宫水三叶的刷题日记

DFS

Author : 宫水三叶
Date : 2021/10/07
QQ Group: 703311589
WeChat : oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「DFS」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「DFS」即可获取最新下载链接。

💡**下面介绍使用本合集的最佳使用实践：**

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「DFS」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **17. 电话号码的字母组合** ，难度为 **中等**。

Tag：「DFS」、「回溯算法」

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

答案可以按「任意顺序」返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例 1：

```
输入：digits = "23"

输出：["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

示例 2：

```
输入：digits = ""

输出：[]
```

示例 3：

```
输入：digits = "2"

输出：["a","b","c"]
```

提示：

- 0 <= digits.length <= 4
- digits[i] 是范围 ['2', '9'] 的一个数字。

回溯算法

对于字符串 `ds` 中的每一位数字，都有其对应的字母映射数组。

在 DFS 中决策每一位数字应该对应哪一个字母，当决策的位数 `i == n`，代表整个 `ds` 字符串都被决策完毕，将决策结果添加到结果集：

代码：

```java
class Solution {
    Map<String, String[]> map = new HashMap<>(){{
        put("2", new String[]{"a", "b", "c"});
        put("3", new String[]{"d", "e", "f"});
        put("4", new String[]{"g", "h", "i"});
        put("5", new String[]{"j", "k", "l"});
        put("6", new String[]{"m", "n", "o"});
        put("7", new String[]{"p", "q", "r", "s"});
        put("8", new String[]{"t", "u", "v"});
        put("9", new String[]{"w", "x", "y", "z"});
    }};
    public List<String> letterCombinations(String ds) {
        int n = ds.length();
        List<String> ans = new ArrayList<>();
        if (n == 0) return ans;
        StringBuilder sb = new StringBuilder();
        dfs(ds, 0, n, sb, ans);
        return ans;
    }
    void dfs(String ds, int i, int n, StringBuilder sb, List<String> ans) {
        if (i == n) {
            ans.add(sb.toString());
            return;
        }
        String key = ds.substring(i, i + 1);
        String[] all = map.get(key);
        for (String item : all) {
            sb.append(item);
            dfs(ds, i + 1, n, sb, ans);
            sb.deleteCharAt(sb.length() - 1);
        }
    }
}
```

- 时间复杂度：`n` 代表字符串 `ds` 的长度，一个数字最多对应 4 个字符（7 对应 “pqrs"），即每个数字最多有 4 个字母需要被决策。复杂度为 $O(4^n)$
- 空间复杂度：有多少种方案，就需要多少空间来存放答案。复杂度为 $O(4^n)$

题目描述

这是 LeetCode 上的 **22. 括号生成** ，难度为 **中等**。

Tag：「DFS」、「回溯算法」

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

示例 1：

```
输入：n = 3

输出：["((()))","(()())","(())()","()(())","()()()"]
```

示例 2：

```
输入：n = 1

输出：["()"]
```

提示：

- 1 <= n <= 8

DFS

既然题目是求所有的方案，那只能爆搜了，爆搜可以使用 `DFS` 来做。

从数据范围 `1 <= n <= 8` 来说，`DFS` 应该是稳稳的 AC。

这题的关键是我们要从题目中发掘一些性质：

1. 括号数为 `n`，那么一个合法的括号组合，应该包含 `n` 个左括号和 `n` 个右括号，组合总长度为 `2n`
2. 一对合法的括号，应该是先出现左括号，再出现右括号。那么意味着**任意一个右括号的左边，至少有一个左括号**

其中性质 2 是比较难想到的，我们可以用反证法来证明性质 2 总是成立：

假设某个右括号不满足「其左边至少有一个左括号」，即其左边没有左括号，那么这个右括号就找不到一个与之对应的左括号进行匹配。

这样的组合必然不是有效的括号组合。

使用我们「20. 有效的括号（简单）」的思路（栈）去验证的话，必然验证不通过。

掌握了这两个性质之后，我们可以设定一个初始值为 0 的得分值，令往组合添加一个 `(` 得分值 + 1，往组合添加一个 `)` 得分值 -1。

这样就有：

1. 一个合法的括号组合，最终得分必然为 0（左括号和右括号的数量相等，对应了性质 1）
2. 整个 `DFS` 过程中，得分值范围在 `[0, n]`（得分不可能超过 `n` 意味着不可能添加数量超过 `n` 的左括号，对应了性质 1；得分不可能为负数，意味着每一个右括号必然有一个左括号进行匹配，对应性质 2）

代码：

```java
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> ans = new ArrayList<>();
        dfs(0, n * 2, 0, n, "", ans);
        return ans;
    }

    /**
    * i: 当前遍历到位置
    * n: 字符总长度
    * score: 当前得分，令 '(' 为 1， ')' 为 -1
    * max: 最大得分值
    * path: 当前的拼接结果
    * ans: 最终结果集
    */
    void dfs(int i, int n, int score, int max, String path, List<String> ans) {
        if (i == n) {
            if (score == 0) ans.add(path);
        } else {
            if (score + 1 <= max) dfs(i + 1, n, score + 1, max, path + "(", ans);
            if (score - 1 >= 0) dfs(i + 1, n, score - 1, max, path + ")", ans);
        }
    }
}
```

- 时间复杂度：放置的左括号数量为 `n`，右括号的个数总是小于等于左括号的个数，典型的卡特兰数问题。复杂度为 $O(C_{2n}^{n})$
- 空间复杂度：$O(1)$

题目描述

这是 LeetCode 上的 **37. 解数独** ，难度为 **困难**。

Tag：「回溯算法」、「DFS」、「数独问题」

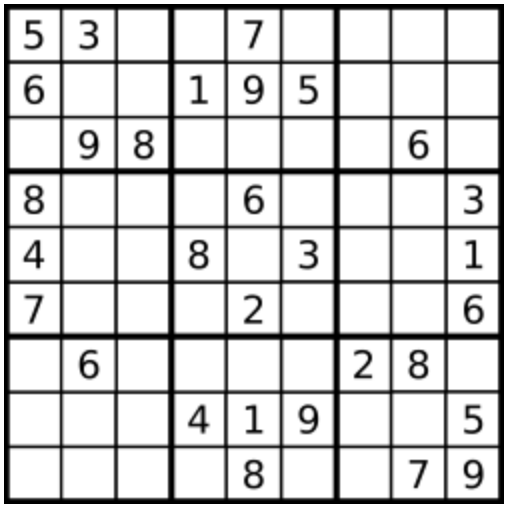
编写一个程序，通过填充空格来解决数独问题。

数独的解法需 遵循如下规则：

1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。（请参考示例图）

数独部分空格内已填入了数字，空白格用 '.' 表示。

示例：



输入：board =
[["5","3",".",".","7",".",".",".",".","."],
["6",".",".","1","9","5",".",".",".","."],
[".","9","8",".",".",".",".","6",".","."],
["8",".",".",".","6",".",".",".","3"],
["4",".",".","8",".","3",".",".","1"],
["7",".",".",".","2",".",".",".","6"],
[".","6",".",".",".",".",".","2","8","."],
[".",".",".","4","1","9",".",".","5"],
[".",".",".",".","8",".",".",".","7","9"]]

输出：[["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],
["1","9","8","3","4","2","5","6","7"],
["8","5","9","7","6","1","4","2","3"],
["4","2","6","8","5","3","7","9","1"],
["7","1","3","9","2","4","8","5","6"],
["9","6","1","5","3","7","2","8","4"],
["2","8","7","4","1","9","6","3","5"],
["3","4","5","2","8","6","1","7","9"]]

解**释**：输入的数独如上**图**所示，唯一有效的解决方案如下所示：

提示：

- board.length == 9
- board[i].length == 9
- board[i][j] 是一位数字或者 '.'
- 题目数据 保证 输入数独仅有一个解

回溯解法

和 N 皇后一样，是一道回溯解法裸题。

上一题「36. 有效的数独（中等）」是让我们判断给定的 `borad` 是否为有效数独。

这题让我们对给定 `board` 求数独，由于 `board` 固定是 `9*9` 的大小，我们可以使用回溯算法去做。

这一类题和 N 皇后一样，属于经典的回溯算法裸题。

这类题都有一个明显的特征，就是数据范围不会很大，如该题限制了范围为 `9*9`，而 N 皇后的 N 一般不会超过 13。

对每一个需要填入数字的位置进行填入，如果发现填入某个数会导致数独解不下去，则进行回溯。

代码：

```Java
class Solution {
    boolean[][] row = new boolean[9][9];
    boolean[][] col = new boolean[9][9];
    boolean[][][] cell = new boolean[3][3][9];
    public void solveSudoku(char[][] board) {
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.') {
                    int t = board[i][j] - '1';
                    row[i][t] = col[j][t] = cell[i / 3][j / 3][t] = true;
                }
            }
        }
        dfs(board, 0, 0);
    }
    boolean dfs(char[][] board, int x, int y) {
        if (y == 9) return dfs(board, x + 1, 0);
        if (x == 9) return true;
        if (board[x][y] != '.') return dfs(board, x, y + 1);
        for (int i = 0; i < 9; i++) {
            if (!row[x][i] && !col[y][i] && !cell[x / 3][y / 3][i]) {
                board[x][y] = (char)(i + '1');
                row[x][i] = col[y][i] = cell[x / 3][y / 3][i] = true;
                if (dfs(board, x, y + 1)) {
                    break;
                } else {
                    board[x][y] = '.';
                    row[x][i] = col[y][i] = cell[x / 3][y / 3][i] = false;
                }
            }
        }
        return board[x][y] != '.';
    }
}
```

- 时间复杂度：在固定 `9*9` 的棋盘里，具有一个枚举方案的最大值（极端情况，假设我们的棋盘刚开始是空的，这时候每一个格子都要枚举，每个格子都有可能从 1 枚举到 9，所以枚举次数为 $9^{9} = 729$），即复杂度不随数据变化而变化。复杂度为 $O(1)$
- 空间复杂度：在固定 `9*9` 的棋盘里，复杂度不随数据变化而变化。复杂度为 $O(1)$

题目描述

这是 LeetCode 上的 **39. 组合总和** ，难度为 **中等**。

Tag：「回溯算法」、「DFS」、「组合总和问题」

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 target）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7,

所求解集为：
[
  [7],
  [2,2,3]
]
```

示例 2：

```
输入：candidates = [2,3,5], target = 8,

所求解集为：
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

提示：

- 1 <= candidates.length <= 30
- 1 <= candidates[i] <= 200
- candidate 中的每个元素都是独一无二的。

- 1 <= target <= 500

DFS + 回溯

这道题很明显就是在考察回溯算法。

还记得三叶之前跟你分享过的 37. 解数独（困难） 吗？

里面有提到我们应该如何快速判断一道题是否应该使用 DFS + 回溯算法来爆搜。

总的来说，你可以从两个方面来考虑：

- **1. 求的是所有的方案，而不是方案数。** 由于求的是所有方案，不可能有什么特别的优化，我们只能进行枚举。这时候可能的解法有动态规划、记忆化搜索、DFS + 回溯算法。
- **2. 通常数据范围不会太大，只有几十。** 如果是动态规划或是记忆化搜索的题的话，由于它们的特点在于低重复/不重复枚举，所以一般数据范围可以出到 10^5 到 10^7，而 DFS + 回溯的话，通常会限制在 30 以内。

这道题数据范围是 30 以内，而且是求所有方案，因此我们使用 DFS + 回溯来求解。

代码：

```Java
class Solution {
    public List<List<Integer>> combinationSum(int[] cs, int t) {
        List<List<Integer>> ans = new ArrayList<>();
        List<Integer> cur = new ArrayList<>();
        dfs(cs, t, 0, ans, cur);
        return ans;
    }

    /**
     * cs: 原数组，从该数组进行选数
     * t: 还剩多少值需要凑成。起始值为 target ，代表还没选择任何数；当 t = 0，代表选择的数凑成了 targe
     * u: 当前决策到 cs[] 中的第几位
     * ans: 最终结果集
     * cur: 当前结果集
     */
    void dfs(int[] cs, int t, int u, List<List<Integer>> ans, List<Integer> cur) {
        if (t == 0) {
            ans.add(new ArrayList<>(cur));
            return;
        }
        if (u == cs.length || t < 0) return;

        // 枚举 cs[u] 的使用次数
        for (int i = 0; cs[u] * i <= t; i++) {
            dfs(cs, t - cs[u] * i, u + 1, ans, cur);
            cur.add(cs[u]);
        }
        // 进行回溯。注意回溯总是将数组的最后一位弹出
        for (int i = 0; cs[u] * i <= t; i++) {
            cur.remove(cur.size() - 1);
        }
    }
}
```

- 时间复杂度：由于每个数字的使用次数不确定，因此无法分析具体的复杂度。但是 DFS 回溯算法通常是指数级别的复杂度（因此数据范围通常为 30 以内）。这里暂定 $O(n * 2^n)$
- 空间复杂度：同上。复杂度为 $O(n * 2^n)$

题目描述

这是 LeetCode 上的 **40. 组合总和 II** ，难度为 **中等**。

Tag：「回溯算法」、「DFS」、「组合总和问题」

给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 解集不能包含重复的组合。

示例 1:

```
输入: candidates = [10,1,2,7,6,1,5], target = 8,

所求解集为:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

示例 2:

```
输入: candidates = [2,5,2,1,2], target = 5,

所求解集为:
[
  [1,2,2],
  [5]
]
```

DFS + 回溯

这道题和「39. 组合总和（中等）」几乎一样。

唯一的不同是这题每个数只能使用一次，而「39. 组合总和（中等）」中可以使用无限次。

我们再来回顾一下应该如何快速判断一道题是否应该使用 DFS + 回溯算法来爆搜。

这个判断方法，最早三叶在 37. 解数独（困难） 讲过。

总的来说，你可以从两个方面来考虑：

- **1. 求的是所有的方案，而不是方案数。** 由于求的是所有方案，不可能有什么特别的优化，我们只能进行枚举。这时候可能的解法有动态规划、记忆化搜索、DFS + 回溯算法。
- **2. 通常数据范围不会太大，只有几十。** 如果是动态规划或是记忆化搜索的题的话，由于它们的特点在于低重复/不重复枚举，所以一般数据范围可以出到 10^5 到 10^7，而 DFS + 回溯的话，通常会限制在 30 以内。

这道题数据范围是 30 以内，而且是求所有方案。因此我们使用 DFS + 回溯来求解。

我们可以接着 39. 组合总和（中等） 的思路来修改：

1. 由于每个数字只能使用一次，我们可以直接在 DFS 中决策某个数是用还是不用。
2. 由于不允许重复答案，可以使用 set 来保存所有合法方案，最终再转为 list 进行返回。当然我们需要先对 cs 进行排序，确保得到的合法方案中数值都是从小到大的。这样 set 才能起到去重的作用。对于 `[1,2,1]` 和 `[1,1,2]`，set 不会认为是相同的数组。

代码：

```java
class Solution {
    public List<List<Integer>> combinationSum2(int[] cs, int t) {
        Arrays.sort(cs);
        Set<List<Integer>> ans = new HashSet<>();
        List<Integer> cur = new ArrayList<>();
        dfs(cs, t, 0, ans, cur);
        return new ArrayList<>(ans);
    }

    /**
     * cs: 原数组，从该数组进行选数
     * t: 还剩多少值需要凑成。起始值为 target ，代表还没选择任何数；当 t = 0，代表选择的数凑成了 targe
     * u: 当前决策到 cs[] 中的第几位
     * ans: 最终结果集
     * cur: 当前结果集
     */
    void dfs(int[] cs, int t, int u, Set<List<Integer>> ans, List<Integer> cur) {
        if (t == 0) {
            ans.add(new ArrayList<>(cur));
            return;
        }
        if (u == cs.length || t < 0) return;

        // 使用 cs[u]
        cur.add(cs[u]);
        dfs(cs, t - cs[u], u + 1, ans, cur);

        // 进行回溯
        cur.remove(cur.size() - 1);
        // 不使用 cs[u]
        dfs(cs, t, u + 1, ans, cur);
    }
}
```

- 时间复杂度：DFS 回溯算法通常是指数级别的复杂度（因此数据范围通常为 30 以内）。这里暂定 $O(n * 2^n)$
- 空间复杂度：同上。复杂度为 $O(n * 2^n)$

题目描述

这是 LeetCode 上的 **301. 删除无效的括号** ，难度为 **困难**。

Tag : 「括号问题」、「回溯算法」、「DFS」

给你一个由若干括号和字母组成的字符串 s ，删除最小数量的无效括号，使得输入的字符串有效。

返回所有可能的结果。答案可以按 任意顺序 返回。

示例 1:

```
输入: "()())()"
输出: ["()()()", "(())()"]
```

示例 2:

```
输入: "(a)())()"
输出: ["(a)()()", "(a())()"]
```

示例 3:

```
输入: ")("
输出: [""]
```

提示：

- 1 <= s.length <= 25
- s 由小写英文字母以及括号 '(' 和 ')' 组成
- s 中至多含 20 个括号

DFS 回溯算法

由于题目要求我们将所有（最长）合法方案输出，因此不可能有别的优化，只能进行「爆搜」。

我们可以使用 DFS 实现回溯搜索。

基本思路：

我们知道所有的合法方案，必然有左括号的数量与右括号数量相等。

首先我们令左括号的得分为 1；右括号的得分为 -1。那么对于合法的方案而言，必定满足最终得分为 0。

同时我们可以预处理出「爆搜」过程的最大得分：`max = min(左括号的数量, 右括号的数量)`

PS.「爆搜」过程的最大得分必然是：合法左括号先全部出现在左边，之后使用最多的合法右括号进行匹配。

枚举过程中出现字符分三种情况：

- 普通字符：无须删除，直接添加
- 左括号：如果当前得分不超过 `max - 1` 时，我们可以选择添加该左括号，也能选择不添加
- 右括号：如果当前得分大于 0（说明有一个左括号可以与之匹配），我们可以选择添加该右括号，也能选择不添加

使用 Set 进行方案去重，`len` 记录「爆搜」过程中的最大子串，然后将所有结果集中长度为 `len` 的子串加入答案：

```java
class Solution {
    int len;
    public List<String> removeInvalidParentheses(String s) {
        char[] cs = s.toCharArray();
        int l = 0, r = 0;
        for (char c : cs) {
            if (c == '(') {
                l++;
            } else if (c == ')') {
                r++;
            }
        }
        int max = Math.min(l, r);
        Set<String> all = new HashSet<>();
        dfs(cs, 0, 0, max, "", all);
        List<String> ans = new ArrayList<>();
        for (String str : all) {
            if (str.length() == len) ans.add(str);
        }
        return ans;
    }
    /**
     * cs: 字符串 s 对应的字符数组
     * u: 当前决策到 cs 的哪一位
     * score: 当前决策方案的得分值（每往 cur 追加一个左括号进行 +1；每往 cur 追加一个右括号进行 -1）
     * max: 整个 dfs 过程的最大得分
     * cur: 当前决策方案
     * ans: 合法方案结果集
     */
    void dfs(char[] cs, int u, int score, int max, String cur, Set<String> ans) {
        if (u == cs.length) {
            if (score == 0 && cur.length() >= len) {
                len = Math.max(len, cur.length());
                ans.add(cur);
            }
            return;
        }
        if (cs[u] == '(') {
            if (score + 1 <= max) dfs(cs, u + 1, score + 1, max, cur + "(", ans);
            dfs(cs, u + 1, score, max, cur, ans);
        } else if (cs[u] == ')') {
            if (score > 0) dfs(cs, u + 1, score - 1, max, cur + ")", ans);
            dfs(cs, u + 1, score, max, cur, ans);
        } else {
            dfs(cs, u + 1, score, max, cur + String.valueOf(cs[u]), ans);
        }
```

```
    }
}
```

- 时间复杂度：不考虑 score 带来的剪枝效果，最坏情况下，每个位置都有两种选择。复杂度为 $O(n * 2^n)$
- 空间复杂度：最大合法方案数与字符串长度最多呈线性关系。复杂度为 $O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **341. 扁平化嵌套列表迭代器** ，难度为 **中等**。

Tag：「DFS」、「队列」、「栈」

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

示例 1:

```
输入: [[1,1],2,[1,1]]
输出: [1,1,2,1,1]
解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,1,2,1,1]。
```

示例 2:

```
输入: [1,[4,[6]]]
输出: [1,4,6]
解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,4,6]。
```

DFS + 队列

由于所有的元素都是在初始化时提供的，因此一个朴素的做法是在初始化的时候进行处理。

由于存在嵌套，比较简单的做法是通过 DFS 进行处理，将元素都放至队列。

代码：

```
public class NestedIterator implements Iterator<Integer> {

    Deque<Integer> queue = new ArrayDeque<>();

    public NestedIterator(List<NestedInteger> nestedList) {
        dfs(nestedList);
    }

    @Override
    public Integer next() {
        return hasNext() ? queue.pollFirst() : -1;
    }

    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    void dfs(List<NestedInteger> list) {
        for (NestedInteger item : list) {
            if (item.isInteger()) {
                queue.addLast(item.getInteger());
            } else {
                dfs(item.getList());
            }
        }
    }
}
```

- 时间复杂度：构建迭代器的复杂度为 $O(n)$，调用 $next()$ 与 $hasNext()$ 的复杂度为 $O(1)$
- 空间复杂度：$O(n)$

递归 + 栈

另外一个做法是，我们不对所有的元素进行预处理。

而是先将所有的 `NestedInteger` 逆序放到栈中，当需要展开的时候才进行展开。

代码：

```
public class NestedIterator implements Iterator<Integer> {

    Deque<NestedInteger> stack = new ArrayDeque<>();

    public NestedIterator(List<NestedInteger> list) {
        for (int i = list.size() - 1; i >= 0; i--) {
            NestedInteger item = list.get(i);
            stack.addLast(item);
        }
    }

    @Override
    public Integer next() {
        return hasNext() ? stack.pollLast().getInteger() : -1;
    }

    @Override
    public boolean hasNext() {
        if (stack.isEmpty()) {
            return false;
        } else {
            NestedInteger item = stack.peekLast();
            if (item.isInteger()) {
                return true;
            } else {
                item = stack.pollLast();
                List<NestedInteger> list = item.getList();
                for (int i = list.size() - 1; i >= 0; i--) {
                    stack.addLast(list.get(i));
                }
                return hasNext();
            }
        }
    }
}
```

- 时间复杂度：构建迭代器的复杂度为 $O(n)$，$hasNext()$ 的复杂度为均摊 $O(1)$，$next()$ 严格按照迭代器的访问顺序（ 先 $hasNext()$ 再 $next()$ ）的话为 $O(1)$，防御性编程生效的情况下为均摊 $O(1)$
- 空间复杂度：$O(n)$

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **403. 青蛙过河** ，难度为 **困难**。

Tag：「DFS」、「BFS」、「记忆化搜索」、「线性 DP」

一只青蛙想要过河。 假定河流被等分为若干个单元格，并且在每一个单元格内都有可能放有一块石子（也有可能没有）。 青蛙可以跳上石子，但是不可以跳入水中。

给你石子的位置列表 stones（用单元格序号 升序 表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一块石子上）。

开始时， 青蛙默认已站在第一块石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格 1 跳至单元格 2 ）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1 个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

示例 1：

```
输入：stones = [0,1,3,5,6,8,12,17]

输出：true

解释：青蛙可以成功过河，按照如下方案跳跃：跳 1 个单位到第 2 块石子, 然后跳 2 个单位到第 3 块石子, 接着 跳 2 个单位到第 4 块
```

示例 2：

```
输入：stones = [0,1,2,3,4,8,9,11]

输出：false

解释：这是因为第 5 和第 6 个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。
```

提示：

- 2 <= stones.length <= 2000
- 0 <= stones[i] <= 2^{31} - 1

- stones[0] == 0

DFS（TLE）

根据题意，我们可以使用 `DFS` 来模拟/爆搜一遍，检查所有的可能性中是否有能到达最后一块石子的。

通常设计 `DFS` 函数时，我们只需要不失一般性的考虑完成第 i 块石子的跳跃需要些什么信息即可：

- 需要知道当前所在位置在哪，也就是需要知道当前石子所在列表中的下标 u。
- 需要知道当前所在位置是经过多少步而来的，也就是需要知道上一步的跳跃步长 k。

代码：

```Java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        // 将石子信息存入哈希表
        // 为了快速判断是否存在某块石子，以及快速查找某块石子所在下标
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        // 根据题意，第一步是固定经过步长 1 到达第一块石子（下标为 1）
        if (!map.containsKey(1)) return false;
        return dfs(ss, ss.length, 1, 1);
    }

    /**
     * 判定是否能够跳到最后一块石子
     * @param ss 石子列表【不变】
     * @param n  石子列表长度【不变】
     * @param u  当前所在的石子的下标
     * @param k  上一次是经过多少步跳到当前位置的
     * @return 是否能跳到最后一块石子
     */
    boolean dfs(int[] ss, int n, int u, int k) {
        if (u == n - 1) return true;
        for (int i = -1; i <= 1; i++) {
            // 如果是原地踏步的话，直接跳过
            if (k + i == 0) continue;
            // 下一步的石子理论编号
            int next = ss[u] + k + i;
            // 如果存在下一步的石子，则跳转到下一步石子，并 DFS 下去
            if (map.containsKey(next)) {
                boolean cur = dfs(ss, n, map.get(next), k + i);
                if (cur) return true;
            }
        }
        return false;
    }
}
```

- 时间复杂度：$O(3^n)$
- 空间复杂度：$O(3^n)$

但数据范围为 10^3，直接使用 DFS 肯定会超时。

我们需要考虑加入「记忆化」功能，或者改为使用带标记的 `BFS`。

记忆化搜索

在考虑加入「记忆化」时，我们只需要将 `DFS` 方法签名中的【可变】参数作为维度，`DFS` 方法中的返回值作为存储值即可。

通常我们会使用「数组」来作为我们缓存中间结果的容器，

对应到本题，就是需要一个 `boolean[石子列表下标][跳跃步数]` 这样的数组，但使用布尔数组作为记忆化容器往往无法区分「状态尚未计算」和「状态已经计算，并且结果为 `false`」两种情况。

因此我们需要转为使用 `int[石子列表下标][跳跃步数]`，默认值 0 代表状态尚未计算，-1 代表计算状态为 `false`，1 代表计算状态为 `true`。

接下来需要估算数组的容量，可以从「数据范围」入手分析。

根据 `2 <= stones.length <= 2000`，我们可以确定第一维（数组下标）的长度为 2009，而另外一维（跳跃步数）是与跳转过程相关的，无法直接确定一个精确边界，但是一个显而易见的事实是，跳到最后一块石子之后的位置是没有意义的，因此我们不会有「跳跃步长」大于「石子列表长度」的情况，因此也可以定为 2009（这里是利用了由下标为 i 的位置发起的跳跃不会超过 $i + 1$ 的性质）。

至此，我们定下来了记忆化容器为 `int[][] cache = new int[2009][2009]`。

但是可以看出，上述确定容器大小的过程还是需要一点点分析 & 经验的。

那么是否有思维难度再低点的方法呢？

答案是有的，直接使用「哈希表」作为记忆化容器。「哈希表」本身属于非定长容器集合，我们不需要分析两个维度的上限到底是多少。

另外，当容器维度较多且上界较大时（例如上述的 `int[2009][2009]`），直接使用「哈希表」可以有效降低「爆空间/时间」的风险（不需要每跑一个样例都创建一个百万级的数组）。

代码：

```java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    // int[][] cache = new int[2009][2009];
    Map<String, Boolean> cache = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        if (!map.containsKey(1)) return false;
        return dfs(ss, ss.length, 1, 1);
    }
    boolean dfs(int[] ss, int n, int u, int k) {
        String key = u + "_" + k;
        // if (cache[u][k] != 0) return cache[u][k] == 1;
        if (cache.containsKey(key)) return cache.get(key);
        if (u == n - 1) return true;
        for (int i = -1; i <= 1; i++) {
            if (k + i == 0) continue;
            int next = ss[u] + k + i;
            if (map.containsKey(next)) {
                boolean cur = dfs(ss, n, map.get(next), k + i);
                // cache[u][k] = cur ? 1 : -1;
                cache.put(key, cur);
                if (cur) return true;
            }
        }
        // cache[u][k] = -1;
        cache.put(key, false);
        return false;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

动态规划

有了「记忆化搜索」的基础，要写写出来动态规划就变得相对简单了。

我们可以从 `DFS` 函数出发，写出「动态规划」解法。

我们的 `DFS` 函数签名为：

```
boolean dfs(int[] ss, int n, int u, int k);
```

其中前两个参数为不变参数，后两个为可变参数，返回值是我们的答案。

因此可以设定为 $f[][]$ 作为动规数组：

1. 第一维为可变参数 u，代表石子列表的下标，范围为数组 `stones` 长度；
2. 第二维为可变参数 k，代表上一步的的跳跃步长，前面也分析过了，最多不超过数组 `stones` 长度。

这样的「状态定义」所代表的含义：当前在第 i 个位置，并且是以步长 k 跳到位置 i 时，是否到达最后一块石子。

那么对于 $f[i][k]$ 是否为真，则取决于上一位置 j 的状态值，结合每次步长的变化为 `[-1,0,1]` 可知：

- 可从 $f[j][k-1]$ 状态而来：先是经过 $k-1$ 的跳跃到达位置 j，再在原步长的基础上 `+1`，跳到了位置 i。
- 可从 $f[j][k]$ 状态而来：先是经过 k 的跳跃到达位置 j，维持原步长不变，跳到了位置 i。
- 可从 $f[j][k+1]$ 状态而来：先是经过 $k+1$ 的跳跃到达位置 j，再在原步长的基础上 `-1`，跳到了位置 i。

只要上述三种情况其中一种为真，则 $f[i][j]$ 为真。

至此，我们解决了动态规划的「状态定义」&「状态转移方程」部分。

但这就结束了吗？还没有。

我们还缺少可让状态递推下去的「有效值」，或者说缺少初始化环节。

因为我们的 $f[i][k]$ 依赖于之前的状态进行“或运算”而来，转移方程本身不会产生 $true$ 值。因此为了让整个「递推」过程可滚动，我们需要先有一个为 $true$ 的状态值。

这时候再回看我们的状态定义：**当前在第 i 个位置，并且是以步长 k 跳到位置 i 时，是否到达最后一块石子。**

显然，我们事先是不可能知道经过「多大的步长」跳到「哪些位置」，最终可以到达最后一块石

子。

这时候需要利用「对偶性」将跳跃过程「翻转」过来分析：

我们知道起始状态是「经过步长为 1」的跳跃到达「位置 1」，如果从起始状态出发，存在一种方案到达最后一块石子的话，那么必然存在一条反向路径，它是以从「最后一块石子」开始，并以「某个步长 k」开始跳跃，最终以回到位置 1。

因此我们可以设 $f[1][1] = true$，作为我们的起始值。

这里本质是利用「路径可逆」的性质，将问题进行了「等效对偶」。表面上我们是进行「正向递推」，但事实上我们是在验证是否存在某条「反向路径」到达位置 1。

建议大家加强理解 ~

代码：

```
class Solution {
    public boolean canCross(int[] ss) {
        int n = ss.length;
        // check first step
        if (ss[1] != 1) return false;
        boolean[][] f = new boolean[n + 1][n + 1];
        f[1][1] = true;
        for (int i = 2; i < n; i++) {
            for (int j = 1; j < i; j++) {
                int k = ss[i] - ss[j];
                // 我们知道从位置 j 到位置 i 是需要步长为 k 的跳跃

                // 而从位置 j 发起的跳跃最多不超过 j + 1
                // 因为每次跳跃，下标至少增加 1，而步长最多增加 1
                if (k <= j + 1) {
                    f[i][k] = f[j][k - 1] || f[j][k] || f[j][k + 1];
                }
            }
        }
        for (int i = 1; i < n; i++) {
            if (f[n - 1][i]) return true;
        }
        return false;
    }
}
```

- 时间复杂度：$O(n^2)$

- 空间复杂度：$O(n^2)$

BFS

事实上，前面我们也说到，解决超时 `DFS` 问题，除了增加「记忆化」功能以外，还能使用带标记的 `BFS`。

因为两者都能解决 `DFS` 的超时原因：大量的重复计算。

但为了「记忆化搜索」&「动态规划」能够更好的衔接，所以我把 `BFS` 放到最后。

如果你能够看到这里，那么这里的 `BFS` 应该看起来会相对轻松。

它更多是作为「记忆化搜索」的另外一种实现形式。

代码：

```Java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        if (!map.containsKey(1)) return false;

        boolean[][] vis = new boolean[n][n];
        Deque<int[]> d = new ArrayDeque<>();
        vis[1][1] = true;
        d.addLast(new int[]{1, 1});

        while (!d.isEmpty()) {
            int[] poll = d.pollFirst();
            int idx = poll[0], k = poll[1];
            if (idx == n - 1) return true;
            for (int i = -1; i <= 1; i++) {
                if (k + i == 0) continue;
                int next = ss[idx] + k + i;
                if (map.containsKey(next)) {
                    int nIdx = map.get(next), nK = k + i;
                    if (nIdx == n - 1) return true;
                    if (!vis[nIdx][nK]) {
                        vis[nIdx][nK] = true;
                        d.addLast(new int[]{nIdx, nK});
                    }
                }
            }
        }

        return false;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

题目描述

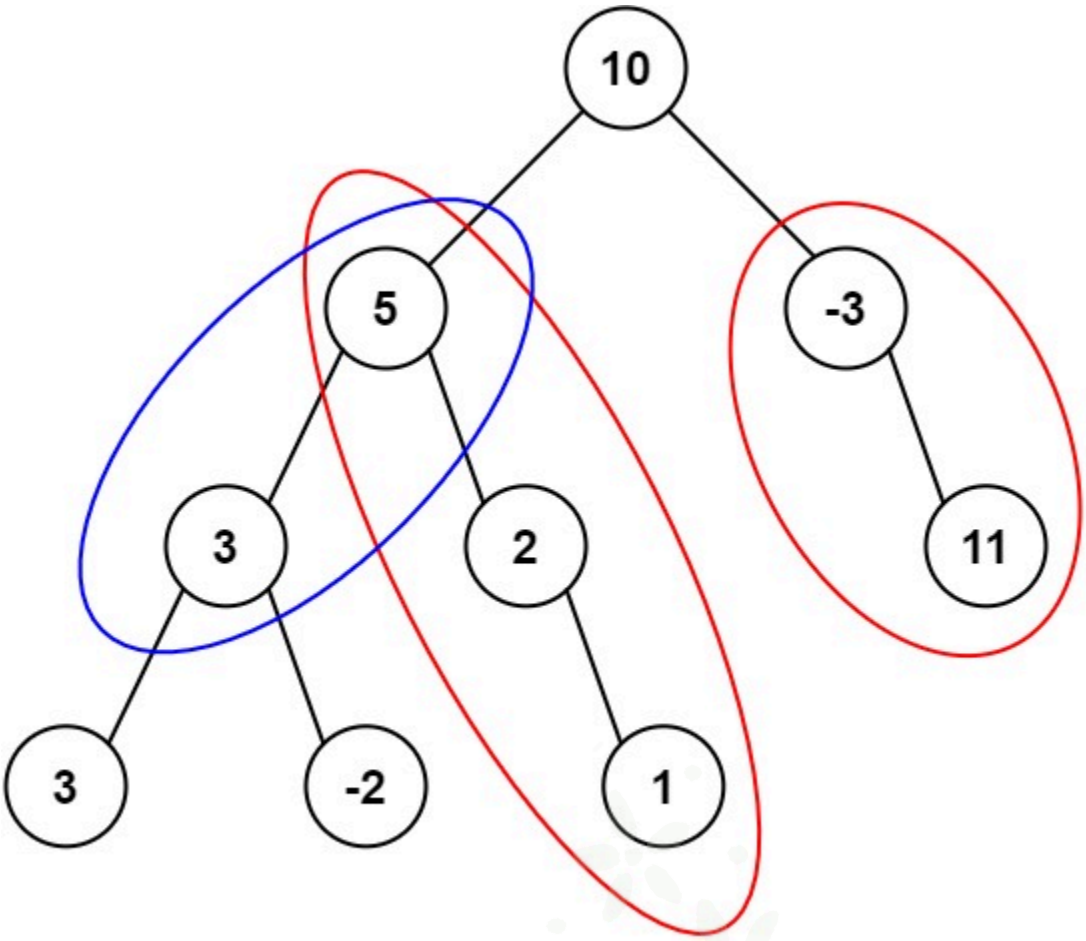
这是 LeetCode 上的 **437. 路径总和 III** ，难度为 **中等**。

Tag：「DFS」、「树的遍历」、「前缀和」

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（只能从父节点到子节点）。

示例 1：



```
输入：root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8

输出：3

解释：和等于 8 的路径有 3 条，如图所示。
```

示例 2：

```
输入：root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

输出：3
```

提示:

- 二叉树的节点个数的范围是 [0,1000]
- -10^9 <= Node.val <= 10^9
- -1000 <= targetSum <= 1000

树的遍历 + DFS

一个朴素的做法是搜索以每个节点为根的（往下的）所有路径，并对路径总和为 $targetSum$ 的路径进行累加统计。

使用 `dfs1` 来搜索所有节点，复杂度为 $O(n)$；在 `dfs1` 中对于每个当前节点，使用 `dfs2` 搜索以其为根的所有（往下的）路径，同时累加路径总和为 $targetSum$ 的所有路径，复杂度为 $O(n)$。

整体复杂度为 $O(n^2)$，数据范围为 10^3，可以过。

代码：

```Java
class Solution {
    int ans, t;
    public int pathSum(TreeNode root, int _t) {
        t = _t;
        dfs1(root);
        return ans;
    }
    void dfs1(TreeNode root) {
        if (root == null) return;
        dfs2(root, root.val);
        dfs1(root.left);
        dfs1(root.right);
    }
    void dfs2(TreeNode root, int val) {
        if (val == t) ans++;
        if (root.left != null) dfs2(root.left, val + root.left.val);
        if (root.right != null) dfs2(root.right, val + root.right.val);
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：忽略递归带来的额外空间开销，复杂度为 $O(1)$

树的遍历 + 前缀和

在「解法一」中，我们统计的是以每个节点为根的（往下的）所有路径，也就是说统计的是以每个节点为「路径开头」的所有合法路径。

本题的一个优化切入点为「路径只能往下」，因此如果我们转换一下，统计以每个节点为「路径结尾」的合法数量的话，配合原本就是「从上往下」进行的数的遍历（最完整的路径必然是从原始根节点到当前节点的唯一路径），相当于只需要在完整路径中找到有多少个节点到当前节点的路径总和为 $targetSum$。

于是这个树上问题彻底转换一维问题：求解从原始起点（根节点）到当前节点 b 的路径中，有多少节点 a 满足 $sum[a...b] = targetSum$，由于从原始起点（根节点）到当前节点的路径唯一，因此这其实是一个「一维前缀和」问题。

具体的，我们可以在进行树的遍历时，记录下从原始根节点 $root$ 到当前节点 cur 路径中，从 $root$ 到任意中间节点 x 的路径总和，配合哈希表，快速找到满足以 cur 为「路径结尾」的、

使得路径总和为 $targetSum$ 的目标「路径起点」有多少个。

> 一些细节：由于我们只能统计往下的路径，但是树的遍历会同时搜索两个方向的子树。因此我们应当在搜索完以某个节点为根的左右子树之后，应当回溯地将路径总和从哈希表中删除，防止统计到跨越两个方向的路径。

代码：

```
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    int ans, t;
    public int pathSum(TreeNode root, int _t) {
        if (root == null) return 0;
        t = _t;
        map.put(0, 1);
        dfs(root, root.val);
        return ans;
    }
    void dfs(TreeNode root, int val) {
        if (map.containsKey(val - t)) ans += map.get(val - t);
        map.put(val, map.getOrDefault(val, 0) + 1);
        if (root.left != null) dfs(root.left, val + root.left.val);
        if (root.right != null) dfs(root.right, val + root.right.val);
        map.put(val, map.getOrDefault(val, 0) - 1);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **494. 目标和** ，难度为 **中等**。

Tag：「DFS」、「记忆化搜索」、「背包 DP」、「01 背包」

给你一个整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 ‘+’ 或 ‘-’ ，然后串联起所有整数，可以构造一个 表达式 ：

- 例如，nums = [2, 1] ，可以在 2 之前添加 ‘+’ ，在 1 之前添加 ‘-’ ，然后串联起来得到表达式 “+2-1” 。
  返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 3 。
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

示例 2：

```
输入：nums = [1], target = 1
输出：1
```

提示：

- 1 <= nums.length <= 20
- 0 <= nums[i] <= 1000
- 0 <= sum(nums[i]) <= 100
- -1000 <= target <= 100

DFS

数据范围只有 20，而且每个数据只有 $+/-$ 两种选择，因此可以直接使用 DFS 进行「爆搜」。

而 DFS 有「使用全局变量维护」和「接收返回值处理」两种形式。

代码：

```Java
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        return dfs(nums, t, 0, 0);
    }
    int dfs(int[] nums, int t, int u, int cur) {
        if (u == nums.length) {
            return cur == t ? 1 : 0;
        }
        int left = dfs(nums, t, u + 1, cur + nums[u]);
        int right = dfs(nums, t, u + 1, cur - nums[u]);
        return left + right;
    }
}
```

```Java
class Solution {
    int ans = 0;
    public int findTargetSumWays(int[] nums, int t) {
        dfs(nums, t, 0, 0);
        return ans;
    }
    void dfs(int[] nums, int t, int u, int cur) {
        if (u == nums.length) {
            ans += cur == t ? 1 : 0;
            return;
        }
        dfs(nums, t, u + 1, cur + nums[u]);
        dfs(nums, t, u + 1, cur - nums[u]);
    }
}
```

- 时间复杂度：$O(2^n)$
- 空间复杂度：忽略递归带来的额外空间消耗。复杂度为 $O(1)$

记忆化搜索

不难发现，在 DFS 的函数签名中只有「数值下标 `u`」和「当前结算结果 `cur`」为可变参数，考虑将其作为记忆化容器的两个维度，返回值作为记忆化容器的记录值。

由于 `cur` 存在负权值，为了方便，我们这里不设计成静态数组，而是使用「哈希表」进行记录。

以上分析都在 （题解）403. 青蛙过河 完整讲过。

代码：

```
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        return dfs(nums, t, 0, 0);
    }
    Map<String, Integer> cache = new HashMap<>();
    int dfs(int[] nums, int t, int u, int cur) {
        String key = u + "_" + cur;
        if (cache.containsKey(key)) return cache.get(key);
        if (u == nums.length) {
            cache.put(key, cur == t ? 1 : 0);
            return cache.get(key);
        }
        int left = dfs(nums, t, u + 1, cur + nums[u]);
        int right = dfs(nums, t, u + 1, cur - nums[u]);
        cache.put(key, left + right);
        return cache.get(key);
    }
}
```

- 时间复杂度：$O(n * \sum_{i=0}^{n-1} abs(nums[i]))$
- 空间复杂度：忽略递归带来的额外空间消耗。复杂度为 $O(n * \sum_{i=0}^{n-1} abs(nums[i]))$

动态规划

能够以「递归」的形式实现动态规划（记忆化搜索），自然也能使用「递推」的方式进行实现。

根据记忆化搜索的分析，我们可以定义：

$f[i][j]$ 代表考虑前 i 个数，当前计算结果为 j 的方案数，令 `nums` 下标从 1 开始。

那么 $f[n][target]$ 为最终答案，$f[0][0] = 1$ 为初始条件：代表不考虑任何数，凑出计算结果为 0 的方案数为 1 种。

根据每个数值只能搭配 $+/-$ 使用，可得状态转移方程：

$$f[i][j] = f[i-1][j-nums[i-1]] + f[i-1][j+nums[i-1]]$$

到这里，既有了「状态定义」和「转移方程」，又有了可以滚动下去的「有效值」（起始条件）。

距离我们完成所有分析还差最后一步。

当使用递推形式时，我们通常会使用「静态数组」来存储动规值，因此还需要考虑维度范围的：

- 第一维为物品数量：范围为 `nums` 数组长度
- 第二维为中间结果：令 `s` 为所有 `nums` 元素的总和（题目给定了 `nums[i]` 为非负数的条件，否则需要对 `nums[i]` 取绝对值再累加），那么中间结果的范围为 $[-s, s]$

因此，我们可以确定动规数组的大小。**同时在转移时，对第二维度的使用做一个 `s` 的右偏移，以确保「负权值」也能够被合理计算/存储。**

代码：

```
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        int n = nums.length;
        int s = 0;
        for (int i : nums) s += Math.abs(i);
        if (t > s) return 0;
        int[][] f = new int[n + 1][2 * s + 1];
        f[0][0 + s] = 1;
        for (int i = 1; i <= n; i++) {
            int x = nums[i - 1];
            for (int j = -s; j <= s; j++) {
                if ((j - x) + s >= 0) f[i][j + s] += f[i - 1][(j - x) + s];
                if ((j + x) + s <= 2 * s) f[i][j + s] += f[i - 1][(j + x) + s];
            }
        }
        return f[n][t + s];
    }
}
```

- 时间复杂度：$O(n * \sum_{i=0}^{n-1} abs(nums[i]))$
- 空间复杂度：$O(n * \sum_{i=0}^{n-1} abs(nums[i]))$

动态规划（优化）

在上述「动态规划」分析中，我们总是尝试将所有的状态值都计算出来，当中包含很多对「目标状态」不可达的“额外”状态值。

即达成某些状态后，不可能再回到我们的「目标状态」。

例如当我们的 $target$ 不为 $-s$ 和 s 时，$-s$ 和 s 就是两个对「目标状态」不可达的“额外”状态值，到达 $-s$ 或 s 已经使用所有数值，对 $target$ 不可达。

那么我们如何规避掉这些“额外”状态值呢？

我们可以从哪些数值使用哪种符号来分析，即划分为「负值部分」&「非负值部分」，令「负值部分」的绝对值总和为 m，即可得：

$$(s-m)-m=s-2*m=target$$

变形得：

$$m=\frac{s-target}{2}$$

问题转换为：**只使用 $+$ 运算符，从 `nums` 凑出 m 的方案数。**

这样「原问题的具体方案」和「转换问题的具体方案」具有一一对应关系：「转换问题」中凑出来的数值部分在实际计算中应用 $-$，剩余部分应用 $+$，从而实现凑出来原问题的 $target$ 值。

另外，由于 `nums` 均为非负整数，因此我们需要确保 $s-target$ 能够被 2 整除。

同时，由于问题转换为 **从 `nums` 中凑出 m 的方案数，因此「状态定义」和「状态转移」都需要进行调整（01 背包求方案数）：**

定义 $f[i][j]$ 为从 `nums` 凑出总和「恰好」为 j 的方案数。

最终答案为 $f[n][m]$，$f[0][0]=1$ 为起始条件：代表不考虑任何数，凑出计算结果为 0 的方案数为 1 种。

每个数值有「选」和「不选」两种决策，转移方程为：

$$f[i][j]=f[i-1][j]+f[i-1][j-nums[i-1]]$$

代码：

```Java
class Solution {
    public int findTargetSumWays(int[] nums, int t) {
        int n = nums.length;
        int s = 0;
        for (int i : nums) s += Math.abs(i);
        if (t > s || (s - t) % 2 != 0) return 0;
        int m = (s - t) / 2;
        int[][] f = new int[n + 1][m + 1];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            int x = nums[i - 1];
            for (int j = 0; j <= m; j++) {
                f[i][j] += f[i - 1][j];
                if (j >= x) f[i][j] += f[i - 1][j - x];
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：$O(n * (\sum_{i=0}^{n-1} abs(nums[i]) - target))$
- 空间复杂度：$O(n * (\sum_{i=0}^{n-1} abs(nums[i]) - target))$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **690. 员工的重要性** ，难度为 **简单**。

Tag：「BFS」、「DFS」、「队列」

给定一个保存员工信息的数据结构，它包含了员工 唯一的 id ，重要度 和 直系下属的 id 。

比如，员工 1 是员工 2 的领导，员工 2 是员工 3 的领导。他们相应的重要度为 15 , 10 , 5 。那么员工 1 的数据结构是 [1, 15, [2]] ，员工 2的 数据结构是 [2, 10, [3]] ，员工 3 的数据结构是 [3, 5, []] 。注意虽然员工 3 也是员工 1 的一个下属，但是由于 并不是直系 下属，因此没有体现在员工 1 的数据结构中。

现在输入一个公司的所有员工信息，以及单个员工 id ，返回这个员工和他所有下属的重要度之和。

示例：

```
输入：[[1, 5, [2, 3]], [2, 3, []], [3, 3, []]], 1

输出：11

解释：
员工 1 自身的重要度是 5 ，他有两个直系下属 2 和 3 ，而且 2 和 3 的重要度均为 3 。因此员工 1 的总重要度是 5 + 3 + 3 = 11
```

提示：

- 一个员工最多有一个 直系 领导，但是可以有多个 直系 下属
- 员工数量不超过 2000 。

递归 / DFS

一个直观的做法是，写一个递归函数来统计某个员工的总和。

统计自身的 $importance$ 值和直系下属的 $importance$ 值。同时如果某个下属还有下属的话，则递归这个过程。

代码：

```Java
class Solution {
    Map<Integer, Employee> map = new HashMap<>();
    public int getImportance(List<Employee> es, int id) {
        int n = es.size();
        for (int i = 0; i < n; i++) map.put(es.get(i).id, es.get(i));
        return getVal(id);
    }
    int getVal(int id) {
        Employee master = map.get(id);
        int ans = master.importance;
        for (int oid : master.subordinates) {
            Employee other = map.get(oid);
            ans += other.importance;
            for (int sub : other.subordinates) ans += getVal(sub);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

迭代 / BFS

另外一个做法是使用「队列」来存储所有将要计算的 $Employee$ 对象，每次弹出时进行统计，并将其「下属」添加到队列尾部。

代码：

```java
class Solution {
    public int getImportance(List<Employee> es, int id) {
        int n = es.size();
        Map<Integer, Employee> map = new HashMap<>();
        for (int i = 0; i < n; i++) map.put(es.get(i).id, es.get(i));
        int ans = 0;
        Deque<Employee> d = new ArrayDeque<>();
        d.addLast(map.get(id));
        while (!d.isEmpty()) {
            Employee poll = d.pollFirst();
            ans += poll.importance;
            for (int oid : poll.subordinates) {
                d.addLast(map.get(oid));
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **778. 水位上升的泳池中游泳** ，难度为 **困难**。

Tag：「最小生成树」、「并查集」、「Kruskal」、「二分」、「BFS」

在一个 N x N 的坐标方格 grid 中，每一个方格的值 grid[i][j] 表示在位置 (i,j) 的平台高度。

现在开始下雨了。当时间为 t 时，此时雨水导致水池中任意位置的水位为 t 。

你可以从一个平台游向四周相邻的任意一个平台，但是前提是此时水位必须同时淹没这两个平台。

假定你可以瞬间移动无限距离，也就是默认在方格内部游动是不耗时的。

当然，在你游泳的时候你必须待在坐标方格里面。

你从坐标方格的左上平台 (0，0) 出发，最少耗时多久你才能到达坐标方格的右下平台 (N-1, N-1)？

示例 1:

```
输入: [[0,2],[1,3]]

输出: 3

解释:
时间为0时，你位于坐标方格的位置为 (0, 0)。
此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时的水位。
等时间到达 3 时，你才可以游向平台 (1, 1). 因为此时的水位是 3，坐标方格中的平台没有比水位 3 更高的，所以你可以游向坐标方格中
```

示例2:

```
输入: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]
输出: 16

解释:
 0  1  2  3  4
             5
12 13 14 15 16
11
10  9  8  7  6
```

提示:

- 2 <= N <= 50.
- grid[i][j] 是 `[0, ..., N*N - 1]` 的排列。

Kruskal

由于在任意点可以往任意方向移动，所以相邻的点（四个方向）之间存在一条无向边。

边的权重 w 是指两点节点中的最大高度。

按照题意，我们需要找的是从左上角点到右下角点的最优路径，其中最优路径是指**途径的边的最大权重值最小**，然后输入最优路径中的最大权重值。

我们可以先遍历所有的点，将所有的边加入集合，存储的格式为数组 $[a, b, w]$ ，代表编号为 a 的点和编号为 b 的点之间的权重为 w（按照题意，w 为两者的最大高度）。

对集合进行排序，按照 w 进行从小到达排序。

当我们有了所有排好序的候选边集合之后，我们可以对边从前往后处理，每次加入一条边之后，使用并查集来查询左上角的点和右下角的点是否连通。

当我们的合并了某条边之后，判定左上角和右下角的点联通，那么该边的权重即是答案。

这道题和前天的 1631. 最小体力消耗路径 几乎是完全一样的思路。

你甚至可以将那题的代码拷贝过来，改一下对于 w 的定义即可。

代码：

```Java
class Solution {
    int n;
    int[] p;
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    boolean query(int a, int b) {
        return find(a) == find(b);
    }
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    public int swimInWater(int[][] grid) {
        n = grid.length;

        // 初始化并查集
        p = new int[n * n];
        for (int i = 0; i < n * n; i++) p[i] = i;

        // 预处理出所有的边
        // edge 存的是 [a, b, w]：代表从 a 到 b 所需要的时间为 w
        // 虽然我们可以往四个方向移动，但是只要对于每个点都添加「向右」和「向下」两条边的话，其实就已经覆盖
        List<int[]> edges =  new ArrayList<>();
        for (int i = 0; i < n ;i++) {
            for (int j = 0; j < n; j++) {
                int idx = getIndex(i, j);
                p[idx] = idx;
                if (i + 1 < n) {
                    int a = idx, b = getIndex(i + 1, j);
                    int w = Math.max(grid[i][j], grid[i + 1][j]);
                    edges.add(new int[]{a, b, w});
                }
                if (j + 1 < n) {
                    int a = idx, b = getIndex(i, j + 1);
                    int w = Math.max(grid[i][j], grid[i][j + 1]);
                    edges.add(new int[]{a, b, w});
                }
            }
        }

        // 根据权值 w 升序
        Collections.sort(edges, (a,b)->a[2]-b[2]);

        // 从「小边」开始添加，当某一条边刚应用之后，恰好使用得「起点」和「结点」联通
```

```
            // 那么代表找到了「最短路径」中的「权重最大的边」
            int start = getIndex(0, 0), end = getIndex(n - 1, n - 1);
            for (int[] edge : edges) {
                int a = edge[0], b = edge[1], w = edge[2];
                union(a, b);
                if (query(start, end)) {
                    return w;
                }
            }
            return -1;
        }
        int getIndex(int i, int j) {
            return i * n + j;
        }
    }
```

节点的数量为 $n * n$，无向边的数量严格为 $2 * n * (n - 1)$，数量级上为 n^2。

- 时间复杂度：获取所有的边复杂度为 $O(n^2)$，排序复杂度为 $O(n^2 \log n)$，遍历得到最终解复杂度为 $O(n^2)$。整体复杂度为 $O(n^2 \log n)$。
- 空间复杂度：使用了并查集数组。复杂度为 $O(n^2)$。

注意：假定 Collections.sort() 使用 Arrays.sort() 中的双轴快排实现。

二分 + BFS/DFS

在与本题类型的 1631. 最小体力消耗路径中，有同学问到是否可以用「二分」。

答案是可以的。

题目给定了 $grid[i][j]$ 的范围是 $[0, n^2 - 1]$，所以答案必然落在此范围。

假设最优解为 min 的话（恰好能到达右下角的时间）。那么小于 min 的时间无法到达右下角，大于 min 的时间能到达右下角。

因此在以最优解 min 为分割点的数轴上具有两段性，可以通过「二分」来找到分割点 min。

> 注意：「二分」的本质是两段性，并非单调性。只要一段满足某个性质，另外一段不满足某个性质，就可以用「二分」。其中 33. 搜索旋转排序数组 是一个很好的说明例子。

接着分析，假设最优解为 min，我们在 $[l, r]$ 范围内进行二分，当前二分到的时间为 mid 时：

1. 能到达右下角：必然有 $min \leqslant mid$，让 $r = mid$
2. 不能到达右下角：必然有 $min > mid$，让 $l = mid + 1$

当确定了「二分」逻辑之后，我们需要考虑如何写 $check$ 函数。

显然 $check$ 应该是一个判断给定 时间/步数 能否从「起点」到「终点」的函数。

我们只需要按照规则走特定步数，边走边检查是否到达终点即可。

实现 $check$ 既可以使用 DFS 也可以使用 BFS。两者思路类似，这里就只以 BFS 为例。

代码：

```Java
class Solution {
    int[][] dirs = new int[][]{{1,0}, {-1,0}, {0,1}, {0,-1}};
    public int swimInWater(int[][] grid) {
        int n = grid.length;
        int l = 0, r = n * n;
        while (l < r) {
            int mid = l + r >> 1;
            if (check(grid, mid)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return r;
    }
    boolean check(int[][] grid, int time) {
        int n = grid.length;
        boolean[][] visited = new boolean[n][n];
        Deque<int[]> queue = new ArrayDeque<>();
        queue.addLast(new int[]{0, 0});
        visited[0][0] = true;
        while (!queue.isEmpty()) {
            int[] pos = queue.pollFirst();
            int x = pos[0], y = pos[1];
            if (x == n - 1 && y == n - 1) return true;

            for (int[] dir : dirs) {
                int newX = x + dir[0], newY = y + dir[1];
                int[] to = new int[]{newX, newY};
                if (inArea(n, newX, newY) && !visited[newX][newY] && canMove(grid, pos, to
                    visited[newX][newY] = true;
                    queue.addLast(to);
                }
            }
        }
        return false;
    }
    boolean inArea(int n, int x, int y) {
        return x >= 0 && x < n && y >= 0 && y < n;
    }
    boolean canMove(int[][] grid, int[] from, int[] to, int time) {
        return time >= Math.max(grid[from[0]][from[1]], grid[to[0]][to[1]]);
    }
}
```

- 时间复杂度：在 $[0, n^2]$ 范围内进行二分，复杂度为 $O(\log n)$；每一次 BFS 最多

有 n^2 个节点入队，复杂度为 $O(n^2)$。整体复杂度为 $O(n^2 \log n)$

- 空间复杂度：使用了 visited 数组。复杂度为 $O(n^2)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

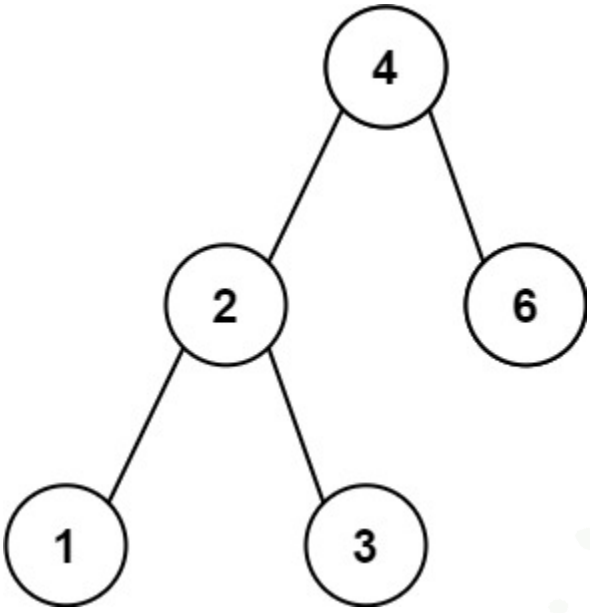
这是 LeetCode 上的 **783. 二叉搜索树节点最小距离** ，难度为 **简单**。

Tag：「树的搜素」、「迭代」、「非迭代」、「中序遍历」、「BFS」、「DFS」

给你一个二叉搜索树的根节点 root ，返回 树中任意两不同节点值之间的最小差值 。

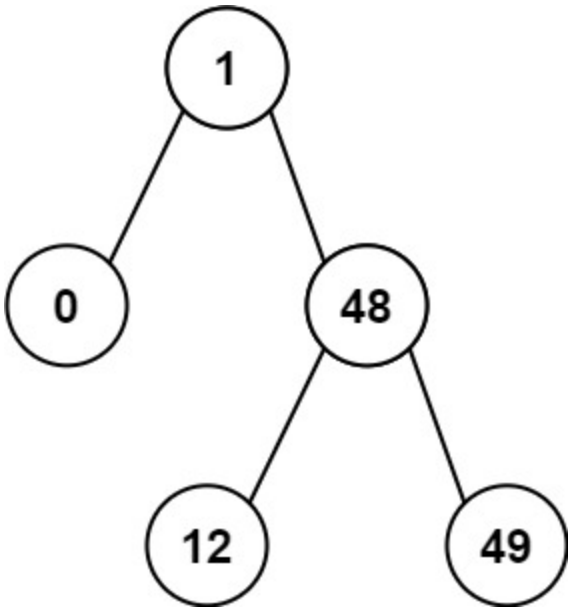
注意：本题与 530：https://leetcode-cn.com/problems/minimum-absolute-difference-in-bst/ 相同

示例 1：




```
输入：root = [4,2,6,1,3]

输出：1
```

示例 2：




```
输入：root = [1,0,48,null,null,12,49]

输出：1
```

提示：

- 树中节点数目在范围 [2, 100] 内
- 0 <= Node.val <= 10^5
- 差值是一个正数，其数值等于两值之差的绝对值

朴素解法（BFS & DFS）

如果不考虑利用二叉搜索树特性的话，一个朴素的做法是将所有节点的 val 存到一个数组中。

对数组进行排序，并获取答案。

将所有节点的 val 存入数组，可以使用 BFS 或者 DFS。

代码：

```Java
class Solution {
    public int minDiffInBST(TreeNode root) {
        List<Integer> list = new ArrayList<>();

        // BFS
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        while (!d.isEmpty()) {
            TreeNode poll = d.pollFirst();
            list.add(poll.val);
            if (poll.left != null) d.addLast(poll.left);
            if (poll.right != null) d.addLast(poll.right);
        }

        // DFS
        // dfs(root, list);

        Collections.sort(list);
        int n = list.size();
        int ans = Integer.MAX_VALUE;
        for (int i = 1; i < n; i++) {
            int cur = Math.abs(list.get(i) - list.get(i - 1));
            ans = Math.min(ans, cur);
        }
        return ans;
    }
    void dfs(TreeNode root, List<Integer> list) {
        list.add(root.val);
        if (root.left != null) dfs(root.left, list);
        if (root.right != null) dfs(root.right, list);
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(n)$

中序遍历（栈模拟 & 递归）

不难发现，在朴素解法中，我们对树进行搜索的目的是为了获取一个「有序序列」，然后从「有序序列」中获取答案。

而二叉搜索树的中序遍历是有序的，因此我们可以直接对「二叉搜索树」进行中序遍历，保存遍

历过程中的相邻元素最小值即是答案。

代码：

```
class Solution {
    int ans = Integer.MAX_VALUE;
    TreeNode prev = null;
    public int minDiffInBST(TreeNode root) {
        // 栈模拟
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }
            root = d.pollLast();
            if (prev != null) {
                ans = Math.min(ans, Math.abs(prev.val - root.val));
            }
            prev = root;
            root = root.right;
        }

        // 递归
        // dfs(root);

        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        dfs(root.left);
        if (prev != null) {
            ans = Math.min(ans, Math.abs(prev.val - root.val));
        }
        prev = root;
        dfs(root.right);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

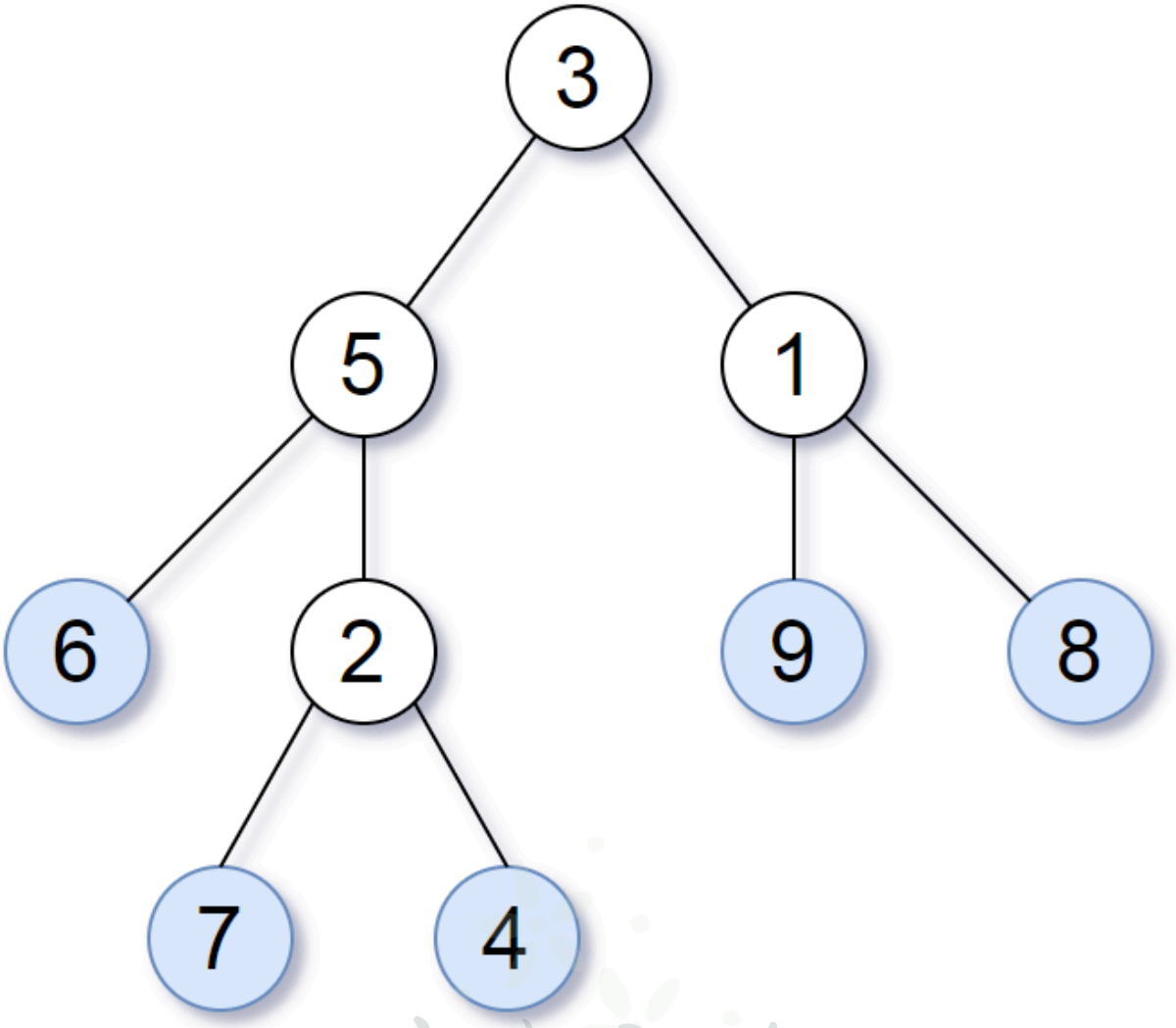
> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **872. 叶子相似的树** ，难度为 **简单**。

Tag：「树的搜索」、「非递归」、「递归」、「DFS」

请考虑一棵二叉树上所有的叶子，这些叶子的值按从左到右的顺序排列形成一个 叶值序列 。

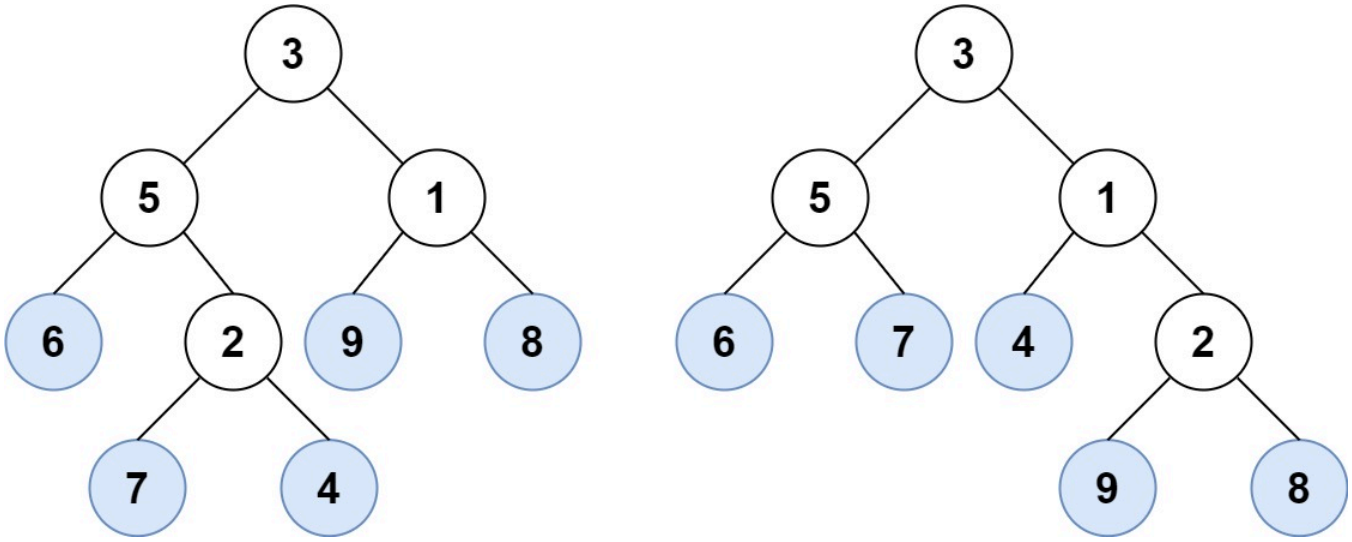


举个例子，如上图所示，给定一棵叶值序列为 (6, 7, 4, 9, 8) 的树。

如果有两棵二叉树的叶值序列是相同，那么我们就认为它们是 叶相似 的。

如果给定的两个根结点分别为 root1 和 root2 的树是叶相似的，则返回 true；否则返回 false 。

示例 1：




```
输入：
root1 = [3,5,1,6,2,9,8,null,null,7,4],
root2 = [3,5,1,6,7,4,2,null,null,null,null,null,null,9,8]

输出：true
```

示例 2：

```
输入：root1 = [1], root2 = [1]

输出：true
```

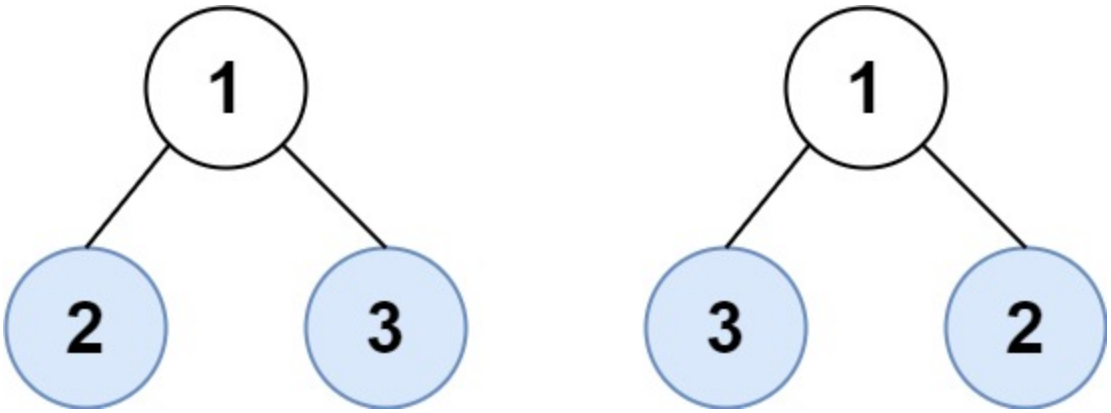
示例 3：

```
输入：root1 = [1], root2 = [2]

输出：false
```

示例 4：

```
输入：root1 = [1,2], root2 = [2,2]

输出：true
```

示例 5：




```
输入：root1 = [1,2,3], root2 = [1,3,2]

输出：false
```

提示：

- 给定的两棵树可能会有 1 到 200 个结点。
- 给定的两棵树上的值介于 0 到 200 之间。

递归

递归写法十分简单，属于树的遍历中最简单的实现方式。

代码：

```Java
class Solution {
    public boolean leafSimilar(TreeNode t1, TreeNode t2) {
        List<Integer> l1 = new ArrayList<>(), l2 = new ArrayList<>();
        dfs(t1, l1);
        dfs(t2, l2);
        if (l1.size() == l2.size()) {
            for (int i = 0; i < l1.size(); i++) {
                if (!l1.get(i).equals(l2.get(i))) return false;
            }
            return true;
        }
        return false;
    }
    void dfs(TreeNode root, List<Integer> list) {
        if (root == null) return;
        if (root.left == null && root.right == null) {
            list.add(root.val);
            return;
        }
        dfs(root.left, list);
        dfs(root.right, list);
    }
}
```

- 时间复杂度：`n` 和 `m` 分别代表两棵树的节点数量。复杂度为 $O(n+m)$
- 空间复杂度：`n` 和 `m` 分别代表两棵树的节点数量，当两棵树都只有一层的情况，所有的节点值都会被存储在 $list$ 中。复杂度为 $O(n+m)$

迭代

迭代其实就是使用「栈」来模拟递归过程，也属于树的遍历中的常见实现形式。

一般简单的面试中如果问到树的遍历，面试官都不会对「递归」解法感到满意，因此掌握「迭代/非递归」写法同样重要。

代码：

```Java
class Solution {
    public boolean leafSimilar(TreeNode t1, TreeNode t2) {
        List<Integer> l1 = new ArrayList<>(), l2 = new ArrayList<>();
        process(t1, l1);
        process(t2, l2);
        if (l1.size() == l2.size()) {
            for (int i = 0; i < l1.size(); i++) {
                if (!l1.get(i).equals(l2.get(i))) return false;
            }
            return true;
        }
        return false;
    }
    void process(TreeNode root, List<Integer> list) {
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }
            root = d.pollLast();
            if (root.left == null && root.right == null) list.add(root.val);
            root = root.right;
        }
    }
}
```

- 时间复杂度：`n` 和 `m` 分别代表两棵树的节点数量。复杂度为 $O(n+m)$
- 空间复杂度：`n` 和 `m` 分别代表两棵树的节点数量，当两棵树都只有一层的情况，所有的节点值都会被存储在 $list$ 中。复杂度为 $O(n+m)$

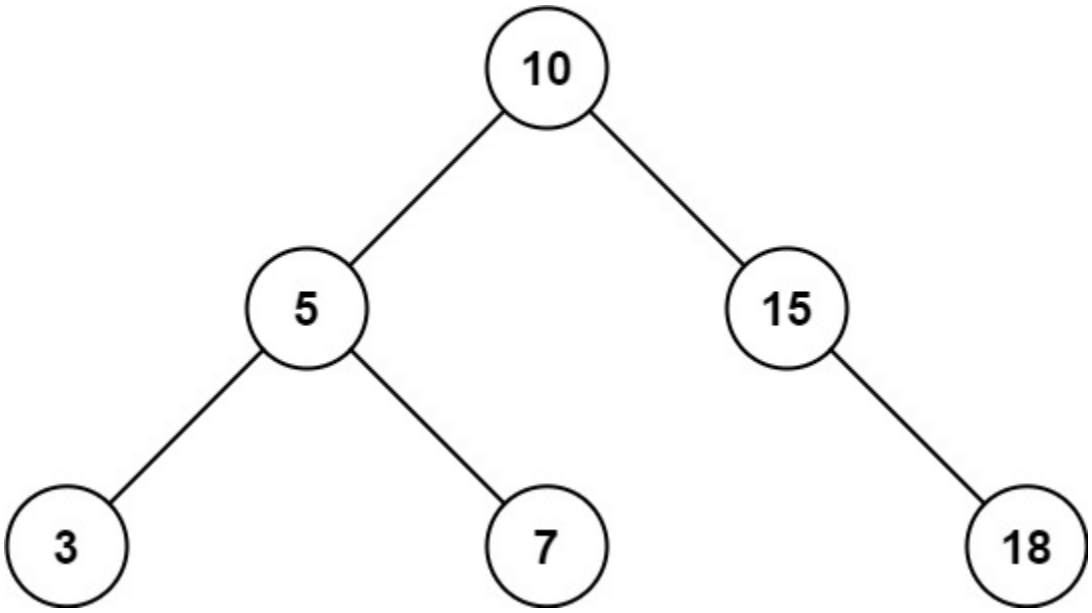
题目描述

这是 LeetCode 上的 **938. 二叉搜索树的范围和** ，难度为 **简单**。

Tag：「树的搜索」、「DFS」、「BFS」

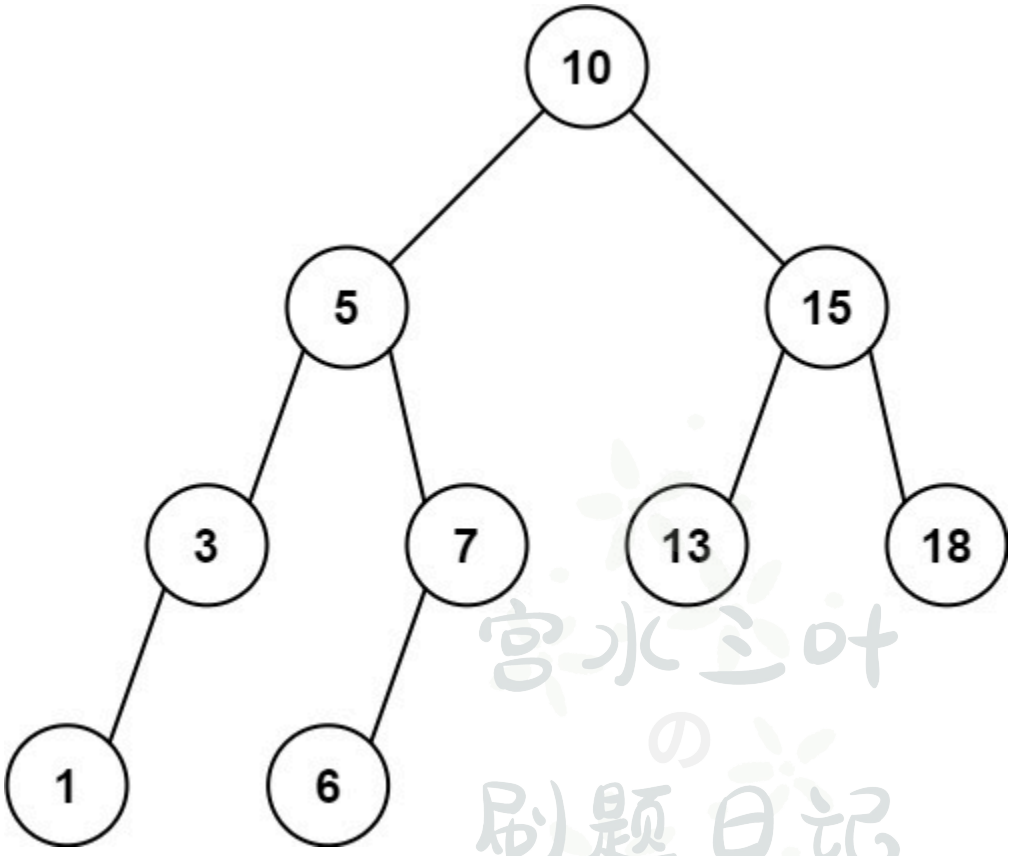
给定二叉搜索树的根结点 root，返回值位于范围 [low, high] 之间的所有结点的值的和。

示例 1：




```
输入：root = [10,5,15,3,7,null,18], low = 7, high = 15

输出：32
```

示例 2：



```
输入：root = [10,5,15,3,7,13,18,1,null,6], low = 6, high = 10

输出：23
```

提示：

- 树中节点数目在范围 [1, 2 * 10^4] 内
- 1 <= Node.val <= 10^5
- 1 <= low <= high <= 10^5
- 所有 Node.val 互不相同

基本思路

这又是众多「二叉搜索树遍历」题目中的一道。

二叉搜索树的中序遍历是有序的。

只要对其进行「中序遍历」即可得到有序列表，在遍历过程中判断节点值是否符合要求，对于符合要求的节点值进行累加即可。

二叉搜索树的「中序遍历」有「迭代」和「递归」两种形式。**由于给定了值范围 $[low, high]$，因此可以在遍历过程中做一些剪枝操作，但并不影响时空复杂度。**

递归

递归写法十分简单，属于树的遍历中最简单的实现方式。

代码：

```Java
class Solution {
    int low, high;
    int ans;
    public int rangeSumBST(TreeNode root, int _low, int _high) {
        low = _low; high = _high;
        dfs(root);
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        dfs(root.left);
        if (low <= root.val && root.val <= high) ans += root.val;
        dfs(root.right);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

迭代

迭代其实就是使用「栈」来模拟递归过程，也属于树的遍历中的常见实现形式。

一般简单的面试中如果问到树的遍历，面试官都不会对「递归」解法感到满意，因此掌握「迭代/非递归」写法同样重要。

代码：

```java
class Solution {
    public int rangeSumBST(TreeNode root, int low, int high) {
        int ans = 0;
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }
            root = d.pollLast();
            if (low <= root.val && root.val <= high) {
                ans += root.val;
            }
            root = root.right;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **987. 二叉树的垂序遍历** ，难度为 **困难**。

Tag：「数据结构运用」、「二叉树」、「哈希表」、「排序」、「优先队列」、「DFS」

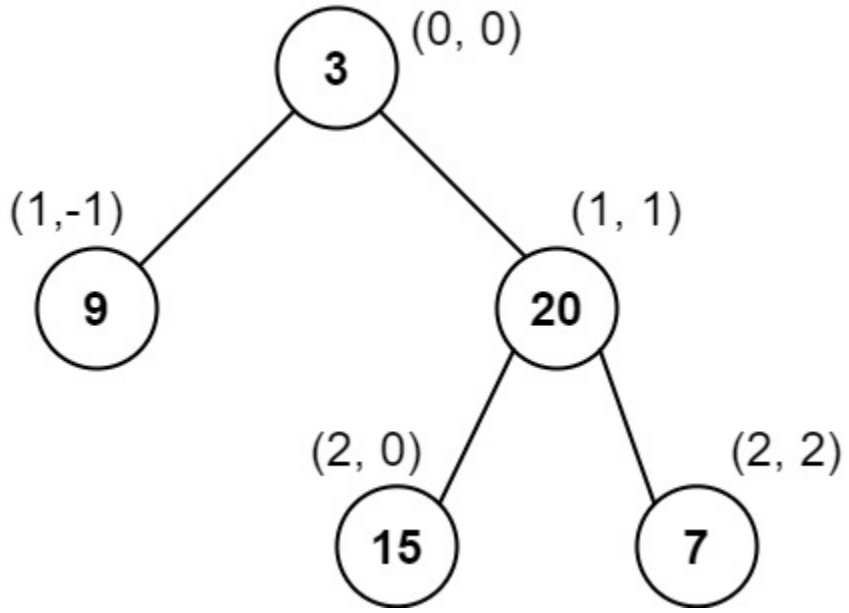
给你二叉树的根结点 root ，请你设计算法计算二叉树的 垂序遍历 序列。

对位于 (row, col) 的每个结点而言，其左右子结点分别位于 (row + 1, col - 1) 和 (row + 1, col + 1) 。树的根结点位于 (0, 0) 。

二叉树的 垂序遍历 从最左边的列开始直到最右边的列结束，按列索引每一列上的所有结点，形成一个按出现位置从上到下排序的有序列表。如果同行同列上有多个结点，则按结点的值从小到大进行排序。

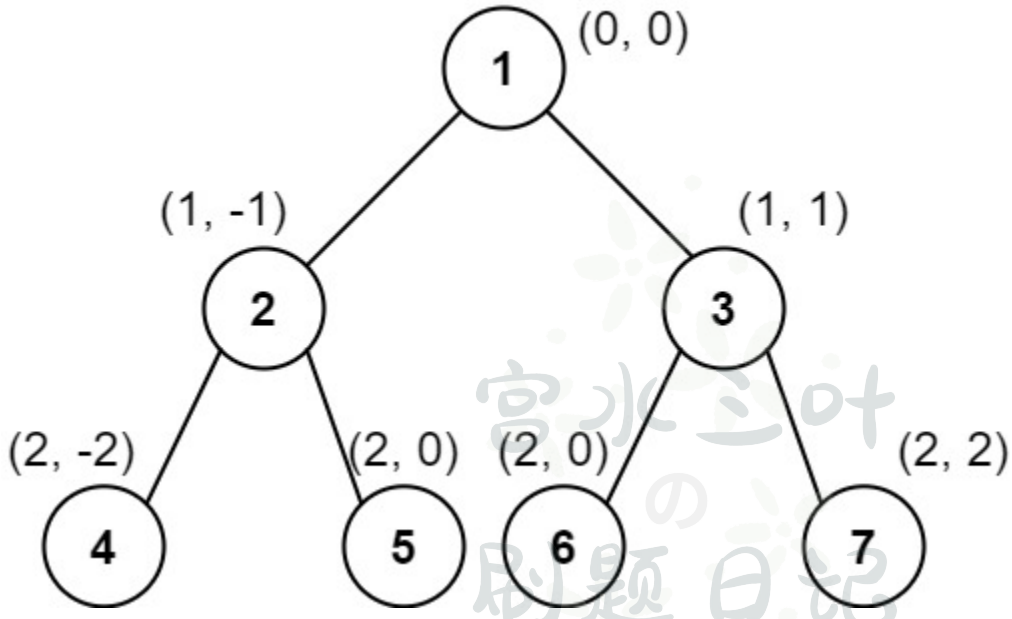
返回二叉树的 垂序遍历 序列。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]

输出：[[9],[3,15],[20],[7]]

解释：
列 -1 ：只有结点 9 在此列中。
列  0 ：只有结点 3 和 15 在此列中，按从上到下顺序。
列  1 ：只有结点 20 在此列中。
列  2 ：只有结点 7 在此列中。
```

示例 2：

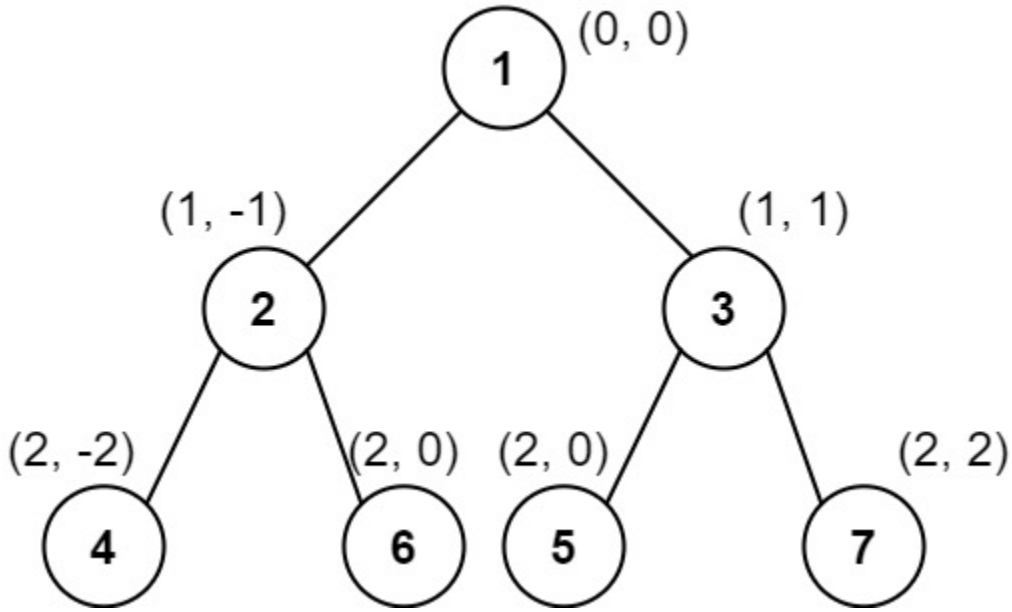


输入：root = [1,2,3,4,5,6,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：
列 -2 ：只有结点 4 在此列中。
列 -1 ：只有结点 2 在此列中。
列 0 ：结点 1 、5 和 6 都在此列中。
1 在上面，所以它出现在前面。
5 和 6 位置都是 (2, 0) ，所以按值从小到大排序，5 在 6 的前面。
列 1 ：只有结点 3 在此列中。
列 2 ：只有结点 7 在此列中。

示例 3：



输入：root = [1,2,3,4,6,5,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：
这个示例实际上与示例 2 完全相同，只是结点 5 和 6 在树中的位置发生了交换。
因为 5 和 6 的位置仍然相同，所以答案保持不变，仍然按值从小到大排序。

提示：

- 树中结点数目总数在范围 [1, 10]

DFS + 哈希表 + 排序

根据题意，我们需要按照优先级「**“列号从小到大”，对于同列节点，“行号从小到大”，对于同列同行元素，“节点值从小到大”**」进行答案构造。

因此我们可以对树进行遍历，遍历过程中记下这些信息 (col, row, val)，然后根据规则进行排序，并构造答案。

我们可以先使用「哈希表」进行存储，最后再进行一次性的排序。

代码：

```Java
class Solution {
    Map<TreeNode, int[]> map = new HashMap<>(); // col, row, val
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        map.put(root, new int[]{0, 0, root.val});
        dfs(root);
        List<int[]> list = new ArrayList<>(map.values());
        Collections.sort(list, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            if (a[1] != b[1]) return a[1] - b[1];
            return a[2] - b[2];
        });
        int n = list.size();
        List<List<Integer>> ans = new ArrayList<>();
        for (int i = 0; i < n; ) {
            int j = i;
            List<Integer> tmp = new ArrayList<>();
            while (j < n && list.get(j)[0] == list.get(i)[0]) tmp.add(list.get(j++)[2]);
            ans.add(tmp);
            i = j;
        }
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return ;
        int[] info = map.get(root);
        int col = info[0], row = info[1], val = info[2];
        if (root.left != null) {
            map.put(root.left, new int[]{col - 1, row + 1, root.left.val});
            dfs(root.left);
        }
        if (root.right != null) {
            map.put(root.right, new int[]{col + 1, row + 1, root.right.val});
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：令总节点数量为 n，填充哈希表时进行树的遍历，复杂度为 $O(n)$；构造答案时需要进行排序，复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

DFS + 优先队列（堆）

显然，最终要让所有节点的相应信息有序，可以使用「优先队列（堆）」边存储边维护有序性。

代码：

```
class Solution {
    PriorityQueue<int[]> q = new PriorityQueue<>((a, b)->{ // col, row, val
        if (a[0] != b[0]) return a[0] - b[0];
        if (a[1] != b[1]) return a[1] - b[1];
        return a[2] - b[2];
    });
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        int[] info = new int[]{0, 0, root.val};
        q.add(info);
        dfs(root, info);
        List<List<Integer>> ans = new ArrayList<>();
        while (!q.isEmpty()) {
            List<Integer> tmp = new ArrayList<>();
            int[] poll = q.peek();
            while (!q.isEmpty() && q.peek()[0] == poll[0]) tmp.add(q.poll()[2]);
            ans.add(tmp);
        }
        return ans;
    }
    void dfs(TreeNode root, int[] fa) {
        if (root.left != null) {
            int[] linfo = new int[]{fa[0] - 1, fa[1] + 1, root.left.val};
            q.add(linfo);
            dfs(root.left, linfo);
        }
        if (root.right != null) {
            int[] rinfo = new int[]{fa[0] + 1, fa[1] + 1, root.right.val};
            q.add(rinfo);
            dfs(root.right, rinfo);
        }
    }
}
```

- 时间复杂度：令总节点数量为 n，将节点信息存入优先队列（堆）复杂度为 $O(n\log n)$；构造答案复杂度为 $O(n\log n)$。整体复杂度为 $O(n\log n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **993. 二叉树的堂兄弟节点** ，难度为 **简单**。

Tag：「树的搜索」、「BFS」、「DFS」

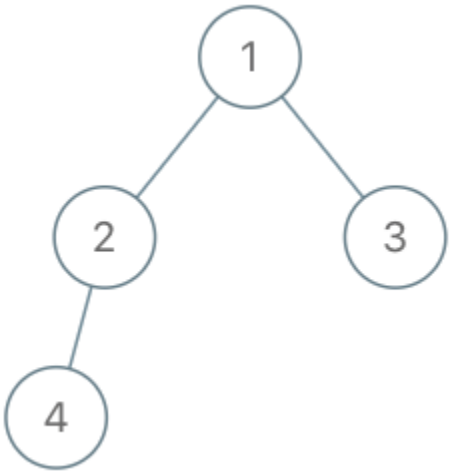
在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 k+1 处。

如果二叉树的两个节点深度相同，但 父节点不同 ，则它们是一对堂兄弟节点。

我们给出了具有唯一值的二叉树的根节点 root ，以及树中两个不同节点的值 x 和 y 。

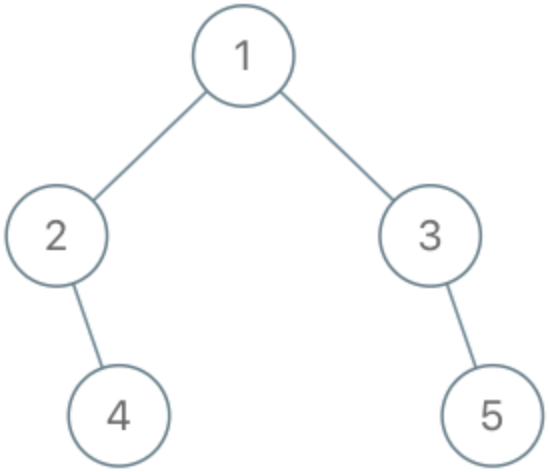
只有与值 x 和 y 对应的节点是堂兄弟节点时，才返回 true 。否则，返回 false。

示例 1：



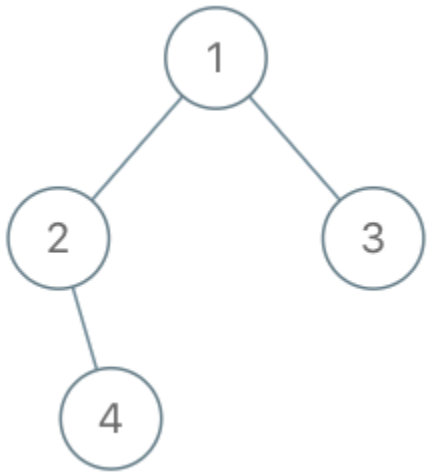

```
输入：root = [1,2,3,4], x = 4, y = 3

输出：false
```

示例 2：




```
输入：root = [1,2,3,null,4,null,5], x = 5, y = 4

输出：true
```

示例 3：




```
输入：root = [1,2,3,null,4], x = 2, y = 3

输出：false
```

提示：

- 二叉树的节点数介于 2 到 100 之间。
- 每个节点的值都是唯一的、范围为 1 到 100 的整数。

DFS

显然，我们希望得到某个节点的「父节点」&「所在深度」，不难设计出如下「DFS 函数签名」：

```
/**
 * 查找 t 的「父节点值」&「所在深度」
 * @param root 当前搜索到的节点
 * @param fa root 的父节点
 * @param depth 当前深度
 * @param t 搜索目标值
 * @return [fa.val, depth]
 */
int[] dfs(TreeNode root, TreeNode fa, int depth, int t);
```

之后按照遍历的逻辑处理即可。

需要注意的时，我们需要区分出「搜索不到」和「搜索对象为 root（没有 fa 父节点）」两种情况。

我们约定使用 -1 代指没有找到目标值 t，使用 0 代表找到了目标值 t，但其不存在父节点。

代码：

```
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
        int[] xi = dfs(root, null, 0, x);
        int[] yi = dfs(root, null, 0, y);
        return xi[1] == yi[1] && xi[0] != yi[0];
    }
    int[] dfs(TreeNode root, TreeNode fa, int depth, int t) {
        if (root == null) return new int[]{-1, -1}; // 使用 -1 代表为搜索不到 t
        if (root.val == t) {
            return new int[]{fa != null ? fa.val : 1, depth}; // 使用 1 代表搜索值 t 为 root
        }
        int[] l = dfs(root.left, root, depth + 1, t);
        if (l[0] != -1) return l;
        return dfs(root.right, root, depth + 1, t);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：忽略递归开销为 $O(1)$，否则为 $O(n)$

BFS

能使用 `DFS`，自然也能使用 `BFS`，两者大同小异。

代码：

```
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
        int[] xi = bfs(root, x);
        int[] yi = bfs(root, y);
        return xi[1] == yi[1] && xi[0] != yi[0];
    }
    int[] bfs(TreeNode root, int t) {
        Deque<Object[]> d = new ArrayDeque<>(); // 存储值为 [cur, fa, depth]
        d.addLast(new Object[]{root, null, 0});
        while (!d.isEmpty()) {
            int size = d.size();
            while (size-- > 0) {
                Object[] poll = d.pollFirst();
                TreeNode cur = (TreeNode)poll[0], fa = (TreeNode)poll[1];
                int depth = (Integer)poll[2];

                if (cur.val == t) return new int[]{fa != null ? fa.val : 0, depth};
                if (cur.left != null) d.addLast(new Object[]{cur.left, cur, depth + 1});
                if (cur.right != null) d.addLast(new Object[]{cur.right, cur, depth + 1});
            }
        }
        return new int[]{-1, -1};
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **1239. 串联字符串的最大长度** ，难度为 **中等**。

Tag：「DFS」、「二进制枚举」、「模拟退火」

给定一个字符串数组 arr，字符串 s 是将 arr 某一子序列字符串连接所得的字符串，如果 s 中的每一个字符都只出现过一次，那么它就是一个可行解。

请返回所有可行解 s 中最长长度。

示例 1：

```
输入：arr = ["un","iq","ue"]
输出：4
解释：所有可能的串联组合是 "","un","iq","ue","uniq" 和 "ique"，最大长度为 4。
```

示例 2：

```
输入：arr = ["cha","r","act","ers"]
输出：6
解释：可能的解答有 "chaers" 和 "acters"。
```

示例 3：

```
输入：arr = ["abcdefghijklmnopqrstuvwxyz"]
输出：26
```

提示：

- 1 <= arr.length <= 16
- 1 <= arr[i].length <= 26
- arr[i] 中只含有小写英文字母

基本分析

根据题意，可以将本题看做一类特殊的「数独问题」：在给定的 `arr` 字符数组中选择，尽可能多的覆盖一个 $1 * 26$ 的矩阵。

对于此类「精确覆盖」问题，换个角度也可以看做「组合问题」。

通常有几种做法：`DFS`、剪枝 `DFS`、二进制枚举、模拟退火、`DLX`。

其中一头一尾解法过于简单和困难，有兴趣的同学自行了解与实现。

基本分析

根据题意，可以将本题看做一类特殊的「数独问题」：在给定的 `arr` 字符数组中选择，尽可能多的覆盖一个 $1 * 26$ 的矩阵。

对于此类「精确覆盖」问题，换个角度也可以看做「组合问题」。

通常有几种做法：`DFS`、剪枝 `DFS`、二进制枚举、模拟退火、`DLX`。

其中一头一尾解法过于简单和困难，有兴趣的同学自行了解与实现。

剪枝 DFS

根据题意，可以有如下的剪枝策略：

1. 预处理掉「本身具有重复字符」的无效字符串，并去重；
2. 由于只关心某个字符是否出现，而不关心某个字符在原字符串的位置，因此可以将字符串使用 `int` 进行表示；
3. 由于使用 `int` 进行表示，因而可以使用「位运算」来判断某个字符是否可以被追加到当前状态中；
4. `DFS` 过程中维护一个 `total`，代表后续未经处理的字符串所剩余的“最大价值”是多少，从而实现剪枝；
5. 使用 `lowbit` 计算某个状态对应的字符长度是多少；
6. 使用「全局哈希表」记录某个状态对应的字符长度是多少（使用 `static` 修饰，确保某个状态在所有测试数据中只会被计算一次）；
7. 【未应用】由于存在第 4 点这样的「更优性剪枝」，理论上我们可以根据「字符串所包含字符数量」进行从大到小排序，然后再进行 `DFS` 这样效果理论上会更好。想象一下如果存在一个包含所有字母的字符串，先选择该字符串，后续所有字符串将不能被添加，那么由它出发的分支数量为 0；而如果一个字符串只包含单个字

母，先决策选择该字符串，那么由它出发的分支数量必然大于 0。但该策略实测效果不好，没有添加到代码中。

代码：

```java
class Solution {
    // 本来想使用如下逻辑将「所有可能用到的状态」打表，实现 O(1) 查询某个状态有多少个字符，但是被卡了
    // static int N = 26, M = (1 << N);
    // static int[] cnt = new int[M];
    // static {
    //     for (int i = 0; i < M; i++) {
    //         for (int j = 0; j < 26; j++) {
    //             if (((i >> j) & 1) == 1) cnt[i]++;
    //         }
    //     }
    // }

    static Map<Integer, Integer> map = new HashMap<>();
    int get(int cur) {
        if (map.containsKey(cur)) {
            return map.get(cur);
        }
        int ans = 0;
        for (int i = cur; i > 0; i -= lowbit(i)) ans++;
        map.put(cur, ans);
        return ans;
    }
    int lowbit(int x) {
        return x & -x;
    }

    int n;
    int ans = Integer.MIN_VALUE;
    int[] hash;
    public int maxLength(List<String> _ws) {
        n = _ws.size();
        HashSet<Integer> set = new HashSet<>();
        for (String s : _ws) {
            int val = 0;
            for (char c : s.toCharArray()) {
                int t = (int)(c - 'a');
                if (((val >> t) & 1) != 0) {
                    val = -1;
                    break;
                }
                val |= (1 << t);
            }
            if (val != -1) set.add(val);
        }

        n = set.size();
```

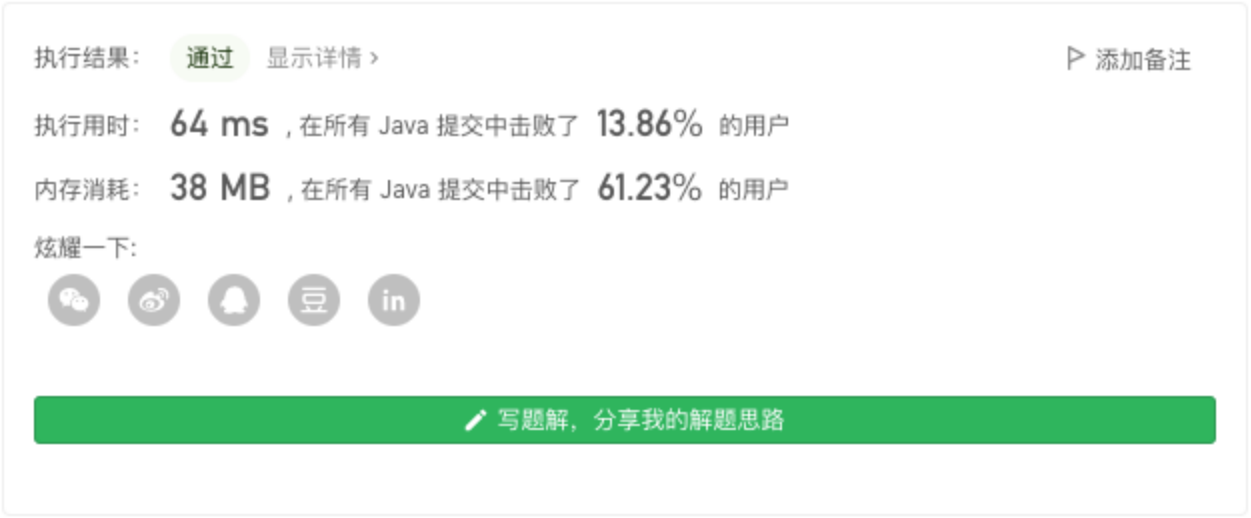
```
        if (n == 0) return 0;
        hash = new int[n];

        int idx = 0;
        int total = 0;
        for (Integer i : set) {
            hash[idx++] = i;
            total |= i;
        }
        dfs(0, 0, total);
        return ans;
    }
    void dfs(int u, int cur, int total) {
        if (get(cur | total) <= ans) return;
        if (u == n) {
            ans = Math.max(ans, get(cur));
            return;
        }
        // 在原有基础上，选择该数字（如果可以）
        if ((hash[u] & cur) == 0) {
            dfs(u + 1, hash[u] | cur, total - (total & hash[u]));
        }
        // 不选择该数字
        dfs(u + 1, cur, total);
    }
}
```

二进制枚举

首先还是对所有字符串进行预处理。

然后使用「二进制枚举」的方式，枚举某个字符串是否被选择。

举个🌰，$(110)_2$ 代表选择前两个字符串，$(011)_2$ 代表选择后两个字符串，这样我们便可以枚举出所有组合方案。



代码：

```java
class Solution {
    static Map<Integer, Integer> map = new HashMap<>();
    int get(int cur) {
        if (map.containsKey(cur)) {
            return map.get(cur);
        }
        int ans = 0;
        for (int i = cur; i > 0; i -= lowbit(i)) ans++;
        map.put(cur, ans);
        return ans;
    }
    int lowbit(int x) {
        return x & -x;
    }

    int n;
    int ans = Integer.MIN_VALUE;
    Integer[] hash;
    public int maxLength(List<String> _ws) {
        n = _ws.size();
        HashSet<Integer> set = new HashSet<>();
        for (String s : _ws) {
            int val = 0;
            for (char c : s.toCharArray()) {
                int t = (int)(c - 'a');
                if (((val >> t) & 1) != 0) {
                    val = -1;
                    break;
                }
                val |= (1 << t);
            }
            if (val != -1) set.add(val);
        }

        n = set.size();
        if (n == 0) return 0;
        hash = new Integer[n];
        int idx = 0;
        for (Integer i : set) hash[idx++] = i;

        for (int i = 0; i < (1 << n); i++) {
            int cur = 0, val = 0;
            for (int j = 0; j < n; j++) {
                if (((i >> j) & 1) == 1) {
                    if ((cur & hash[j]) == 0) {
                        cur |= hash[j];
```

```
                    val += get(hash[j]);
                } else {
                    cur = -1;
                    break;
                }
            }
        }
        if (cur != -1) ans = Math.max(ans, val);
    }
    return ans;
  }
}
```

模拟退火

事实上，可以将原问题看作求「最优前缀序列」问题，从而使用「模拟退火」进行求解。

具体的，我们可以定义「最优前缀序列」为 **组成最优解所用到的字符串均出现在排列的前面。**

举个🌰，假如构成最优解使用到的字符串集合为 `[a,b,c]`，那么对应 `[a,b,c,...]`、`[a,c,b,...]` 均称为「最优前缀序列」。

不难发现，答案与最优前缀序列是一对多关系，这指导我们可以将「参数」调得宽松一些。

具有「一对多」关系的问题十分适合使用「模拟退火」，使用「模拟退火」可以轻松将本题 `arr.length` 数据范围上升到 60 甚至以上。

调整成比较宽松的参数可以跑赢「二进制枚举」，但为了以后增加数据不容易被 hack，还是使用 `N=400` & `fa=0.90` 的搭配。

「模拟退火」的几个参数的作用在 这里 说过了，不再赘述。

代码：

```java
class Solution {
    static Map<Integer, Integer> map = new HashMap<>();
    int get(int cur) {
        if (map.containsKey(cur)) {
            return map.get(cur);
        }
        int ans = 0;
        for (int i = cur; i > 0; i -= lowbit(i)) ans++;
        map.put(cur, ans);
        return ans;
    }
    int lowbit(int x) {
        return x & -x;
    }

    int n;
    int ans = Integer.MIN_VALUE;
    Random random = new Random(20210619);
    double hi = 1e4, lo = 1e-4, fa = 0.90;
    int N = 400;
    int calc() {
        int mix = 0, cur = 0;
        for (int i = 0; i < n; i++) {
            int hash = ws[i];
            if ((mix & hash) == 0) {
                mix |= hash;
                cur += get(hash);
            } else {
                break;
            }
        }
        ans = Math.max(ans, cur);
        return cur;
    }
    void swap(int[] arr, int i, int j) {
        int c = arr[i];
        arr[i] = arr[j];
        arr[j] = c;
    }
    void sa() {
        for (double t = hi; t > lo; t *= fa) {
            int a = random.nextInt(n), b = random.nextInt(n);
            int prev = calc();
            swap(ws, a, b);
            int cur = calc();
            int diff = prev - cur;
```

```
                if (Math.log(diff / t) >= random.nextDouble()) {
                    swap(ws, a, b);
                }
            }
        }
        int[] ws;
        public int maxLength(List<String> _ws) {
            // 预处理字符串：去重，剔除无效字符
            // 结果这一步后：N 可以下降到 100；fa 可以下降到 0.70，耗时约为 78 ms
            // 为了预留将来添加测试数据，题解还是保持 N = 400 & fa = 0.90 的配置
            n = _ws.size();
            HashSet<Integer> set = new HashSet<>();
            for (String s : _ws) {
                int val = 0;
                for (char c : s.toCharArray()) {
                    int t = (int)(c - 'a');
                    if (((val >> t) & 1) != 0) {
                        val = -1;
                        break;
                    }
                    val |= (1 << t);
                }
                if (val != -1) set.add(val);
            }

            n = set.size();
            if (n == 0) return 0;
            ws = new int[n];
            int idx = 0;
            for (Integer i : set) ws[idx++] = i;

            while (N-- > 0) sa();
            return ans;
        }
    }
```

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1723. 完成所有工作的最短时间** ，难度为 **困难**。

Tag：「DFS」、「模拟退火」

给你一个整数数组 jobs ，其中 jobs[i] 是完成第 i 项工作要花费的时间。

请你将这些工作分配给 k 位工人。所有工作都应该分配给工人，且每项工作只能分配给一位工人。

工人的 工作时间 是完成分配给他们的所有工作花费时间的总和。

请你设计一套最佳的工作分配方案，使工人的 最大工作时间 得以 最小化 。

返回分配方案中尽可能「最小」的 最大工作时间 。

示例 1：

```
输入：jobs = [3,2,3], k = 3

输出：3

解释：给每位工人分配一项工作，最大工作时间是 3 。
```

示例 2：

```
输入：jobs = [1,2,4,7,8], k = 2

输出：11

解释：按下述方式分配工作：
1 号工人：1、2、8（工作时间 = 1 + 2 + 8 = 11）
2 号工人：4、7（工作时间 = 4 + 7 = 11）
最大工作时间是 11 。
```

提示：

- 1 <= k <= jobs.length <= 12
- 1 <= jobs[i] <= 10^7

DFS（TLE）

一看数据范围只有 12，我猜不少同学上来就想 `DFS`，但是注意 `n` 和 `k` 同等规模的，爆搜（`DFS`）的复杂度是 $O(k^n)$ 的。

那么极限数据下的计算量为 12^{12}，远超运算量 10^7。

抱着侥幸的心理一运行，很顺利的卡在了 $43/60$ 个数据：

```
[254,256,256,254,251,256,254,253,255,251,251,255] // n = 12
10 // k = 10
```

代码：

```
class Solution {
    int[] jobs;
    int n, k;
    int ans = 0x3f3f3f3f;
    public int minimumTimeRequired(int[] _jobs, int _k) {
        jobs = _jobs;
        n = jobs.length;
        k = _k;
        int[] sum = new int[k];
        dfs(0, sum, 0);
        return ans;
    }
    /**
     * u   : 当前处理到那个 job
     * sum : 工人的分配情况          例如：sum[0] = x 代表 0 号工人工作量为 x
     * max : 当前的「最大工作时间」
     */
    void dfs(int u, int[] sum, int max) {
        if (max >= ans) return;
        if (u == n) {
            ans = max;
            return;
        }
        for (int i = 0; i < k; i++) {
            sum[i] += jobs[u];
            dfs(u + 1, sum, Math.max(sum[i], max));
            sum[i] -= jobs[u];
        }
    }
}
```

- 时间复杂度：$O(k^n)$
- 空间复杂度：$O(k)$

优先分配「空闲工人」的 DFS

那么 `DFS` 就没法过了吗？

除了 `max >= ans` 以外，我们还要做些别的剪枝吗？

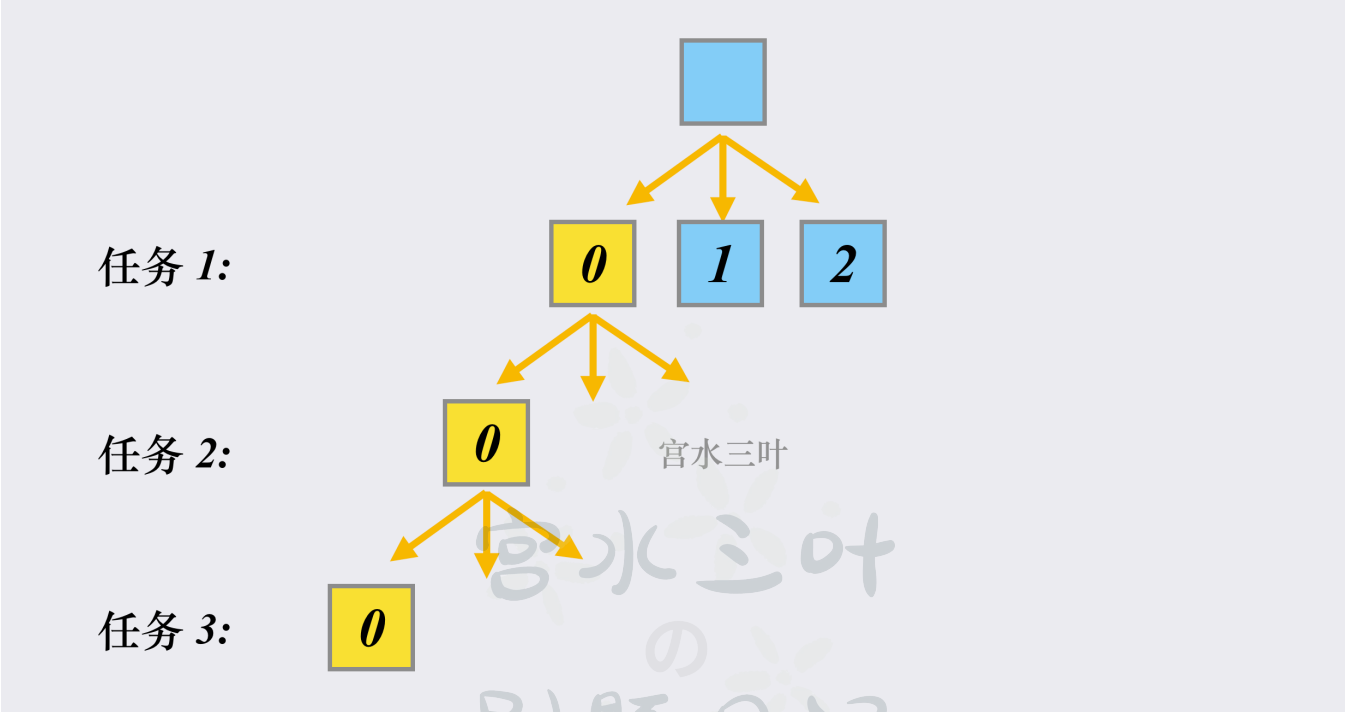
我们可以重新审视一下这道题。

题目其实是让我们将 n 个数分为 k 份，并且尽可能让 k 份平均。这样的「最大工作时间」才是最小的。

但在朴素的 `DFS` 中，我们是将每个任务依次分给每个工人，并递归此过程。

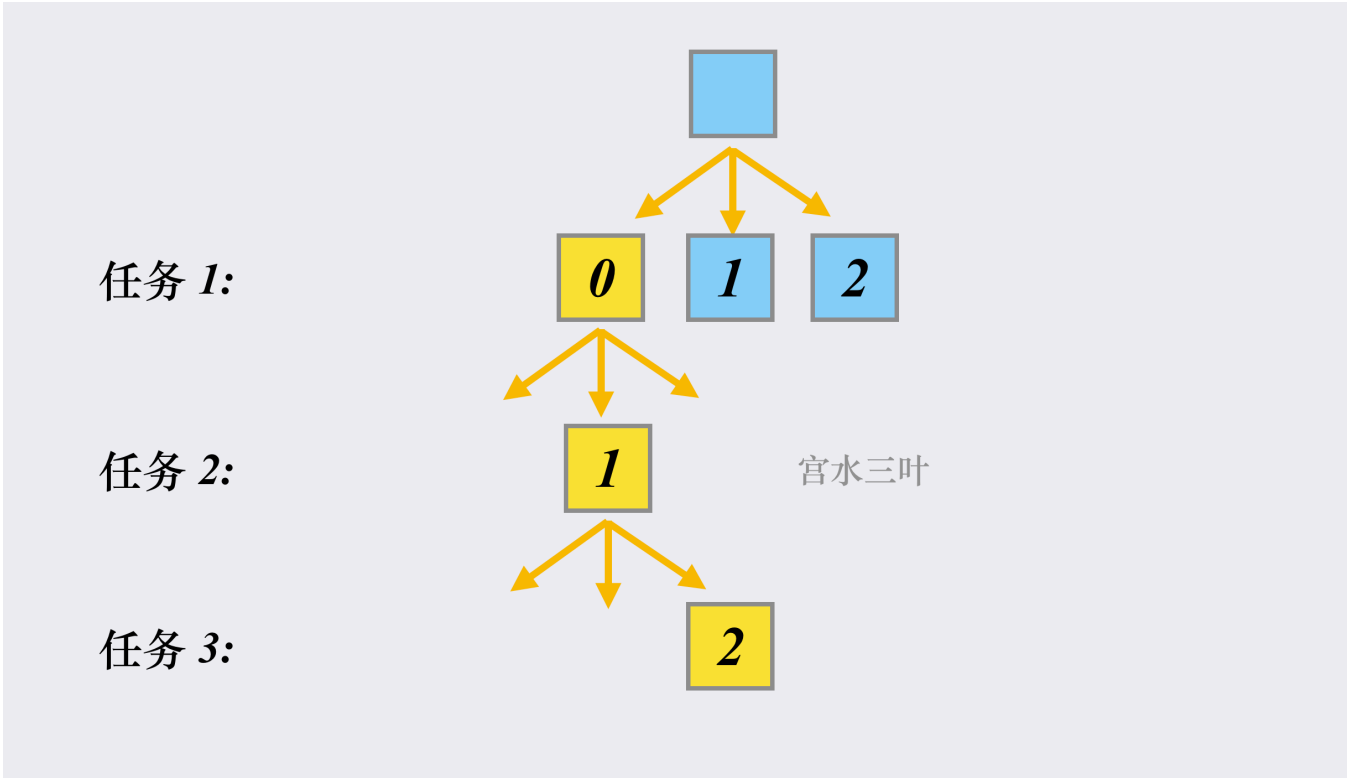
对应的递归树其实是一颗高度为 n 的 k 阶树。

所以其实我们第一次更新的 `ans` 其实是「最差」的答案（所有的任务都会分配给 0 号工人），最差的 `ans` 为所有的 `job` 的总和（带编号的方块代表工人）：



因此我们朴素版的 `DFS` 其实是弱化了 `max >= ans` 剪枝效果的。

那么想要最大化剪枝效果，并且尽量让 k 份平均的话，我们应当调整我们对于「递归树」的搜索方向：将任务优先分配给「空闲工人」（带编号的方块代表工人）：



树还是那棵树，但是搜索调整分配优先级后，我们可以在首次取得一个「较好」的答案，来增强我们的 `max >= ans` 剪枝效益。

事实上，当做完这个调整，我们能实现从 TLE 到 99% 的提升 🤣🤣

代码：

```
class Solution {
    int[] jobs;
    int n, k;
    int ans = 0x3f3f3f3f;
    public int minimumTimeRequired(int[] _jobs, int _k) {
        jobs = _jobs;
        n = jobs.length;
        k = _k;
        int[] sum = new int[k];
        dfs(0, 0, sum, 0);
        return ans;
    }
    /**
     * 【补充说明】不理解可以看看下面的「我猜你问」的 Q5 哦 ~
     *
     * u     : 当前处理到那个 job
     * used  : 当前分配给了多少个工人了
     * sum   : 工人的分配情况            例如：sum[0] = x 代表 0 号工人工作量为 x
     * max   : 当前的「最大工作时间」
     */
    void dfs(int u, int used, int[] sum, int max) {
        if (max >= ans) return;
        if (u == n) {
            ans = max;
            return;
        }
        // 优先分配给「空闲工人」
        if (used < k) {
            sum[used] = jobs[u];
            dfs(u + 1, used + 1, sum, Math.max(sum[used], max));
            sum[used] = 0;
        }
        for (int i = 0; i < used; i++) {
            sum[i] += jobs[u];
            dfs(u + 1, used, sum, Math.max(sum[i], max));
            sum[i] -= jobs[u];
        }
    }
}
```

- 时间复杂度：$O(k^n)$
- 空间复杂度：$O(k)$

模拟退火

事实上，这道题还能使用「模拟退火」进行求解。

因为将 n 个数划分为 k 份，等效于用 n 个数构造出一个「特定排列」，然后对「特定排列」进行固定模式的任务分配逻辑，就能实现「答案」与「最优排列」的对应关系。

基于此，我们可以使用「模拟退火」进行求解。

单次迭代的基本流程：

1. 随机选择两个下标，计算「交换下标元素前对应序列的得分」&「交换下标元素后对应序列的得分」
2. 如果温度下降（交换后的序列更优），进入下一次迭代
3. 如果温度上升（交换前的序列更优），以「一定的概率」恢复现场（再交换回来）

代码：

```java
class Solution {
    int[] jobs;
    int[] works = new int[20];
    int n, k;
    int ans = 0x3f3f3f3f;
    Random random = new Random(20210508);
    // 最高温/最低温/变化速率（以什么速度进行退火，系数越低退火越快，迭代次数越少，落入「局部最优」（WA）的
    double hi = 1e4, lo = 1e-4, fa = 0.90;
    // 迭代次数，与变化速率同理
    int N = 400;

    // 计算当前 jobs 序列对应的最小「最大工作时间 」是多少
    int calc() {
        Arrays.fill(works, 0);
        for (int i = 0; i < n; i++) {
            // [固定模式分配逻辑] : 每次都找最小的 worker 去分配
            int idx = 0, cur = works[idx];
            for (int j = 0; j < k; j++) {
                if (works[j] < cur) {
                    cur = works[j];
                    idx = j;
                }
            }
            works[idx] += jobs[i];
        }
        int cur = 0;
        for (int i = 0; i < k; i++) cur = Math.max(cur, works[i]);
        ans = Math.min(ans, cur);
        return cur;
    }
    void swap(int[] arr, int i, int j) {
        int c = arr[i];
        arr[i] = arr[j];
        arr[j] = c;
    }
    void sa() {
        for (double t = hi; t > lo; t *= fa) {
            int a = random.nextInt(n), b = random.nextInt(n);
            int prev = calc(); // 退火前
            swap(jobs, a, b);
            int cur = calc(); // 退火后
            int diff = prev - cur;
            // 退火为负收益（温度上升），以一定概率回退现场
            if (Math.log(diff / t) < random.nextDouble()) {
                swap(jobs, a, b);
            }
```

```
        }
    }
    public int minimumTimeRequired(int[] _jobs, int _k) {
        jobs = _jobs;
        n = jobs.length;
        k = _k;
        while (N-- > 0) sa();
        return ans;
    }
}
```

我猜你问

Q0. 模拟退火有何风险？

随机算法，会面临 `WA` 和 `TLE` 风险。

Q1. 模拟退火中的参数如何敲定的？

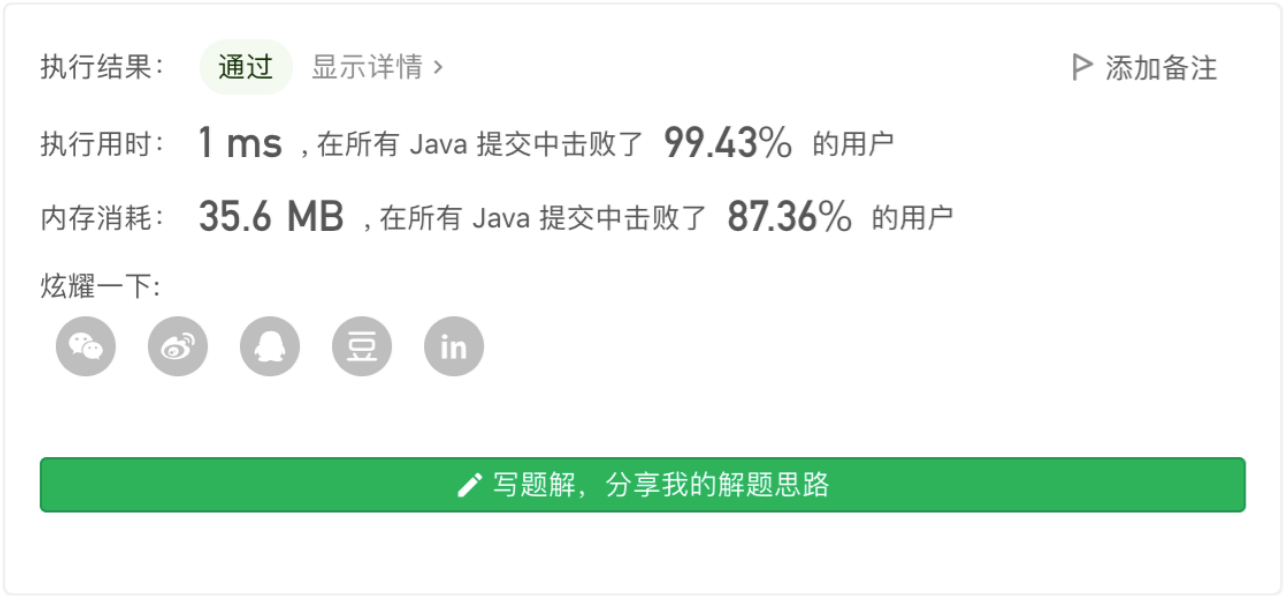
根据经验猜的，然后提交。根据结果是 `WA` 还是 `TLE` 来决定之后的调参方向。如果是 `WA` 说明部分数据落到了「局部最优」或者尚未达到「全局最优」。

Q2. 参数如何调整？

如果是 `WA` 了，一般我是优先调大 fa 参数，使降温变慢，来变相增加迭代次数；如果是 `TLE` 了，一般是优先调小 fa 参数，使降温变快，减小迭代次数。总迭代参数 `N` 也是同理。

可以简单理解调大 fa 代表将「大步」改为「baby step」，防止越过全局最优，同时增加总执行步数。

可以结合我不同的 fa 参数的提交结果来感受下：



Q3. 关于「模拟退火」正确性？

随机种子不变，测试数据不变，迭代参数不变，那么退火的过程就是恒定的，必然都能找到这些测试样例的「全局最优」。

Q4. 需要掌握「模拟退火」吗？

还是那句话，特别特别特别有兴趣的可以去了解一下。

但绝对是在你已经彻底理解「剪枝 DFS」和我没写的「状态压缩 DP」之后再去了解。

Q5. 在「剪枝 DFS」中为什么「优先分配空闲工人」的做法是对的？

首先要明确，递归树还是那棵递归树。

所谓的「优先分配空闲工人」它并不是「贪心模拟」思路，而只是一个「调整搜索顺序」的做法。

「优先分配空闲工人」不代表不会将任务分配给有工作的工人，仅仅代表我们先去搜索那些「优先分配空闲工人」的方案。

然后将得到的「合法解」配合 `max >= ans` 去剪枝掉那些「必然不是最优解」的方案。

本质上，我们并没有主动的否决某些方案（也就是我们并没有改动递归树），我们只是调整了搜索顺序来剪枝掉了一些「必然不是最优」的搜索路径。

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1766. 互质树** ，难度为 **困难**。

Tag：「DFS」

给你一个 n 个节点的树（也就是一个无环连通无向图），节点编号从 0 到 n - 1 ，且恰好有 n - 1 条边，每个节点有一个值。树的 根节点 为 0 号点。

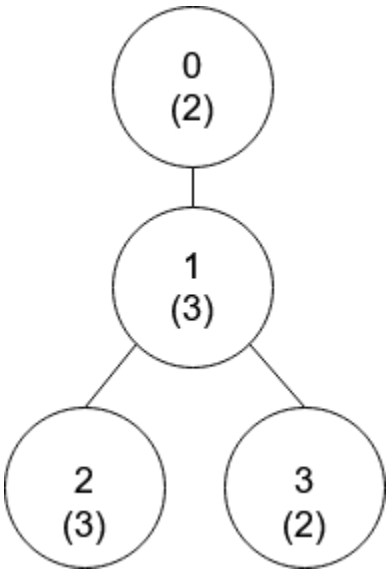
给你一个整数数组 nums 和一个二维数组 edges 来表示这棵树。nums[i] 表示第 i 个点的值，edges[j] = [uj, vj] 表示节点 uj 和节点 vj 在树中有一条边。

当 gcd(x, y) == 1 ，我们称两个数 x 和 y 是 互质的 ，其中 gcd(x, y) 是 x 和 y 的 最大公约数 。

从节点 i 到 根 最短路径上的点都是节点 i 的祖先节点。一个节点 不是 它自己的祖先节点。

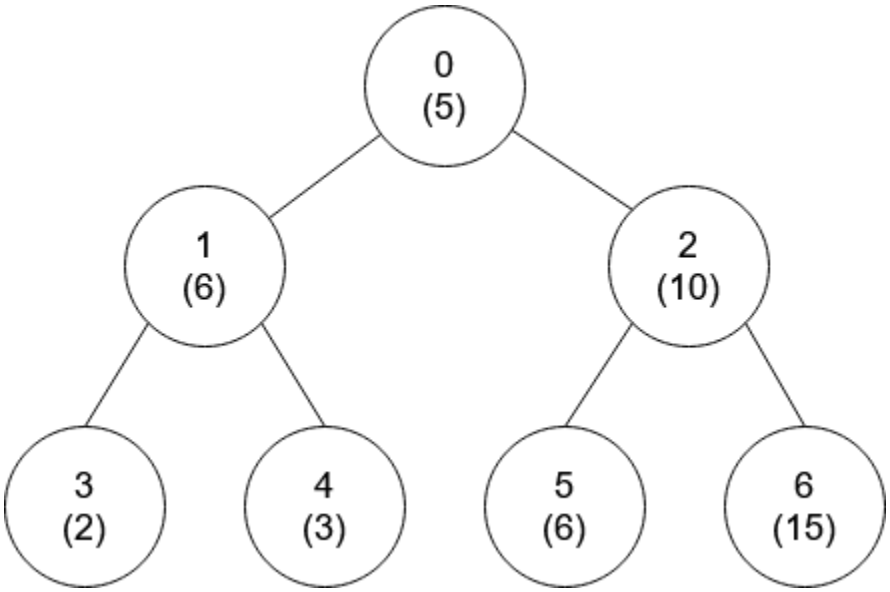
请你返回一个大小为 n 的数组 ans ，其中 ans[i]是离节点 i 最近的祖先节点且满足 nums[i] 和 nums[ans[i]] 是 互质的 ，如果不存在这样的祖先节点，ans[i] 为 -1 。

示例 1：



输入：nums = [2,3,3,2], edges = [[0,1],[1,2],[1,3]]
输出：[-1,0,0,1]
解释：上图中，每个节点的值在括号中表示。
- 节点 0 没有互质祖先。
- 节点 1 只有一个祖先节点 0 。它们的值是互质的（gcd(2,3) == 1）。
- 节点 2 有两个祖先节点，分别是节点 1 和节点 0 。节点 1 的值与它的值不是互质的（gcd(3,3) == 3）但节点 0 的值是互质的(gcd
- 节点 3 有两个祖先节点，分别是节点 1 和节点 0 。它与节点 1 互质（gcd(3,2) == 1），所以节点 1 是离它最近的符合要求的祖先节

示例 2：



输入：nums = [5,6,10,2,3,6,15], edges = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6]]
输出：[-1,0,-1,0,0,0,-1]

提示：

- nums.length == n
- 1 <= nums[i] <= 50
- 1 <= n <= 10^5
- edges.length == n - 1
- edges[j].length == 2
- 0 <= uj, vj < n
- uj != vj

基本思路

题目描述很长，但其实就是说每个节点从下往上找，找到最近的「与其互质」的节点。

数据范围是 10^5，如果每个节点都直接往上找最近「互质」祖宗节点的话，当树为线性时，复杂度是 $O(n^2)$ ，会超时。

因此我们要利用 nums[i] 范围只有 50 的特性。

我们可以先预处理除 [1, 50] 范围内的每个数，求出他们互质的数有哪些，存到一个字典里。

那么对于某个节点而言，假设节点的值为 `x` ，所在层数为 `y`。

那么问题转化为求与 `x` 互质的数有哪些，最近的在哪一层。

用 `dep[x]` 表示距离值为 `x` 的节点最近的层是多少；`pos[x]` 代表具体的节点编号。

DFS 解法

代码：

```java
class Solution {
    int[] ans;
    Map<Integer, List<Integer>> map = new HashMap<>(); // 边映射
    Map<Integer, List<Integer>> val = new HashMap<>(); // 互质数字典
    int[] dep;
    int[] pos = new int[52];
    public int[] getCoprimes(int[] nums, int[][] edges) {
        int n = nums.length;
        ans = new int[n];
        dep = new int[n];
        Arrays.fill(ans, - 1);
        Arrays.fill(pos, -1);

        for (int[] edge : edges) {
            int a = edge[0], b = edge[1];
            List<Integer> alist = map.getOrDefault(a, new ArrayList<>());
            alist.add(b);
            map.put(a, alist);
            List<Integer> blist = map.getOrDefault(b, new ArrayList<>());
            blist.add(a);
            map.put(b, blist);
        }

        for (int i = 1; i <= 50; i++) {
            for (int j = 1; j <= 50; j++) {
                if (gcd(i, j) == 1) {
                    List<Integer> list = val.getOrDefault(i, new ArrayList<>());
                    list.add(j);
                    val.put(i, list);
                }
            }
        }

        dfs(nums, 0, -1);
        return ans;
    }
    void dfs(int[] nums, int u, int form) {
        int t = nums[u];
        for (int v : val.get(t)) {
            if (pos[v] == -1) continue;
            if (ans[u] == -1 || dep[ans[u]] < dep[pos[v]]) ans[u] = pos[v];
        }
        int p = pos[t];
        pos[t] = u;

        for (int i : map.get(u)) {
```

```
            if (i == form) continue;
            dep[i] = dep[u] + 1;
            dfs(nums, i, u);
        }
        pos[t] = p;
    }
    int gcd(int a, int b) {
        if (b == 0) return a;
        if (a == 0) return b;
        return gcd(b, a % b);
    }
}
```

- 时间复杂度：对于每个节点而言，会检查与其数值互质的数有哪些，在哪层。最坏情况下会检查 50 个互质数（当前数值为 1）。复杂度为 $O(n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「DFS」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

"给作者手机充个电"

YOLO 的赞赏码