 2 + 1, m + 1, r, ql, qr));
    return res;
} };
```

本页要点

注意 query 里空局部对象 `Info res;` 的默认值就是 {0, INT_MIN}！

例 2 | 核心代码（3/4）：HLD 两遍 DFS 剖分

```
void dfs1(int u, int p) {
    fa[u] = p; dep[u] = dep[p] + 1;
    for (int v : g[u]) if (v != p) {
        dfs1(v, u); sz[u] += sz[v];
        if (sz[v] > sz[heavy[u]]) heavy[u] = v;      // 选子树最大的儿子作重儿子
    }
}
void dfs2(int u, int h) {
    head[u] = h; pos[u] = ++timer;                   // 先重后轻，使重链在 DFS 序上连
    续
    if (heavy[u]) dfs2(heavy[u], h);
    for (int v : g[u]) if (v != fa[u] && v != heavy[u]) dfs2(v, v);
}
```

本页要点

与 code/hld_tree_stats.cpp 一致；注意初始化时 sz 全设为 1。

例 2 | 核心代码（4/4）：路径合并与主函数处理

```
Info path_query(int u, int v, SegTree &seg) {
    Info res;
    while (head[u] != head[v]) {
        if (dep[head[u]] < dep[head[v]]) swap(u, v); // 永远选链顶更深的一侧往上跳
        res = merge_info(res, seg.query(1, 1, n, pos[head[u]], pos[u]));
        u = fa[head[u]];
    }
    if (dep[u] > dep[v]) swap(u, v);
    return merge_info(res, seg.query(1, 1, n, pos[u], pos[v]));
}
```

- main 中先 `val[hld.pos[i]] = w[i];` 建树；再 CHANGE 改 `hld.pos[u]`，QMAX/QSUM 调 `path_query`。

本页要点

完整可编译源码位于 `code/hld_tree_stats.cpp`，核心逻辑与上述四页完全一致。

例 2 | 易错与 3 组自测边界

常见失分点

- 全负数点权坑：若 `Info::mx` 或 `path_query` 初始值写成 `0`，全负路径会错输出 `0`（WA）；
- 改点忘映射下标：`CHANGE u t` 必须传 `hld.pos[u]`，传成 `u` 会改到别的点（WA）。

自测场景	测试输入简述	预期输出
全负权小树	$n=2$，边 1-2，点权 -5，-10，查 `QMAX 1 2`	-5
单点路径	查 `QMAX 2 2` 与 `QSUM 2 2`（$w_2=-10$）	-10、-10
修改后重查	`CHANGE 2 7` 后再查 `QMAX 1 2` 与 `QSUM 1 2`	7、2

本页要点

凡是涉及树上最大值查询，必造一组全负数节点权值的小树测试！

倍增 LCA 与树链剖分高频错误对照表

题目 / 场景	常见错误写法	正确做法与原因
倍增数组循环顺序	外层枚举节点 u，内层枚举幂次 j	跳 2^j 依赖所有点跳 2^{j-1}，必须外层枚举 j！
同深度后忘了特判	跳齐深度后没写 `if (u == v) return u;`	本身就是直系祖先时继续往上找会跳错成它爹！
同步跳正序枚举	`for (int j = 0; j < K; ++j)`	只有从 `K-1` 倒序到 `0` 才能贪心停在 LCA 下方 1 步！
树剖爬跳比错深度	比较 `dep[u] < dep[v]` 而非链顶	可能把较浅链顶先跳飞，必须比 `dep[head[u]] < dep[head[v]]`
线段树错用原编号	直接用节点编号 `u` 查/改线段树	树剖后连续区间对应 DFS 序 `pos[u]`，必须映射后再传！
负数最大点权初值	路径最大值初值设为 `0`	全负数树路径最大点权 < 0 会错输出 0，初值设 `INT_MIN`

本页要点

树剖时刻分清“原节点编号 u”与“DFS 序位置 pos[u]”两套坐标系！

评测结果定位指南（WA / TLE / RE 快速排查）

评测结果	最常见诱因	30 秒定位方法
WA	漏判 `u == v`；正序同步跳；`dep[u]` 代替 `dep[head[u]]`；最大值初值写 0	手造两组小树：直系祖孙 (2, 4)；全负点权 -5，-10，逐行打印调试。
TLE	没关同步 `cin/cout` 或用了 `endl`；树剖线段树 `build` 重复建树	检查是否加了 `ios::sync_with_stdio(false)` 并用 `'\n'` 换行。
RE	$n = 10^5$ 长链递归 DFS 爆栈；数组 `up[K][n+1]` 的 `K` 或线段树 `4*n` 开小	改用显式栈非递归遍历建树；检查 `2^K > n` 与 `4*n+4`。

本页要点

遇到 **WA** 先测“直系祖孙 + 全负权”，遇到 **RE** 先查“长链爆栈 + 数组大小”！

提交前检查清单（六必查）

1. 倍增表的第一维 K 是否满足 $2^K > n$（如 $n = 10^5$ 至少开 18 层），且外层循环是 `j`？
2. 求 LCA 第一步把深节点抬平后，是否紧跟着写了 `if (u == v) return u;`？
3. 求 LCA 第二步协同跳跃循环是否是从高幂次 `K - 1` 倒序跑到 `0`？
4. 树链剖分建线段树时，是不是用新下标 `pos[u]` 填初值，并在 `change/query` 传参 `pos[u]`？
5. `while (head[u] != head[v])` 中是不是挑 `dep[head[u]]` 更大的先向上爬？
6. 若节点权值可为负，线段树或路径查询 `mx` 的空区间默认值有没有设为 `INT_MIN`？

最后一步

造一组只有 2 个点且权值为负的小树、以及一条长链测直系祖孙 LCA 后再提交！

本章全部练习与完成标准

1. **[基础]** 点的距离（学校 OJ：1552 | 点的距离，配套源码 `code/lca_dist.cpp`）：倍增 LCA 求无权树两点距离。
2. **[提高]** 暗的连锁（学校 OJ：1553 | 暗的连锁）：树上差分 + 倍增 LCA 经典应用。
3. **[提高]** 树的统计（学校 OJ：1560 | 树的统计，配套源码 `code/hld_tree_stats.cpp`）：树链剖分 + 线段树单点修改、路径最值与路径和。

完成标准

- 能独立写出零 BUG 的倍增 LCA，并在“直系祖孙”与“10^5 长链”样例下验对；
- 能完整梳理树剖两遍 DFS 各个数组的含义，独立 AC 上述 3 道精选习题。

课后安排与下一章预告

- 课后复习：在纸上画一棵 7 个点的树，亲手模拟两遍重链剖分，标出每个点的 `sz`，`heavy`，`head`，`pos`。
- 拓展练习：把 `code/lca_dist.cpp` 扩展成支持带边权的版本，并尝试用树剖顺带实现子树查询 `[pos[u], pos[u]+sz[u]-1]`。
- 下一章预告：第 7 章《平衡树 Treap》——数据结构篇收官！当既要动态插入删除、又要查排名与第 k 大，且不想被有序输入卡成 $O(n)$ 单链时，看随机优先级如何让搜索树稳稳保持 $O(\log n)$ 平衡！

本页要点

承前启后：倍增解决静态树上查询，树剖打通树与线段树的桥梁，下一章解决动态有序集合！

本章一句话

核心总结

倍增靠 2^k 跨越实现树上快速会师，树剖靠重链把树压成线段直接复用线段树！

谢谢大家！