动态规划 知识点

主讲人：陈驰水

邮箱：walter.chen.bj@gmail.com

目录

- 从记忆化搜索到动态规划
- 经典线性 DP 问题
- 网格 DP 问题
- 背包问题
- 状态机 DP 问题
- 划分 DP 问题

扩展

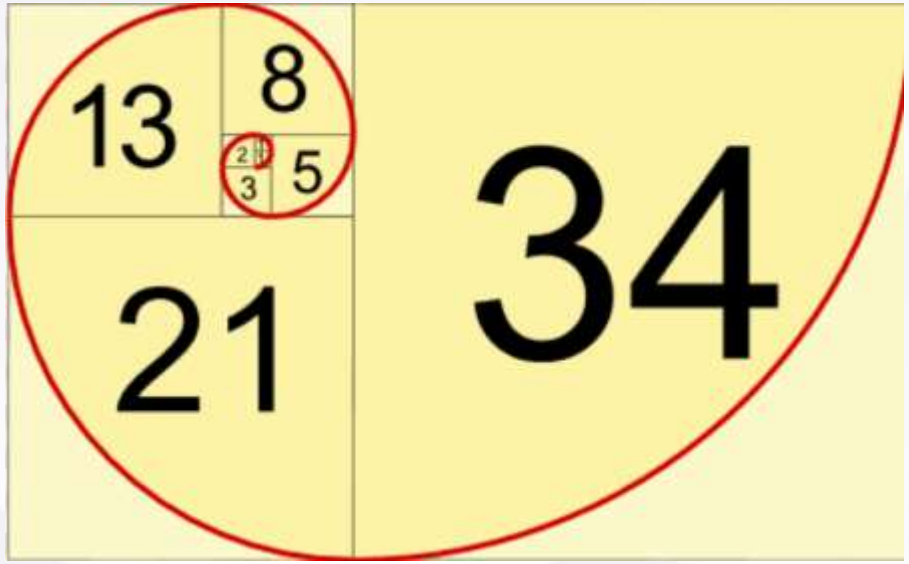
- 区间 DP 问题
- 数位 DP 问题
- 状态压缩 DP 问题
- 树上 DP 问题

引入：斐波那契数列

斐波那契数 （通常用 F(n) 表示）形成的序列称为 斐波那契数列 。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

$F(0) = 0$，$F(1) = 1$

$F(n) = F(n - 1) + F(n - 2)$，其中 $n > 1$



从记忆化搜索到动态规划

对于这个问题我们第一直觉就是可以用「递归」来解。代码如右图所示：

```
int func(int x) {
    if (x == 1 || x == 2) {
        return 1;
    }
    return func( x: x - 1) + func( x: x - 2);
}
```

但是进一步想，递归的做法是最好的吗？大家想一想时间复杂度是多少？

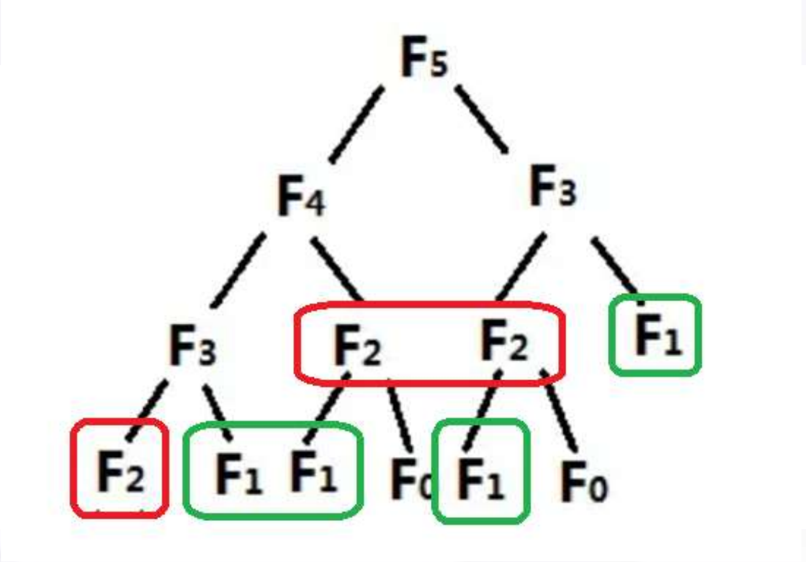
$$O(2^n)$$

从记忆化搜索到动态规划

斐波那契的二叉树结构

显然，对于每次进入递归的过程中，将延伸出两次递归，因此为了方便起见我们将其画成二叉树结构。

通过观察，我们发现二叉树中有很多重复的节点，这些节点被反复的计算。我们可以通过备忘录来解决这个问题，使得复杂度变为线性。



从记忆化搜索到动态规划

记忆化搜索

```
// 用哈希表作为备忘录
unordered_map<int, int>memo;

int func(int x) {
    // 进入递归后先读取哈希表
    if (memo.count(x)) {
        return memo[x];
    }
    if (x == 1 || x == 2) {
        return 1;
    }
    // 将计算结构存入哈希表
    return memo[x] = func( x: x - 1) + func( x: x - 2);
}
```

动态规划

记忆化搜索并不能解决所有问题，并且书写也相对麻烦，因此我们可以将其转化为「动态规划」

核心为两点：

1. **状态转移方程**
2. **初始状态**

```cpp
// 用哈希表作为备忘录
unordered_map<int, int>memo;

int func(int x) {
    // 进入递归后先读取哈希表
    if (memo.count(x)) {
        return memo[x];
    }
    if (x == 1 || x == 2) {
        return 1;
    }
    // 将计算结构存入哈希表
    return memo[x] = func( x: x - 1) + func( x: x - 2);
}
```

初始状态

状态转移方程

从记忆化搜索到动态规划

```cpp
// 用哈希表作为备忘录
unordered_map<int, int>memo;

int func(int x) {
    // 进入递归后先读取哈希表
    if (memo.count(x)) {
        return memo[x];
    }
    if (x == 1 || x == 2) {
        return 1;
    }
    // 将计算结构存入哈希表
    return memo[x] = func( x: x - 1) + func( x: x - 2);
}
```

初始状态

状态转移方程

```cpp
int main() {
    int n = 30; // 我要求的斐波那契项数
    // dp 数组 就是之前的备忘录
    vector<int>dp( n: n + 1,  value: 0);
    // 初始状态
    dp[0] = 1;
    dp[1] = 1;
    // 自底向上执行状态转移方程
    for(int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    cout << dp[n]; // dp[n] 就是想要的答案
    return 0;
}
```

此题更适合自底向上遍历，具体要根据题目要求

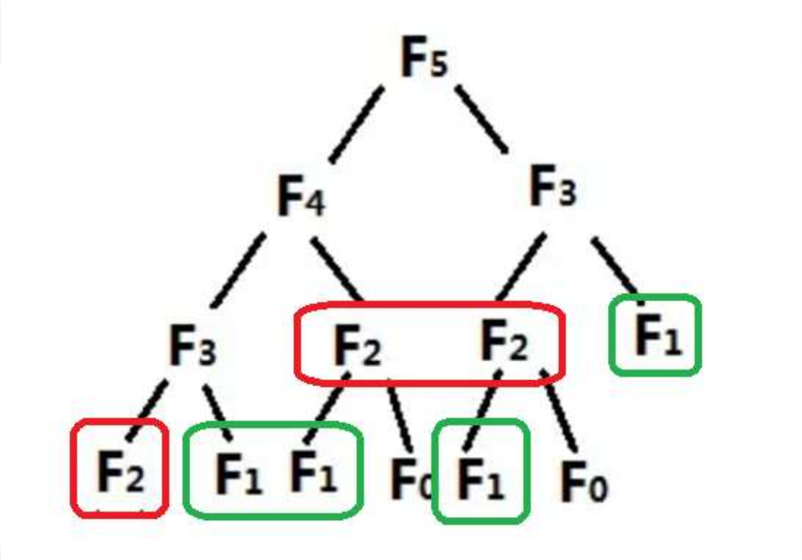
时间复杂度都是 O(n)

斐波那契数列的扩展思考

1. 能否用 O(1) 的空间实现?

2. 当斐波那契要求的项数更大的时候，有没有办法获得比线性复杂度更低的时间复杂度?

提示：用矩阵快速幂



练习题：

1. 使用最小花费爬楼梯（斐波那契进阶）

   https://leetcode.cn/problems/min-cost-climbing-stairs/description/

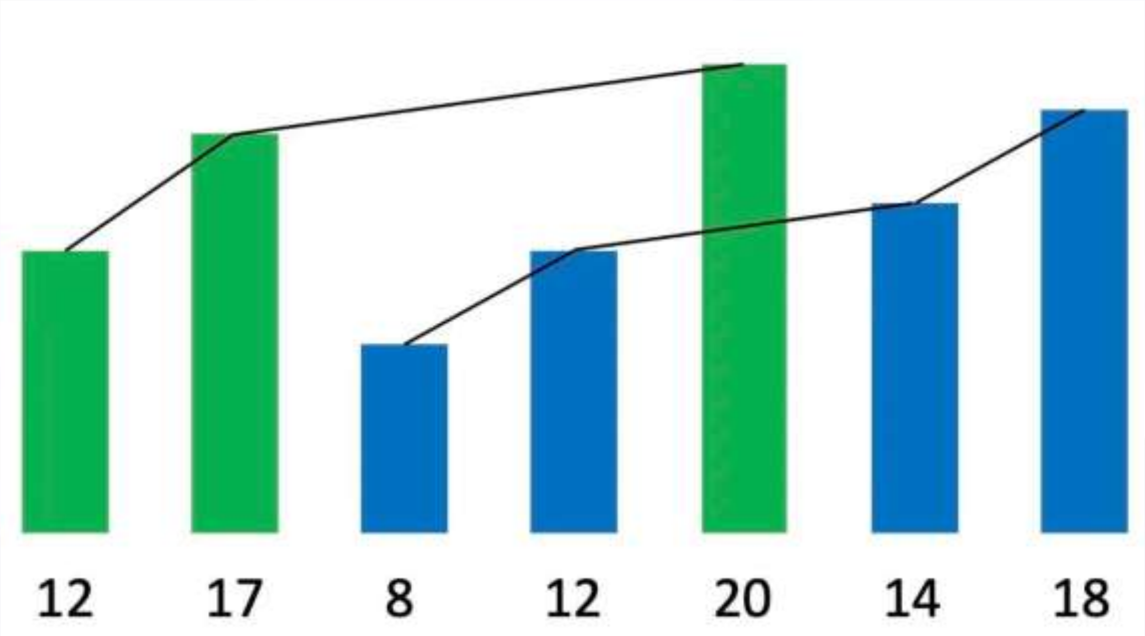
2. 打家劫舍（经典动态规划）

   https://leetcode.cn/problems/house-robber/

经典线性 DP 问题

最长递增子序列（LIS）

子序列 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。

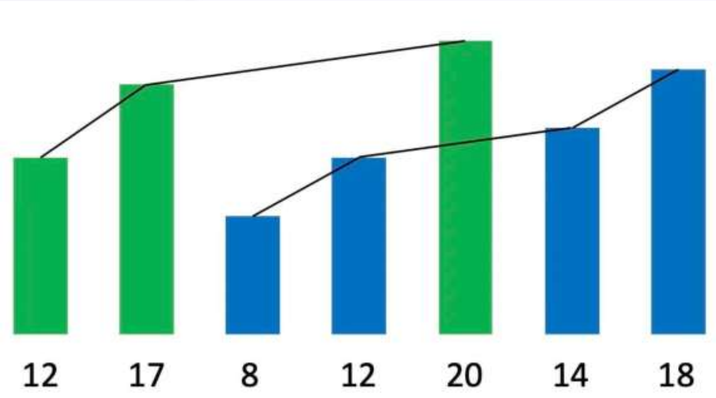


线性 DP 求解

1. 定义 dp[i] 表示为以 a[i] 为末尾的最长上升子序列长度
2. 状态转移方程：

   $dp[i] = \max\{dp[j] + 1\}$ 其中 $j < i$ 且 $a[j] < a[i]$

3. 初始状态设置 dp[i] 全部为 1



线性 DP 求解

1. 定义 dp[i] 表示为以 a[i] 为末尾的最长上升子序列长度
2. 状态转移方程：

   $dp[i] = \max\{dp[j] + 1\}$ 其中 $j < i$ 且 $a[j] < a[i]$

3. 初始状态设置 dp[i] 全部为 1

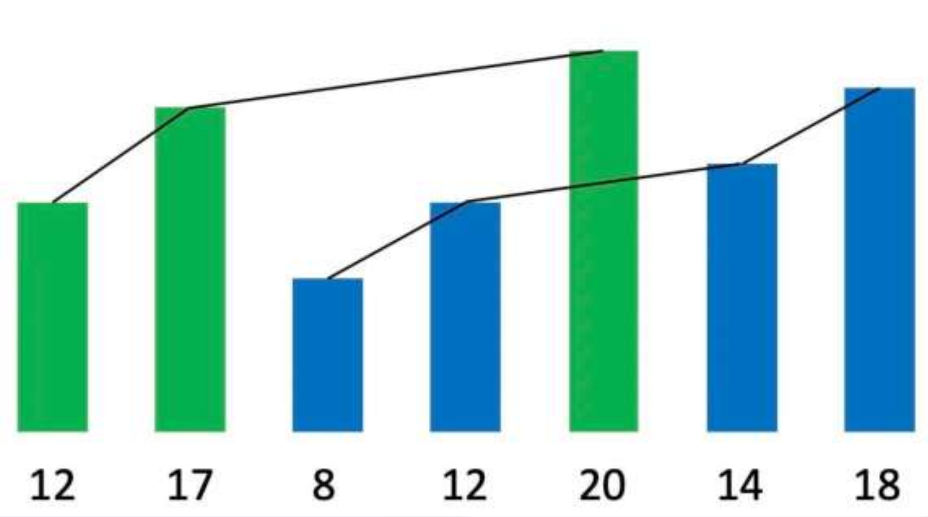
时间复杂度：$\mathcal{O}(n^2)$

```cpp
int LIS(vector<int>& nums) {
    // 此题dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度。
    int n = nums.size();
    vector<int> dp(n, value: 1); // 最小结果是 1 ，是初始状态
    for (int i = 0; i < n; i++) {
        int tail = nums[i];
        // 下面是找最大的 dp[j] 是状态转移方程
        for (int j = 0; j < i; j++) {
            if (tail > nums[j]) // 可以直接拼接出新子序列
                dp[i] = max(dp[i], dp[j] + 1);
        }
    }
    // 此题 dp[n - 1] 不一定是最大的，要遍历找出 dp 数组中的最大值
    int ans = 0;
    for (auto i :int : dp) {
        ans = max(ans, i);
    }
    return ans;
}
```

最长递增子序列（LIS）扩展思考

能不能用 O(n * log(n)) 的时间复杂度完成？

提示：二分查找



经典线性 DP 问题

练习题：

1. 俄罗斯套娃信封问题（二维 LIS）

   https://leetcode.cn/circle/discuss/tXLS3i/

2. 最长公共子序列（LCS问题）

   https://leetcode.cn/problems/longest-common-subsequence/description/

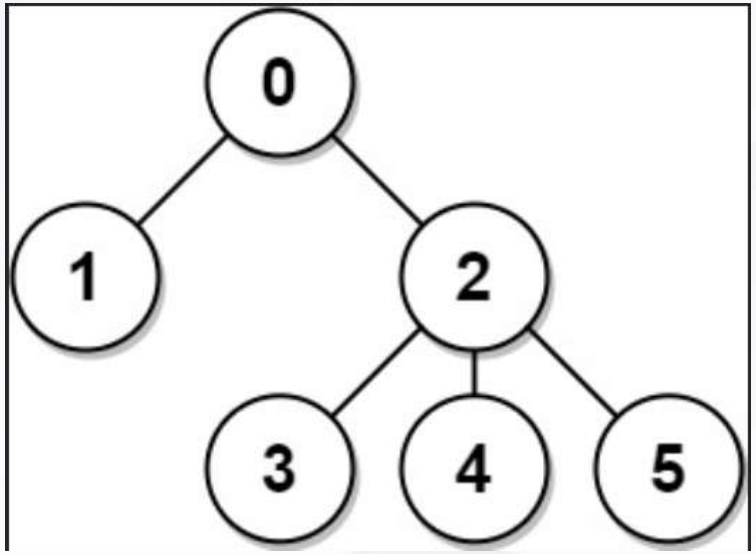
树上 DP 问题

换根 DP

给定一个无向、连通的树。树中有 n 个标记为 0...n-1 的节点以及 n-1 条边 。

给定整数 n 和数组 edges ， edges[i] = [ai, bi] 表示树中的节点 ai 和 bi 之间有一条边。

返回长度为 n 的数组 answer ，其中 answer[i] 是树中第 i 个节点与所有其他节点之间的距离之和。



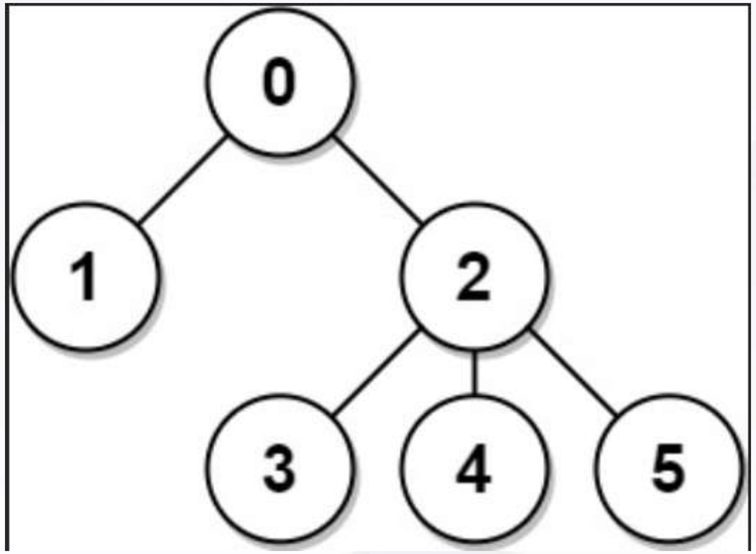
dist(0,1) + dist(0,2) +
dist(0,3) + dist(0,4) + dist(0,5)

换根 DP

给定一个无向、连通的树。树中有 n 个标记为 0...n-1 的节点以及 n-1 条边 。

给定整数 n 和数组 edges ， edges[i] = [ai, bi] 表示树中的节点 ai 和 bi 之间有一条边。

返回长度为 n 的数组 answer ，其中 answer[i] 是树中第 i 个节点与所有其他节点之间的距离之和。



dist(0,1) + dist(0,2) +
dist(0,3) + dist(0,4) + dist(0,5)

换根 DP

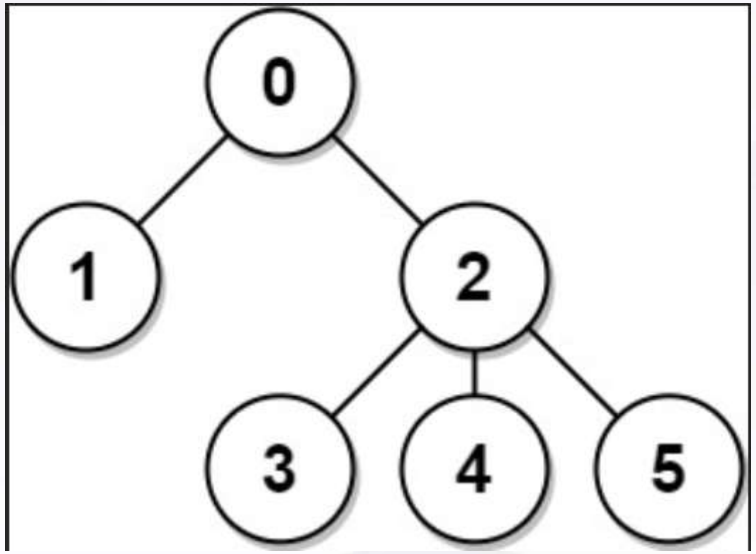
从 0 出发 DFS , 累加 0 到每个点的距离，得到ans[0]。

DFS的同时，计算出每棵子树的大小size[i]。

然后从 0 出发再次 DFS ,设 y 是 x 的儿子，那么:

ans[y] = ans[x] + n - 2 * size[y]。 （状态转移方程）

利用该公式可以自顶向下递推得到每个ans[i]。



dist(0,1) + dist(0,2) +
dist(0,3) + dist(0,4) + dist(0,5)

换根 DP

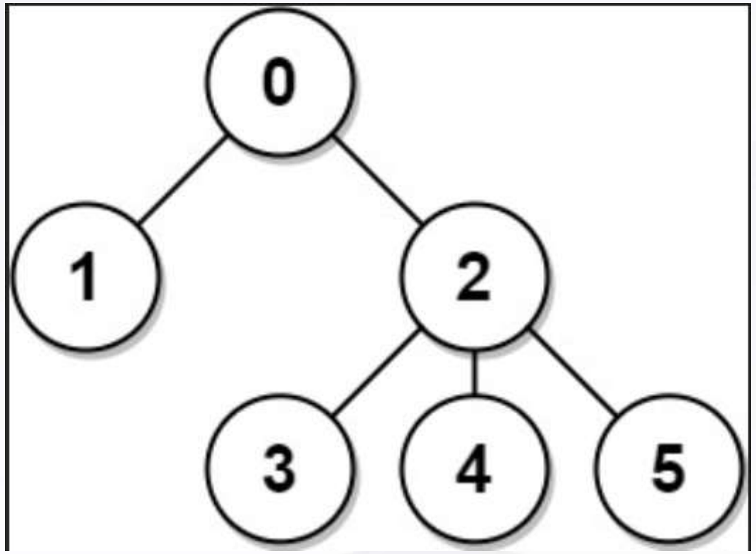
1. 从 0 出发 DFS，累加 0 到每个点的距离，得到ans[0]。

DFS的同时，计算出每棵子树的大小size[i]。

2. 然后从 0 出发再次 DFS，设 y 是 x 的儿子，那么：

ans[y] = ans[x] + n - 2 * size[y]。 （状态转移方程）

3. 利用该公式可以自顶向下递推得到每个ans[i]。

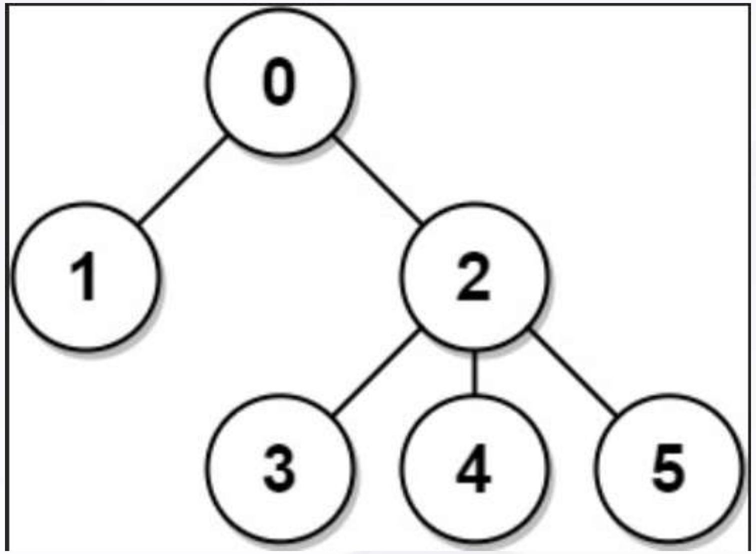


dist(0,1) + dist(0,2) +
dist(0,3) + dist(0,4) + dist(0,5)

树上 DP 问题

```cpp
// ans 和 size 是表示换根 dp 的核心
vector<int>ans; // 不同节点为根得到的答案
vector<int>sizeTree; // 每颗子树的大小
vector<vector<int>>graph;
```

```cpp
// 从 0 节点遍历，同时计算总路径和与 size 数组
// 邻接表存储的树可以用标记 father 代替 visited 数组
void getTreeNum(int root, int father, int depth) {
    // 先计算的是节点 0 为根的路径和
    ans[0] += depth;
    for (auto next :int  : graph[root]) {
        if (next != father) {
            // 后根遍历
            getTreeNum( root: next,  father: root,  depth: depth + 1);
            sizeTree[root] += sizeTree[next];
        }
    }
}
```

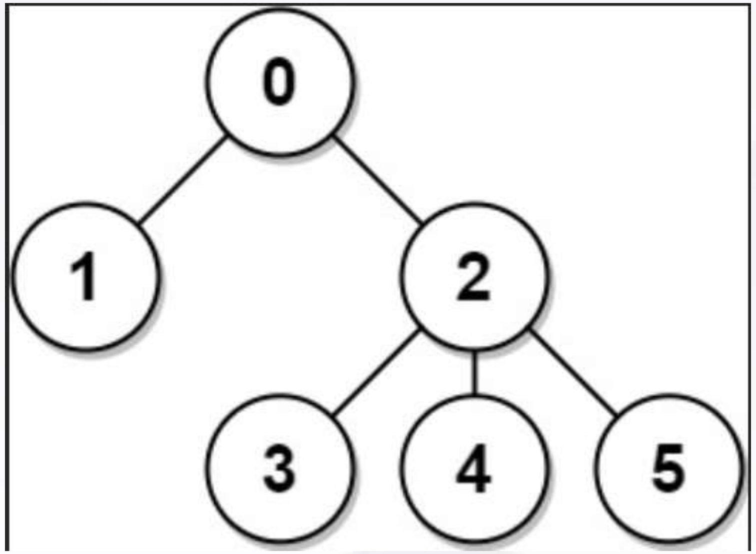


dist(0,1) + dist(0,2) +
dist(0,3) + dist(0,4) + dist(0,5)

树上 DP 问题

```
// 进行换根操作
void exchangeRoot(int root, int father, int n) {
    for (auto next :int  : graph[root]) {
        if (next != father) {
            // 换根 dp 的核心，推导见题解
            ans[next] = ans[root] + n - 2 * sizeTree[next];
            exchangeRoot( root: next,  father: root, n);
        }
    }
}

vector<int> sumOfDistancesInTree(int n, vector<vector<int>>& edges) {
    // 转化为邻接表
    graph.resize( new_size: n);
    for (auto edge :vector<int>  : edges) {
        int from = edge[0], to = edge[1];
        graph[from].push_back(to);
        graph[to].push_back(from);
    }
    ans.resize( new_size: n);
    sizeTree.resize( new_size: n,  x: 1); // 每棵子树的大小初始化为 1
    getTreeNum( root: 0,  father: -1,  depth: 0);
    exchangeRoot( root: 0,  father: -1, n);
    return ans;
}
```



dist(0,1) + dist(0,2) +
dist(0,3) + dist(0,4) + dist(0,5)

To be continued