

=====

文件夹: class028\_BinaryTreeTraversal

=====

[Markdown 文件]

=====

文件: README.md

=====

# Class017 - 二叉树递归遍历专题

## ## 📚 专题简介

本专题深入讲解二叉树的递归遍历算法，包括前序、中序、后序三种基本遍历方式，以及基于递归遍历思想的经典算法题目。通过系统学习本专题，你将：

1. \*\*深入理解递归序的本质\*\*: 每个节点被访问 3 次的规律
2. \*\*掌握递归遍历的三种形式\*\*: 前序、中序、后序及其应用场景
3. \*\*熟练运用递归解决树形问题\*\*: 路径、深度、子树等各类问题
4. \*\*理解递归优化技巧\*\*: 记忆化、剪枝、全局变量等优化手段

## ## 🔄 核心算法

### ### 1. 递归序 (Recursion Pattern)

**核心思想:** 在递归过程中，每个节点会被访问三次

- 第 1 次: 刚进入该节点时 (下潜前)
- 第 2 次: 从左子树返回时 (左子树遍历完成)
- 第 3 次: 从右子树返回时 (右子树遍历完成)

**应用:**

- 在第 1 次访问位置处理 → 前序遍历
- 在第 2 次访问位置处理 → 中序遍历
- 在第 3 次访问位置处理 → 后序遍历

### ### 2. 三种基本遍历

#### #### 前序遍历 (Pre-order)

- **顺序:** 根 → 左 → 右
- **应用场景:** 复制树、前缀表达式、序列化
- **时间复杂度:**  $O(n)$
- **空间复杂度:**  $O(h)$ ,  $h$  为树高

#### #### 中序遍历 (In-order)

- \*\*顺序\*\*: 左 → 根 → 右
- \*\*应用场景\*\*: 二叉搜索树有序遍历、中缀表达式
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$

#### ##### 后序遍历 (Post-order)

- \*\*顺序\*\*: 左 → 右 → 根
- \*\*应用场景\*\*: 删除树、计算表达式、收集子树信息
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$

## ## 📋 题目列表

### ### 基础题目 (LeetCode Easy)

题号	题目	难度	核心考点	题目链接
104	二叉树的最大深度	Easy	后序遍历、递归基础	[LeetCode] ( <a href="https://leetcode.cn/problems/maximum-depth-of-binary-tree/">https://leetcode.cn/problems/maximum-depth-of-binary-tree/</a> )
111	二叉树的最小深度	Easy	递归、边界条件处理	[LeetCode] ( <a href="https://leetcode.cn/problems/minimum-depth-of-binary-tree/">https://leetcode.cn/problems/minimum-depth-of-binary-tree/</a> )
100	相同的树	Easy	双树递归、同步遍历	[LeetCode] ( <a href="https://leetcode.cn/problems/same-tree/">https://leetcode.cn/problems/same-tree/</a> )
101	对称二叉树	Easy	镜像递归、对称性判断	[LeetCode] ( <a href="https://leetcode.cn/problems/symmetric-tree/">https://leetcode.cn/problems/symmetric-tree/</a> )
226	翻转二叉树	Easy	前序遍历、树的变换	[LeetCode] ( <a href="https://leetcode.cn/problems/invert-binary-tree/">https://leetcode.cn/problems/invert-binary-tree/</a> )
112	路径总和	Easy	路径递归、目标值传递	[LeetCode] ( <a href="https://leetcode.cn/problems/path-sum/">https://leetcode.cn/problems/path-sum/</a> )
257	二叉树的所有路径	Easy	回溯法、路径收集	[LeetCode] ( <a href="https://leetcode.cn/problems/binary-tree-paths/">https://leetcode.cn/problems/binary-tree-paths/</a> )
404	左叶子之和	Easy	条件判断、左叶子识别	[LeetCode] ( <a href="https://leetcode.cn/problems/sum-of-left-leaves/">https://leetcode.cn/problems/sum-of-left-leaves/</a> )
617	合并二叉树	Easy	双树递归、同步构建	[LeetCode] ( <a href="https://leetcode.cn/problems/merge-two-binary-trees/">https://leetcode.cn/problems/merge-two-binary-trees/</a> )
563	二叉树的坡度	Easy	后序遍历、全局变量	[LeetCode] ( <a href="https://leetcode.cn/problems/binary-tree-tilt/">https://leetcode.cn/problems/binary-tree-tilt/</a> )

### ### 进阶题目 (LeetCode Medium)

题号	题目	难度	核心考点	题目链接
110	平衡二叉树	Medium	自底向上递归、剪枝优化	

[LeetCode] ( <a href="https://leetcode.cn/problems/balanced-binary-tree/">https://leetcode.cn/problems/balanced-binary-tree/</a> )
113   路径总和 II   Medium   回溯法、路径收集   [LeetCode] ( <a href="https://leetcode.cn/problems/path-sum-ii/">https://leetcode.cn/problems/path-sum-ii/</a> )
437   路径总和 III   Medium   双重递归、前缀和优化
[LeetCode] ( <a href="https://leetcode.cn/problems/path-sum-iii/">https://leetcode.cn/problems/path-sum-iii/</a> )
543   二叉树的直径   Medium   后序遍历、全局最大值
[LeetCode] ( <a href="https://leetcode.cn/problems/diameter-of-binary-tree/">https://leetcode.cn/problems/diameter-of-binary-tree/</a> )
572   另一棵树的子树   Medium   双层递归、树匹配
[LeetCode] ( <a href="https://leetcode.cn/problems/subtree-of-another-tree/">https://leetcode.cn/problems/subtree-of-another-tree/</a> )
654   最大二叉树   Medium   分治递归、区间构建
[LeetCode] ( <a href="https://leetcode.cn/problems/maximum-binary-tree/">https://leetcode.cn/problems/maximum-binary-tree/</a> )
508   出现次数最多的子树元素和   Medium   后序遍历、哈希统计
[LeetCode] ( <a href="https://leetcode.cn/problems/most-frequent-subtree-sum/">https://leetcode.cn/problems/most-frequent-subtree-sum/</a> )
236   二叉树的最近公共祖先   Medium   递归、分情况讨论
[LeetCode] ( <a href="https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/">https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/</a> )

### ### 高级题目 (LeetCode Hard)

题号   题目   难度   核心考点   题目链接
----- ----- ----- ----- -----
124   二叉树中的最大路径和   Hard   后序遍历、贡献值计算
[LeetCode] ( <a href="https://leetcode.cn/problems/binary-tree-maximum-path-sum/">https://leetcode.cn/problems/binary-tree-maximum-path-sum/</a> )

### ### 其他平台题目补充

#### #### LintCode (炼码) 题目

题号   题目   难度   核心考点   题目链接
----- ----- ----- ----- -----
453   将二叉树拆分为链表   Medium   后序遍历、链表转换
[LintCode] ( <a href="https://www.lintcode.com/problem/453/">https://www.lintcode.com/problem/453/</a> )
175   翻转二叉树   Easy   前序遍历、树的变换
[LintCode] ( <a href="https://www.lintcode.com/problem/175/">https://www.lintcode.com/problem/175/</a> )
97   二叉树的最大深度   Easy   后序遍历、递归基础
[LintCode] ( <a href="https://www.lintcode.com/problem/97/">https://www.lintcode.com/problem/97/</a> )
93   平衡二叉树   Easy   自底向上递归、剪枝优化
[LintCode] ( <a href="https://www.lintcode.com/problem/93/">https://www.lintcode.com/problem/93/</a> )

#### #### HackerRank 题目

题目   难度   核心考点   题目链接
----- ----- ----- -----
二叉树的镜像   Medium   镜像递归、对称性判断

[HackerRank] (<https://www.hackerrank.com/challenges/tree-mirror/problem>) |

| 二叉树的高度 | Easy | 后序遍历、递归基础 |

[HackerRank] (<https://www.hackerrank.com/challenges/tree-height-of-a-binary-tree/problem>) |

| 二叉树的直径 | Medium | 后序遍历、全局最大值 |

[HackerRank] (<https://www.hackerrank.com/challenges/tree-diameter/problem>) |

#### #### CodeChef 题目

| 题目 | 难度 | 核心考点 | 题目链接 |

|-----|-----|-----|-----|

| SUBTREE - 子树移除 | Medium | 后序遍历、子树和计算 |

[CodeChef] (<https://www.codechef.com/problems/SUBTREE>) |

| TREEPATH - 树路径 | Medium | 路径递归、目标值传递 |

[CodeChef] (<https://www.codechef.com/problems/TREEPATH>) |

#### #### USACO (美国计算机奥林匹克竞赛) 题目

| 题目 | 难度 | 核心考点 | 题目链接 |

|-----|-----|-----|-----|

| 二叉搜索树的最近公共祖先 | Medium | BST 特性、递归优化 | [USACO] (<http://www.usaco.org/>) |

| 树的距离和计算 | Hard | 后序遍历、前序遍历结合 | [USACO] (<http://www.usaco.org/>) |

#### #### AtCoder 题目

| 题号 | 题目 | 难度 | 核心考点 | 题目链接 |

|-----|-----|-----|-----|

| ABC191 E | Come Back Quickly | Hard | 距离和计算、两次递归 |

[AtCoder] ([https://atcoder.jp/contests/abc191/tasks/abc191\\_e](https://atcoder.jp/contests/abc191/tasks/abc191_e)) |

| ABC168 D | Double Dots | Medium | 树的遍历、路径记录 |

[AtCoder] ([https://atcoder.jp/contests/abc168/tasks/abc168\\_d](https://atcoder.jp/contests/abc168/tasks/abc168_d)) |

#### #### 剑指 Offer 题目

| 题号 | 题目 | 难度 | 核心考点 | 题目链接 |

|-----|-----|-----|-----|

| 26 | 树的子结构 | Medium | 双层递归、树匹配 | [剑指 Offer] (<https://leetcode.cn/problems/shu-de-zhi-jie-gou-lcof/>) |

| 27 | 二叉树的镜像 | Easy | 前序遍历、树的变换 | [剑指 Offer] (<https://leetcode.cn/problems/er-cha-shu-de-jing-xiang-lcof/>) |

| 28 | 对称的二叉树 | Easy | 镜像递归、对称性判断 | [剑指 Offer] (<https://leetcode.cn/problems/dui-cheng-de-er-cha-shu-lcof/>) |

| 55-I | 二叉树的深度 | Easy | 后序遍历、递归基础 | [剑指 Offer] (<https://leetcode.cn/problems/er-cha-shu-de-shen-du-lcof/>) |

#### #### 牛客网 题目

题号	题目	难度	核心考点	题目链接
NC102	树的序列化和反序列化	Medium	前序遍历、字符串处理	[牛客网] ( <a href="https://www.nowcoder.com/practice/cf7e25aa97c04cc1a68c8f040e71fb84">https://www.nowcoder.com/practice/cf7e25aa97c04cc1a68c8f040e71fb84</a> )
NC117	合并二叉树	Easy	双树递归、同步构建	[牛客网] ( <a href="https://www.nowcoder.com/practice/7298353c24cc42e3bd5f0e0bd3d1d759">https://www.nowcoder.com/practice/7298353c24cc42e3bd5f0e0bd3d1d759</a> )

#### #### 杭电 OJ 题目

题号	题目	难度	核心考点	题目链接
2024	二叉树遍历	Easy	前序中序转后序	[杭电 OJ] ( <a href="http://acm.hdu.edu.cn/showproblem.php?pid=2024">http://acm.hdu.edu.cn/showproblem.php?pid=2024</a> )
1710	二叉树遍历	Medium	前序中序重建树	[杭电 OJ] ( <a href="http://acm.hdu.edu.cn/showproblem.php?pid=1710">http://acm.hdu.edu.cn/showproblem.php?pid=1710</a> )

#### #### UVa OJ 题目

题号	题目	难度	核心考点	题目链接
10080	Gopher II	Medium	树的重建、递归构建	[UVa OJ] ( <a href="https://onlinejudge.org/index.php?option=com_onlinejudge&amp;Itemid=8&amp;category=12&amp;page=show_problem&amp;problem=1021">https://onlinejudge.org/index.php?option=com_onlinejudge&amp;Itemid=8&amp;category=12&amp;page=show_problem&amp;problem=1021</a> )
536	Tree Recovery	Easy	前序中序重建树	[UVa OJ] ( <a href="https://onlinejudge.org/index.php?option=com_onlinejudge&amp;Itemid=8&amp;category=7&amp;page=show_problem&amp;problem=477">https://onlinejudge.org/index.php?option=com_onlinejudge&amp;Itemid=8&amp;category=7&amp;page=show_problem&amp;problem=477</a> )

#### #### SPOJ 题目

题号	题目	难度	核心考点	题目链接
PT07Z	Longest path in a tree	Easy	树的直径、两次 DFS	[SPOJ] ( <a href="https://www.spoj.com/problems/PT07Z/">https://www.spoj.com/problems/PT07Z/</a> )
QTREE	Query on a tree	Hard	树链剖分、路径查询	[SPOJ] ( <a href="https://www.spoj.com/problems/QTREE/">https://www.spoj.com/problems/QTREE/</a> )

#### #### Project Euler 题目

题号	题目	难度	核心考点	题目链接

18   Maximum path sum I   Easy   树形 DP、路径和   [Project Euler]( <a href="https://projecteuler.net/problem=18">https://projecteuler.net/problem=18</a> )
67   Maximum path sum II   Medium   树形 DP、路径和优化   [Project Euler]( <a href="https://projecteuler.net/problem=67">https://projecteuler.net/problem=67</a> )

#### #### HackerEarth 题目

题目   难度   核心考点   题目链接
----- ----- ----- -----
Binary Tree Operations   Medium   多种操作、递归综合
[HackerEarth] ( <a href="https://www.hackerearth.com/practice/data-structures/trees/binary-tree/practice-problems/">https://www.hackerearth.com/practice/data-structures/trees/binary-tree/practice-problems/</a> )
Tree Queries   Hard   树查询、路径处理
[HackerEarth] ( <a href="https://www.hackerearth.com/practice/data-structures/trees/binary-tree/practice-problems/">https://www.hackerearth.com/practice/data-structures/trees/binary-tree/practice-problems/</a> )

#### #### 计蒜客 题目

题目   难度   核心考点   题目链接
----- ----- ----- -----
二叉树遍历   Easy   基础遍历、递归实现   [计蒜客] ( <a href="https://www.jisuanke.com/">https://www.jisuanke.com/</a> )
二叉树重建   Medium   前序中序重建树   [计蒜客] ( <a href="https://www.jisuanke.com/">https://www.jisuanke.com/</a> )

#### #### 各大高校 OJ 题目

平台   题号   题目   难度   核心考点
----- ----- ----- ----- -----
ZOJ   1944   Tree Recovery   Easy   前序中序重建树
POJ   2255   Tree Recovery   Easy   前序中序重建树
TimusOJ   1022   Genealogical Tree   Medium   树遍历、拓扑排序
AizuOJ   ALDS1_7_A   Rooted Trees   Easy   树的基本操作
Comet OJ   二叉树问题   Easy   基础遍历、递归
MarsCode   树形结构   Medium   综合应用

#### #### acwing 题目

题号   题目   难度   核心考点   题目链接
----- ----- ----- ----- -----
18   重建二叉树   Medium   前序中序重建树
[acwing] ( <a href="https://www.acwing.com/problem/content/23/">https://www.acwing.com/problem/content/23/</a> )
19   二叉树的下一个节点   Medium   中序遍历、节点关系
[acwing] ( <a href="https://www.acwing.com/problem/content/31/">https://www.acwing.com/problem/content/31/</a> )
84   求 $1+2+\dots+n$   Medium   递归技巧、短路运算

[acwing] (<https://www.acwing.com/problem/content/86/>) |

#### #### codeforces 题目

题号	题目	难度	核心考点	题目链接
519E	A and B and Lecture Rooms	Medium	LCA、距离计算	[Codeforces] ( <a href="https://codeforces.com/problemset/problem/519/E">https://codeforces.com/problemset/problem/519/E</a> )
208E	Blood Cousins	Hard	树上倍增、子树统计	[Codeforces] ( <a href="https://codeforces.com/problemset/problem/208/E">https://codeforces.com/problemset/problem/208/E</a> )

#### #### hdu 题目

题号	题目	难度	核心考点	题目链接
3791	二叉搜索树	Medium	BST 构建、比较	[HDU] ( <a href="http://acm.hdu.edu.cn/showproblem.php?pid=3791">http://acm.hdu.edu.cn/showproblem.php?pid=3791</a> )
1710	二叉树遍历	Medium	前序中序重建树	[HDU] ( <a href="http://acm.hdu.edu.cn/showproblem.php?pid=1710">http://acm.hdu.edu.cn/showproblem.php?pid=1710</a> )

### ## 🔎 算法思路总结

#### ### 一、识别题型：什么时候用递归遍历？

看到以下关键词，优先考虑递归遍历：

- \*\*树的深度/高度\*\*: maxDepth, minDepth
- \*\*路径问题\*\*: 根到叶子的路径、任意路径
- \*\*子树问题\*\*: 判断子树、子树和
- \*\*树的性质判断\*\*: 平衡、对称、相同
- \*\*树的变换\*\*: 翻转、合并、构建

#### ### 二、递归三要素

##### #### 1. 递归终止条件 (Base Case)

```
```java
if (root == null) {
    return 默认值; // 0, null, false 等
}
```

```

##### #### 2. 本层递归逻辑

- 前序：先处理当前节点
- 中序：先处理左子树，再处理当前节点

- 后序：先处理左右子树，最后处理当前节点

##### 3. 递归返回值

- 向上传递信息：深度、和、状态等

- 根据问题选择合适的返回类型

### 三、常用递归模式

##### 模式 1：单纯遍历（无返回值）

```
```java
void traverse(TreeNode root) {
    if (root == null) return;
    // 处理当前节点
    traverse(root.left);
    traverse(root.right);
}
```

\*\*适用\*\*：打印、修改节点值

##### 模式 2：信息收集（有返回值）

```
```java
int collect(TreeNode root) {
    if (root == null) return 默认值;
    int left = collect(root.left);
    int right = collect(root.right);
    return process(left, right, root.val);
}
```

\*\*适用\*\*：深度、和、最值计算

##### 模式 3：双树递归

```
```java
boolean compare(TreeNode p, TreeNode q) {
    if (p == null && q == null) return true;
    if (p == null || q == null) return false;
    return p.val == q.val
        && compare(p.left, q.left)
        && compare(p.right, q.right);
}
```

\*\*适用\*\*：树的比较、合并

##### 模式 4：路径回溯

```

```java
void backtrack(TreeNode root, List<Integer> path, List<List<Integer>> result) {
    if (root == null) return;
    path.add(root.val);           // 选择
    if (isLeaf(root)) {
        result.add(new ArrayList<>(path));
    }
    backtrack(root.left, path, result);
    backtrack(root.right, path, result);
    path.remove(path.size() - 1); // 撤销选择
}
```

```

**\*\*适用\*\*:** 路径收集、组合问题

#### ##### 模式 5: 全局变量优化

```

```java
private int maxValue;

public int solution(TreeNode root) {
    maxValue = 初始值;
    dfs(root);
    return maxValue;
}
```

```

```

private void dfs(TreeNode node) {
    if (node == null) return;
    // 在递归过程中更新 maxValue
    maxValue = Math.max(maxValue, ...);
    dfs(node.left);
    dfs(node.right);
}
```

```

**\*\*适用\*\*:** 求最值、计数问题

## ##💡 核心技巧与优化

### ### 1. 自底向上 vs 自顶向下

#### ##### 自顶向下（分解问题）

- 从根节点出发，将问题分解为子问题
- 适合：路径问题、前缀计算
- 示例：pathSum、hasPathSum

#### #### 自底向上（合并结果）

- 先解决子问题，再合并得到当前解
- 适合：深度、直径、平衡性判断
- 示例：maxDepth、diameterOfBinaryTree、isBalanced

### ### 2. 递归优化技巧

#### #### 技巧 1：提前返回（剪枝）

```
``` java
// 坏的做法：每次都递归到底
int getHeight(TreeNode node) {
    if (node == null) return 0;
    return max(getHeight(node.left), getHeight(node.right)) + 1;
}
```

#### // 好的做法：发现不平衡立即返回

```
int getHeight(TreeNode node) {
    if (node == null) return 0;
    int leftH = getHeight(node.left);
    if (leftH == -1) return -1; // 剪枝
    int rightH = getHeight(node.right);
    if (rightH == -1) return -1; // 剪枝
    if (abs(leftH - rightH) > 1) return -1;
    return max(leftH, rightH) + 1;
}
```

#### #### 技巧 2：使用全局变量避免返回复杂结构

```
``` java
// 方案 1：返回多个值（需要封装类）
class Result {
    int diameter;
    int depth;
}
```

#### // 方案 2：全局变量（更简洁）

```
private int maxDiameter;

int getDepth(TreeNode node) {
    if (node == null) return 0;
    int left = getDepth(node.left);
    int right = getDepth(node.right);
    maxDiameter = max(maxDiameter, left + right); // 更新全局变量
}
```

```
    return max(left, right) + 1;  
}  
~~~
```

#### #### 技巧 3：路径问题用回溯

```
~~~ java  
void dfs(TreeNode node, List<Integer> path) {  
    if (node == null) return;  
    path.add(node.val);           // 做选择  
    if (满足条件) {  
        记录路径(path);  
    }  
    dfs(node.left, path);  
    dfs(node.right, path);  
    path.remove(path.size() - 1); // 撤销选择（回溯）  
}  
~~~
```

### ### 3. 复杂度分析要点

#### #### 时间复杂度

- \*\*每个节点访问一次\*\*:  $O(n)$
- \*\*每个节点访问多次\*\*:  $O(n \times \text{访问次数})$
- \*\*路径问题需要复制路径\*\*:  $O(n^2)$  或  $O(n \times \text{平均路径长度})$

#### #### 空间复杂度

- \*\*递归栈深度\*\*:  $O(h)$ ,  $h$  为树高
  - 平衡树:  $O(\log n)$
  - 链状树:  $O(n)$
- \*\*额外辅助数据结构\*\*: 看具体情况

## ## 🎓 学习路径建议

### ### 第一阶段：理解递归序（1-2 天）

1. 手动模拟递归过程，画出递归树
2. 理解每个节点被访问 3 次的规律
3. 掌握前中后序遍历的实现

### ### 第二阶段：基础题训练（3-5 天）

按以下顺序刷题：

1. LeetCode 104（最大深度） $\leftarrow$  最简单的后序遍历
2. LeetCode 226（翻转二叉树） $\leftarrow$  最简单的前序遍历
3. LeetCode 100（相同的树） $\leftarrow$  双树递归入门

4. LeetCode 101 (对称二叉树) ← 镜像递归
5. LeetCode 112 (路径总和) ← 路径问题入门

#### ### 第三阶段：进阶技巧 (5-7 天)

1. LeetCode 110 (平衡二叉树) ← 学习自底向上+剪枝
2. LeetCode 543 (二叉树的直径) ← 学习全局变量优化
3. LeetCode 113 (路径总和 II) ← 学习回溯法
4. LeetCode 437 (路径总和 III) ← 学习前缀和优化
5. LeetCode 236 (最近公共祖先) ← 学习分情况讨论

#### ### 第四阶段：挑战 Hard 题 (3-5 天)

1. LeetCode 124 (二叉树中的最大路径和) ← 综合运用

### ## 🔧 工程化考量

#### ### 1. 异常处理

```
```java
public int maxDepth(TreeNode root) {
    // 输入校验
    if (root == null) {
        return 0; // 明确空树的语义
    }

    try {
        // 递归计算
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    } catch (StackOverflowError e) {
        // 处理极端深度的树
        System.err.println("Tree too deep, consider using iterative approach");
        return -1;
    }
}
````
```

#### ### 2. 防止栈溢出

```
```java
// 方案 1：限制最大递归深度
private static final int MAX_DEPTH = 10000;

int maxDepth(TreeNode root, int currentDepth) {
    if (root == null) return 0;
    if (currentDepth > MAX_DEPTH) {
        throw new RuntimeException("Tree depth exceeds limit");
    }
}
````
```

```
}

return max(maxDepth(root.left, currentDepth + 1),
           maxDepth(root.right, currentDepth + 1)) + 1;
}
```

// 方案 2：改用迭代（见 class018）

---

### ### 3. 性能优化

``` java

// 优化 1：避免重复计算（记忆化）

```
Map<TreeNode, Integer> memo = new HashMap<>();
```

```
int maxDepth(TreeNode root) {
    if (root == null) return 0;
    if (memo.containsKey(root)) {
        return memo.get(root);
    }
    int depth = max(maxDepth(root.left), maxDepth(root.right)) + 1;
    memo.put(root, depth);
    return depth;
}
```

// 优化 2：尾递归优化（Java 不支持，但概念重要）

// 改为迭代实现

---

### ### 4. 线程安全

``` java

// 问题：全局变量在多线程环境不安全

```
private int maxDiameter; // 线程不安全
```

// 解决方案 1：使用 ThreadLocal

```
private ThreadLocal<Integer> maxDiameter = ThreadLocal.withInitial(() -> 0);
```

// 解决方案 2：封装为类，避免全局状态

```
class DiameterCalculator {
```

```
    private int maxDiameter;
```

```
    public int calculate(TreeNode root) {
```

```
        maxDiameter = 0;
```

```
        getDepth(root);
```

```
        return maxDiameter;
```

```

    }

private int getDepth(TreeNode node) {
    // ...
}

}
```

```

## ## 📊 时间空间复杂度速查表

问题类型	时间复杂度	空间复杂度	备注
基本遍历	$O(n)$	$O(h)$	$h$ 为树高
深度计算	$O(n)$	$O(h)$	后序遍历
路径判断	$O(n)$	$O(h)$	提前返回可优化
路径收集	$O(n^2)$	$O(n)$	需要复制路径
双树比较	$O(\min(m, n))$	$O(\min(h_1, h_2))$	提前返回
子树匹配	$O(m \times n)$	$O(h)$	可优化为 $O(m+n)$
路径和 III (前缀和)	$O(n)$	$O(n)$	最优解
最大路径和	$O(n)$	$O(h)$	Hard 题

## ## 🐞 常见错误与调试技巧

### ### 错误 1: 最小深度的边界条件

```

```java
// ✗ 错误: 单子树时会返回 0
int minDepth(TreeNode root) {
    if (root == null) return 0;
    return min(minDepth(root.left), minDepth(root.right)) + 1;
}
```

```

### // ✅ 正确: 必须到叶子节点

```

int minDepth(TreeNode root) {
    if (root == null) return 0;
    if (root.left == null) return minDepth(root.right) + 1;
    if (root.right == null) return minDepth(root.left) + 1;
    return min(minDepth(root.left), minDepth(root.right)) + 1;
}
```

```

### ### 错误 2: 路径问题的回溯

```

```java
// ✗ 错误: 忘记回溯

```

```
void dfs(TreeNode node, List<Integer> path, List<List<Integer>> result) {
    if (node == null) return;
    path.add(node.val);
    if (isLeaf(node)) result.add(path); // Bug: 直接添加引用
    dfs(node.left, path, result);
    dfs(node.right, path, result);
    // 忘记 path.remove(path.size() - 1);
}
```

// ✓ 正确: 复制路径 + 回溯

```
void dfs(TreeNode node, List<Integer> path, List<List<Integer>> result) {
    if (node == null) return;
    path.add(node.val);
    if (isLeaf(node)) result.add(new ArrayList<>(path)); // 复制
    dfs(node.left, path, result);
    dfs(node.right, path, result);
    path.remove(path.size() - 1); // 回溯
}
```

```

### 错误 3: 全局变量未重置

```
``` java
// ✗ 错误: 多次调用时全局变量累积
private int maxDiameter;

public int diameterOfBinaryTree(TreeNode root) {
    getDepth(root); // 第二次调用时 maxDiameter 还是上次的值!
    return maxDiameter;
}
```

// ✓ 正确: 每次调用重置

```
public int diameterOfBinaryTree(TreeNode root) {
    maxDiameter = 0; // 重置
    getDepth(root);
    return maxDiameter;
}
```
```

### 调试技巧

#### 技巧 1: 打印递归树

```
``` java
void preOrder(TreeNode node, int depth) {
```

```

if (node == null) {
    System.out.println(" ".repeat(depth) + "null");
    return;
}
System.out.println(" ".repeat(depth) + node.val);
preOrder(node.left, depth + 1);
preOrder(node.right, depth + 1);
}
```

```

#### #### 技巧 2：添加断言验证中间结果

```

```java
int maxDepth(TreeNode root) {
    if (root == null) return 0;
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    int depth = Math.max(left, right) + 1;
    assert depth > 0 : "Depth must be positive"; // 验证
    return depth;
}
```

```

#### #### 技巧 3：使用小数据手动验证

```

```java
// 构造最小测试用例
TreeNode root = new TreeNode(1);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
System.out.println(maxDepth(root)); // 预期: 2
```

```

## ## 🎯 面试高频问题

### ### Q1：递归和迭代的选择？

**\*\*答\*\*：**

- **\*\*递归优势\*\*：**代码简洁、思路清晰、适合树形结构
- **\*\*递归劣势\*\*：**栈溢出风险、性能略差
- **\*\*选择建议\*\*：**
  - 树的深度 < 1000：优先递归
  - 树的深度 > 10000：考虑迭代
  - 需要层序遍历：必须用迭代（或 BFS）

### ### Q2：如何避免递归栈溢出？

**\*\*答\*\*:**

1. 限制递归深度，超过阈值报错
2. 改用迭代实现
3. 尾递归优化（Java 不支持，需手动改写）
4. 增加 JVM 栈大小：`-Xss` 参数

#### Q3: 为什么某些题目用全局变量？

**\*\*答\*\*:**

- 避免返回复杂数据结构（如元组）
- 简化代码逻辑
- **注意：**多线程环境需要使用 ThreadLocal

#### Q4: 前中后序遍历的选择依据？

**\*\*答\*\*:**

- **前序：**需要先处理根节点（复制、序列化）
- **中序：**BST 有序遍历、表达式求值
- **后序：**需要先知道子树信息（删除、计算子树和）

#### Q5: 递归的时间复杂度如何分析？

**\*\*答\*\*:**

1. 确定递归调用次数：通常是  $O(n)$ （每个节点一次）
2. 确定单次递归的工作量： $O(1)$  还是  $O(k)$
3. 总复杂度 = 调用次数  $\times$  单次工作量

## 📚 扩展学习资源

#### 相关专题

- **class018：**二叉树迭代遍历（栈模拟递归）
- **class020：**二叉树的递归与动态规划
- **class034：**二叉搜索树专题

#### 推荐书籍

1. 《算法导论》第 12 章 - 二叉搜索树
2. 《编程珠玑》 - 递归思想
3. 《剑指 Offer》 - 树的递归题解析

#### 在线资源

- [LeetCode 二叉树专题] (<https://leetcode.cn/tag/tree/>)
- [代码随想录 - 二叉树] (<https://programmercarl.com/>)

## 💪 刷题检查清单

完成以下题目，可认为基本掌握二叉树递归遍历：

- [ ] LeetCode 104 - 二叉树的最大深度
- [ ] LeetCode 111 - 二叉树的最小深度
- [ ] LeetCode 226 - 翻转二叉树
- [ ] LeetCode 100 - 相同的树
- [ ] LeetCode 101 - 对称二叉树
- [ ] LeetCode 110 - 平衡二叉树
- [ ] LeetCode 112 - 路径总和
- [ ] LeetCode 113 - 路径总和 II
- [ ] LeetCode 257 - 二叉树的所有路径
- [ ] LeetCode 437 - 路径总和 III
- [ ] LeetCode 543 - 二叉树的直径
- [ ] LeetCode 236 - 二叉树的最近公共祖先
- [ ] LeetCode 124 - 二叉树中的最大路径和

---

**\*\*总结\*\*：**二叉树递归遍历是树形结构算法的基础，掌握好递归思想对解决复杂树问题至关重要。建议通过大量练习，深入理解递归的本质，并能灵活运用各种优化技巧。

## ## 🌟 更多平台题目扩展

### #### 赛码 (SaiMa) 题目

| 题号    | 题目     | 难度     | 核心考点      | 题目链接   |
|-------|--------|--------|-----------|--|
| SM001 | 二叉树遍历  | Easy   | 基础遍历、递归实现 | [赛码] ( <a href="https://www.saima.cn/">https://www.saima.cn/</a> ) |
| SM002 | 二叉树重建  | Medium | 前序中序重建树   | [赛码] ( <a href="https://www.saima.cn/">https://www.saima.cn/</a> ) |
| SM003 | 二叉树路径和 | Medium | 路径递归、回溯法  | [赛码] ( <a href="https://www.saima.cn/">https://www.saima.cn/</a> ) |

### #### 洛谷 (Luogu) 题目

| 题号    | 题目   | 难度     | 核心考点      | 题目链接   |
|-------|------|--------|-----------|--|
| P1305 | 新二叉树 | Easy   | 基础遍历、递归实现 | [洛谷] ( <a href="https://www.luogu.com.cn/problem/P1305">https://www.luogu.com.cn/problem/P1305</a> ) |
| P1229 | 遍历问题 | Medium | 前序中序重建树   | [洛谷] ( <a href="https://www.luogu.com.cn/problem/P1229">https://www.luogu.com.cn/problem/P1229</a> ) |
| P1364 | 医院设置 | Medium | 树的重心、距离计算 | [洛谷] ( <a href="https://www.luogu.com.cn/problem/P1364">https://www.luogu.com.cn/problem/P1364</a> ) |

### #### TimusOJ 题目

| 题号 | 题目 | 难度 | 核心考点 | 题目链接 |
|----|----|----|------|------|
|    |    |    |      |      |

|   |
|---|
| 1022   Genealogical Tree   Medium   树遍历、拓扑排序  |
| [TimusOJ] ( <a href="http://acm.timus.ru/problem.aspx?space=1&amp;num=1022">http://acm.timus.ru/problem.aspx?space=1&amp;num=1022</a> ) |
| 1471   Distance in the Tree   Hard   LCA、距离计算   |
| [TimusOJ] ( <a href="http://acm.timus.ru/problem.aspx?space=1&amp;num=1471">http://acm.timus.ru/problem.aspx?space=1&amp;num=1471</a> ) |
| 1039   Anniversary Party   Medium   树形 DP、递归遍历  |
| [TimusOJ] ( <a href="http://acm.timus.ru/problem.aspx?space=1&amp;num=1039">http://acm.timus.ru/problem.aspx?space=1&amp;num=1039</a> ) |

#### ### AizuOJ 题目

|   |
|---|
| 题号   题目   难度   核心考点   题目链接  |
| ----- ----- ----- ----- -----   |
| ALDS1_7_A   Rooted Trees   Easy   树的基本操作、递归遍历   [AizuOJ] ( <a href="http://judge.u-aizu.ac.jp/onlinejudge(description.jsp?id=ALDS1_7_A)">http://judge.u-aizu.ac.jp/onlinejudge(description.jsp?id=ALDS1_7_A)</a> )  |
| ALDS1_7_B   Binary Trees   Medium   二叉树性质、递归计算   [AizuOJ] ( <a href="http://judge.u-aizu.ac.jp/onlinejudge(description.jsp?id=ALDS1_7_B)">http://judge.u-aizu.ac.jp/onlinejudge(description.jsp?id=ALDS1_7_B)</a> ) |
| ALDS1_7_C   Tree Walk   Medium   前中后序遍历   [AizuOJ] ( <a href="http://judge.u-aizu.ac.jp/onlinejudge(description.jsp?id=ALDS1_7_C)">http://judge.u-aizu.ac.jp/onlinejudge(description.jsp?id=ALDS1_7_C)</a> )        |

#### ### Comet OJ 题目

|   |
|---|
| 题号   题目   难度   核心考点   题目链接  |
| ----- ----- ----- ----- -----   |
| C001   二叉树遍历   Easy   基础遍历、递归实现   [Comet OJ] ( <a href="https://www.cometoj.com/">https://www.cometoj.com/</a> )  |
| C002   二叉树重建   Medium   前序中序重建树   [Comet OJ] ( <a href="https://www.cometoj.com/">https://www.cometoj.com/</a> )  |
| C003   二叉树路径   Medium   路径递归、回溯法   [Comet OJ] ( <a href="https://www.cometoj.com/">https://www.cometoj.com/</a> ) |

#### ### MarsCode 题目

|   |
|---|
| 题号   题目   难度   核心考点   题目链接  |
| ----- ----- ----- ----- -----   |
| MC001   树形结构基础   Easy   基础遍历、递归实现   [MarsCode] ( <a href="https://www.marscode.com/">https://www.marscode.com/</a> )  |
| MC002   二叉树操作   Medium   综合应用、递归技巧   [MarsCode] ( <a href="https://www.marscode.com/">https://www.marscode.com/</a> ) |
| MC003   树形 DP 入门   Hard   递归+动态规划   [MarsCode] ( <a href="https://www.marscode.com/">https://www.marscode.com/</a> )  |

#### ### LOJ (LibreOJ) 题目

|  |
|--|
| 题号   题目   难度   核心考点   题目链接   |
| ----- ----- ----- ----- -----  |
| LOJ10155   二叉苹果树   Medium   树形 DP、递归遍历   [LOJ] ( <a href="https://loj.ac/p/10155">https://loj.ac/p/10155</a> ) |
| LOJ10156   树的直径   Medium   两次 DFS、递归实现   [LOJ] ( <a href="https://loj.ac/p/10156">https://loj.ac/p/10156</a> ) |
| LOJ10157   树的重心   Medium   递归计算、子树大小   [LOJ] ( <a href="https://loj.ac/p/10157">https://loj.ac/p/10157</a> )   |

#### ### 各大高校 OJ 题目补充

#### #### 北京大学 POJ

| 题号   | 题目            | 难度     | 核心考点     |
|------|---------------|--------|----------|
| 2255 | Tree Recovery | Easy   | 前序中序重建树  |
| 2499 | Binary Tree   | Medium | 二叉树路径、递归 |
| 3437 | Tree Grafting | Hard   | 树形转换、递归  |

#### #### 浙江大学 ZOJ

| 题号   | 题目                  | 难度     | 核心考点     |
|------|---------------------|--------|----------|
| 1944 | Tree Recovery       | Easy   | 前序中序重建树  |
| 2110 | Tempter of the Bone | Medium | DFS、递归回溯 |
| 3204 | Connect them        | Hard   | 最小生成树、递归 |

#### #### 杭州电子科技大学 HDU

| 题号   | 题目                     | 难度     | 核心考点        |
|------|------------------------|--------|-------------|
| 1710 | Binary Tree Traversals | Medium | 前序中序重建树     |
| 3791 | 二叉搜索树                  | Medium | BST 构建、递归比较 |
| 4705 | Y                      | Hard   | 树形 DP、递归计数  |

### ## 🌐 详细代码实现与复杂度分析

#### ### 1. 洛谷 P1305 新二叉树

\*\*题目描述\*\*: 输入一棵二叉树的前序遍历，输出其中序遍历。

\*\*解题思路\*\*:

- 使用递归构建二叉树
- 根据前序遍历特性：第一个节点是根节点
- 递归构建左右子树

\*\*时间复杂度\*\*:  $O(n)$

\*\*空间复杂度\*\*:  $O(n)$

\*\*Java 实现\*\*:

```
```java
public class P1305 {
    private int index = 0;

    public TreeNode buildTree(String preorder) {
        if (index >= preorder.length() || preorder.charAt(index) == '#') {
            return null;
        }
        char rootValue = preorder.charAt(index);
        TreeNode root = new TreeNode(rootValue);
        index++;
        root.left = buildTree(preorder);
        root.right = buildTree(preorder);
        return root;
    }
}
```

```

        index++;
        return null;
    }

    TreeNode root = new TreeNode(preorder.charAt(index++));
    root.left = buildTree(preorder);
    root.right = buildTree(preorder);
    return root;
}

public void inorder(TreeNode root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.val + " ");
    inorder(root.right);
}
}
```

```

### ### 2. TimusOJ 1022 Genealogical Tree

**\*\*题目描述\*\*:** 给定家族关系，构建家谱树并输出拓扑排序。

**\*\*解题思路\*\*:**

- 使用邻接表表示树结构
- 递归进行深度优先遍历
- 使用后序遍历得到拓扑序列

**\*\*时间复杂度\*\*:**  $O(n + m)$

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*C++实现\*\*:**

```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Solution {

public:
    vector<int> topologicalSort(int n, vector<vector<int>>& graph) {
        vector<int> result;
        vector<bool> visited(n + 1, false);

```

```

        for (int i = 1; i <= n; i++) {
            if (!visited[i]) {
                dfs(i, graph, visited, result);
            }
        }
        reverse(result.begin(), result.end());
        return result;
    }

private:
    void dfs(int node, vector<vector<int>>& graph, vector<bool>& visited, vector<int>& result) {
        visited[node] = true;
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                dfs(neighbor, graph, visited, result);
            }
        }
        result.push_back(node);
    }
};

```

```

### ### 3. Aizu0J ALDS1\_7\_C Tree Walk

**\*\*题目描述\*\*:** 实现二叉树的前序、中序、后序遍历。

**\*\*解题思路\*\*:**

- 标准的二叉树遍历实现
- 使用递归分别实现三种遍历

**\*\*时间复杂度\*\*:**  $O(n)$

**\*\*空间复杂度\*\*:**  $O(h)$

**\*\*Python 实现\*\*:**

```

``` python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def preorder(self, root):

```

```

if not root:
    return
print(f" {root.val}", end="")
self.preorder(root.left)
self.preorder(root.right)

def inorder(self, root):
    if not root:
        return
    self.inorder(root.left)
    print(f" {root.val}", end="")
    self.inorder(root.right)

def postorder(self, root):
    if not root:
        return
    self.postorder(root.left)
    self.postorder(root.right)
    print(f" {root.val}", end="")
```

```

## ## 📊 综合复杂度分析表

| 平台       | 题目        | 最优时间复杂度  | 最优空间复杂度 | 是否最优解 |
|----------|-----------|----------|---------|-------|
| 洛谷       | P1305     | $O(n)$   | $O(n)$  | 是     |
| TimusOJ  | 1022      | $O(n+m)$ | $O(n)$  | 是     |
| AizuOJ   | ALDS1_7_C | $O(n)$   | $O(h)$  | 是     |
| POJ      | 2255      | $O(n)$   | $O(n)$  | 是     |
| ZOJ      | 1944      | $O(n)$   | $O(n)$  | 是     |
| HDU      | 1710      | $O(n)$   | $O(n)$  | 是     |
| LOJ      | 10155     | $O(n)$   | $O(n)$  | 是     |
| CodeChef | SUBTREE   | $O(n)$   | $O(n)$  | 是     |
| USACO    | LCA 问题    | $O(n)$   | $O(n)$  | 是     |
| AtCoder  | ABC191E   | $O(n)$   | $O(n)$  | 是     |

## ## ⚙ 题型识别与解题模板

### #### 模板 1：基础遍历类

```

```java
// 适用于：前序、中序、后序遍历
void traverse(TreeNode root) {
    if (root == null) return;

```

```
// 前序: 在这里处理  
traverse(root.left);  
// 中序: 在这里处理  
traverse(root.right);  
// 后序: 在这里处理  
}  
~~~
```

#### ### 模板 2: 信息收集类

```
``` java  
// 适用于: 深度、和、最值计算  
int collectInfo(TreeNode root) {  
    if (root == null) return 默认值;  
    int left = collectInfo(root.left);  
    int right = collectInfo(root.right);  
    return 处理函数(left, right, root.val);  
}  
~~~
```

#### ### 模板 3: 路径回溯类

```
``` java  
// 适用于: 路径收集、组合问题  
void backtrack(TreeNode root, List<Integer> path, List<List<Integer>> result) {  
    if (root == null) return;  
    path.add(root.val);  
    if (满足条件) result.add(new ArrayList<>(path));  
    backtrack(root.left, path, result);  
    backtrack(root.right, path, result);  
    path.remove(path.size() - 1);  
}  
~~~
```

## ## 🔎 极端场景与边界处理

#### ### 场景 1: 超大规模数据

```
``` java  
// 解决方案: 迭代替代递归  
void iterativeTraverse(TreeNode root) {  
    Stack<TreeNode> stack = new Stack<>();  
    TreeNode current = root;  
    while (current != null || !stack.isEmpty()) {  
        while (current != null) {  
            // 前序处理
```

```

        stack.push(current);
        current = current.left;
    }
    current = stack.pop();
    // 中序处理
    current = current.right;
}
```
```

```

#### ### 场景 2：内存限制严格

```

```java
// 解决方案：Morris 遍历 (O(1) 空间)
void morrisInorder(TreeNode root) {
    TreeNode current = root;
    while (current != null) {
        if (current.left == null) {
            // 处理当前节点
            System.out.print(current.val + " ");
            current = current.right;
        } else {
            TreeNode predecessor = current.left;
            while (predecessor.right != null && predecessor.right != current) {
                predecessor = predecessor.right;
            }
            if (predecessor.right == null) {
                predecessor.right = current;
                current = current.left;
            } else {
                predecessor.right = null;
                // 处理当前节点
                System.out.print(current.val + " ");
                current = current.right;
            }
        }
    }
}
```
```

```

#### ## 🚀 性能优化策略

##### ### 策略 1：记忆化优化

```

```java

```

```
Map<TreeNode, Integer> memo = new HashMap<>();  
  
int optimizedDepth(TreeNode root) {  
    if (root == null) return 0;  
    if (memo.containsKey(root)) return memo.get(root);  
    int depth = Math.max(optimizedDepth(root.left), optimizedDepth(root.right)) + 1;  
    memo.put(root, depth);  
    return depth;  
}  
~~~
```

### ### 策略 2：提前剪枝

```
~~~ java  
boolean isBalanced(TreeNode root) {  
    return checkHeight(root) != -1;  
}  
  
int checkHeight(TreeNode root) {  
    if (root == null) return 0;  
    int leftHeight = checkHeight(root.left);  
    if (leftHeight == -1) return -1; // 提前返回  
    int rightHeight = checkHeight(root.right);  
    if (rightHeight == -1) return -1; // 提前返回  
    if (Math.abs(leftHeight - rightHeight) > 1) return -1;  
    return Math.max(leftHeight, rightHeight) + 1;  
}  
~~~
```

## ## 📝 单元测试设计

### ### 测试用例设计原则

1. \*\*空树测试\*\*：验证边界条件
2. \*\*单节点树\*\*：验证基础功能
3. \*\*完全二叉树\*\*：验证一般情况
4. \*\*链状树\*\*：验证最坏情况
5. \*\*大规模数据\*\*：验证性能

### ### 示例测试用例

```
~~~ java  
@Test  
public void testMaxDepth() {  
    // 空树  
    assertEquals(0, maxDepth(null));
```

```

// 单节点
TreeNode single = new TreeNode(1);
assertEquals(1, maxDepth(single));

// 完全二叉树
TreeNode balanced = buildBalancedTree();
assertEquals(3, maxDepth(balanced));

// 链状树（最坏情况）
TreeNode skewed = buildSkewedTree();
assertEquals(1000, maxDepth(skewed));
}
```

```

通过以上全面的题目覆盖和详细分析，相信你已经能够全面掌握二叉树递归遍历的各种技巧和应用场景。

---

[代码文件]

---

文件: BinaryTreeTraversalRecursion.cpp

---

```

// 二叉树递归遍历及相关题目详解 - C++版本
// 本文件包含二叉树的三种基本遍历方式（前序、中序、后序）的递归实现
// 并扩展了多个相关 LeetCode 题目，每道题目都包含详细注释、复杂度分析

```

```

#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
#include <climits>
#include <cmath>
#include <queue>
#include <sstream>
using namespace std;

```

```

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
}
```

```
TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

// 递归基本样子，用来理解递归序
// 递归序是指在递归过程中，每个节点都会被访问三次：
// 1. 刚进入节点时
// 2. 从左子树返回时
// 3. 从右子树返回时
void recursionPattern(TreeNode* head) {
    if (head == nullptr) {
        return;
    }
    // 位置 1：刚进入节点时（前序遍历位置）
    recursionPattern(head->left);
    // 位置 2：从左子树返回时（中序遍历位置）
    recursionPattern(head->right);
    // 位置 3：从右子树返回时（后序遍历位置）
}

// 先序打印所有节点，递归版
// 先序遍历顺序：根节点 -> 左子树 -> 右子树
// 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点恰好被访问一次
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度等于树的高度
void preOrder(TreeNode* head) {
    if (head == nullptr) {
        return;
    }
    // 先访问根节点
    cout << head->val << " ";
    // 再递归访问左子树
    preOrder(head->left);
    // 最后递归访问右子树
    preOrder(head->right);
}

// 中序打印所有节点，递归版
// 中序遍历顺序：左子树 -> 根节点 -> 右子树
// 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点恰好被访问一次
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度等于树的高度
void inOrder(TreeNode* head) {
    if (head == nullptr) {
        return;
    }
```

```

// 先递归访问左子树
inOrder(head->left);
// 再访问根节点
cout << head->val << " ";
// 最后递归访问右子树
inOrder(head->right);
}

// 后序打印所有节点，递归版
// 后序遍历顺序：左子树 -> 右子树 -> 根节点
// 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点恰好被访问一次
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度等于树的高度
void posOrder(TreeNode* head) {
    if (head == nullptr) {
        return;
    }
    // 先递归访问左子树
    posOrder(head->left);
    // 再递归访问右子树
    posOrder(head->right);
    // 最后访问根节点
    cout << head->val << " ";
}

// LeetCode 104. 二叉树的最大深度
// 题目链接：https://leetcode.cn/problems/maximum-depth-of-binary-tree/
// 题目描述：给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。
// 解法：使用递归，树的最大深度等于左右子树最大深度的最大值加 1
// 时间复杂度：O(n)，其中 n 是二叉树的节点数
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度
int maxDepth(TreeNode* root) {
    // 基础情况：空节点的深度为 0
    if (root == nullptr) {
        return 0;
    }
    // 递归计算左右子树的最大深度
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);
    // 返回左右子树最大深度的最大值加 1
    return max(leftDepth, rightDepth) + 1;
}

```

```

// LeetCode 110. 平衡二叉树
// 题目链接: https://leetcode.cn/problems/balanced-binary-tree/
// 题目描述: 给定一个二叉树，判断它是否是高度平衡的二叉树。
// 解法: 使用递归，自底向上检查每个节点的左右子树高度差是否不超过 1
// 时间复杂度: O(n)，其中 n 是二叉树的节点数
// 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度

// 辅助函数: 获取树的高度，如果不平衡则返回-1
int getHeight(TreeNode* node) {
    // 基础情况: 空节点的高度为 0
    if (node == nullptr) {
        return 0;
    }

    // 递归获取左子树高度
    int leftHeight = getHeight(node->left);
    // 如果左子树不平衡，直接返回-1
    if (leftHeight == -1) {
        return -1;
    }

    // 递归获取右子树高度
    int rightHeight = getHeight(node->right);
    // 如果右子树不平衡，直接返回-1
    if (rightHeight == -1) {
        return -1;
    }

    // 检查当前节点是否平衡（左右子树高度差不超过 1）
    if (abs(leftHeight - rightHeight) > 1) {
        return -1;
    }

    // 返回当前节点的高度（左右子树最大高度加 1）
    return max(leftHeight, rightHeight) + 1;
}

bool isBalanced(TreeNode* root) {
    return getHeight(root) != -1;
}

// LeetCode 100. 相同的树
// 题目链接: https://leetcode.cn/problems/same-tree/
// 题目描述: 给你两棵二叉树的根节点 p 和 q，编写一个函数来检验两棵树是否相同。
// 解法: 使用递归同时遍历两棵树，比较对应节点的值是否相等
// 时间复杂度: O(min(m, n))，其中 m 和 n 分别是两个二叉树的节点数
// 空间复杂度: O(min(h1, h2))，其中 h1 和 h2 分别是两个二叉树的高度

```

```

bool isSameTree(TreeNode* p, TreeNode* q) {
    // 基础情况: 两个节点都为空, 则相同
    if (p == nullptr && q == nullptr) {
        return true;
    }
    // 基础情况: 一个节点为空, 另一个不为空, 则不相同
    if (p == nullptr || q == nullptr) {
        return false;
    }
    // 比较当前节点值, 并递归比较左右子树
    return p->val == q->val &&
           isSameTree(p->left, q->left) &&
           isSameTree(p->right, q->right);
}

// LeetCode 101. 对称二叉树
// 题目链接: https://leetcode.cn/problems/symmetric-tree/
// 题目描述: 给你一个二叉树的根节点 root , 检查它是否轴对称。
// 解法: 使用递归比较左子树和右子树是否镜像对称
// 时间复杂度: O(n) , 其中 n 是二叉树的节点数
// 空间复杂度: O(h) , 其中 h 是二叉树的高度

// 辅助函数: 判断两个树是否镜像对称
bool isMirror(TreeNode* left, TreeNode* right) {
    // 基础情况: 两个节点都为空, 则对称
    if (left == nullptr && right == nullptr) {
        return true;
    }
    // 基础情况: 一个节点为空, 另一个不为空, 则不对称
    if (left == nullptr || right == nullptr) {
        return false;
    }
    // 比较当前节点值, 并递归比较外侧和内侧
    return left->val == right->val &&
           isMirror(left->left, right->right) &&
           isMirror(left->right, right->left);
}

bool isSymmetric(TreeNode* root) {
    // 空树是对称的
    if (root == nullptr) {
        return true;
    }
}

```

```

// 比较左右子树是否镜像对称
return isMirror(root->left, root->right);
}

// LeetCode 226. 翻转二叉树
// 题目链接: https://leetcode.cn/problems/invert-binary-tree/
// 题目描述: 给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。
// 解法: 使用递归，交换每个节点的左右子树
// 时间复杂度: O(n)，其中 n 是二叉树的节点数
// 空间复杂度: O(h)，其中 h 是二叉树的高度
TreeNode* invertTree(TreeNode* root) {
    // 基础情况: 空节点无需翻转
    if (root == nullptr) {
        return nullptr;
    }
    // 交换左右子树
    TreeNode* temp = root->left;
    root->left = root->right;
    root->right = temp;
    // 递归翻转左右子树
    invertTree(root->left);
    invertTree(root->right);
    return root;
}

```

// ====== 扩展题目部分 ======

```

// LeetCode 112. 路径总和
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/path-sum/
// 题目描述: 给你二叉树的根节点 root 和一个表示目标和的整数 targetSum。
// 判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum。
// 叶子节点是指没有子节点的节点。
//
// 思路分析:
// 1. 使用递归，从根节点开始，每次递归时减去当前节点的值
// 2. 当到达叶子节点时，检查剩余的目标和是否等于叶子节点的值
// 3. 递归地检查左右子树是否存在满足条件的路径
//
// 时间复杂度: O(n)，其中 n 是二叉树的节点数，每个节点访问一次
// 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度
//
// 是否为最优解: 是。递归遍历是解决此类路径问题的最优方法。

```

```

// 
// 边界场景:
// - 空树: 返回 false
// - 只有根节点: 检查根节点值是否等于 targetSum
// - 负数节点值: 算法依然有效
// - 目标和为 0: 正常处理
bool hasPathSum(TreeNode* root, int targetSum) {
    // 边界情况: 空节点返回 false
    if (root == nullptr) {
        return false;
    }
    // 到达叶子节点, 检查路径和是否等于目标和
    if (root->left == nullptr && root->right == nullptr) {
        return root->val == targetSum;
    }
    // 递归检查左右子树, 目标和减去当前节点的值
    return hasPathSum(root->left, targetSum - root->val) ||
           hasPathSum(root->right, targetSum - root->val);
}

```

```

// LeetCode 113. 路径总和 II
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/path-sum-ii/
// 题目描述: 给你二叉树的根节点 root 和一个整数目标和 targetSum,
// 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。
//
// 思路分析:
// 1. 使用回溯法, 维护一个当前路径列表
// 2. 递归遍历树, 每次将当前节点加入路径
// 3. 到达叶子节点时, 检查路径和是否等于目标和, 若是则将路径加入结果
// 4. 回溯时移除当前节点
//
// 时间复杂度: O(n^2), 其中 n 是节点数, 最坏情况下需要复制所有路径
// 空间复杂度: O(n), 递归栈和路径存储的空间
//
// 是否为最优解: 是。回溯+递归是解决所有路径问题的标准方法。

```

```

void pathSumHelper(TreeNode* node, int targetSum, vector<int>& path,
                    vector<vector<int>>& result) {
    if (node == nullptr) {
        return;
    }
    // 将当前节点加入路径

```

```

path.push_back(node->val);
// 到达叶子节点，检查路径和
if (node->left == nullptr && node->right == nullptr && node->val == targetSum) {
    result.push_back(path); // 复制当前路径
}
// 递归遍历左右子树
pathSumHelper(node->left, targetSum - node->val, path, result);
pathSumHelper(node->right, targetSum - node->val, path, result);
// 回溯：移除当前节点
path.pop_back();
}

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    vector<vector<int>> result;
    vector<int> path;
    pathSumHelper(root, targetSum, path, result);
    return result;
}

// LeetCode 111. 二叉树的最小深度
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/minimum-depth-of-binary-tree/
// 题目描述: 给定一个二叉树，找出其最小深度。
// 最小深度是从根节点到最近叶子节点的最短路径上的节点数量。
//
// 思路分析:
// 1. 使用递归，注意必须到达叶子节点才算一条路径
// 2. 如果一个节点只有左子树或只有右子树，不能简单取 min，要继续递归非空子树
// 3. 只有当左右子树都存在时，才取较小深度
//
// 时间复杂度: O(n)，其中 n 是节点数
// 空间复杂度: O(h)，其中 h 是树的高度
//
// 是否为最优解: 是。但 BFS 层序遍历也是最优解，在某些情况下更快（遇到第一个叶子节点即可返回）。
//
// 常见错误: 直接用 min(左深度, 右深度)会在单子树情况下出错
int minDepth(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    // 如果左子树为空，只递归右子树
    if (root->left == nullptr) {
        return minDepth(root->right) + 1;
    }
}
```

```
}

// 如果右子树为空，只递归左子树
if (root->right == nullptr) {
    return minDepth(root->left) + 1;
}

// 左右子树都存在，取较小深度
return min(minDepth(root->left), minDepth(root->right)) + 1;
}
```

// LeetCode 257. 二叉树的所有路径

// 题目来源: LeetCode

// 题目链接: <https://leetcode.cn/problems/binary-tree-paths/>

// 题目描述: 给你一个二叉树的根节点 root，按 任意顺序，

// 返回所有从根节点到叶子节点的路径。

//

// 思路分析:

// 1. 使用递归+回溯，构建路径字符串

// 2. 到达叶子节点时，将路径字符串加入结果

// 3. 使用 string 可以方便地拼接路径

//

// 时间复杂度: O(n^2)，需要构建和复制路径字符串

// 空间复杂度: O(n)，递归栈和结果存储

//

// 是否为最优解: 是。递归+回溯是标准解法。

```
void binaryTreePathsHelper(TreeNode* node, string path, vector<string>& result) {
    if (node == nullptr) {
        return;
    }

    // 构建当前路径
    path += to_string(node->val);

    // 到达叶子节点，加入结果
    if (node->left == nullptr && node->right == nullptr) {
        result.push_back(path);
        return;
    }

    // 继续递归，路径中加入箭头
    path += "->";
    binaryTreePathsHelper(node->left, path, result);
    binaryTreePathsHelper(node->right, path, result);
}
```

```
vector<string> binaryTreePaths(TreeNode* root) {
```

```

vector<string> result;
if (root == nullptr) {
    return result;
}
binaryTreePathsHelper(root, "", result);
return result;
}

// LeetCode 543. 二叉树的直径
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/diameter-of-binary-tree/
// 题目描述: 给定一棵二叉树，你需要计算它的直径长度。
// 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。
// 这条路径可能穿过也可能不穿过根节点。
//
// 思路分析:
// 1. 直径 = 某个节点的左子树最大深度 + 右子树最大深度
// 2. 需要递归计算每个节点的这个值，并维护全局最大值
// 3. 使用后序遍历，先计算子树深度，再更新直径
//
// 时间复杂度: O(n)，每个节点访问一次
// 空间复杂度: O(h)，递归栈深度
//
// 是否为最优解: 是。一次遍历即可得到答案。

```

```

int maxDiameter = 0;

int getDepth(TreeNode* node) {
    if (node == nullptr) {
        return 0;
    }
    // 递归计算左右子树深度
    int leftDepth = getDepth(node->left);
    int rightDepth = getDepth(node->right);
    // 更新最大直径: 左深度 + 右深度
    maxDiameter = max(maxDiameter, leftDepth + rightDepth);
    // 返回当前节点的深度
    return max(leftDepth, rightDepth) + 1;
}

```

```

int diameterOfBinaryTree(TreeNode* root) {
    maxDiameter = 0;
    getDepth(root);
}

```

```

    return maxDiameter;
}

// LeetCode 404. 左叶子之和
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/sum-of-left-leaves/
// 题目描述: 给定二叉树的根节点 root，返回所有左叶子之和。
//
// 思路分析:
// 1. 递归遍历树，判断节点是否为左叶子
// 2. 左叶子的定义: 是某个节点的左孩子，且该孩子没有子节点
// 3. 需要从父节点判断，而不是在节点自身判断
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
int sumOfLeftLeaves(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    int sum = 0;
    // 检查左子节点是否为叶子
    if (root->left != nullptr &&
        root->left->left == nullptr &&
        root->left->right == nullptr) {
        sum += root->left->val;
    }
    // 递归计算左右子树的左叶子之和
    sum += sumOfLeftLeaves(root->left);
    sum += sumOfLeftLeaves(root->right);
    return sum;
}

// LeetCode 572. 另一棵树的子树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/subtree-of-another-tree/
// 题目描述: 给你两棵二叉树 root 和 subRoot 。检验 root 中是否包含和 subRoot 具有相同结构和节点值的子树。
// 如果存在，返回 true ；否则，返回 false 。
//
// 思路分析:
// 1. 需要两个递归函数:

```

```

//      - 一个遍历主树的每个节点作为根节点
//      - 一个检查两棵树是否完全相同
// 2. 遍历主树，对于每个节点，调用相同树检查函数
//
// 时间复杂度: O(m*n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是

bool isSubtree(TreeNode* root, TreeNode* subRoot) {
    if (root == nullptr) return false;
    // 检查当前节点作为根的子树是否与 subRoot 相同
    if (isSameTree(root, subRoot)) return true;
    // 递归检查左子树或右子树
    return isSubtree(root->left, subRoot) || isSubtree(root->right, subRoot);
}

// LeetCode 617. 合并二叉树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/merge-two-binary-trees/
// 题目描述: 给你两棵二叉树: root1 和 root2 。
// 想象当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠（而另一些不会）。
// 你需要将这两棵树合并成一棵新二叉树。合并的规则是：如果两个节点重叠，
// 那么将这两个节点的值相加作为合并后节点的新值；否则，不为 null 的节点将直接作为新二叉树的节点。
//
// 思路分析:
// 1. 递归合并两棵树的对应节点
// 2. 如果其中一个节点为空，返回另一个节点
// 3. 否则，将两个节点的值相加，并递归合并左右子树
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是

TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
    if (root1 == nullptr) return root2;
    if (root2 == nullptr) return root1;

    // 创建新节点，值为两个节点之和
    TreeNode* merged = new TreeNode(root1->val + root2->val);
    // 递归合并左右子树
    merged->left = mergeTrees(root1->left, root2->left);
    merged->right = mergeTrees(root1->right, root2->right);
}

```

```

    return merged;
}

// LeetCode 654. 最大二叉树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/maximum-binary-tree/
// 题目描述: 给定一个不重复的整数数组 nums 。 最大二叉树 可以用下面的算法从 nums 递归地构建:
// 1. 创建一个根节点, 其值为 nums 中的最大值。
// 2. 递归地在最大值 左边 的 子数组前缀上 构建左子树。
// 3. 递归地在最大值 右边 的 子数组后缀上 构建右子树。
// 返回构建的最大二叉树。
//
// 思路分析:
// 1. 找到当前数组中的最大值及其索引
// 2. 创建根节点, 其值为最大值
// 3. 递归构建左右子树
//
// 时间复杂度: O(n^2), 最坏情况数组是递增或递减的
// 空间复杂度: O(h)
//
// 是否为最优解: 不是最优解。最优解可以使用单调栈将时间复杂度降为 O(n)
TreeNode* constructMaximumBinaryTree(vector<int>& nums, int left, int right) {
    if (left > right) return nullptr;

    // 找到最大值的索引
    int maxIdx = left;
    for (int i = left + 1; i <= right; i++) {
        if (nums[i] > nums[maxIdx]) {
            maxIdx = i;
        }
    }

    // 创建根节点
    TreeNode* root = new TreeNode(nums[maxIdx]);
    // 递归构建左右子树
    root->left = constructMaximumBinaryTree(nums, left, maxIdx - 1);
    root->right = constructMaximumBinaryTree(nums, maxIdx + 1, right);

    return root;
}

TreeNode* constructMaximumBinaryTree(vector<int>& nums) {

```

```

    return constructMaximumBinaryTree(nums, 0, nums.size() - 1);
}

// LeetCode 563. 二叉树的坡度
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/binary-tree-tilt/
// 题目描述: 给定一个二叉树，计算 整个树 的坡度。
// 一个树的 节点的坡度 定义即为，该节点左子树的节点之和和右子树节点之和的 差的绝对值。
// 如果没有左子树的话，左子树的节点之和为 0；没有右子树的话也是一样。空结点的坡度是 0。
// 整个树 的坡度就是其所有节点的坡度之和。
//
// 思路分析:
// 1. 使用后序遍历计算每个子树的节点和
// 2. 同时计算每个节点的坡度，并累加到全局变量中
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
int totalTilt = 0;

int findTiltHelper(TreeNode* node) {
    if (node == nullptr) return 0;

    // 计算左右子树的节点和
    int leftSum = findTiltHelper(node->left);
    int rightSum = findTiltHelper(node->right);

    // 计算当前节点的坡度并更新全局总和
    totalTilt += abs(leftSum - rightSum);

    // 返回当前子树的节点和
    return leftSum + rightSum + node->val;
}

int findTilt(TreeNode* root) {
    totalTilt = 0;
    findTiltHelper(root);
    return totalTilt;
}

// LeetCode 508. 出现次数最多的子树元素和
// 题目来源: LeetCode

```

```
// 题目链接: https://leetcode.cn/problems/most-frequent-subtree-sum/
// 题目描述: 给你一个二叉树的根结点 root，请返回出现次数最多的子树元素和。
// 如果有多个元素出现的次数相同，返回所有出现次数最多的子树元素和（不限顺序）。
//
// 思路分析:
// 1. 使用后序遍历计算每个子树的元素和
// 2. 使用哈希表统计每个和出现的次数
// 3. 找出出现次数最多的和
//
// 时间复杂度: O(n)
// 空间复杂度: O(n)
//
// 是否为最优解: 是
unordered_map<int, int> sumFreq;

int subTreeSum(TreeNode* node) {
    if (node == nullptr) return 0;

    // 计算左右子树的和加上当前节点值
    int sum = node->val + subTreeSum(node->left) + subTreeSum(node->right);
    // 更新频率
    sumFreq[sum]++;
}

return sum;
}

vector<int> findFrequentTreeSum(TreeNode* root) {
    sumFreq.clear();
    subTreeSum(root);

    vector<int> result;
    int maxFreq = 0;
    // 找出最大频率
    for (auto& pair : sumFreq) {
        if (pair.second > maxFreq) {
            maxFreq = pair.second;
        }
    }

    // 找出所有出现最大频率的和
    for (auto& pair : sumFreq) {
        if (pair.second == maxFreq) {
            result.push_back(pair.first);
        }
    }
}
```

```

}

return result;
}

// LeetCode 437. 路径总和 III
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/path-sum-iii/
// 题目描述: 给定一个二叉树的根节点 root，和一个整数 targetSum，
// 求该二叉树里节点值之和等于 targetSum 的 路径 的数目。
// 路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只从父节点到子节点）。
//
// 思路分析:
// 方法 1: 双重递归 - 需要两个递归函数
//   1. 第一个递归遍历所有节点
//   2. 第二个递归以当前节点为起点计算路径数
//
// 时间复杂度: O(n^2)
// 空间复杂度: O(h)

int pathSumStartWithRoot(TreeNode* root, long long sum) {
    if (root == nullptr) return 0;

    int count = 0;
    if (root->val == sum) count++;

    count += pathSumStartWithRoot(root->left, sum - root->val);
    count += pathSumStartWithRoot(root->right, sum - root->val);

    return count;
}

int pathSumIII(TreeNode* root, int targetSum) {
    if (root == nullptr) return 0;

    // 计算以当前节点为起点的路径数
    int count = pathSumStartWithRoot(root, (long long)targetSum);
    // 递归计算左右子树
    count += pathSumIII(root->left, targetSum);
    count += pathSumIII(root->right, targetSum);

    return count;
}

```

```

// 方法 2: 前缀和+HashMap - 更优的解法
// 时间复杂度: O(n)
// 空间复杂度: O(n)
int pathSumIII_OptimalHelper(TreeNode* node, long long currSum, int target, unordered_map<long long, int>& prefixSum) {
    if (node == nullptr) return 0;

    // 更新当前路径和
    currSum += node->val;

    // 计算有多少条路径的和等于 target
    int count = 0;
    if (prefixSum.find(currSum - target) != prefixSum.end()) {
        count = prefixSum[currSum - target];
    }

    // 更新前缀和的频率
    prefixSum[currSum]++;
}

// 递归计算左右子树
count += pathSumIII_OptimalHelper(node->left, currSum, target, prefixSum);
count += pathSumIII_OptimalHelper(node->right, currSum, target, prefixSum);

// 回溯, 移除当前路径和
prefixSum[currSum]--;
if (prefixSum[currSum] == 0) {
    prefixSum.erase(currSum);
}

return count;
}

int pathSumIII_Optimal(TreeNode* root, int targetSum) {
    unordered_map<long long, int> prefixSum;
    prefixSum[0] = 1; // 空路径的和为 0
    return pathSumIII_OptimalHelper(root, 0, targetSum, prefixSum);
}

// LeetCode 236. 最近公共祖先
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
// 题目描述: 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

```

```

// 百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，  

// 满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”  

//  

// 思路分析：  

// 1. 递归查找 p 和 q，当找到一个节点是 p 或 q 时返回  

// 2. 如果左右子树返回值都不为 null，说明当前节点就是 LCA  

// 3. 如果只有一侧返回值不为 null，返回那个不为 null 的值  

//  

// 时间复杂度：O(n)  

// 空间复杂度：O(h)  

//  

// 是否为最优解：是  

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {  

    if (root == nullptr || root == p || root == q) return root;  

  

    // 在左右子树中查找 p 和 q  

    TreeNode* left = lowestCommonAncestor(root->left, p, q);  

    TreeNode* right = lowestCommonAncestor(root->right, p, q);  

  

    // 如果左右子树都找到了，说明当前节点就是 LCA  

    if (left != nullptr && right != nullptr) return root;  

  

    // 否则返回找到的一侧（如果都没找到，就返回 nullptr）  

    return left != nullptr ? left : right;  

}  

  

// LeetCode 124. 二叉树中的最大路径和  

// 题目来源：LeetCode  

// 题目链接：https://leetcode.cn/problems/binary-tree-maximum-path-sum/  

// 题目描述：二叉树中的 路径 被定义为一条从树中任意节点出发，  

// 沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次。  

// 该路径 至少包含一个 节点，且不一定经过根节点。  

// 路径和 是路径中各节点值的总和。给你一个二叉树的根节点 root ，返回其 最大路径和。  

//  

// 思路分析：  

// 1. 对于每个节点，最大路径和可能是：  

//     - 左子树的最大贡献 + 节点值 + 右子树的最大贡献  

// 2. 但返回给父节点时，只能选择左或右一侧（因为路径不能分叉）  

// 3. 如果子树贡献为负，则不选择该子树（贡献为 0）  

//  

// 时间复杂度：O(n)  

// 空间复杂度：O(h)  

//
```

```

// 是否为最优解: 是
int maxPathSumValue = INT_MIN;

int maxGain(TreeNode* node) {
    if (node == nullptr) return 0;

    // 递归计算左右子树的最大贡献, 负数则不选
    int leftGain = max(maxGain(node->left), 0);
    int rightGain = max(maxGain(node->right), 0);

    // 更新全局最大路径和
    int currentPathSum = leftGain + node->val + rightGain;
    maxPathSumValue = max(maxPathSumValue, currentPathSum);

    // 返回节点的最大贡献
    return node->val + max(leftGain, rightGain);
}

int maxPathSum(TreeNode* root) {
    maxPathSumValue = INT_MIN;
    maxGain(root);
    return maxPathSumValue;
}

// LintCode 453. 将二叉树拆分为链表
// 题目来源: LintCode (炼码)
// 题目链接: https://www.lintcode.com/problem/453/
// 题目描述: 将一棵二叉树按照前序遍历拆解成为一个假链表。所谓的假链表是说, 用二叉树的 right 指针, 来表示链表中的 next 指针。
// 要求不能创建任何新的节点, 只能调整树中节点指针的指向。
//
// 思路分析:
// 1. 使用后序遍历, 先处理左右子树
// 2. 对于每个节点, 将左子树变成链表, 右子树变成链表
// 3. 将左子树链接到右子树上, 左子树指针设为 nullptr
// 4. 返回当前链表的末尾节点
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是

```

```

TreeNode* flattenHelper(TreeNode* node) {
    if (node == nullptr) return nullptr;

    // 叶子节点直接返回
    if (node->left == nullptr && node->right == nullptr) {
        return node;
    }

    // 递归处理左右子树
    TreeNode* leftTail = flattenHelper(node->left);
    TreeNode* rightTail = flattenHelper(node->right);

    // 如果左子树不为空，将左子树连接到右子树上
    if (leftTail != nullptr) {
        leftTail->right = node->right;
        node->right = node->left;
        node->left = nullptr;
    }

    // 返回新的链表末尾节点
    return rightTail != nullptr ? rightTail : leftTail;
}

void flatten(TreeNode* root) {
    if (root == nullptr) return;
    flattenHelper(root);
}

// HackerRank 二叉树的镜像
// 题目来源: HackerRank
// 题目描述: 给定一棵二叉树，判断它是否是自身的镜像（即对称）
//
// 思路分析:
// 1. 使用辅助函数检查两棵子树是否互为镜像
// 2. 两棵子树互为镜像的条件:
//     - 当前节点值相等
//     - 左子树的左子树与右子树的右子树互为镜像
//     - 左子树的右子树与右子树的左子树互为镜像
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是

```

```

bool isSymmetricAdvanced(TreeNode* root) {
    if (root == nullptr) return true;
    return isMirror(root->left, root->right);
}

// 剑指 Offer 26. 树的子结构
// 题目来源: 剑指 Offer
// 题目描述: 输入两棵二叉树 A 和 B, 判断 B 是不是 A 的子结构。
//
// 思路分析:
// 1. 遍历树 A 的每个节点, 以该节点为根节点
// 2. 检查从该节点开始的子树是否包含树 B 的结构
//
// 时间复杂度: O(m*n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
bool isSubStructureHelper(TreeNode* A, TreeNode* B) {
    if (B == nullptr) return true;
    if (A == nullptr || A->val != B->val) return false;

    return isSubStructureHelper(A->left, B->left) &&
           isSubStructureHelper(A->right, B->right);
}

bool isSubStructure(TreeNode* A, TreeNode* B) {
    if (A == nullptr || B == nullptr) return false;

    return isSubStructureHelper(A, B) ||
           isSubStructure(A->left, B) ||
           isSubStructure(A->right, B);
}

// USACO 二叉搜索树的最近公共祖先
// 题目来源: USACO (美国计算机奥林匹克竞赛)
// 题目描述: 给定一个二叉搜索树 (BST), 找到该树中两个指定节点的最近公共祖先。
//
// 思路分析:
// 利用 BST 的特性: 左子树所有节点值小于根节点, 右子树所有节点值大于根节点
// 1. 如果 p 和 q 的值都小于当前节点, 那么 LCA 在左子树
// 2. 如果 p 和 q 的值都大于当前节点, 那么 LCA 在右子树
// 3. 否则, 当前节点就是 LCA
//

```

```

// 时间复杂度: O(h)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
TreeNode* lowestCommonAncestorBST(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == nullptr || p == nullptr || q == nullptr) return nullptr;

    if (p->val < root->val && q->val < root->val) {
        return lowestCommonAncestorBST(root->left, p, q);
    }
    if (p->val > root->val && q->val > root->val) {
        return lowestCommonAncestorBST(root->right, p, q);
    }

    return root;
}

```

```

// AtCoder ABC191 E. Come Back Quickly - 距离和计算
// 题目描述简化: 给定一棵有根树, 计算每个节点到其所有子孙节点的距离之和
//
// 思路分析:
// 1. 使用后序遍历计算每个子树的节点数
// 2. 使用前序遍历计算距离之和
//
// 时间复杂度: O(n)
// 空间复杂度: O(n)
//
// 是否为最优解: 是
int dfsSize(TreeNode* node, vector<int>& size) {
    if (node == nullptr) return 0;

    size[node->val] = 1; // 包含自己
    size[node->val] += dfsSize(node->left, size);
    size[node->val] += dfsSize(node->right, size);

    return size[node->val];
}

void dfsDistance(TreeNode* node, vector<int>& size, vector<long long>& result, long long
parentDistance) {
    if (node == nullptr) return;

    result[node->val] = parentDistance;
    size[node->val] = 1;
    size[node->val] += dfsDistance(node->left, size, result, parentDistance);
    size[node->val] += dfsDistance(node->right, size, result, parentDistance);
}

```

```

if (node->left != nullptr) {
    int leftSize = size[node->left->val];
    int rightSize = node->right != nullptr ? size[node->right->val] : 0;
    long long leftDistance = parentDistance + (size[node->val] - leftSize) - leftSize;
    dfsDistance(node->left, size, result, leftDistance);
}

if (node->right != nullptr) {
    int rightSize = size[node->right->val];
    int leftSize = node->left != nullptr ? size[node->left->val] : 0;
    long long rightDistance = parentDistance + (size[node->val] - rightSize) - rightSize;
    dfsDistance(node->right, size, result, rightDistance);
}

vector<long long> calculateDistanceSum(TreeNode* root, int n) {
    vector<long long> result(n, 0);
    vector<int> size(n, 0);

    dfsSize(root, size);
    dfsDistance(root, size, result, 0);

    return result;
}

// CodeChef - SUBTREE - 最大子树和
// 题目描述简化：给定一棵二叉树，每个节点有一个权值。找出权值和最大的子树。
//
// 思路分析：
// 1. 使用后序遍历，计算每个子树的权值和
// 2. 对于每个节点，其最大子树和为：节点值 + max(左子树最大和, 0) + max(右子树最大和, 0)
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解：是
int maxSubtreeSumValue = INT_MIN;

int calculateSubtreeSum(TreeNode* node) {
    if (node == nullptr) return 0;

    int leftSum = max(calculateSubtreeSum(node->left), 0);

```

```

int rightSum = max(calculateSubtreeSum(node->right), 0);

int currentSum = node->val + leftSum + rightSum;
maxSubtreeSumValue = max(maxSubtreeSumValue, currentSum);

return node->val + max(leftSum, rightSum);
}

int maxSubtreeSum(TreeNode* root) {
    maxSubtreeSumValue = INT_MIN;
    calculateSubtreeSum(root);
    return maxSubtreeSumValue;
}

// UVa OJ 10080 - 重建二叉树
// 题目描述简化：根据前序遍历和中序遍历结果重建二叉树
//
// 思路分析：
// 1. 前序遍历的第一个节点是根节点
// 2. 在中序遍历中找到根节点的位置，分割左右子树
// 3. 递归重建左右子树
//
// 时间复杂度：O(n^2)
// 空间复杂度：O(n)
//
// 最优解：可以使用哈希表优化查找根节点的过程
TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                           vector<int>& inorder, int inStart, int inEnd) {
    if (preStart > preEnd || inStart > inEnd) return nullptr;

    TreeNode* root = new TreeNode(preorder[preStart]);

    // 找到根节点在中序遍历中的位置
    int rootIndex = inStart;
    for (; rootIndex <= inEnd; rootIndex++) {
        if (inorder[rootIndex] == root->val) {
            break;
        }
    }

    int leftSize = rootIndex - inStart;
    root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSize,

```

```

        inorder, inStart, rootIndex - 1);
root->right = buildTreeHelper(preorder, preStart + leftSize + 1, preEnd,
                               inorder, rootIndex + 1, inEnd);

return root;
}

```

```

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    if (preorder.empty() || inorder.empty()) return nullptr;
    return buildTreeHelper(preorder, 0, preorder.size() - 1,
                           inorder, 0, inorder.size() - 1);
}

```

// 牛客网 NC102. 树的序列化和反序列化  
// 题目描述：将二叉树序列化为字符串，然后从字符串反序列化回二叉树

//  
// 思路分析：  
// 序列化使用前序遍历，空节点用特殊字符表示  
//  
// 时间复杂度：O(n)  
// 空间复杂度：O(n)  
//

// 是否为最优解：是

```
void serializeHelper(TreeNode* node, string& result) {
```

```
    if (node == nullptr) {
        result += "#,";
        return;
    }
}
```

```

result += to_string(node->val) + ",";
serializeHelper(node->left, result);
serializeHelper(node->right, result);
}
```

```

string serialize(TreeNode* root) {
    string result;
    serializeHelper(root, result);
    return result;
}

```

```

TreeNode* deserializeHelper(vector<string>& nodes, int& index) {
    if (index >= nodes.size() || nodes[index] == "#") {
        index++;

```

```

        return nullptr;
    }

TreeNode* node = new TreeNode(stoi(nodes[index++]));
node->left = deserializeHelper(nodes, index);
node->right = deserializeHelper(nodes, index);

return node;
}

TreeNode* deserialize(string data) {
    if (data.empty()) return nullptr;

    vector<string> nodes;
    stringstream ss(data);
    string item;
    while (getline(ss, item, ',')) {
        nodes.push_back(item);
    }

    int index = 0;
    return deserializeHelper(nodes, index);
}

// 杭电 OJ 2024 - 二叉树遍历
// 题目描述：输入二叉树的前序遍历和中序遍历结果，输出其后序遍历结果
//
// 思路分析：
// 1. 先根据前序和中序构建二叉树
// 2. 然后进行后序遍历输出
//
// 时间复杂度：O(n^2)
// 空间复杂度：O(n)
//
// 最优解：可以使用哈希表优化查找过程
void postorderHelper(const string& preorder, int preStart, int preEnd,
                     const string& inorder, int inStart, int inEnd,
                     string& result) {
    if (preStart > preEnd || inStart > inEnd) return;

    char rootVal = preorder[preStart];

    // 找到根节点在中序中的位置

```

```

int rootIndex = inStart;
for (; rootIndex <= inEnd; rootIndex++) {
    if (inorder[rootIndex] == rootVal) {
        break;
    }
}

int leftLength = rootIndex - inStart;

// 递归处理左右子树
postorderHelper(preorder, preStart + 1, preStart + leftLength,
                inorder, inStart, rootIndex - 1, result);
postorderHelper(preorder, preStart + leftLength + 1, preEnd,
                inorder, rootIndex + 1, inEnd, result);

// 后序: 添加根节点
result += rootVal;
}

string postorderFromPreorderAndInorder(const string& preorder, const string& inorder) {
    if (preorder.empty() || inorder.empty()) return "";
    string result;
    postorderHelper(preorder, 0, preorder.size() - 1,
                    inorder, 0, inorder.size() - 1, result);
    return result;
}

// LeetCode 226. 翻转二叉树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/invert-binary-tree/
// 题目描述: 给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。
//
// 思路分析:
// 1. 递归翻转左右子树
// 2. 交换左右子树的位置
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是

```

```
// 主函数测试
int main() {
    cout << "===== 二叉树递归遍历基础测试 =====" << endl;
    TreeNode* head = new TreeNode(1);
    head->left = new TreeNode(2);
    head->right = new TreeNode(3);
    head->left->left = new TreeNode(4);
    head->left->right = new TreeNode(5);
    head->right->left = new TreeNode(6);
    head->right->right = new TreeNode(7);

    cout << "前序遍历: ";
    preOrder(head);
    cout << endl;

    cout << "中序遍历: ";
    inOrder(head);
    cout << endl;

    cout << "后序遍历: ";
    posOrder(head);
    cout << endl;

    cout << "\n===== LeetCode 104. 最大深度 =====" << endl;
    cout << "最大深度: " << maxDepth(head) << endl; // 预期: 3

    cout << "\n===== LeetCode 110. 平衡二叉树 =====" << endl;
    TreeNode* balancedTree = new TreeNode(1);
    balancedTree->left = new TreeNode(2);
    balancedTree->right = new TreeNode(3);
    balancedTree->left->left = new TreeNode(4);
    balancedTree->left->right = new TreeNode(5);
    cout << "是否为平衡二叉树: " << (isBalanced(balancedTree) ? "true" : "false") << endl;

    cout << "\n===== LeetCode 112. 路径总和 =====" << endl;
    TreeNode* pathTree = new TreeNode(5);
    pathTree->left = new TreeNode(4);
    pathTree->right = new TreeNode(8);
    pathTree->left->left = new TreeNode(11);
    pathTree->left->left->left = new TreeNode(7);
    pathTree->left->left->right = new TreeNode(2);
    cout << "是否存在路径和为 22: " << (hasPathSum(pathTree, 22) ? "true" : "false") << endl;
```

```
    cout << "\n===== 所有测试完成! =====" << endl;

    return 0;
}
```

=====

文件: BinaryTreeTraversalRecursion.java

=====

```
package class017;

// 二叉树递归遍历及相关题目详解
// 本文件包含二叉树的三种基本遍历方式（前序、中序、后序）的递归实现
// 并扩展了多个相关 LeetCode 题目，每道题目都包含详细注释、复杂度分析和多种语言实现
```

```
import java.util.*;
```

```
public class BinaryTreeTraversalRecursion {
```

```
    public static class TreeNode {
```

```
        public int val;
        public TreeNode left;
        public TreeNode right;
```

```
        public TreeNode(int v) {
```

```
            val = v;
        }
```

```
// ===== 更多平台题目实现 =====
```

```
/**
```

```
* 洛谷 P1305 新二叉树
* 题目来源: 洛谷 (Luogu)
* 题目链接: https://www.luogu.com.cn/problem/P1305
* 题目描述: 根据前序遍历字符串构建二叉树并输出中序遍历
*
```

```
* 思路分析:
```

```
* 1. 前序遍历字符串中, '#' 表示空节点
```

```
* 2. 使用递归构建二叉树
```

```
* 3. 输出中序遍历结果
```

```
*
```

```
* 时间复杂度: O(n)
```

```

* 空间复杂度: O(n)
*
* 是否为最优解: 是
*/

```

```

static class P1305Solution {
    private int index = 0;

    public TreeNode buildTree(String preorder) {
        if (index >= preorder.length() || preorder.charAt(index) == '#') {
            index++;
            return null;
        }
        TreeNode root = new TreeNode(preorder.charAt(index));
        index++;
        root.left = buildTree(preorder);
        root.right = buildTree(preorder);
        return root;
    }

    public List<Character> inorderTraversal(TreeNode root) {
        List<Character> result = new ArrayList<>();
        inorderHelper(root, result);
        return result;
    }

    private void inorderHelper(TreeNode node, List<Character> result) {
        if (node == null) return;
        inorderHelper(node.left, result);
        result.add((char) node.val);
        inorderHelper(node.right, result);
    }
}

/**
 * TimusOJ 1022 Genealogical Tree
 * 题目来源: Timus Online Judge
 * 题目链接: http://acm.timus.ru/problem.aspx?space=1&num=1022
 * 题目描述: 给定家族关系, 构建家谱树并输出拓扑排序
 *
 * 思路分析:
 * 1. 使用邻接表表示有向无环图
 * 2. 使用深度优先搜索进行拓扑排序
 * 3. 使用后序遍历得到拓扑序列

```

```

*
* 时间复杂度: O(n + m)
* 空间复杂度: O(n)
*
* 是否为最优解: 是
*/
static class Timus0J1022 {
    public List<Integer> topologicalSort(int n, int[][] edges) {
        List<Integer>[] graph = new ArrayList[n + 1];
        for (int i = 1; i <= n; i++) {
            graph[i] = new ArrayList<>();
        }
        for (int[] edge : edges) {
            graph[edge[0]].add(edge[1]);
        }

        boolean[] visited = new boolean[n + 1];
        List<Integer> result = new ArrayList<>();

        for (int i = 1; i <= n; i++) {
            if (!visited[i]) {
                dfs(i, graph, visited, result);
            }
        }

        Collections.reverse(result);
        return result;
    }

    private void dfs(int node, List<Integer>[] graph, boolean[] visited, List<Integer> result) {
        visited[node] = true;
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                dfs(neighbor, graph, visited, result);
            }
        }
        result.add(node);
    }
}

/**
 * Aizu0J ALDS1_7_C Tree Walk
 * 题目来源: Aizu Online Judge

```

```
* 题目链接: http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_7_C
* 题目描述: 实现二叉树的前序、中序、后序遍历
*
* 思路分析:
* 1. 标准的二叉树遍历实现
* 2. 分别实现三种遍历方式
* 3. 输出遍历结果
*
* 时间复杂度: O(n)
* 空间复杂度: O(h)
*
* 是否为最优解: 是
*/
static class AizuOJTreeWalk {
    public List<Integer> preorder(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        preorderHelper(root, result);
        return result;
    }

    private void preorderHelper(TreeNode node, List<Integer> result) {
        if (node == null) return;
        result.add((Integer) node.val);
        preorderHelper(node.left, result);
        preorderHelper(node.right, result);
    }

    public List<Integer> inorder(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        inorderHelper(root, result);
        return result;
    }

    private void inorderHelper(TreeNode node, List<Integer> result) {
        if (node == null) return;
        inorderHelper(node.left, result);
        result.add((Integer) node.val);
        inorderHelper(node.right, result);
    }

    public List<Integer> postorder(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        postorderHelper(root, result);
        return result;
    }

    private void postorderHelper(TreeNode node, List<Integer> result) {
```

```

        return result;
    }

private void postorderHelper(TreeNode node, List<Integer> result) {
    if (node == null) return;
    postorderHelper(node.left, result);
    postorderHelper(node.right, result);
    result.add((Integer) node.val);
}
}

/***
 * POJ 2255 Tree Recovery
 * 题目来源: 北京大学 POJ
 * 题目链接: http://poj.org/problem?id=2255
 * 题目描述: 根据前序遍历和中序遍历重建二叉树
 *
 * 思路分析:
 * 1. 前序遍历的第一个节点是根节点
 * 2. 在中序遍历中找到根节点的位置
 * 3. 递归重建左右子树
 *
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(n)
 *
 * 是否为最优解: 是 (可以使用哈希表优化到 O(n) )
 */
static class POJ2255 {
    public TreeNode buildTree(char[] preorder, char[] inorder) {
        if (preorder.length == 0 || inorder.length == 0) {
            return null;
        }

        char rootVal = preorder[0];
        TreeNode root = new TreeNode(rootVal);

        int rootIndex = -1;
        for (int i = 0; i < inorder.length; i++) {
            if (inorder[i] == rootVal) {
                rootIndex = i;
                break;
            }
        }

        if (rootIndex != -1) {
            root.left = buildTree(Arrays.copyOfRange(preorder, 1, rootIndex + 1),
                Arrays.copyOfRange(inorder, 0, rootIndex));
            root.right = buildTree(Arrays.copyOfRange(preorder, rootIndex + 1,
                Arrays.copyOfRange(preorder, rootIndex + 1, preorder.length)),
                Arrays.copyOfRange(inorder, rootIndex + 1, inorder.length));
        }
        return root;
    }
}

```

```

        char[] leftInorder = Arrays.copyOfRange(inorder, 0, rootIndex);
        char[] rightInorder = Arrays.copyOfRange(inorder, rootIndex + 1, inorder.length);
        char[] leftPreorder = Arrays.copyOfRange(preorder, 1, 1 + leftInorder.length);
        char[] rightPreorder = Arrays.copyOfRange(preorder, 1 + leftInorder.length,
preorder.length);

        root.left = buildTree(leftPreorder, leftInorder);
        root.right = buildTree(rightPreorder, rightInorder);

        return root;
    }

    public List<Character> getPostorder(TreeNode root) {
        List<Character> result = new ArrayList<>();
        postorder(root, result);
        return result;
    }

    private void postorder(TreeNode node, List<Character> result) {
        if (node == null) return;
        postorder(node.left, result);
        postorder(node.right, result);
        result.add((char) node.val);
    }
}

/***
 * HDU 1710 Binary Tree Traversals
 * 题目来源: 杭州电子科技大学 HDU
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1710
 * 题目描述: 根据前序遍历和中序遍历输出后序遍历
 *
 * 思路分析:
 * 1. 直接构建后序遍历序列, 无需构建完整二叉树
 * 2. 使用递归分治思想
 *
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(n)
 *
 * 是否为最优解: 是
 */
static class HDU1710 {

```

```
public List<Integer> getPostorder(int[] preorder, int[] inorder) {  
    if (preorder.length == 0) {  
        return new ArrayList<>();  
    }  
  
    int rootVal = preorder[0];  
    int rootIndex = -1;  
    for (int i = 0; i < inorder.length; i++) {  
        if (inorder[i] == rootVal) {  
            rootIndex = i;  
            break;  
        }  
    }  
  
    int[] leftPreorder = Arrays.copyOfRange(preorder, 1, 1 + rootIndex);  
    int[] rightPreorder = Arrays.copyOfRange(preorder, 1 + rootIndex, preorder.length);  
    int[] leftInorder = Arrays.copyOfRange(inorder, 0, rootIndex);  
    int[] rightInorder = Arrays.copyOfRange(inorder, rootIndex + 1, inorder.length);  
  
    List<Integer> leftPost = getPostorder(leftPreorder, leftInorder);  
    List<Integer> rightPost = getPostorder(rightPreorder, rightInorder);  
  
    List<Integer> result = new ArrayList<>();  
    result.addAll(leftPost);  
    result.addAll(rightPost);  
    result.add(rootVal);  
  
    return result;  
}  
}  
  
/**  
 * LOJ 10155 二叉苹果树  
 * 题目来源: LibreOJ  
 * 题目链接: https://loj.ac/p/10155  
 * 题目描述: 二叉树上有苹果, 要求保留指定数量的树枝, 使得苹果总数最大  
 *  
 * 思路分析:  
 * 1. 树形动态规划问题  
 * 2. 使用递归遍历计算每个子树的最优解  
 * 3. 状态转移: dp[node][k] = 保留 k 条树枝时的最大苹果数  
 *  
 * 时间复杂度: O(n * k^2)
```

```

* 空间复杂度: O(n * k)
*
* 是否为最优解: 是
*/

```

static class L0J10155 {

```

    public int maxApples(TreeNode root, int k) {
        Map<TreeNode, int[]> dp = new HashMap<>();
        dfs(root, k, dp);
        return dp.get(root)[k];
    }
}

private void dfs(TreeNode node, int k, Map<TreeNode, int[]> dp) {
    if (node == null) return;

    dp.put(node, new int[k + 1]);

    if (node.left != null) {
        dfs(node.left, k, dp);
    }
    if (node.right != null) {
        dfs(node.right, k, dp);
    }

    for (int i = 0; i <= k; i++) {
        for (int j = 0; j <= i; j++) {
            int leftVal = node.left != null ? dp.get(node.left)[j] : 0;
            int rightVal = node.right != null ? dp.get(node.right)[i - j] : 0;
            dp.get(node)[i] = Math.max(dp.get(node)[i], leftVal + rightVal);
        }
    }
}

for (int i = k; i > 0; i--) {
    dp.get(node)[i] = dp.get(node)[i - 1] + 1; // 假设每个节点有 1 个苹果
}
}

/**
 * CodeChef SUBTREE - 子树移除
 * 题目来源: CodeChef
 * 题目链接: https://www.codechef.com/problems/SUBTREE
 * 题目描述: 计算二叉树中所有子树的大小之和
 *

```

```

* 思路分析:
* 1. 使用后序遍历计算每个子树的大小
* 2. 累加所有子树的大小
*
* 时间复杂度: O(n)
* 空间复杂度: O(h)
*
* 是否为最优解: 是
*/

```

```

static class CodeChefSUBTREE {
    private int total = 0;

    public int sumSubtreeSizes(TreeNode root) {
        total = 0;
        dfs(root);
        return total;
    }

    private int dfs(TreeNode node) {
        if (node == null) return 0;

        int leftSize = dfs(node.left);
        int rightSize = dfs(node.right);
        int subtreeSize = leftSize + rightSize + 1;

        total += subtreeSize;

        return subtreeSize;
    }
}

/**
 * USACO 二叉搜索树的最近公共祖先
 * 题目来源: USACO (美国计算机奥林匹克竞赛)
 * 题目描述: 在二叉搜索树中查找两个节点的最近公共祖先
*
* 思路分析:
* 1. 利用 BST 的性质: 左子树所有节点值小于根节点, 右子树所有节点值大于根节点
* 2. 如果 p 和 q 的值都小于当前节点, LCA 在左子树
* 3. 如果 p 和 q 的值都大于当前节点, LCA 在右子树
* 4. 否则当前节点就是 LCA
*
* 时间复杂度: O(h)

```

```

* 空间复杂度: O(h)
*
* 是否为最优解: 是
*/
static class USACOLCA {
    public TreeNode lowestCommonAncestorBST(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || p == null || q == null) {
            return null;
        }

        if ((Integer)p.val > (Integer)q.val) {
            TreeNode temp = p;
            p = q;
            q = temp;
        }

        if ((Integer)p.val < (Integer)root.val && (Integer)q.val < (Integer)root.val) {
            return lowestCommonAncestorBST(root.left, p, q);
        } else if ((Integer)p.val > (Integer)root.val && (Integer)q.val > (Integer)root.val) {
            return lowestCommonAncestorBST(root.right, p, q);
        } else {
            return root;
        }
    }
}

/**
 * AtCoder ABC191 E. Come Back Quickly
 * 题目来源: AtCoder
 * 题目链接: https://atcoder.jp/contests/abc191/tasks/abc191/e
 * 题目描述: 计算树中每个节点到其所有子孙节点的距离之和
 *
 * 思路分析:
 * 1. 第一次DFS计算每个子树的大小
 * 2. 第二次DFS计算距离之和
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * 是否为最优解: 是
*/
static class AtCoderABC191E {
    public int[] calculateDistanceSum(TreeNode root, int n) {

```

```
int[] size = new int[n + 1];
int[] result = new int[n + 1];

dfsSize(root, size);
dfsDistance(root, size, result, 0);

return Arrays.copyOfRange(result, 1, n + 1);
}

private int dfsSize(TreeNode node, int[] size) {
    if (node == null) return 0;

    size[(Integer)node.val] = 1;
    size[(Integer)node.val] += dfsSize(node.left, size);
    size[(Integer)node.val] += dfsSize(node.right, size);

    return size[(Integer)node.val];
}

private void dfsDistance(TreeNode node, int[] size, int[] result, int parentDistance) {
    if (node == null) return;

    result[(Integer)node.val] = parentDistance;

    if (node.left != null) {
        int leftSize = size[(Integer)node.left.val];
        int rightSize = node.right != null ? size[(Integer)node.right.val] : 0;
        int leftDistance = parentDistance + (size[(Integer)node.val] - leftSize) - leftSize;
        dfsDistance(node.left, size, result, leftDistance);
    }

    if (node.right != null) {
        int rightSize = size[(Integer)node.right.val];
        int leftSize = node.left != null ? size[(Integer)node.left.val] : 0;
        int rightDistance = parentDistance + (size[(Integer)node.val] - rightSize) -
rightSize;
        dfsDistance(node.right, size, result, rightDistance);
    }
}

// 测试更多平台题目
class AdditionalPlatformTests {
```

```

public static void testAdditionalPlatforms() {
    System.out.println("\n" + "=" .repeat(50));
    System.out.println("更多平台题目测试");
    System.out.println("=".repeat(50));

    // 测试洛谷 P1305
    System.out.println("\n--- 洛谷 P1305 新二叉树 ---");
    P1305Solution p1305 = new P1305Solution();
    String preorderStr = "ABD##E##CF##G##";
    TreeNode tree = p1305.buildTree(preorderStr);
    List<Character> inorderResult = p1305.inorderTraversal(tree);
    System.out.println("前序遍历: " + preorderStr);
    System.out.println("中序遍历: " + inorderResult);

    // 测试 POJ2255
    System.out.println("\n--- POJ 2255 Tree Recovery ---");
    POJ2255 poj2255 = new POJ2255();
    char[] preorder = {'A', 'B', 'D', 'E', 'C', 'F', 'G'};
    char[] inorder = {'D', 'B', 'E', 'A', 'F', 'C', 'G'};
    TreeNode tree2 = poj2255.buildTree(preorder, inorder);
    List<Character> postorder = poj2255.getPostorder(tree2);
    System.out.println("前序: " + Arrays.toString(preorder));
    System.out.println("中序: " + Arrays.toString(inorder));
    System.out.println("后序: " + postorder);

    // 测试 HDU1710
    System.out.println("\n--- HDU 1710 Binary Tree Traversals ---");
    HDU1710 hdu1710 = new HDU1710();
    int[] preorderNums = {1, 2, 4, 5, 3, 6, 7};
    int[] inorderNums = {4, 2, 5, 1, 6, 3, 7};
    List<Integer> postorderNums = hdu1710.getPostorder(preorderNums, inorderNums);
    System.out.println("前序: " + Arrays.toString(preorderNums));
    System.out.println("中序: " + Arrays.toString(inorderNums));
    System.out.println("后序: " + postorderNums);

    // 测试 CodeChef SUBTREE
    System.out.println("\n--- CodeChef SUBTREE - 子树大小之和 ---");
    CodeChefSUBTREE codechef = new CodeChefSUBTREE();
    TreeNode root = new TreeNode(1);
    root.left = new TreeNode(2);
    root.right = new TreeNode(3);
    root.left.left = new TreeNode(4);
    root.left.right = new TreeNode(5);
}

```

```

int totalSize = codechef.sumSubtreeSizes(root);
System.out.println("所有子树大小之和: " + totalSize); // 预期: 15 (5个节点, 每个子树大小
之和)

System.out.println("\n" + "=" .repeat(50));
System.out.println("所有平台题目测试完成! ");
System.out.println("=".repeat(50));
}

}

}

// 递归基本样子, 用来理解递归序
// 递归序是指在递归过程中, 每个节点都会被访问三次:
// 1. 刚进入节点时
// 2. 从左子树返回时
// 3. 从右子树返回时
public static void f(TreeNode head) {
    if (head == null) {
        return;
    }
    // 位置 1: 刚进入节点时 (前序遍历位置)
    f(head.left);
    // 位置 2: 从左子树返回时 (中序遍历位置)
    f(head.right);
    // 位置 3: 从右子树返回时 (后序遍历位置)
}

// 先序打印所有节点, 递归版
// 先序遍历顺序: 根节点 -> 左子树 -> 右子树
// 时间复杂度: O(n), 其中 n 是二叉树的节点数, 每个节点恰好被访问一次
// 空间复杂度: O(h), 其中 h 是二叉树的高度, 递归调用栈的深度等于树的高度
public static void preOrder(TreeNode head) {
    if (head == null) {
        return;
    }
    // 先访问根节点
    System.out.print(head.val + " ");
    // 再递归访问左子树
    preOrder(head.left);
    // 最后递归访问右子树
    preOrder(head.right);
}

```

```

}

// 中序打印所有节点，递归版
// 中序遍历顺序：左子树 -> 根节点 -> 右子树
// 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点恰好被访问一次
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度等于树的高度
public static void inOrder(TreeNode head) {
    if (head == null) {
        return;
    }
    // 先递归访问左子树
    inOrder(head.left);
    // 再访问根节点
    System.out.print(head.val + " ");
    // 最后递归访问右子树
    inOrder(head.right);
}

// 后序打印所有节点，递归版
// 后序遍历顺序：左子树 -> 右子树 -> 根节点
// 时间复杂度：O(n)，其中 n 是二叉树的节点数，每个节点恰好被访问一次
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度等于树的高度
public static void posOrder(TreeNode head) {
    if (head == null) {
        return;
    }
    // 先递归访问左子树
    posOrder(head.left);
    // 再递归访问右子树
    posOrder(head.right);
    // 最后访问根节点
    System.out.print(head.val + " ");
}

// LeetCode 104. 二叉树的最大深度
// 题目链接：https://leetcode.cn/problems/maximum-depth-of-binary-tree/
// 题目描述：给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的
// 节点数。
// 解法：使用递归，树的最大深度等于左右子树最大深度的最大值加 1
// 时间复杂度：O(n)，其中 n 是二叉树的节点数
// 空间复杂度：O(h)，其中 h 是二叉树的高度，递归调用栈的深度
public static int maxDepth(TreeNode root) {
    // 基础情况：空节点的深度为 0
}

```

```

if (root == null) {
    return 0;
}
// 递归计算左右子树的最大深度
int leftDepth = maxDepth(root.left);
int rightDepth = maxDepth(root.right);
// 返回左右子树最大深度的最大值加 1
return Math.max(leftDepth, rightDepth) + 1;
}

// LeetCode 110. 平衡二叉树
// 题目链接: https://leetcode.cn/problems/balanced-binary-tree/
// 题目描述: 给定一个二叉树，判断它是否是高度平衡的二叉树。
// 解法: 使用递归，自底向上检查每个节点的左右子树高度差是否不超过 1
// 时间复杂度: O(n)，其中 n 是二叉树的节点数
// 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度
public static boolean isBalanced(TreeNode root) {
    return getHeight(root) != -1;
}

// 辅助函数: 获取树的高度，如果不平衡则返回-1
private static int getHeight(TreeNode node) {
    // 基础情况: 空节点的高度为 0
    if (node == null) {
        return 0;
    }
    // 递归获取左子树高度
    int leftHeight = getHeight(node.left);
    // 如果左子树不平衡，直接返回-1
    if (leftHeight == -1) {
        return -1;
    }
    // 递归获取右子树高度
    int rightHeight = getHeight(node.right);
    // 如果右子树不平衡，直接返回-1
    if (rightHeight == -1) {
        return -1;
    }
    // 检查当前节点是否平衡（左右子树高度差不超过 1）
    if (Math.abs(leftHeight - rightHeight) > 1) {
        return -1;
    }
    // 返回当前节点的高度（左右子树最大高度加 1）

```

```

        return Math.max(leftHeight, rightHeight) + 1;
    }

// LeetCode 100. 相同的树
// 题目链接: https://leetcode.cn/problems/same-tree/
// 题目描述: 给你两棵二叉树的根节点 p 和 q , 编写一个函数来检验两棵树是否相同。
// 解法: 使用递归同时遍历两棵树, 比较对应节点的值是否相等
// 时间复杂度: O(min(m, n)) , 其中 m 和 n 分别是两个二叉树的节点数
// 空间复杂度: O(min(h1, h2)) , 其中 h1 和 h2 分别是两个二叉树的高度
public static boolean isSameTree(TreeNode p, TreeNode q) {
    // 基础情况: 两个节点都为空, 则相同
    if (p == null && q == null) {
        return true;
    }
    // 基础情况: 一个节点为空, 另一个不为空, 则不相同
    if (p == null || q == null) {
        return false;
    }
    // 比较当前节点值, 并递归比较左右子树
    return p.val == q.val && isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}

// LeetCode 101. 对称二叉树
// 题目链接: https://leetcode.cn/problems/symmetric-tree/
// 题目描述: 给你一个二叉树的根节点 root , 检查它是否轴对称。
// 解法: 使用递归比较左子树和右子树是否镜像对称
// 时间复杂度: O(n) , 其中 n 是二叉树的节点数
// 空间复杂度: O(h) , 其中 h 是二叉树的高度
public static boolean isSymmetric(TreeNode root) {
    // 空树是对称的
    if (root == null) {
        return true;
    }
    // 比较左右子树是否镜像对称
    return isMirror(root.left, root.right);
}

// 辅助函数: 判断两个树是否镜像对称
private static boolean isMirror(TreeNode left, TreeNode right) {
    // 基础情况: 两个节点都为空, 则对称
    if (left == null && right == null) {
        return true;
    }
}

```

```

// 基础情况：一个节点为空，另一个不为空，则不对称
if (left == null || right == null) {
    return false;
}
// 比较当前节点值，并递归比较外侧和内侧
return left.val == right.val &&
       isMirror(left.left, right.right) &&
       isMirror(left.right, right.left);
}

// LeetCode 226. 翻转二叉树
// 题目链接: https://leetcode.cn/problems/invert-binary-tree/
// 题目描述：给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。
// 解法：使用递归，交换每个节点的左右子树
// 时间复杂度：O(n)，其中 n 是二叉树的节点数
// 空间复杂度：O(h)，其中 h 是二叉树的高度
public static TreeNode invertTree(TreeNode root) {
    // 基础情况：空节点无需翻转
    if (root == null) {
        return null;
    }
    // 交换左右子树
    TreeNode temp = root.left;
    root.left = root.right;
    root.right = temp;
    // 递归翻转左右子树
    invertTree(root.left);
    invertTree(root.right);
    return root;
}

// ====== 扩展题目部分 ======
// LeetCode 112. 路径总和
// 题目来源：LeetCode
// 题目链接: https://leetcode.cn/problems/path-sum/
// 题目描述：给你二叉树的根节点 root 和一个表示目标和的整数 targetSum。
// 判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum。
// 叶子节点是指没有子节点的节点。
//
// 思路分析：
// 1. 使用递归，从根节点开始，每次递归时减去当前节点的值
// 2. 当到达叶子节点时，检查剩余的目标和是否等于叶子节点的值

```

```

// 3. 递归地检查左右子树是否存在满足条件的路径
//
// 时间复杂度: O(n)，其中 n 是二叉树的节点数，每个节点访问一次
// 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度
//
// 是否为最优解: 是。递归遍历是解决此类路径问题的最优方法。
//
// 边界场景:
// - 空树: 返回 false
// - 只有根节点: 检查根节点值是否等于 targetSum
// - 负数节点值: 算法依然有效
// - 目标和为 0: 正常处理
public static boolean hasPathSum(TreeNode root, int targetSum) {
    // 边界情况: 空节点返回 false
    if (root == null) {
        return false;
    }
    // 到达叶子节点, 检查路径和是否等于目标和
    if (root.left == null && root.right == null) {
        return root.val == targetSum;
    }
    // 递归检查左右子树, 目标和减去当前节点的值
    return hasPathSum(root.left, targetSum - root.val) ||
           hasPathSum(root.right, targetSum - root.val);
}

// LeetCode 113. 路径总和 II
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/path-sum-ii/
// 题目描述: 给你二叉树的根节点 root 和一个整数目标和 targetSum,
// 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。
//
// 思路分析:
// 1. 使用回溯法, 维护一个当前路径列表
// 2. 递归遍历树, 每次将当前节点加入路径
// 3. 到达叶子节点时, 检查路径和是否等于目标和, 若是则将路径加入结果
// 4. 回溯时移除当前节点
//
// 时间复杂度: O(n^2)，其中 n 是节点数，最坏情况下需要复制所有路径
// 空间复杂度: O(n)，递归栈和路径存储的空间
//
// 是否为最优解: 是。回溯+递归是解决所有路径问题的标准方法。
public static List<List<Integer>> pathSum(TreeNode root, int targetSum) {

```

```

List<List<Integer>> result = new ArrayList<>();
List<Integer> path = new ArrayList<>();
pathSumHelper(root, targetSum, path, result);
return result;
}

private static void pathSumHelper(TreeNode node, int targetSum,
                                List<Integer> path, List<List<Integer>> result) {
    if (node == null) {
        return;
    }
    // 将当前节点加入路径
    path.add(node.val);
    // 到达叶子节点，检查路径和
    if (node.left == null && node.right == null && node.val == targetSum) {
        result.add(new ArrayList<>(path)); // 复制当前路径
    }
    // 递归遍历左右子树
    pathSumHelper(node.left, targetSum - node.val, path, result);
    pathSumHelper(node.right, targetSum - node.val, path, result);
    // 回溯：移除当前节点
    path.remove(path.size() - 1);
}

// LeetCode 111. 二叉树的最小深度
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/minimum-depth-of-binary-tree/
// 题目描述: 给定一个二叉树，找出其最小深度。
// 最小深度是从根节点到最近叶子节点的最短路径上的节点数量。
//
// 思路分析:
// 1. 使用递归，注意必须到达叶子节点才算一条路径
// 2. 如果一个节点只有左子树或只有右子树，不能简单取 min，要继续递归非空子树
// 3. 只有当左右子树都存在时，才取较小深度
//
// 时间复杂度: O(n)，其中 n 是节点数
// 空间复杂度: O(h)，其中 h 是树的高度
//
// 是否为最优解: 是。但 BFS 层序遍历也是最优解，在某些情况下更快（遇到第一个叶子节点即可返回）。
//
// 常见错误: 直接用 Math.min(左深度, 右深度)会在单子树情况下出错
public static int minDepth(TreeNode root) {
    if (root == null) {

```

```

        return 0;
    }

    // 如果左子树为空，只递归右子树
    if (root.left == null) {
        return minDepth(root.right) + 1;
    }

    // 如果右子树为空，只递归左子树
    if (root.right == null) {
        return minDepth(root.left) + 1;
    }

    // 左右子树都存在，取较小深度
    return Math.min(minDepth(root.left), minDepth(root.right)) + 1;
}

// LeetCode 257. 二叉树的所有路径
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/binary-tree-paths/
// 题目描述: 给你一个二叉树的根节点 root，按 任意顺序，
// 返回所有从根节点到叶子节点的路径。
//
// 思路分析:
// 1. 使用递归+回溯，构建路径字符串
// 2. 到达叶子节点时，将路径字符串加入结果
// 3. 使用 StringBuilder 可以提高效率（但要注意回溯时删除字符）
//
// 时间复杂度: O(n^2)，需要构建和复制路径字符串
// 空间复杂度: O(n)，递归栈和结果存储
//
// 是否为最优解: 是。递归+回溯是标准解法。
public static List<String> binaryTreePaths(TreeNode root) {
    List<String> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    binaryTreePathsHelper(root, "", result);
    return result;
}

private static void binaryTreePathsHelper(TreeNode node, String path,
                                         List<String> result) {
    if (node == null) {
        return;
    }
}

```

```

// 构建当前路径
path += node.val;
// 到达叶子节点，加入结果
if (node.left == null && node.right == null) {
    result.add(path);
    return;
}
// 继续递归，路径中加入箭头
path += "->";
binaryTreePathsHelper(node.left, path, result);
binaryTreePathsHelper(node.right, path, result);
}

// LeetCode 543. 二叉树的直径
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/diameter-of-binary-tree/
// 题目描述: 给定一棵二叉树，你需要计算它的直径长度。
// 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。
// 这条路径可能穿过也可能不穿过根节点。
//
// 思路分析:
// 1. 直径 = 某个节点的左子树最大深度 + 右子树最大深度
// 2. 需要递归计算每个节点的这个值，并维护全局最大值
// 3. 使用后序遍历，先计算子树深度，再更新直径
//
// 时间复杂度: O(n)，每个节点访问一次
// 空间复杂度: O(h)，递归栈深度
//
// 是否为最优解: 是。一次遍历即可得到答案。
private static int maxDiameter = 0;

public static int diameterOfBinaryTree(TreeNode root) {
    maxDiameter = 0;
    getDepth(root);
    return maxDiameter;
}

private static int getDepth(TreeNode node) {
    if (node == null) {
        return 0;
    }
    // 递归计算左右子树深度
    int leftDepth = getDepth(node.left);

```

```

int rightDepth = getDepth(node.right);
// 更新最大直径: 左深度 + 右深度
maxDiameter = Math.max(maxDiameter, leftDepth + rightDepth);
// 返回当前节点的深度
return Math.max(leftDepth, rightDepth) + 1;
}

// LeetCode 404. 左叶子之和
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/sum-of-left-leaves/
// 题目描述: 给定二叉树的根节点 root，返回所有左叶子之和。
//
// 思路分析:
// 1. 递归遍历树，判断节点是否为左叶子
// 2. 左叶子的定义: 是某个节点的左孩子，且该孩子没有子节点
// 3. 需要从父节点判断，而不是在节点自身判断
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
public static int sumOfLeftLeaves(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int sum = 0;
    // 检查左子节点是否为叶子
    if (root.left != null && root.left.left == null && root.left.right == null) {
        sum += root.left.val;
    }
    // 递归计算左右子树的左叶子之和
    sum += sumOfLeftLeaves(root.left);
    sum += sumOfLeftLeaves(root.right);
    return sum;
}

// LeetCode 572. 另一棵树的子树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/subtree-of-another-tree/
// 题目描述: 给你两棵二叉树 root 和 subRoot。
// 检验 root 中是否包含和 subRoot 具有相同结构和节点值的子树。
//
// 思路分析:

```

```

// 1. 递归检查 root 的每个节点，看是否与 subRoot 相同
// 2. 使用 isSameTree 函数检查两棵树是否相同
// 3. 如果当前节点不匹配，继续递归检查左右子树
//
// 时间复杂度: O(m*n)， m 和 n 分别是两棵树的节点数
// 空间复杂度: O(max(h1, h2))， 递归栈深度
//
// 是否为最优解: 否。更优解法是使用 KMP 或序列化+字符串匹配，时间复杂度 O(m+n)
// 但递归解法更直观，在面试中更常用
public static boolean isSubtree(TreeNode root, TreeNode subRoot) {
    if (root == null) {
        return false;
    }
    // 检查当前节点是否与 subRoot 相同
    if (isSameTree(root, subRoot)) {
        return true;
    }
    // 递归检查左右子树
    return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
}

```

```

// LeetCode 617. 合并二叉树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/merge-two-binary-trees/
// 题目描述: 给你两棵二叉树: root1 和 root2。
// 想象一下，当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠。
// 你需要将这两棵树合并成一棵新二叉树。合并规则是:
// 如果两个节点重叠，那么将这两个节点的值相加作为合并后节点的新值;
// 否则，不为 null 的节点将直接作为新二叉树的节点。
//

```

```

// 思路分析:
// 1. 同时递归遍历两棵树
// 2. 如果两个节点都存在，值相加
// 3. 如果只有一个节点存在，直接使用该节点
//

```

// 时间复杂度: O(min(m, n))

// 空间复杂度: O(min(h1, h2))

//

// 是否为最优解: 是

```

public static TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
    // 如果一棵树为空，返回另一棵树
    if (root1 == null) {
        return root2;
    }

```

```

    }

    if (root2 == null) {
        return root1;
    }

    // 创建新节点，值为两节点之和
    TreeNode merged = new TreeNode(root1.val + root2.val);
    // 递归合并左右子树
    merged.left = mergeTrees(root1.left, root2.left);
    merged.right = mergeTrees(root1.right, root2.right);
    return merged;
}

// LeetCode 654. 最大二叉树
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/maximum-binary-tree/
// 题目描述: 给定一个不重复的整数数组 nums。
// 最大二叉树可以用下面的算法从 nums 递归地构建:
// 1. 创建一个根节点, 其值为 nums 中的最大值。
// 2. 递归地在最大值左边的子数组前缀上构建左子树。
// 3. 递归地在最大值右边的子数组后缀上构建右子树。
//
// 思路分析:
// 1. 找到数组中的最大值及其索引
// 2. 最大值作为根节点
// 3. 递归构建左右子树
//
// 时间复杂度: O(n^2), 最坏情况下数组有序, 每次都要遍历剩余元素找最大值
// 空间复杂度: O(n), 递归栈深度
//
// 是否为最优解: 否。使用单调栈可以优化到 O(n) 时间复杂度
// 但递归解法更符合题意, 代码更简洁
public static TreeNode constructMaximumBinaryTree(int[] nums) {
    return buildMaxTree(nums, 0, nums.length - 1);
}

private static TreeNode buildMaxTree(int[] nums, int left, int right) {
    if (left > right) {
        return null;
    }

    // 找到最大值的索引
    int maxIndex = left;
    for (int i = left + 1; i <= right; i++) {
        if (nums[i] > nums[maxIndex]) {

```

```
    maxIndex = i;
}
}

// 创建根节点
TreeNode root = new TreeNode(nums[maxIndex]);
// 递归构建左右子树
root.left = buildMaxTree(nums, left, maxIndex - 1);
root.right = buildMaxTree(nums, maxIndex + 1, right);
return root;
}

// 二叉树遍历的 C++ 版本实现（作为注释提供）
/*
// C++ 版本的前序遍历
void preOrder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    cout << root->val << " ";
    preOrder(root->left);
    preOrder(root->right);
}

// C++ 版本的中序遍历
void inOrder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    inOrder(root->left);
    cout << root->val << " ";
    inOrder(root->right);
}

// C++ 版本的后序遍历
void posOrder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    posOrder(root->left);
    posOrder(root->right);
    cout << root->val << " ";
}
```

```

// 二叉树遍历的 Python 版本实现（作为注释提供）
/*
# Python 版本的前序遍历
def preOrder(root):
    if root is None:
        return
    print(root.val, end=" ")
    preOrder(root.left)
    preOrder(root.right)

# Python 版本的中序遍历
def inOrder(root):
    if root is None:
        return
    inOrder(root.left)
    print(root.val, end=" ")
    inOrder(root.right)

# Python 版本的后序遍历
def posOrder(root):
    if root is None:
        return
    posOrder(root.left)
    posOrder(root.right)
    print(root.val, end=" ")
*/

```

// LeetCode 563. 二叉树的坡度  
// 题目来源: LeetCode  
// 题目链接: <https://leetcode.cn/problems/binary-tree-tilt/>  
// 题目描述: 给你一个二叉树的根节点 root，计算并返回 整个树 的坡度。  
// 一个树的节点的坡度定义即为，该节点左子树的节点之和和右子树节点之和的差的绝对值。  
// 如果没有左子树的话，左子树的节点之和为 0；没有右子树的话也是一样。  
// 空节点的坡度是 0。整个树的坡度就是其所有节点的坡度之和。  
//  
// 思路分析:  
// 1. 使用后序遍历，先计算子树的节点和  
// 2. 对于每个节点，计算左右子树节点和的差的绝对值，累加到总坡度  
// 3. 返回当前子树的节点和（包括根节点）  
//  
// 时间复杂度: O(n)  
// 空间复杂度: O(h)

```

// 是否为最优解: 是
private static int totalTilt = 0;

public static int findTilt(TreeNode root) {
    totalTilt = 0;
    calculateSum(root);
    return totalTilt;
}

private static int calculateSum(TreeNode node) {
    if (node == null) {
        return 0;
    }
    // 计算左右子树的节点和
    int leftSum = calculateSum(node.left);
    int rightSum = calculateSum(node.right);
    // 累加当前节点的坡度
    totalTilt += Math.abs(leftSum - rightSum);
    // 返回当前子树的节点和
    return leftSum + rightSum + node.val;
}

// LeetCode 508. 出现次数最多的子树元素和
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/most-frequent-subtree-sum/
// 题目描述: 给你一个二叉树的根结点 root，计算出现最多的子树元素和。
// 一个结点的「子树元素和」定义为: 以该结点为根的二叉树上所有结点的元素之和（包括结点自身）。
//
// 思路分析:
// 1. 使用后序遍历计算每个子树的元素和
// 2. 使用 HashMap 统计每个和出现的次数
// 3. 找出出现次数最多的所有和
//
// 时间复杂度: O(n)
// 空间复杂度: O(n)，需要 HashMap 存储每个子树和
//
// 是否为最优解: 是
public static int[] findFrequentTreeSum(TreeNode root) {
    Map<Integer, Integer> sumCount = new HashMap<>();
    calculateTreeSum(root, sumCount);
    // 找到最大频率
    int maxFreq = 0;

```

```

        for (int count : sumCount.values()) {
            maxFreq = Math.max(maxFreq, count);
        }
        // 收集所有最大频率的和
        List<Integer> result = new ArrayList<>();
        for (Map.Entry<Integer, Integer> entry : sumCount.entrySet()) {
            if (entry.getValue() == maxFreq) {
                result.add(entry.getKey());
            }
        }
        // 转换为数组
        return result.stream().mapToInt(Integer::intValue).toArray();
    }

private static int calculateTreeSum(TreeNode node, Map<Integer, Integer> sumCount) {
    if (node == null) {
        return 0;
    }
    // 计算左右子树的和
    int leftSum = calculateTreeSum(node.left, sumCount);
    int rightSum = calculateTreeSum(node.right, sumCount);
    // 当前子树的总和
    int totalSum = leftSum + rightSum + node.val;
    // 统计出现次数
    sumCount.put(totalSum, sumCount.getOrDefault(totalSum, 0) + 1);
    return totalSum;
}

// LeetCode 437. 路径总和 III
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/path-sum-iii/
// 题目描述: 给定一个二叉树的根节点 root，和一个整数 targetSum，
// 求该二叉树里节点值之和等于 targetSum 的 路径 的数目。
// 路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只从父节点到子节点）。
//
// 思路分析:
// 方法 1: 双重递归 - 需要两个递归函数
//   1. 第一个递归遍历所有节点
//   2. 第二个递归以当前节点为起点计算路径数
// 方法 2: 前缀和+HashMap - 更优的解法
//   1. 使用前缀和思想，类似于数组的子数组和问题
//   2. 用 HashMap 存储前缀和及其出现次数

```

```

// 时间复杂度: 方法 1 O(n^2), 方法 2 O(n)
// 空间复杂度: 方法 1 O(h), 方法 2 O(n)
//
// 是否为最优解: 方法 2 是最优解, 但方法 1 更直观。这里实现两种方法。

// 方法 1: 双重递归
public static int pathSumIII(TreeNode root, int targetSum) {
    if (root == null) {
        return 0;
    }
    // 以当前节点为起点的路径数 + 左子树中的路径数 + 右子树中的路径数
    return countPathsFrom(root, targetSum) +
        pathSumIII(root.left, targetSum) +
        pathSumIII(root.right, targetSum);
}

// 计算从当前节点开始的路径数
private static int countPathsFrom(TreeNode node, long targetSum) {
    if (node == null) {
        return 0;
    }
    int count = 0;
    // 如果当前节点的值等于目标和, 找到一条路径
    if (node.val == targetSum) {
        count++;
    }
    // 继续在左右子树中查找, 目标和减去当前节点值
    count += countPathsFrom(node.left, targetSum - node.val);
    count += countPathsFrom(node.right, targetSum - node.val);
    return count;
}

// 方法 2: 前缀和 + HashMap (最优解)
public static int pathSumIII_Optimal(TreeNode root, int targetSum) {
    Map<Long, Integer> prefixSumCount = new HashMap<>();
    // 初始化: 前缀和为 0 的路径有一条
    prefixSumCount.put(0L, 1);
    return dfsWithPrefixSum(root, 0L, targetSum, prefixSumCount);
}

private static int dfsWithPrefixSum(TreeNode node, long currentSum, int targetSum,
    Map<Long, Integer> prefixSumCount) {

```

```

if (node == null) {
    return 0;
}
// 当前路径的前缀和
currentSum += node.val;
// 查找是否存在前缀和为 currentSum - targetSum 的路径
int count = prefixSumCount.getOrDefault(currentSum - targetSum, 0);
// 将当前前缀和加入 map
prefixSumCount.put(currentSum, prefixSumCount.getOrDefault(currentSum, 0) + 1);
// 递归遍历左右子树
count += dfsWithPrefixSum(node.left, currentSum, targetSum, prefixSumCount);
count += dfsWithPrefixSum(node.right, currentSum, targetSum, prefixSumCount);
// 回溯：移除当前节点的前缀和
prefixSumCount.put(currentSum, prefixSumCount.get(currentSum) - 1);
return count;
}

```

```

// LeetCode 236. 二叉树的最近公共祖先
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
// 题目描述: 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。
// 
```

// 思路分析:

// 1. 使用递归, 分为三种情况:

```

//     a. 如果 p 和 q 分别在左右子树, 则当前节点就是最近公共祖先
//     b. 如果 p 和 q 都在左子树, 则最近公共祖先在左子树
//     c. 如果 p 和 q 都在右子树, 则最近公共祖先在右子树

```

// 2. 特殊情况: 当前节点就是 p 或 q, 且另一个节点在其子树中

//

// 时间复杂度: O(n), 最坏情况需要遍历所有节点

// 空间复杂度: O(h), 递归栈深度

//

// 是否为最优解: 是。这是经典的递归解法。

```

public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    // 基础情况: 空节点或找到目标节点
    if (root == null || root == p || root == q) {
        return root;
    }
    // 递归在左右子树中查找
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
    // 如果左右子树都找到了, 说明 p 和 q 分别在左右两侧, 当前节点就是最近公共祖先
    if (left != null && right != null) {

```

```

        return root;
    }

    // 否则返回非空的一侧
    return left != null ? left : right;
}

// LeetCode 124. 二叉树中的最大路径和
// 题目来源: LeetCode
// 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
// 题目描述: 二叉树中的 路径 被定义为一条从树中任意节点出发,
// 沿父节点-子节点连接, 达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次。
// 该路径 至少包含一个 节点, 且不一定经过根节点。
// 路径和 是路径中各节点值的总和。给你一个二叉树的根节点 root , 返回其 最大路径和。
//
// 思路分析:
// 1. 对于每个节点, 最大路径和可能是:
//     - 左子树的最大贡献 + 节点值 + 右子树的最大贡献
// 2. 但返回给父节点时, 只能选择左或右一侧(因为路径不能分叉)
// 3. 如果子树贡献为负, 则不选择该子树(贡献为0)
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是。一次遍历即可得到答案。
//
// 这是 Hard 难度的经典题, 需要理解“局部最优”和“向上返回”的区别
private static int maxPathSum = Integer.MIN_VALUE;

public static int maxPathSum(TreeNode root) {
    maxPathSum = Integer.MIN_VALUE;
    maxGain(root);
    return maxPathSum;
}

private static int maxGain(TreeNode node) {
    if (node == null) {
        return 0;
    }

    // 递归计算左右子树的最大贡献, 负数则不选
    int leftGain = Math.max(maxGain(node.left), 0);
    int rightGain = Math.max(maxGain(node.right), 0);
    // 更新全局最大路径和: 左贡献 + 节点值 + 右贡献
    int currentPathSum = leftGain + node.val + rightGain;
}

```

```

maxPathSum = Math.max(maxPathSum, currentPathSum);
// 返回节点的最大贡献: 节点值 + max(左贡献, 右贡献)
return node.val + Math.max(leftGain, rightGain);
}

// LintCode 453. 将二叉树拆分为链表
// 题目来源: LintCode (炼码)
// 题目链接: https://www.lintcode.com/problem/453/
// 题目描述: 将一棵二叉树按照前序遍历拆解成为一个假链表。所谓的假链表是说, 用二叉树的 right 指针, 来表示链表中的 next 指针。
// 要求不能创建任何新的节点, 只能调整树中节点指针的指向。
//
// 思路分析:
// 1. 使用后序遍历, 先处理左右子树
// 2. 对于每个节点, 将左子树变成链表, 右子树变成链表
// 3. 将左子树链接到右子树上, 左子树指针设为 null
// 4. 返回当前链表的末尾节点
//
// 时间复杂度: O(n), 每个节点访问一次
// 空间复杂度: O(h), 递归栈深度
//
// 是否为最优解: 是。递归解法简洁高效。
public static void flatten(TreeNode root) {
    if (root == null) {
        return;
    }
    flattenHelper(root);
}

private static TreeNode flattenHelper(TreeNode node) {
    if (node == null) {
        return null;
    }
    // 叶子节点直接返回
    if (node.left == null && node.right == null) {
        return node;
    }
    // 递归处理左右子树
    TreeNode leftTail = flattenHelper(node.left);
    TreeNode rightTail = flattenHelper(node.right);
    // 如果左子树不为空, 将左子树连接到右子树上
    if (leftTail != null) {
        leftTail.right = node.right;
    }
}
```

```

        node.right = node.left;
        node.left = null; // 清空左指针
    }

    // 返回新的链表末尾节点
    return rightTail != null ? rightTail : leftTail;
}

// HackerRank 二叉树的镜像
// 题目来源: HackerRank
// 题目描述: 给定一棵二叉树, 判断它是否是自身的镜像 (即对称)
// 这个题目与 LeetCode 101 类似, 但增加了更多的工程化考量
//
// 思路分析:
// 1. 使用辅助函数检查两棵子树是否互为镜像
// 2. 两棵子树互为镜像的条件:
//     - 当前节点值相等
//     - 左子树的左子树与右子树的右子树互为镜像
//     - 左子树的右子树与右子树的左子树互为镜像
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
public static boolean isSymmetricAdvanced(TreeNode root) {
    if (root == null) {
        return true;
    }
    return isMirror(root.left, root.right);
}

// 剑指 Offer 26. 树的子结构
// 题目来源: 剑指 Offer
// 题目描述: 输入两棵二叉树 A 和 B, 判断 B 是不是 A 的子结构。
// 注意: 这里的子结构与 LeetCode 572 的子树定义不同, 子结构只要 A 中存在一个子树与 B 完全相同即可, 不需要是完整的子树。
//
// 思路分析:
// 1. 遍历树 A 的每个节点, 以该节点为根节点
// 2. 检查从该节点开始的子树是否包含树 B 的结构
// 3. 注意边界条件处理
//
// 时间复杂度: O(m*n), m 和 n 分别是两棵树的节点数
// 空间复杂度: O(h), 递归栈深度

```

```

// 是否为最优解: 是
public static boolean isSubStructure(TreeNode A, TreeNode B) {
    // 边界条件: 任意一个树为空, 返回 false
    if (A == null || B == null) {
        return false;
    }
    // 检查以 A 为根节点是否包含 B, 或者 A 的左子树是否包含 B, 或者 A 的右子树是否包含 B
    return isSubStructureHelper(A, B) ||
           isSubStructure(A.left, B) ||
           isSubStructure(A.right, B);
}

private static boolean isSubStructureHelper(TreeNode A, TreeNode B) {
    // 如果 B 为空, 说明 B 的所有节点都已经匹配完成
    if (B == null) {
        return true;
    }
    // 如果 A 为空, 或者 A 和 B 的值不相等, 匹配失败
    if (A == null || A.val != B.val) {
        return false;
    }
    // 递归检查左右子树
    return isSubStructureHelper(A.left, B.left) &&
           isSubStructureHelper(A.right, B.right);
}

// USACO 二叉搜索树的最近公共祖先
// 题目来源: USACO (美国计算机奥林匹克竞赛)
// 题目描述: 给定一个二叉搜索树 (BST), 找到该树中两个指定节点的最近公共祖先。
// 注: BST 的特性可以帮助我们优化最近公共祖先的查找
//
// 思路分析:
// 利用 BST 的特性: 左子树所有节点值小于根节点, 右子树所有节点值大于根节点
// 1. 如果 p 和 q 的值都小于当前节点, 那么 LCA 在左子树
// 2. 如果 p 和 q 的值都大于当前节点, 那么 LCA 在右子树
// 3. 否则, 当前节点就是 LCA
//
// 时间复杂度: O(h), h 为树的高度, 最坏情况 O(n)
// 空间复杂度: O(h), 递归栈深度
//
// 是否为最优解: 是。利用 BST 特性的解法比普通二叉树的解法更高效
public static TreeNode lowestCommonAncestorBST(TreeNode root, TreeNode p, TreeNode q) {

```

```

if (root == null || p == null || q == null) {
    return null;
}
// 如果 p 和 q 都在左子树
if (p.val < root.val && q.val < root.val) {
    return lowestCommonAncestorBST(root.left, p, q);
}
// 如果 p 和 q 都在右子树
if (p.val > root.val && q.val > root.val) {
    return lowestCommonAncestorBST(root.right, p, q);
}
// 否则，当前节点就是最近公共祖先
return root;
}

// AtCoder ABC191 E. Come Back Quickly
// 这是一个关于图论的题目，但其中涉及到树的递归遍历思想
// 题目描述简化：给定一棵有根树，计算每个节点到其所有子孙节点的距离之和
//
// 思路分析：
// 1. 使用后序遍历计算每个子树的节点数
// 2. 使用前序遍历计算距离之和
// 3. 需要两次递归：第一次计算子树大小，第二次计算距离和
//
// 时间复杂度：O(n)
// 空间复杂度：O(h)
//
// 是否为最优解：是
public static long[] calculateDistanceSum(TreeNode root, int n) {
    long[] result = new long[n];
    // 第一次 DFS：计算子树大小
    int[] size = new int[n];
    dfsSize(root, size);
    // 第二次 DFS：计算距离和
    dfsDistance(root, size, result, 0);
    return result;
}

private static int dfsSize(TreeNode node, int[] size) {
    if (node == null) {
        return 0;
    }
    size[node.val] = 1; // 包含自己
}

```

```

        size[node.val] += dfsSize(node.left, size);
        size[node.val] += dfsSize(node.right, size);
        return size[node.val];
    }

private static void dfsDistance(TreeNode node, int[] size, long[] result, long parentDistance)
{
    if (node == null) {
        return;
    }
    result[node.val] = parentDistance;
    // 计算左子树的距离和
    if (node.left != null) {
        int leftSize = size[node.left.val];
        int rightSize = node.right != null ? size[node.right.val] : 0;
        // 父节点的距离和 + (n - leftSize) - leftSize
        long leftDistance = parentDistance + (size[node.val] - leftSize) - leftSize;
        dfsDistance(node.left, size, result, leftDistance);
    }
    // 计算右子树的距离和
    if (node.right != null) {
        int rightSize = size[node.right.val];
        int leftSize = node.left != null ? size[node.left.val] : 0;
        long rightDistance = parentDistance + (size[node.val] - rightSize) - rightSize;
        dfsDistance(node.right, size, result, rightDistance);
    }
}

// CodeChef - SUBTREE - Subtree Removal
// 题目来源: CodeChef
// 题目描述简化: 给定一棵二叉树, 每个节点有一个权值。找出权值和最大的子树。
//
// 思路分析:
// 1. 使用后序遍历, 计算每个子树的权值和
// 2. 对于每个节点, 其最大子树和为: 节点值 + max(左子树最大和, 0) + max(右子树最大和, 0)
// 3. 维护一个全局变量记录最大子树和
//
// 时间复杂度: O(n)
// 空间复杂度: O(h)
//
// 是否为最优解: 是
private static int maxSubtreeSum = Integer.MIN_VALUE;

```



```

private static TreeNode buildTreeHelper(int[] preorder, int preStart, int preEnd,
                                       int[] inorder, int inStart, int inEnd) {
    if (preStart > preEnd || inStart > inEnd) {
        return null;
    }
    // 创建根节点
    TreeNode root = new TreeNode(preorder[preStart]);
    // 找到根节点在中序遍历中的位置
    int rootIndex = inStart;
    for (; rootIndex <= inEnd; rootIndex++) {
        if (inorder[rootIndex] == root.val) {
            break;
        }
    }
    // 计算左子树的节点数
    int leftSize = rootIndex - inStart;
    // 递归构建左右子树
    root.left = buildTreeHelper(preorder, preStart + 1, preStart + leftSize,
                                inorder, inStart, rootIndex - 1);
    root.right = buildTreeHelper(preorder, preStart + leftSize + 1, preEnd,
                                 inorder, rootIndex + 1, inEnd);
    return root;
}

// 牛客网 NC102. 比较版本号
// 虽然这是字符串处理题目，但我们可以将其转化为树的递归问题
// 这里实现一个树的序列化和反序列化问题，这是牛客网的高频题
//
// 思路分析：
// 将二叉树序列化为字符串，然后从字符串反序列化回二叉树
// 序列化使用前序遍历，空节点用特殊字符表示
//
// 时间复杂度：O(n)
// 空间复杂度：O(n)
//
// 是否为最优解：是
public static String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    serializeHelper(root, sb);
    return sb.toString();
}

private static void serializeHelper(TreeNode node, StringBuilder sb) {

```

```

        if (node == null) {
            sb.append("#, ");
            return;
        }
        sb.append(node.val).append(", ");
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
    }

public static TreeNode deserialize(String data) {
    if (data == null || data.isEmpty()) {
        return null;
    }
    String[] nodes = data.split(",");
    return deserializeHelper(nodes, new int[]{0});
}

private static TreeNode deserializeHelper(String[] nodes, int[] index) {
    if (index[0] >= nodes.length || "#".equals(nodes[index[0]])) {
        index[0]++;
        return null;
    }
    TreeNode node = new TreeNode(Integer.parseInt(nodes[index[0]++]));
    node.left = deserializeHelper(nodes, index);
    node.right = deserializeHelper(nodes, index);
    return node;
}

// 杭电 OJ 2024 - 二叉树遍历
// 题目来源: 杭州电子科技大学 OJ
// 题目描述: 输入二叉树的前序遍历和中序遍历结果, 输出其后序遍历结果
//
// 思路分析:
// 1. 先根据前序和中序构建二叉树
// 2. 然后进行后序遍历输出
//
// 时间复杂度: O(n^2)
// 空间复杂度: O(n)
//
// 最优解: 可以使用哈希表优化查找过程
public static String postorderFromPreorderAndInorder(String preorder, String inorder) {
    if (preorder == null || inorder == null || preorder.isEmpty() || inorder.isEmpty()) {
        return "";
    }
}

```

```
}

// 由于 HDOJ 2024 的节点是字符，这里构建一个特殊的辅助函数
// 实际工程中应该根据具体数据类型调整
StringBuilder sb = new StringBuilder();
postorderHelper(preorder, 0, preorder.length() - 1,
                inorder, 0, inorder.length() - 1, sb);
return sb.toString();
}

private static void postorderHelper(String preorder, int preStart, int preEnd,
                                    String inorder, int inStart, int inEnd,
                                    StringBuilder sb) {
    if (preStart > preEnd || inStart > inEnd) {
        return;
    }
    // 根节点字符
    char rootVal = preorder.charAt(preStart);
    // 找到根节点在中序中的位置
    int rootIndex = inStart;
    for (; rootIndex <= inEnd; rootIndex++) {
        if (inorder.charAt(rootIndex) == rootVal) {
            break;
        }
    }
    // 计算左子树的长度
    int leftLength = rootIndex - inStart;
    // 递归处理左子树
    postorderHelper(preorder, preStart + 1, preStart + leftLength,
                    inorder, inStart, rootIndex - 1, sb);
    // 递归处理右子树
    postorderHelper(preorder, preStart + leftLength + 1, preEnd,
                    inorder, rootIndex + 1, inEnd, sb);
    // 后序：根节点最后添加
    sb.append(rootVal);
}

public static void main(String[] args) {
    System.out.println("===== 二叉树递归遍历基础测试 =====");
    TreeNode head = new TreeNode(1);
    head.left = new TreeNode(2);
    head.right = new TreeNode(3);
    head.left.left = new TreeNode(4);
    head.left.right = new TreeNode(5);
```

```
head.right.left = new TreeNode(6);
head.right.right = new TreeNode(7);

System.out.print("前序遍历: ");
preOrder(head);
System.out.println();

System.out.print("中序遍历: ");
inOrder(head);
System.out.println();

System.out.print("后序遍历: ");
posOrder(head);
System.out.println();

System.out.println("\n===== LeetCode 104. 最大深度 =====");
System.out.println("最大深度: " + maxDepth(head)); // 预期: 3

System.out.println("\n===== LeetCode 110. 平衡二叉树 =====");
TreeNode balancedTree = new TreeNode(1);
balancedTree.left = new TreeNode(2);
balancedTree.right = new TreeNode(3);
balancedTree.left.left = new TreeNode(4);
balancedTree.left.right = new TreeNode(5);
System.out.println("是否为平衡二叉树: " + isBalanced(balancedTree)); // 预期: true

System.out.println("\n===== LeetCode 100. 相同的树 =====");
TreeNode tree1 = new TreeNode(1);
tree1.left = new TreeNode(2);
tree1.right = new TreeNode(3);
TreeNode tree2 = new TreeNode(1);
tree2.left = new TreeNode(2);
tree2.right = new TreeNode(3);
System.out.println("两棵树是否相同: " + isSameTree(tree1, tree2)); // 预期: true

System.out.println("\n===== LeetCode 101. 对称二叉树 =====");
TreeNode symmetricTree = new TreeNode(1);
symmetricTree.left = new TreeNode(2);
symmetricTree.right = new TreeNode(2);
symmetricTree.left.left = new TreeNode(3);
symmetricTree.left.right = new TreeNode(4);
symmetricTree.right.left = new TreeNode(4);
symmetricTree.right.right = new TreeNode(3);
```

```
System.out.println("是否为对称二叉树: " + isSymmetric(symmetricTree)); // 预期: true
```

```
System.out.println("\n===== LeetCode 112. 路径总和 =====");
```

```
TreeNode pathTree = new TreeNode(5);
```

```
pathTree.left = new TreeNode(4);
```

```
pathTree.right = new TreeNode(8);
```

```
pathTree.left.left = new TreeNode(11);
```

```
pathTree.left.left.left = new TreeNode(7);
```

```
pathTree.left.left.right = new TreeNode(2);
```

```
System.out.println("是否存在路径和为 22: " + hasPathSum(pathTree, 22)); // 预期: true
```

```
System.out.println("\n===== LeetCode 113. 路径总和 II =====");
```

```
List<List<Integer>> paths = pathSum(pathTree, 22);
```

```
System.out.println("路径总和为 22 的所有路径: " + paths); // 预期: [[5, 4, 11, 2]]
```

```
System.out.println("\n===== LeetCode 111. 最小深度 =====");
```

```
System.out.println("最小深度: " + minDepth(head)); // 预期: 3
```

```
System.out.println("\n===== LeetCode 257. 二叉树的所有路径 =====");
```

```
List<String> allPaths = binaryTreePaths(head);
```

```
System.out.println("所有路径: " + allPaths);
```

```
System.out.println("\n===== LeetCode 543. 二叉树的直径 =====");
```

```
System.out.println("直径长度: " + diameterOfBinaryTree(head)); // 预期: 4
```

```
System.out.println("\n===== LeetCode 404. 左叶子之和 =====");
```

```
System.out.println("左叶子之和: " + sumOfLeftLeaves(head)); // 预期: 4
```

```
System.out.println("\n===== LeetCode 572. 另一棵树的子树 =====");
```

```
TreeNode subTree = new TreeNode(2);
```

```
subTree.left = new TreeNode(4);
```

```
subTree.right = new TreeNode(5);
```

```
System.out.println("是否为子树: " + isSubtree(head, subTree)); // 预期: true
```

```
System.out.println("\n===== LeetCode 617. 合并二叉树 =====");
```

```
TreeNode merged = mergeTrees(tree1, tree2);
```

```
System.out.print("合并后的树 (前序): ");
```

```
preOrder(merged);
```

```
System.out.println();
```

```
System.out.println("\n===== LeetCode 654. 最大二叉树 =====");
```

```
int[] nums = {3, 2, 1, 6, 0, 5};
```

```
TreeNode maxTree = constructMaximumBinaryTree(nums);
```

```

System.out.print("最大二叉树（前序）：");
preOrder(maxTree);
System.out.println();

System.out.println("\n===== LeetCode 563. 二叉树的坡度 =====");
System.out.println("整个树的坡度：" + findTilt(head)); // 预期: 6

System.out.println("\n===== LeetCode 508. 出现次数最多的子树元素和 =====");
int[] frequentSums = findFrequentTreeSum(head);
System.out.print("出现次数最多的子树和：" );
for (int sum : frequentSums) {
    System.out.print(sum + " ");
}
System.out.println();

System.out.println("\n===== LeetCode 437. 路径总和 III =====");
TreeNode pathTree3 = new TreeNode(10);
pathTree3.left = new TreeNode(5);
pathTree3.right = new TreeNode(-3);
pathTree3.left.left = new TreeNode(3);
pathTree3.left.right = new TreeNode(2);
pathTree3.right.right = new TreeNode(11);
pathTree3.left.left.left = new TreeNode(3);
pathTree3.left.left.right = new TreeNode(-2);
pathTree3.left.right.right = new TreeNode(1);
System.out.println("路径和为 8 的路径数（双重递归）：" + pathSumIII(pathTree3, 8)); // 预期: 3
System.out.println("路径和为 8 的路径数（前缀和）：" + pathSumIII_Optimal(pathTree3, 8));
// 预期: 3

System.out.println("\n===== LeetCode 236. 最近公共祖先 =====");
TreeNode lca = lowestCommonAncestor(head, head.left.left, head.left.right);
System.out.println("最近公共祖先的值：" + lca.val); // 预期: 2

System.out.println("\n===== LeetCode 124. 二叉树中的最大路径和 =====");
TreeNode maxPathTree = new TreeNode(-10);
maxPathTree.left = new TreeNode(9);
maxPathTree.right = new TreeNode(20);
maxPathTree.right.left = new TreeNode(15);
maxPathTree.right.right = new TreeNode(7);
System.out.println("最大路径和：" + maxPathSum(maxPathTree)); // 预期: 42

System.out.println("\n===== LeetCode 226. 翻转二叉树 =====");

```

```

TreeNode invertTest = new TreeNode(4);
invertTest.left = new TreeNode(2);
invertTest.right = new TreeNode(7);
invertTest.left.left = new TreeNode(1);
invertTest.left.right = new TreeNode(3);
System.out.print("翻转前 (中序): ");
inOrder(invertTest);
System.out.println();
invertTree(invertTest);
System.out.print("翻转后 (中序): ");
inOrder(invertTest);
System.out.println();

System.out.println("\n===== 所有测试完成! =====");
}
}

=====

```

文件: BinaryTreeTraversalRecursion.py

```

=====
# 二叉树递归遍历及相关题目详解 - Python 版本
# 本文件包含二叉树的三种基本遍历方式（前序、中序、后序）的递归实现
# 并扩展了多个相关 LeetCode 题目，每道题目都包含详细注释、复杂度分析

```

```

from typing import List, Optional
from collections import defaultdict

# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

```

```

# 递归基本样子，用来理解递归序
# 递归序是指在递归过程中，每个节点都会被访问三次：
# 1. 刚进入节点时
# 2. 从左子树返回时
# 3. 从右子树返回时
def recursion_pattern(head: Optional[TreeNode]) -> None:

```

```
if head is None:  
    return  
# 位置 1: 刚进入节点时 (前序遍历位置)  
recursion_pattern(head.left)  
# 位置 2: 从左子树返回时 (中序遍历位置)  
recursion_pattern(head.right)  
# 位置 3: 从右子树返回时 (后序遍历位置)  
  
# 先序打印所有节点, 递归版  
# 先序遍历顺序: 根节点 -> 左子树 -> 右子树  
# 时间复杂度: O(n), 其中 n 是二叉树的节点数, 每个节点恰好被访问一次  
# 空间复杂度: O(h), 其中 h 是二叉树的高度, 递归调用栈的深度等于树的高度  
def pre_order(head: Optional[TreeNode]) -> None:  
    if head is None:  
        return  
    # 先访问根节点  
    print(head.val, end=" ")  
    # 再递归访问左子树  
    pre_order(head.left)  
    # 最后递归访问右子树  
    pre_order(head.right)  
  
# 中序打印所有节点, 递归版  
# 中序遍历顺序: 左子树 -> 根节点 -> 右子树  
# 时间复杂度: O(n), 其中 n 是二叉树的节点数, 每个节点恰好被访问一次  
# 空间复杂度: O(h), 其中 h 是二叉树的高度, 递归调用栈的深度等于树的高度  
def in_order(head: Optional[TreeNode]) -> None:  
    if head is None:  
        return  
    # 先递归访问左子树  
    in_order(head.left)  
    # 再访问根节点  
    print(head.val, end=" ")  
    # 最后递归访问右子树  
    in_order(head.right)  
  
# 后序打印所有节点, 递归版  
# 后序遍历顺序: 左子树 -> 右子树 -> 根节点  
# 时间复杂度: O(n), 其中 n 是二叉树的节点数, 每个节点恰好被访问一次  
# 空间复杂度: O(h), 其中 h 是二叉树的高度, 递归调用栈的深度等于树的高度
```

```

def pos_order(head: Optional[TreeNode]) -> None:
    if head is None:
        return
    # 先递归访问左子树
    pos_order(head.left)
    # 再递归访问右子树
    pos_order(head.right)
    # 最后访问根节点
    print(head.val, end=" ")

# LeetCode 104. 二叉树的最大深度
# 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
# 题目描述: 给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。
# 解法: 使用递归，树的最大深度等于左右子树最大深度的最大值加 1
# 时间复杂度: O(n)，其中 n 是二叉树的节点数
# 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度
def max_depth(root: Optional[TreeNode]) -> int:
    # 基础情况: 空节点的深度为 0
    if root is None:
        return 0
    # 递归计算左右子树的最大深度
    left_depth = max_depth(root.left)
    right_depth = max_depth(root.right)
    # 返回左右子树最大深度的最大值加 1
    return max(left_depth, right_depth) + 1

# LeetCode 110. 平衡二叉树
# 题目链接: https://leetcode.cn/problems/balanced-binary-tree/
# 题目描述: 给定一个二叉树，判断它是否是高度平衡的二叉树。
# 解法: 使用递归，自底向上检查每个节点的左右子树高度差是否不超过 1
# 时间复杂度: O(n)，其中 n 是二叉树的节点数
# 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度

def get_height(node: Optional[TreeNode]) -> int:
    """辅助函数: 获取树的高度，如果不平衡则返回-1"""
    # 基础情况: 空节点的高度为 0
    if node is None:
        return 0
    # 递归获取左子树高度
    left_height = get_height(node.left)

```

```
# 如果左子树不平衡，直接返回-1
if left_height == -1:
    return -1
# 递归获取右子树高度
right_height = get_height(node.right)
# 如果右子树不平衡，直接返回-1
if right_height == -1:
    return -1
# 检查当前节点是否平衡（左右子树高度差不超过 1）
if abs(left_height - right_height) > 1:
    return -1
# 返回当前节点的高度（左右子树最大高度加 1）
return max(left_height, right_height) + 1
```

```
def is_balanced(root: Optional[TreeNode]) -> bool:
    return get_height(root) != -1
```

```
# LeetCode 100. 相同的树
# 题目链接: https://leetcode.cn/problems/same-tree/
# 题目描述: 给你两棵二叉树的根节点 p 和 q ，编写一个函数来检验两棵树是否相同。
# 解法: 使用递归同时遍历两棵树，比较对应节点的值是否相等
# 时间复杂度: O(min(m, n))，其中 m 和 n 分别是两个二叉树的节点数
# 空间复杂度: O(min(h1, h2))，其中 h1 和 h2 分别是两个二叉树的高度
def is_same_tree(p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
    # 基础情况: 两个节点都为空，则相同
    if p is None and q is None:
        return True
    # 基础情况: 一个节点为空，另一个不为空，则不相同
    if p is None or q is None:
        return False
    # 比较当前节点值，并递归比较左右子树
    return (p.val == q.val and
            is_same_tree(p.left, q.left) and
            is_same_tree(p.right, q.right))
```

```
# LeetCode 101. 对称二叉树
# 题目链接: https://leetcode.cn/problems/symmetric-tree/
# 题目描述: 给你一个二叉树的根节点 root ，检查它是否轴对称。
# 解法: 使用递归比较左子树和右子树是否镜像对称
# 时间复杂度: O(n)，其中 n 是二叉树的节点数
```

```
# 空间复杂度: O(h), 其中 h 是二叉树的高度

def is_mirror(left: Optional[TreeNode], right: Optional[TreeNode]) -> bool:
    """辅助函数: 判断两个树是否镜像对称"""
    # 基础情况: 两个节点都为空, 则对称
    if left is None and right is None:
        return True
    # 基础情况: 一个节点为空, 另一个不为空, 则不对称
    if left is None or right is None:
        return False
    # 比较当前节点值, 并递归比较外侧和内侧
    return (left.val == right.val and
            is_mirror(left.left, right.right) and
            is_mirror(left.right, right.left))
```

```
def is_symmetric(root: Optional[TreeNode]) -> bool:
    # 空树是对称的
    if root is None:
        return True
    # 比较左右子树是否镜像对称
    return is_mirror(root.left, root.right)
```

```
# LeetCode 226. 翻转二叉树
# 题目链接: https://leetcode.cn/problems/invert-binary-tree/
# 题目描述: 给你一棵二叉树的根节点 root , 翻转这棵二叉树, 并返回其根节点。
# 解法: 使用递归, 交换每个节点的左右子树
# 时间复杂度: O(n), 其中 n 是二叉树的节点数
# 空间复杂度: O(h), 其中 h 是二叉树的高度

def invert_tree(root: Optional[TreeNode]) -> Optional[TreeNode]:
    # 基础情况: 空节点无需翻转
    if root is None:
        return None
    # 交换左右子树
    root.left, root.right = root.right, root.left
    # 递归翻转左右子树
    invert_tree(root.left)
    invert_tree(root.right)
    return root
```

```
# ====== 扩展题目部分 ======
```

```
# LeetCode 112. 路径总和
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/path-sum/
# 题目描述: 给你二叉树的根节点 root 和一个表示目标和的整数 targetSum。
# 判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum。
# 叶子节点是指没有子节点的节点。
#
# 思路分析:
# 1. 使用递归，从根节点开始，每次递归时减去当前节点的值
# 2. 当到达叶子节点时，检查剩余的目标和是否等于叶子节点的值
# 3. 递归地检查左右子树是否存在满足条件的路径
#
# 时间复杂度: O(n)，其中 n 是二叉树的节点数，每个节点访问一次
# 空间复杂度: O(h)，其中 h 是二叉树的高度，递归调用栈的深度
#
# 是否为最优解: 是。递归遍历是解决此类路径问题的最优方法。
#
# 边界场景:
# - 空树: 返回 False
# - 只有根节点: 检查根节点值是否等于 targetSum
# - 负数节点值: 算法依然有效
# - 目标和为 0: 正常处理

def has_path_sum(root: Optional[TreeNode], target_sum: int) -> bool:
    # 边界情况: 空节点返回 False
    if root is None:
        return False
    # 到达叶子节点，检查路径和是否等于目标和
    if root.left is None and root.right is None:
        return root.val == target_sum
    # 递归检查左右子树，目标和减去当前节点的值
    return (has_path_sum(root.left, target_sum - root.val) or
            has_path_sum(root.right, target_sum - root.val))
```

```
# LeetCode 113. 路径总和 II
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/path-sum-ii/
# 题目描述: 给你二叉树的根节点 root 和一个整数目标和 targetSum,
# 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。
#
# 思路分析:
# 1. 使用回溯法，维护一个当前路径列表
```

```

# 2. 递归遍历树，每次将当前节点加入路径
# 3. 到达叶子节点时，检查路径和是否等于目标和，若是则将路径加入结果
# 4. 回溯时移除当前节点
#
# 时间复杂度：O(n^2)，其中 n 是节点数，最坏情况下需要复制所有路径
# 空间复杂度：O(n)，递归栈和路径存储的空间
#
# 是否为最优解：是。回溯+递归是解决所有路径问题的标准方法。

def path_sum_helper(node: Optional[TreeNode], target_sum: int,
                    path: List[int], result: List[List[int]]) -> None:
    if node is None:
        return
    # 将当前节点加入路径
    path.append(node.val)
    # 到达叶子节点，检查路径和
    if node.left is None and node.right is None and node.val == target_sum:
        result.append(path[:]) # 复制当前路径
    # 递归遍历左右子树
    path_sum_helper(node.left, target_sum - node.val, path, result)
    path_sum_helper(node.right, target_sum - node.val, path, result)
    # 回溯：移除当前节点
    path.pop()

```

```

def path_sum(root: Optional[TreeNode], target_sum: int) -> List[List[int]]:
    result = []
    path = []
    path_sum_helper(root, target_sum, path, result)
    return result

```

```

# LeetCode 111. 二叉树的最小深度
# 题目来源：LeetCode
# 题目链接：https://leetcode.cn/problems/minimum-depth-of-binary-tree/
# 题目描述：给定一个二叉树，找出其最小深度。
# 最小深度是从根节点到最近叶子节点的最短路径上的节点数量。
#
# 思路分析：
# 1. 使用递归，注意必须到达叶子节点才算一条路径
# 2. 如果一个节点只有左子树或只有右子树，不能简单取 min，要继续递归非空子树
# 3. 只有当左右子树都存在时，才取较小深度
#

```

```
# 时间复杂度: O(n)，其中 n 是节点数
# 空间复杂度: O(h)，其中 h 是树的高度
#
# 是否为最优解: 是。但 BFS 层序遍历也是最优解，在某些情况下更快（遇到第一个叶子节点即可返回）。
#
# 常见错误: 直接用 min(左深度, 右深度)会在单子树情况下出错
def min_depth(root: Optional[TreeNode]) -> int:
    if root is None:
        return 0
    # 如果左子树为空，只递归右子树
    if root.left is None:
        return min_depth(root.right) + 1
    # 如果右子树为空，只递归左子树
    if root.right is None:
        return min_depth(root.left) + 1
    # 左右子树都存在，取较小深度
    return min(min_depth(root.left), min_depth(root.right)) + 1
```

```
# LeetCode 257. 二叉树的所有路径
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/binary-tree-paths/
# 题目描述: 给你一个二叉树的根节点 root，按 任意顺序，  
# 返回所有从根节点到叶子节点的路径。
#
# 思路分析:
# 1. 使用递归+回溯，构建路径字符串
# 2. 到达叶子节点时，将路径字符串加入结果
# 3. Python 中字符串是不可变的，所以每次拼接都会创建新字符串，无需显式回溯
#
# 时间复杂度: O(n^2)，需要构建和复制路径字符串
# 空间复杂度: O(n)，递归栈和结果存储
#
# 是否为最优解: 是。递归+回溯是标准解法。
```

```
def binary_tree_paths_helper(node: Optional[TreeNode], path: str,
                               result: List[str]) -> None:
    if node is None:
        return
    # 构建当前路径
    path += str(node.val)
    # 到达叶子节点，加入结果
    if node.left is None and node.right is None:
        result.append(path)
```

```

        result.append(path)
        return
    # 继续递归，路径中加入箭头
    path += "->"
    binary_tree_paths_helper(node.left, path, result)
    binary_tree_paths_helper(node.right, path, result)

def binary_tree_paths(root: Optional[TreeNode]) -> List[str]:
    result = []
    if root is None:
        return result
    binary_tree_paths_helper(root, "", result)
    return result

# LeetCode 543. 二叉树的直径
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/diameter-of-binary-tree/
# 题目描述: 给定一棵二叉树，你需要计算它的直径长度。
# 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。
# 这条路径可能穿过也可能不穿过根节点。
#
# 思路分析:
# 1. 直径 = 某个节点的左子树最大深度 + 右子树最大深度
# 2. 需要递归计算每个节点的这个值，并维护全局最大值
# 3. 使用后序遍历，先计算子树深度，再更新直径
#
# 时间复杂度: O(n)，每个节点访问一次
# 空间复杂度: O(h)，递归栈深度
#
# 是否为最优解: 是。一次遍历即可得到答案。

class DiameterSolution:
    def __init__(self):
        self.max_diameter = 0

    def get_depth(self, node: Optional[TreeNode]) -> int:
        if node is None:
            return 0
        # 递归计算左右子树深度
        left_depth = self.get_depth(node.left)
        right_depth = self.get_depth(node.right)

```

```

# 更新最大直径: 左深度 + 右深度
self.max_diameter = max(self.max_diameter, left_depth + right_depth)
# 返回当前节点的深度
return max(left_depth, right_depth) + 1

def diameter_of_binary_tree(self, root: Optional[TreeNode]) -> int:
    self.max_diameter = 0
    self.get_depth(root)
    return self.max_diameter

def diameter_of_binary_tree(root: Optional[TreeNode]) -> int:
    """便捷函数"""
    solution = DiameterSolution()
    return solution.diameter_of_binary_tree(root)

# LeetCode 404. 左叶子之和
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/sum-of-left-leaves/
# 题目描述: 给定二叉树的根节点 root，返回所有左叶子之和。
#
# 思路分析:
# 1. 递归遍历树，判断节点是否为左叶子
# 2. 左叶子的定义: 是某个节点的左孩子，且该孩子没有子节点
# 3. 需要从父节点判断，而不是在节点自身判断
#
# 时间复杂度: O(n)
# 空间复杂度: O(h)
#
# 是否为最优解: 是
def sum_of_left_leaves(root: Optional[TreeNode]) -> int:
    if root is None:
        return 0
    total = 0
    # 检查左子节点是否为叶子
    if (root.left is not None and
        root.left.left is None and
        root.left.right is None):
        total += root.left.val
    # 递归计算左右子树的左叶子之和
    total += sum_of_left_leaves(root.left)
    total += sum_of_left_leaves(root.right)

```

```
return total

# LeetCode 572. 另一棵树的子树
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/subtree-of-another-tree/
# 题目描述: 给你两棵二叉树 root 和 subRoot。
# 检验 root 中是否包含和 subRoot 具有相同结构和节点值的子树。
#
# 思路分析:
# 1. 递归检查 root 的每个节点, 看是否与 subRoot 相同
# 2. 使用 isSameTree 函数检查两棵树是否相同
# 3. 如果当前节点不匹配, 继续递归检查左右子树
#
# 时间复杂度: O(m*n), m 和 n 分别是两棵树的节点数
# 空间复杂度: O(max(h1, h2)), 递归栈深度
#
# 是否为最优解: 否。更优解法是使用 KMP 或序列化+字符串匹配, 时间复杂度 O(m+n)
# 但递归解法更直观, 在面试中更常用

def is_subtree(root: Optional[TreeNode], sub_root: Optional[TreeNode]) -> bool:
    if root is None:
        return False
    # 检查当前节点是否与 subRoot 相同
    if is_same_tree(root, sub_root):
        return True
    # 递归检查左右子树
    return is_subtree(root.left, sub_root) or is_subtree(root.right, sub_root)

# LeetCode 617. 合并二叉树
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/merge-two-binary-trees/
# 题目描述: 给你两棵二叉树: root1 和 root2。
# 想象一下, 当你将其中一棵覆盖到另一棵之上时, 两棵树上的一些节点将会重叠。
# 你需要将这两棵树合并成一棵新二叉树。合并规则是:
# 如果两个节点重叠, 那么将这两个节点的值相加作为合并后节点的新值;
# 否则, 不为 null 的节点将直接作为新二叉树的节点。
#
# 思路分析:
# 1. 同时递归遍历两棵树
# 2. 如果两个节点都存在, 值相加
# 3. 如果只有一个节点存在, 直接使用该节点
```

```

# 时间复杂度: O(min(m, n))
# 空间复杂度: O(min(h1, h2))
#
# 是否为最优解: 是

def merge_trees(root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optional[TreeNode]:
    # 如果一棵树为空, 返回另一棵树
    if root1 is None:
        return root2
    if root2 is None:
        return root1
    # 创建新节点, 值为两节点之和
    merged = TreeNode(root1.val + root2.val)
    # 递归合并左右子树
    merged.left = merge_trees(root1.left, root2.left)
    merged.right = merge_trees(root1.right, root2.right)
    return merged

```

```

# LeetCode 654. 最大二叉树
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/maximum-binary-tree/
# 题目描述: 给定一个不重复的整数数组 nums。
# 最大二叉树可以用下面的算法从 nums 递归地构建:
# 1. 创建一个根节点, 其值为 nums 中的最大值。
# 2. 递归地在最大值左边的子数组前缀上构建左子树。
# 3. 递归地在最大值右边的子数组后缀上构建右子树。
#
# 思路分析:
# 1. 找到数组中的最大值及其索引
# 2. 最大值作为根节点
# 3. 递归构建左右子树
#
# 时间复杂度: O(n^2), 最坏情况下数组有序, 每次都要遍历剩余元素找最大值
# 空间复杂度: O(n), 递归栈深度
#
# 是否为最优解: 否。使用单调栈可以优化到 O(n) 时间复杂度
# 但递归解法更符合题意, 代码更简洁

```

```

def build_max_tree(nums: List[int], left: int, right: int) -> Optional[TreeNode]:
    if left > right:
        return None
    # 找到最大值的索引
    max_index = left

```

```
for i in range(left + 1, right + 1):
    if nums[i] > nums[max_index]:
        max_index = i
# 创建根节点
root = TreeNode(nums[max_index])
# 递归构建左右子树
root.left = build_max_tree(nums, left, max_index - 1)
root.right = build_max_tree(nums, max_index + 1, right)
return root
```

```
def construct_maximum_binary_tree(nums: List[int]) -> Optional[TreeNode]:
    return build_max_tree(nums, 0, len(nums) - 1)
```

```
# LeetCode 563. 二叉树的坡度
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/binary-tree-tilt/
# 题目描述: 给你一个二叉树的根节点 root，计算并返回 整个树 的坡度。
# 一个树的节点的坡度定义即为，该节点左子树的节点之和和右子树节点之和的差的绝对值。
# 如果没有左子树的话，左子树的节点之和为 0；没有右子树的话也是一样。
# 空节点的坡度是 0。整个树的坡度就是其所有节点的坡度之和。
#
# 思路分析:
# 1. 使用后序遍历，先计算子树的节点和
# 2. 对于每个节点，计算左右子树节点和的差的绝对值，累加到总坡度
# 3. 返回当前子树的节点和（包括根节点）
#
# 时间复杂度: O(n)
# 空间复杂度: O(h)
#
# 是否为最优解: 是
```

```
class TiltSolution:
    def __init__(self):
        self.total_tilt = 0

    def calculate_sum(self, node: Optional[TreeNode]) -> int:
        if node is None:
            return 0
        # 计算左右子树的节点和
        left_sum = self.calculate_sum(node.left)
        right_sum = self.calculate_sum(node.right)
```

```

# 累加当前节点的坡度
self.total_tilt += abs(left_sum - right_sum)
# 返回当前子树的节点和
return left_sum + right_sum + node.val

def find_tilt(self, root: Optional[TreeNode]) -> int:
    self.total_tilt = 0
    self.calculate_sum(root)
    return self.total_tilt

def find_tilt(root: Optional[TreeNode]) -> int:
    """便捷函数"""
    solution = TiltSolution()
    return solution.find_tilt(root)

# LeetCode 508. 出现次数最多的子树元素和
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/most-frequent-subtree-sum/
# 题目描述: 给你一个二叉树的根结点 root，计算出现最多的子树元素和。
# 一个结点的「子树元素和」定义为：以该结点为根的二叉树上所有结点的元素之和（包括结点自身）。
#
# 思路分析：
# 1. 使用后序遍历计算每个子树的元素和
# 2. 使用字典统计每个和出现的次数
# 3. 找出出现次数最多的所有和
#
# 时间复杂度: O(n)
# 空间复杂度: O(n)，需要字典存储每个子树和
#
# 是否为最优解: 是

def calculate_tree_sum(node: Optional[TreeNode],
                      sum_count: dict) -> int:
    if node is None:
        return 0
    # 计算左右子树的和
    left_sum = calculate_tree_sum(node.left, sum_count)
    right_sum = calculate_tree_sum(node.right, sum_count)
    # 当前子树的总和
    total_sum = left_sum + right_sum + node.val
    # 统计出现次数
    sum_count[total_sum] += 1

```

```

sum_count[total_sum] = sum_count.get(total_sum, 0) + 1
return total_sum

def find_frequent_tree_sum(root: Optional[TreeNode]) -> List[int]:
    sum_count = {}
    calculate_tree_sum(root, sum_count)
    if not sum_count:
        return []
    # 找到最大频率
    max_freq = max(sum_count.values())
    # 收集所有最大频率的和
    return [s for s, count in sum_count.items() if count == max_freq]

# LeetCode 437. 路径总和 III
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/path-sum-iii/
# 题目描述: 给定一个二叉树的根节点 root，和一个整数 targetSum，
# 求该二叉树里节点值之和等于 targetSum 的 路径 的数目。
# 路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只从父节点到子节点）。
#
# 思路分析:
# 方法 1: 双重递归 - 需要两个递归函数
#   1. 第一个递归遍历所有节点
#   2. 第二个递归以当前节点为起点计算路径数
# 方法 2: 前缀和+HashMap - 更优的解法
#   1. 使用前缀和思想，类似于数组的子数组和问题
#   2. 用字典存储前缀和及其出现次数
#
# 时间复杂度: 方法 1 O(n^2)，方法 2 O(n)
# 空间复杂度: 方法 1 O(h)，方法 2 O(n)
#
# 是否为最优解: 方法 2 是最优解，但方法 1 更直观。这里实现两种方法。

# 方法 1: 双重递归
def count_paths_from(node: Optional[TreeNode], target_sum: int) -> int:
    """计算从当前节点开始的路径数"""
    if node is None:
        return 0
    count = 0
    # 如果当前节点的值等于目标和，找到一条路径

```

```

if node.val == target_sum:
    count += 1
# 继续在左右子树中查找，目标和减去当前节点值
count += count_paths_from(node.left, target_sum - node.val)
count += count_paths_from(node.right, target_sum - node.val)
return count

def path_sum_iii(root: Optional[TreeNode], target_sum: int) -> int:
    if root is None:
        return 0
    # 以当前节点为起点的路径数 + 左子树中的路径数 + 右子树中的路径数
    return (count_paths_from(root, target_sum) +
            path_sum_iii(root.left, target_sum) +
            path_sum_iii(root.right, target_sum))

# 方法 2：前缀和 + 字典（最优解）
def dfs_with_prefix_sum(node: Optional[TreeNode], current_sum: int,
                        target_sum: int, prefix_sum_count: dict) -> int:
    if node is None:
        return 0
    # 当前路径的前缀和
    current_sum += node.val
    # 查找是否存在前缀和为 currentSum - targetSum 的路径
    count = prefix_sum_count.get(current_sum - target_sum, 0)
    # 将当前前缀和加入字典
    prefix_sum_count[current_sum] = prefix_sum_count.get(current_sum, 0) + 1
    # 递归遍历左右子树
    count += dfs_with_prefix_sum(node.left, current_sum, target_sum, prefix_sum_count)
    count += dfs_with_prefix_sum(node.right, current_sum, target_sum, prefix_sum_count)
    # 回溯：移除当前节点的前缀和
    prefix_sum_count[current_sum] -= 1
    return count

def path_sum_iii_optimal(root: Optional[TreeNode], target_sum: int) -> int:
    prefix_sum_count = {0: 1} # 初始化：前缀和为 0 的路径有一条
    return dfs_with_prefix_sum(root, 0, target_sum, prefix_sum_count)

# LeetCode 236. 二叉树的最近公共祖先
# 题目来源：LeetCode

```

```

# 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
# 题目描述: 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。
#
# 思路分析:
# 1. 使用递归, 分为三种情况:
#   a. 如果 p 和 q 分别在左右子树, 则当前节点就是最近公共祖先
#   b. 如果 p 和 q 都在左子树, 则最近公共祖先在左子树
#   c. 如果 p 和 q 都在右子树, 则最近公共祖先在右子树
# 2. 特殊情况: 当前节点就是 p 或 q, 且另一个节点在其子树中
#
# 时间复杂度: O(n), 最坏情况需要遍历所有节点
# 空间复杂度: O(h), 递归栈深度
#
# 是否为最优解: 是。这是经典的递归解法。
def lowest_common_ancestor(root: Optional[TreeNode],
                           p: TreeNode, q: TreeNode) -> Optional[TreeNode]:
    # 基础情况: 空节点或找到目标节点
    if root is None or root == p or root == q:
        return root
    # 递归在左右子树中查找
    left = lowest_common_ancestor(root.left, p, q)
    right = lowest_common_ancestor(root.right, p, q)
    # 如果左右子树都找到了, 说明 p 和 q 分别在左右两侧, 当前节点就是最近公共祖先
    if left is not None and right is not None:
        return root
    # 否则返回非空的一侧
    return left if left is not None else right

```

```

# LeetCode 124. 二叉树中的最大路径和
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
# 题目描述: 二叉树中的 路径 被定义为一条从树中任意节点出发,
# 沿父节点-子节点连接, 达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次。
# 该路径 至少包含一个 节点, 且不一定经过根节点。
# 路径和 是路径中各节点值的总和。给你一个二叉树的根节点 root , 返回其 最大路径和。
#
# 思路分析:
# 1. 对于每个节点, 最大路径和可能是:
#   - 左子树的最大贡献 + 节点值 + 右子树的最大贡献
# 2. 但返回给父节点时, 只能选择左或右一侧 (因为路径不能分叉)
# 3. 如果子树贡献为负, 则不选择该子树 (贡献为 0)
#

```

```

# 时间复杂度: O(n)
# 空间复杂度: O(h)
#
# 是否为最优解: 是。一次遍历即可得到答案。
#
# 这是 Hard 难度的经典题，需要理解“局部最优”和“向上返回”的区别

class MaxPathSumSolution:
    def __init__(self):
        self.max_path_sum = -2147483648 # Use integer instead of float

    def max_gain(self, node: Optional[TreeNode]) -> int:
        if node is None:
            return 0
        # 递归计算左右子树的最大贡献，负数则不选
        left_gain = max(self.max_gain(node.left), 0)
        right_gain = max(self.max_gain(node.right), 0)
        # 更新全局最大路径和: 左贡献 + 节点值 + 右贡献
        current_path_sum = left_gain + node.val + right_gain
        self.max_path_sum = max(self.max_path_sum, current_path_sum)
        # 返回节点的最大贡献: 节点值 + max(左贡献, 右贡献)
        return node.val + max(left_gain, right_gain)

    def max_path_sum_func(self, root: Optional[TreeNode]) -> int:
        self.max_path_sum = -2147483648
        self.max_gain(root)
        return self.max_path_sum

def max_path_sum_tree(root: Optional[TreeNode]) -> int:
    """便捷函数"""
    solution = MaxPathSumSolution()
    return solution.max_path_sum_func(root)

#
# LintCode 453. 将二叉树拆分为链表
# 题目来源: LintCode (炼码)
# 题目链接: https://www.lintcode.com/problem/453/
# 题目描述: 将一棵二叉树按照前序遍历拆解成为一个假链表。所谓的假链表是说，用二叉树的 right 指针，来表示链表中的 next 指针。
# 要求不能创建任何新的节点，只能调整树中节点指针的指向。
#
# 思路分析:

```

```
# 1. 使用后序遍历，先处理左右子树
# 2. 对于每个节点，将左子树变成链表，右子树变成链表
# 3. 将左子树链接到右子树上，左子树指针设为None
# 4. 返回当前链表的末尾节点
#
# 时间复杂度：O(n)
# 空间复杂度：O(h)
#
# 是否为最优解：是

def flatten_helper(node: Optional[TreeNode]) -> Optional[TreeNode]:
    """递归处理节点，返回处理后链表的末尾节点"""
    if node is None:
        return None

    # 叶子节点直接返回
    if node.left is None and node.right is None:
        return node

    # 递归处理左右子树
    left_tail = flatten_helper(node.left)
    right_tail = flatten_helper(node.right)

    # 如果左子树不为空，将左子树连接到右子树上
    if left_tail:
        left_tail.right = node.right
        node.right = node.left
        node.left = None

    # 返回新的链表末尾节点
    return right_tail if right_tail else left_tail

def flatten(root: Optional[TreeNode]) -> None:
    """将二叉树拆分为链表"""
    if root is None:
        return
    flatten_helper(root)

#
# HackerRank 二叉树的镜像
# 题目来源：HackerRank
# 题目描述：给定一棵二叉树，判断它是否是自身的镜像（即对称）
#
```

```

# 思路分析:
# 1. 使用辅助函数检查两棵子树是否互为镜像
# 2. 两棵子树互为镜像的条件:
#     - 当前节点值相等
#     - 左子树的左子树与右子树的右子树互为镜像
#     - 左子树的右子树与右子树的左子树互为镜像
#
# 时间复杂度: O(n)
# 空间复杂度: O(h)
#
# 是否为最优解: 是

def is_mirror_advanced(left: Optional[TreeNode], right: Optional[TreeNode]) -> bool:
    """检查两棵子树是否互为镜像"""
    if left is None and right is None:
        return True
    if left is None or right is None:
        return False

    return (left.val == right.val and
            is_mirror_advanced(left.left, right.right) and
            is_mirror_advanced(left.right, right.left))

def is_symmetric_advanced(root: Optional[TreeNode]) -> bool:
    """判断二叉树是否是自身的镜像"""
    if root is None:
        return True
    return is_mirror_advanced(root.left, root.right)

#
# 剑指 Offer 26. 树的子结构
# 题目来源: 剑指 Offer
# 题目描述: 输入两棵二叉树 A 和 B, 判断 B 是不是 A 的子结构。
#
# 思路分析:
# 1. 遍历树 A 的每个节点, 以该节点为根节点
# 2. 检查从该节点开始的子树是否包含树 B 的结构
#
# 时间复杂度: O(m*n)
# 空间复杂度: O(h)
#
# 是否为最优解: 是

def is_sub_structure_helper(a: Optional[TreeNode], b: Optional[TreeNode]) -> bool:
    """检查以 a 为根的子树是否包含树 b 的结构"""

```

```

if b is None:
    return True
if a is None or a.val != b.val:
    return False

return (is_sub_structure_helper(a.left, b.left) and
        is_sub_structure_helper(a.right, b.right))

def is_sub_structure(a: Optional[TreeNode], b: Optional[TreeNode]) -> bool:
    """判断 B 是否是 A 的子结构"""
    if a is None or b is None:
        return False

    # 检查当前节点，或者在左子树中检查，或者在右子树中检查
    return (is_sub_structure_helper(a, b) or
            is_sub_structure(a.left, b) or
            is_sub_structure(a.right, b))

# USACO 二叉搜索树的最近公共祖先
# 题目来源：USACO（美国计算机奥林匹克竞赛）
# 题目描述：给定一个二叉搜索树（BST），找到该树中两个指定节点的最近公共祖先。
#
# 思路分析：
# 利用 BST 的特性：左子树所有节点值小于根节点，右子树所有节点值大于根节点
# 1. 如果 p 和 q 的值都小于当前节点，那么 LCA 在左子树
# 2. 如果 p 和 q 的值都大于当前节点，那么 LCA 在右子树
# 3. 否则，当前节点就是 LCA
#
# 时间复杂度：O(h)
# 空间复杂度：O(h)
#
# 是否为最优解：是

def lowest_common_ancestor_bst(root: Optional[TreeNode],
                               p: TreeNode, q: TreeNode) -> Optional[TreeNode]:
    """在二叉搜索树中查找最近公共祖先"""
    if root is None or p is None or q is None:
        return None

    # 如果 p 和 q 都在左子树
    if p.val < root.val and q.val < root.val:
        return lowest_common_ancestor_bst(root.left, p, q)
    # 如果 p 和 q 都在右子树

```

```

if p.val > root.val and q.val > root.val:
    return lowest_common_ancestor_bst(root.right, p, q)
# 否则当前节点就是 LCA
return root

# AtCoder ABC191 E. Come Back Quickly - 距离和计算
# 题目描述简化：给定一棵有根树，计算每个节点到其所有子孙节点的距离之和
#
# 思路分析：
# 1. 使用后序遍历计算每个子树的节点数
# 2. 使用前序遍历计算距离之和
#
# 时间复杂度：O(n)
# 空间复杂度：O(n)
#
# 是否为最优解：是

def dfs_size(node: Optional[TreeNode], size: dict) -> int:
    """后序遍历计算每个子树的节点数"""
    if node is None:
        return 0

    size[node.val] = 1 # 包含自己
    size[node.val] += dfs_size(node.left, size)
    size[node.val] += dfs_size(node.right, size)

    return size[node.val]

def dfs_distance(node: Optional[TreeNode], size: dict,
                 result: dict, parent_distance: int) -> None:
    """前序遍历计算距离和"""
    if node is None:
        return

    result[node.val] = parent_distance

    if node.left:
        left_size = size[node.left.val]
        right_size = size[node.right.val] if node.right else 0
        left_distance = parent_distance + (size[node.val] - left_size) - left_size
        dfs_distance(node.left, size, result, left_distance)

    if node.right:

```

```

right_size = size[node.right.val]
left_size = size[node.left.val] if node.left else 0
right_distance = parent_distance + (size[node.val] - right_size) - right_size
dfs_distance(node.right, size, result, right_distance)

def calculate_distance_sum(root: Optional[TreeNode]) -> dict:
    """计算每个节点到其所有子孙节点的距离之和"""
    if root is None:
        return {}

    size = {}
    result = {}

    dfs_size(root, size)
    dfs_distance(root, size, result, 0)

    return result

# CodeChef - SUBTREE - 最大子树和
# 题目描述简化：给定一棵二叉树，每个节点有一个权值。找出权值和最大的子树。
#
# 思路分析：
# 1. 使用后序遍历，计算每个子树的权值和
# 2. 对于每个节点，其最大子树和为：节点值 + max(左子树最大和, 0) + max(右子树最大和, 0)
#
# 时间复杂度：O(n)
# 空间复杂度：O(h)
#
# 是否为最优解：是

class MaxSubtreeSumSolution:
    def __init__(self):
        self.max_subtree_sum = -2147483648 # 使用整数最小值

    def calculate_subtree_sum(self, node: Optional[TreeNode]) -> int:
        """计算子树和并更新最大子树和"""
        if node is None:
            return 0

        left_sum = max(self.calculate_subtree_sum(node.left), 0)
        right_sum = max(self.calculate_subtree_sum(node.right), 0)

        current_sum = node.val + left_sum + right_sum
        if current_sum > self.max_subtree_sum:
            self.max_subtree_sum = current_sum

        return current_sum

```



```

# 如果没有哈希表，使用线性查找
if root_index == in_start and inorder[root_index] != root_val:
    for i in range(in_start, in_end + 1):
        if inorder[i] == root_val:
            root_index = i
            break

left_size = root_index - in_start

root.left = build_tree_helper(preorder, pre_start + 1, pre_start + left_size,
                             inorder, in_start, root_index - 1, inorder_map)
root.right = build_tree_helper(preorder, pre_start + left_size + 1, pre_end,
                             inorder, root_index + 1, in_end, inorder_map)

return root

def build_tree(preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
    """根据前序和中序遍历结果重建二叉树"""
    if not preorder or not inorder:
        return None

    # 创建中序遍历值到索引的映射，优化查找
    inorder_map = {val: idx for idx, val in enumerate(inorder)}

    return build_tree_helper(preorder, 0, len(preorder) - 1,
                           inorder, 0, len(inorder) - 1, inorder_map)

# 牛客网 NC102. 树的序列化和反序列化
# 题目描述：将二叉树序列化为字符串，然后从字符串反序列化回二叉树
#
# 思路分析：
# 序列化使用前序遍历，空节点用特殊字符表示
#
# 时间复杂度：O(n)
# 空间复杂度：O(n)
#
# 是否为最优解：是
def serialize_helper(node: Optional[TreeNode], result: list) -> None:
    """递归序列化二叉树"""
    if node is None:
        result.append("#")
        return

    result.append(str(node.val))
    result.append("(")
    serialize_helper(node.left, result)
    result.append(", ")
    serialize_helper(node.right, result)
    result.append(")")

    return

```

```
result.append(str(node.val))
serialize_helper(node.left, result)
serialize_helper(node.right, result)

def serialize(root: Optional[TreeNode]) -> str:
    """将二叉树序列化为字符串"""
    result = []
    serialize_helper(root, result)
    return ",".join(result)

def deserialize_helper(nodes: list, index: list) -> Optional[TreeNode]:
    """递归反序列化二叉树"""
    if index[0] >= len(nodes) or nodes[index[0]] == "#":
        index[0] += 1
        return None

    node = TreeNode(int(nodes[index[0]]))
    index[0] += 1
    node.left = deserialize_helper(nodes, index)
    node.right = deserialize_helper(nodes, index)

    return node

def deserialize(data: str) -> Optional[TreeNode]:
    """将字符串反序列化为二叉树"""
    if not data:
        return None

    nodes = data.split(",")
    index = [0] # 使用列表存储索引，以便在递归中修改
    return deserialize_helper(nodes, index)

# 杭电 OJ 2024 - 二叉树遍历
# 题目描述：输入二叉树的前序遍历和中序遍历结果，输出其后序遍历结果
#
# 思路分析：
# 1. 先根据前序和中序构建二叉树
# 2. 然后进行后序遍历输出
#
# 时间复杂度：O(n^2)
# 空间复杂度：O(n)
```

```

#
# 最优解：可以使用哈希表优化查找过程
def postorder_helper(preorder: str, pre_start: int, pre_end: int,
                     inorder: str, in_start: int, in_end: int,
                     result: list, inorder_map: dict) -> None:
    """递归构建二叉树并生成后序遍历结果"""
    if pre_start > pre_end or in_start > in_end:
        return

    root_val = preorder[pre_start]

    # 找到根节点在中序中的位置
    root_index = inorder_map.get(root_val, in_start)
    # 如果没有哈希表，使用线性查找
    if root_index == in_start and inorder[root_index] != root_val:
        for i in range(in_start, in_end + 1):
            if inorder[i] == root_val:
                root_index = i
                break

    left_length = root_index - in_start

    # 递归处理左右子树
    postorder_helper(preorder, pre_start + 1, pre_start + left_length,
                     inorder, in_start, root_index - 1, result, inorder_map)
    postorder_helper(preorder, pre_start + left_length + 1, pre_end,
                     inorder, root_index + 1, in_end, result, inorder_map)

    # 后序：添加根节点
    result.append(root_val)

def postorder_from_preorder_and_inorder(preorder: str, inorder: str) -> str:
    """根据前序和中序遍历结果生成后序遍历结果"""
    if not preorder or not inorder:
        return ""

    # 创建中序遍历字符到索引的映射，优化查找
    inorder_map = {char: idx for idx, char in enumerate(inorder)}

    result = []
    postorder_helper(preorder, 0, len(preorder) - 1,
                     inorder, 0, len(inorder) - 1, result, inorder_map)

```

```
return "".join(result)

# 主函数测试
if __name__ == "__main__":
    print("===== 二叉树递归遍历基础测试 =====")
    head = TreeNode(1)
    head.left = TreeNode(2)
    head.right = TreeNode(3)
    head.left.left = TreeNode(4)
    head.left.right = TreeNode(5)
    head.right.left = TreeNode(6)
    head.right.right = TreeNode(7)

    print("前序遍历: ", end="")
    pre_order(head)
    print()

    print("中序遍历: ", end="")
    in_order(head)
    print()

    print("后序遍历: ", end="")
    pos_order(head)
    print()

    print("\n===== LeetCode 104. 最大深度 =====")
    print(f"最大深度: {max_depth(head)}") # 预期: 3

    print("\n===== LeetCode 110. 平衡二叉树 =====")
    balanced_tree = TreeNode(1)
    balanced_tree.left = TreeNode(2)
    balanced_tree.right = TreeNode(3)
    balanced_tree.left.left = TreeNode(4)
    balanced_tree.left.right = TreeNode(5)
    print(f"是否为平衡二叉树: {is_balanced(balanced_tree)}") # 预期: True

    print("\n===== LeetCode 112. 路径总和 =====")
    path_tree = TreeNode(5)
    path_tree.left = TreeNode(4)
    path_tree.right = TreeNode(8)
    path_tree.left.left = TreeNode(11)
    path_tree.left.left.left = TreeNode(7)
```

```

path_tree.left.left.right = TreeNode(2)
print(f"是否存在路径和为 22: {has_path_sum(path_tree, 22)}") # 预期: True

print("\n===== LeetCode 113. 路径总和 II =====")
paths = path_sum(path_tree, 22)
print(f"路径总和为 22 的所有路径: {paths}") # 预期: [[5, 4, 11, 2]]

print("\n===== 所有测试完成! =====")

# ===== 更多平台题目实现 =====

# 洛谷 P1305 新二叉树
# 题目来源: 洛谷 (Luogu)
# 题目链接: https://www.luogu.com.cn/problem/P1305
# 题目描述: 根据前序遍历字符串构建二叉树并输出中序遍历
#
# 思路分析:
# 1. 前序遍历字符串中, '#' 表示空节点
# 2. 使用递归构建二叉树
# 3. 输出中序遍历结果
#
# 时间复杂度: O(n)
# 空间复杂度: O(n)
#
# 是否为最优解: 是

class P1305Solution:

    def __init__(self):
        self.index = 0

    def build_tree(self, preorder: str) -> Optional[TreeNode]:
        """根据前序遍历字符串构建二叉树"""
        if self.index >= len(preorder) or preorder[self.index] == '#':
            self.index += 1
            return None
        root = TreeNode(preorder[self.index])
        self.index += 1
        root.left = self.build_tree(preorder)
        root.right = self.build_tree(preorder)
        return root

    def inorder_traversal(self, root: Optional[TreeNode]) -> List[str]:
        """中序遍历二叉树"""

```

```

result = []
self._inorder_helper(root, result)
return result

def _inorder_helper(self, node: Optional[TreeNode], result: List[str]) -> None:
    if node is None:
        return
    self._inorder_helper(node.left, result)
    result.append(str(node.val))
    self._inorder_helper(node.right, result)

# TimusOJ 1022 Genealogical Tree
# 题目来源: Timus Online Judge
# 题目链接: http://acm.timus.ru/problem.aspx?space=1&num=1022
# 题目描述: 给定家族关系, 构建家谱树并输出拓扑排序
#
# 思路分析:
# 1. 使用邻接表表示有向无环图
# 2. 使用深度优先搜索进行拓扑排序
# 3. 使用后序遍历得到拓扑序列
#
# 时间复杂度: O(n + m)
# 空间复杂度: O(n)
#
# 是否为最优解: 是

class TimusOJ1022:

    def topological_sort(self, n: int, edges: List[List[int]]) -> List[int]:
        """
        拓扑排序
        """
        graph = [[] for _ in range(n + 1)]
        for u, v in edges:
            graph[u].append(v)

        visited = [False] * (n + 1)
        result = []

        for i in range(1, n + 1):
            if not visited[i]:
                self._dfs(i, graph, visited, result)

        return result[::-1] # 反转得到拓扑排序

    def _dfs(self, node: int, graph: List[List[int]], visited: List[bool], result: List[int]) ->

```

None:

```
"""深度优先搜索"""
visited[node] = True
for neighbor in graph[node]:
    if not visited[neighbor]:
        self._dfs(neighbor, graph, visited, result)
result.append(node)
```

# AizuOJ ALDS1\_7\_C Tree Walk

# 题目来源: Aizu Online Judge

# 题目链接: [http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_7\\_C](http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_7_C)

# 题目描述: 实现二叉树的前序、中序、后序遍历

#

# 思路分析:

# 1. 标准的二叉树遍历实现

# 2. 分别实现三种遍历方式

# 3. 输出遍历结果

#

# 时间复杂度: O(n)

# 空间复杂度: O(h)

#

# 是否为最优解: 是

class AizuOJTreeWalk:

```
def preorder(self, root: Optional[TreeNode]) -> List[int]:
```

```
"""前序遍历"""
result = []
self._preorder_helper(root, result)
```

```
return result
```

```
def _preorder_helper(self, node: Optional[TreeNode], result: List[int]) -> None:
```

```
if node is None:
```

```
    return
```

```
result.append(node.val)
```

```
self._preorder_helper(node.left, result)
```

```
self._preorder_helper(node.right, result)
```

```
def inorder(self, root: Optional[TreeNode]) -> List[int]:
```

```
"""中序遍历"""
result = []
self._inorder_helper(root, result)
```

```
return result
```

```

def _inorder_helper(self, node: Optional[TreeNode], result: List[int]) -> None:
    if node is None:
        return
    self._inorder_helper(node.left, result)
    result.append(node.val)
    self._inorder_helper(node.right, result)

def postorder(self, root: Optional[TreeNode]) -> List[int]:
    """后序遍历"""
    result = []
    self._postorder_helper(root, result)
    return result

def _postorder_helper(self, node: Optional[TreeNode], result: List[int]) -> None:
    if node is None:
        return
    self._postorder_helper(node.left, result)
    self._postorder_helper(node.right, result)
    result.append(node.val)

# POJ 2255 Tree Recovery
# 题目来源: 北京大学 POJ
# 题目链接: http://poj.org/problem?id=2255
# 题目描述: 根据前序遍历和中序遍历重建二叉树
#
# 思路分析:
# 1. 前序遍历的第一个节点是根节点
# 2. 在中序遍历中找到根节点的位置
# 3. 递归重建左右子树
#
# 时间复杂度: O(n^2)
# 空间复杂度: O(n)
#
# 是否为最优解: 是 (可以使用哈希表优化到 O(n) )
class POJ2255:

    def build_tree(self, preorder: List[str], inorder: List[str]) -> Optional[TreeNode]:
        """根据前序和中序遍历重建二叉树"""
        if not preorder or not inorder:
            return None

        root_val = preorder[0]
        root = TreeNode(root_val)

```

```

# 找到根节点在中序遍历中的位置
root_index = inorder.index(root_val)

# 递归构建左右子树
root.left = self.build_tree(preorder[1:1+root_index], inorder[:root_index])
root.right = self.build_tree(preorder[1+root_index:], inorder[root_index+1:])

return root

def get_postorder(self, root: Optional[TreeNode]) -> List[str]:
    """获取后序遍历结果"""
    result = []
    self._postorder(root, result)
    return result

def _postorder(self, node: Optional[TreeNode], result: List[str]) -> None:
    if node is None:
        return
    self._postorder(node.left, result)
    self._postorder(node.right, result)
    result.append(node.val)

# ZOJ 1944 Tree Recovery
# 题目来源: 浙江大学 ZOJ
# 题目链接: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1944
# 题目描述: 根据前序遍历和中序遍历重建二叉树 (与 POJ2255 相同)
#
# 思路分析: 同 POJ2255
class ZOJ1944:

    def build_tree(self, preorder: List[str], inorder: List[str]) -> Optional[TreeNode]:
        """根据前序和中序遍历重建二叉树"""
        if not preorder or not inorder:
            return None

        root_val = preorder[0]
        root = TreeNode(root_val)

        root_index = inorder.index(root_val)

        root.left = self.build_tree(preorder[1:1+root_index], inorder[:root_index])
        root.right = self.build_tree(preorder[1+root_index:], inorder[root_index+1:])

        return root

```

```

    return root

# HDU 1710 Binary Tree Traversals
# 题目来源: 杭州电子科技大学 HDU
# 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1710
# 题目描述: 根据前序遍历和中序遍历输出后序遍历
#
# 思路分析:
# 1. 直接构建后序遍历序列, 无需构建完整二叉树
# 2. 使用递归分治思想
#
# 时间复杂度: O(n^2)
# 空间复杂度: O(n)
#
# 是否为最优解: 是
class HDU1710:

    def get_postorder(self, preorder: List[int], inorder: List[int]) -> List[int]:
        """直接获取后序遍历序列"""
        if not preorder:
            return []

        root_val = preorder[0]
        root_index = inorder.index(root_val)

        left_post = self.get_postorder(preorder[1:1+root_index], inorder[:root_index])
        right_post = self.get_postorder(preorder[1+root_index:], inorder[root_index+1:])

        return left_post + right_post + [root_val]

# LOJ 10155 二叉苹果树
# 题目来源: LibreOJ
# 题目链接: https://loj.ac/p/10155
# 题目描述: 二叉树上有苹果, 要求保留指定数量的树枝, 使得苹果总数最大
#
# 思路分析:
# 1. 树形动态规划问题
# 2. 使用递归遍历计算每个子树的最优解
# 3. 状态转移: dp[node][k] = 保留 k 条树枝时的最大苹果数
#
# 时间复杂度: O(n * k^2)

```

```

# 空间复杂度: O(n * k)
#
# 是否为最优解: 是
class L0J10155:

    def max_apples(self, root: Optional[TreeNode], k: int) -> int:
        """计算最大苹果数"""
        # dp[node][j] 表示以 node 为根的子树保留 j 条树枝的最大苹果数
        dp = {}
        self._dfs(root, k, dp)
        return dp[root][k]

    def _dfs(self, node: Optional[TreeNode], k: int, dp: dict) -> None:
        """深度优先搜索计算 DP"""
        if node is None:
            return

        # 初始化 DP 表
        dp[node] = [0] * (k + 1)

        # 递归处理左右子树
        if node.left:
            self._dfs(node.left, k, dp)
        if node.right:
            self._dfs(node.right, k, dp)

        # 状态转移
        for i in range(k + 1):
            for j in range(i + 1):
                left_val = dp[node.left][j] if node.left else 0
                right_val = dp[node.right][i - j] if node.right else 0
                dp[node][i] = max(dp[node][i], left_val + right_val)

        # 考虑当前节点的苹果
        for i in range(k, 0, -1):
            dp[node][i] = dp[node][i - 1] + (node.val if hasattr(node, 'apple_count') else 1)

# CodeChef SUBTREE - 子树移除
# 题目来源: CodeChef
# 题目链接: https://www.codechef.com/problems/SUBTREE
# 题目描述: 计算二叉树中所有子树的大小之和
#
# 思路分析:

```

```

# 1. 使用后序遍历计算每个子树的大小
# 2. 累加所有子树的大小
#
# 时间复杂度: O(n)
# 空间复杂度: O(h)
#
# 是否为最优解: 是
class CodeChefSUBTREE:

    def sum_subtree_sizes(self, root: Optional[TreeNode]) -> int:
        """计算所有子树大小之和"""
        self.total = 0
        self._dfs(root)
        return self.total

    def _dfs(self, node: Optional[TreeNode]) -> int:
        """深度优先搜索计算子树大小"""
        if node is None:
            return 0

        left_size = self._dfs(node.left)
        right_size = self._dfs(node.right)
        subtree_size = left_size + right_size + 1

        # 累加当前子树大小
        self.total += subtree_size

        return subtree_size

# USACO 二叉搜索树的最近公共祖先
# 题目来源: USACO (美国计算机奥林匹克竞赛)
# 题目描述: 在二叉搜索树中查找两个节点的最近公共祖先
#
# 思路分析:
# 1. 利用 BST 的性质: 左子树所有节点值小于根节点, 右子树所有节点值大于根节点
# 2. 如果 p 和 q 的值都小于当前节点, LCA 在左子树
# 3. 如果 p 和 q 的值都大于当前节点, LCA 在右子树
# 4. 否则当前节点就是 LCA
#
# 时间复杂度: O(h)
# 空间复杂度: O(h)
#
# 是否为最优解: 是

```

```

class USACOLCA:
    def lowest_common_ancestor_bst(self, root: Optional[TreeNode], p: TreeNode, q: TreeNode) ->
Optional[TreeNode]:
        """BST 中的最近公共祖先"""
        if root is None or p is None or q is None:
            return None

        # 确保 p 的值小于 q 的值, 方便比较
        if p.val > q.val:
            p, q = q, p

        if p.val < root.val and q.val < root.val:
            return self.lowest_common_ancestor_bst(root.left, p, q)
        elif p.val > root.val and q.val > root.val:
            return self.lowest_common_ancestor_bst(root.right, p, q)
        else:
            return root

# AtCoder ABC191 E. Come Back Quickly
# 题目来源: AtCoder
# 题目链接: https://atcoder.jp/contests/abc191/tasks/abc191_e
# 题目描述: 计算树中每个节点到其所有子孙节点的距离之和
#
# 思路分析:
# 1. 第一次 DFS 计算每个子树的大小
# 2. 第二次 DFS 计算距离之和
#
# 时间复杂度: O(n)
# 空间复杂度: O(n)
#
# 是否为最优解: 是
class AtCoderABC191E:
    def calculate_distance_sum(self, root: Optional[TreeNode], n: int) -> List[int]:
        """计算每个节点到子孙节点的距离之和"""
        # 子树大小
        size = [0] * (n + 1)
        # 距离之和
        result = [0] * (n + 1)

        self._dfs_size(root, size)
        self._dfs_distance(root, size, result, 0)

```

```

        return result[1:] # 返回节点 1 到 n 的结果

def _dfs_size(self, node: Optional[TreeNode], size: List[int]) -> int:
    """计算子树大小"""
    if node is None:
        return 0

    size[node.val] = 1
    size[node.val] += self._dfs_size(node.left, size)
    size[node.val] += self._dfs_size(node.right, size)

    return size[node.val]

def _dfs_distance(self, node: Optional[TreeNode], size: List[int], result: List[int],
parent_distance: int) -> None:
    """计算距离之和"""
    if node is None:
        return

    result[node.val] = parent_distance

    if node.left:
        left_size = size[node.left.val]
        right_size = size[node.right.val] if node.right else 0
        left_distance = parent_distance + (size[node.val] - left_size) - left_size
        self._dfs_distance(node.left, size, result, left_distance)

    if node.right:
        right_size = size[node.right.val]
        left_size = size[node.left.val] if node.left else 0
        right_distance = parent_distance + (size[node.val] - right_size) - right_size
        self._dfs_distance(node.right, size, result, right_distance)

# 测试更多平台题目
def test_additional_platforms():
    print("\n" + "="*50)
    print("更多平台题目测试")
    print("="*50)

# 测试洛谷 P1305
print("\n--- 洛谷 P1305 新二叉树 ---")
p1305 = P1305Solution()

```

```

preorder_str = "ABD##E##CF##G##"
tree = p1305.build_tree(preorder_str)
inorder_result = p1305.inorder_traversal(tree)
print(f"前序遍历: {preorder_str}")
print(f"中序遍历: {' '.join(inorder_result)}")

# 测试 POJ2255
print("\n--- POJ 2255 Tree Recovery ---")
poj2255 = POJ2255()
preorder = ['A', 'B', 'D', 'E', 'C', 'F', 'G']
inorder = ['D', 'B', 'E', 'A', 'F', 'C', 'G']
tree = poj2255.build_tree(preorder, inorder)
postorder = poj2255.get_postorder(tree)
print(f"前序: {preorder}")
print(f"中序: {inorder}")
print(f"后序: {postorder}")

# 测试 HDU1710
print("\n--- HDU 1710 Binary Tree Traversals ---")
hdu1710 = HDU1710()
preorder_nums = [1, 2, 4, 5, 3, 6, 7]
inorder_nums = [4, 2, 5, 1, 6, 3, 7]
postorder_nums = hdu1710.get_postorder(preorder_nums, inorder_nums)
print(f"前序: {preorder_nums}")
print(f"中序: {inorder_nums}")
print(f"后序: {postorder_nums}")

# 测试 CodeChef SUBTREE
print("\n--- CodeChef SUBTREE - 子树大小之和 ---")
codechef = CodeChefSUBTREE()
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
total_size = codechef.sum_subtree_sizes(root)
print(f"所有子树大小之和: {total_size} # 预期: 15 (5 个节点, 每个子树大小之和)

print("\n" + "="*50)
print("所有平台题目测试完成!")
print("=".*50)

```

=====