宫水三叶的刷题日记

BFS

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「BFS」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「BFS」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「BFS」；
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 **90. 子集 II** ，难度为 **中等**。

Tag：「位运算」、「回溯算法」、「状态压缩」、「DFS」

给你一个整数数组 nums ，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。

解集 不能 包含重复的子集。返回的解集中，子集可以按 任意顺序 排列。

示例 1：

```
输入：nums = [1,2,2]

输出：[[],[1],[1,2],[1,2,2],[2],[2,2]]
```

示例 2：

```
输入：nums = [0]

输出：[[],[0]]
```

提示：

- 1 <= nums.length <= 10
- -10 <= nums[i] <= 10

回溯解法（Set）

由于是求所有的方案，而且数据范围只有 10，可以直接用爆搜来做。

同时由于答案中不能包含相同的方案，因此我们可以先对原数组进行排序，从而确保所有爆搜出来的方案，都具有单调性，然后配合 Set 进行去重。

代码：

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

    /**
     * @param nums 原输入数组
     * @param u 当前决策到原输入数组中的哪一位
     * @param cur 当前方案
     * @param ans 最终结果集
     */
    void dfs(int[] nums, int u, List<Integer> cur, Set<List<Integer>> ans) {
        // 所有位置都决策完成，将当前方案放入结果集
        if (nums.length == u) {
            ans.add(new ArrayList<>(cur));
            return;
        }

        // 选择当前位置的元素，往下决策
        cur.add(nums[u]);
        dfs(nums, u + 1, cur, ans);

        // 不选当前位置的元素（回溯），往下决策
        cur.remove(cur.size() - 1);
        dfs(nums, u + 1, cur, ans);
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$，爆搜复杂度为 (2^n)，每个方案通过深拷贝存入答案，复杂度为 $O(n)$。整体复杂度为 $(n * 2^n)$
- 空间复杂度：总共有 2^n 个方案，每个方案最多占用 $O(n)$ 空间，整体复杂度为 $(n * 2^n)$

回溯解法

我们知道使用 Set 虽然是 $O(1)$ 操作，但是只是均摊 $O(1)$。

因此我们来考虑不使用 Set 的做法。

我们使用 Set 的目的是为了去重，那什么时候会导致的重复呢？

其实就是相同的元素，不同的决策方案对应同样的结果。

举个🌰，[1,1,1] 的数据，只选择第一个和只选择第三个（不同的决策方案），结果是一样的。

因此如果我们希望去重的话，不能单纯的利用「某个下标是否被选择」来进行决策，而是要找到某个数值的连续一段，根据该数值的选择次数类进行决策。

还是那个🌰，[1,1,1] 的数据，我们可以需要找到数值为 1 的连续一段，然后决策选择 0 次、选择 1 次、选择 2 次 ... 从而确保不会出现重复

也就是说，将决策方案从「某个下标是否被选择」修改为「相同的数值被选择的个数」。这样肯定不会出现重复，因为 [1,1,1] 不会因为只选择第一个和只选择第三个产生两个 [1] 的方案，只会因为 1 被选择一次，产生一个 [1] 的方案。

代码：

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

    /**
     * @param nums 原输入数组
     * @param u 当前决策到原输入数组中的哪一位
     * @param cur 当前方案
     * @param ans 最终结果集
     */
    void dfs(int[] nums, int u, List<Integer> cur, List<List<Integer>> ans) {
        // 所有位置都决策完成，将当前方案放入结果集
        int n = nums.length;
        if (n == u) {
            ans.add(new ArrayList<>(cur));
            return;
        }

        // 记录当前位置是什么数值（令数值为 t），并找出数值为 t 的连续一段
        int t = nums[u];
        int last = u;
        while (last < n && nums[last] == nums[u]) last++;

        // 不选当前位置的元素，直接跳到 last 往下决策
        dfs(nums, last, cur, ans);

        // 决策选择不同个数的 t 的情况：选择 1 个、2 个、3 个 ... k 个
        for (int i = u; i < last; i++) {
            cur.add(nums[i]);
            dfs(nums, last, cur, ans);
        }

        // 回溯对数值 t 的选择
        for (int i = u; i < last; i++) {
            cur.remove(cur.size() - 1);
        }
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$，爆搜复杂度为 (2^n)，每个方案通过深拷贝存入答案，复杂度为 $O(n)$。整体复杂度为 $(n * 2^n)$

- 空间复杂度：总共有 2^n 个方案，每个方案最多占用 $O(n)$ 空间，整体复杂度为 $(n * 2^n)$

状态压缩解法（Set）

由于长度只有 10，我们可以使用一个 int 的后 10 位来代表每位数组成员是否被选择。

同样，我们也需要先对原数组进行排序，再配合 Set 来进行去重。

代码：

```
class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);
        int n = nums.length;
        Set<List<Integer>> ans = new HashSet<>();
        List<Integer> cur = new ArrayList<>();

        // 枚举 i 代表，枚举所有的选择方案状态
        // 例如 [1,2]，我们有 []、[1]、[2]、[1,2] 几种方案，分别对应了 00、10、01、11 几种状态
        for (int i = 0; i < (1 << n); i++) {
            cur.clear();
            // 对当前状态进行诸位检查，如果当前状态为 1 代表被选择，加入当前方案中
            for (int j = 0; j < n; j++) {
                int t = (i >> j) & 1;
                if (t == 1) cur.add(nums[j]);
            }
            // 将当前方案中加入结果集
            ans.add(new ArrayList<>(cur));
        }
        return new ArrayList<>(ans);
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$，爆搜复杂度为 (2^n)，每个方案通过深拷贝存入答案，复杂度为 $O(n)$。整体复杂度为 $(n * 2^n)$
- 空间复杂度：总共有 2^n 个方案，每个方案最多占用 $O(n)$ 空间，整体复杂度为 $(n * 2^n)$

> **🍥 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 **297. 二叉树的序列化与反序列化** ，难度为 **困难**。

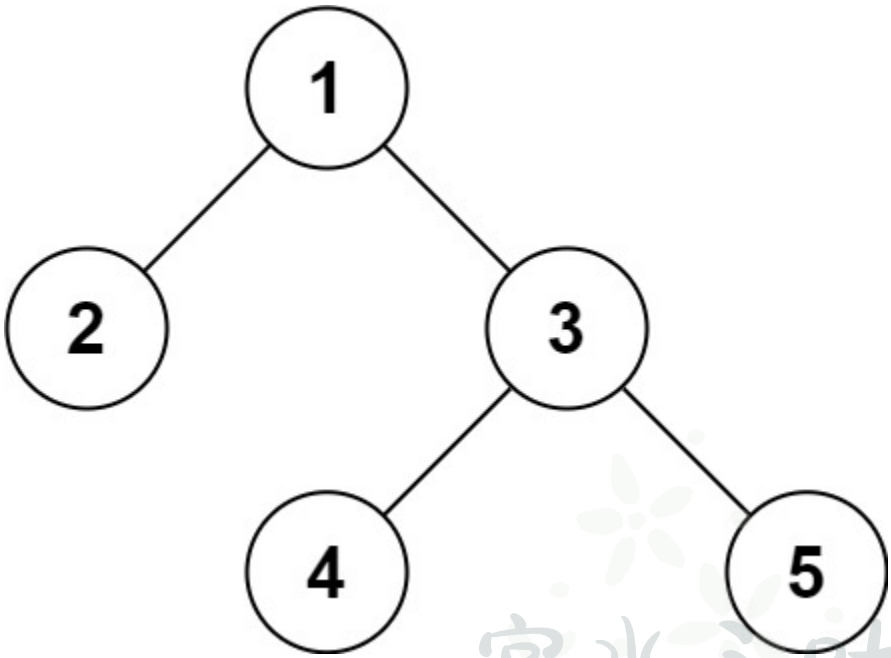
Tag：「二叉树」、「层序遍历」

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

提示: 输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

示例 1：




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

示例 2：

```
输入：root = []
输出：[]
```

示例 3：

```
输入：root = [1]
输出：[1]
```

示例 4：

```
输入：root = [1,2]
输出：[1,2]
```

提示：

- 树中结点数在范围 $[0, 10^4]$ 内
- -1000 <= Node.val <= 1000

基本思路

无论使用何种「遍历方式」进行二叉树存储，为了方便，我们都需要对空节点有所表示。

其实题目本身的样例就给我们提供了很好的思路：**使用层序遍历的方式进行存储，对于某个叶子节点的空节点进行存储，同时确保不递归存储空节点对应的子节点。**

层序遍历

根据节点值的数据范围 `-1000 <= Node.val <= 1000`（我是在 297. 二叉树的序列化与反序列化 看的，你也可以不使用数字，使用某个特殊字符进行表示，只要能在反序列时有所区分即可），我们可以建立占位节点 `emptyNode` 用来代指空节点（`emptyNode.val = INF`）。

序列化：先特判掉空树的情况，之后就是常规的层序遍历逻辑：

1. 起始时，将 `root` 节点入队；
2. 从队列中取出节点，检查节点是否有左/右节点：

- 如果有的话，将值追加序列化字符中（注意使用分隔符），并将节点入队；
- 如果没有，检查当前节点是否为 `emptyNode` 节点，如果不是 `emptyNode` 说明是常规的叶子节点，需要将其对应的空节点进行存储，即将 `emptyNode` 入队；

3. 循环流程 2，直到整个队列为空。

反序列：同理，怎么「序列化」就怎么进行「反序列」即可：

1. 起始时，构造出 `root` 并入队；
2. 每次从队列取出元素时，同时从序列化字符中截取两个值（对应左右节点），检查是否为 `INF`，若不为 `INF` 则构建对应节点；
3. 循环流程 2，直到整个序列化字符串被处理完（注意跳过最后一位分隔符）。

代码：

```Java
public class Codec {
    int INF = -2000;
    TreeNode emptyNode = new TreeNode(INF);
    public String serialize(TreeNode root) {
        if (root == null) return "";

        StringBuilder sb = new StringBuilder();
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        while (!d.isEmpty()) {
            TreeNode poll = d.pollFirst();
            sb.append(poll.val + "_");
            if (!poll.equals(emptyNode)) {
                d.addLast(poll.left != null ? poll.left : emptyNode);
                d.addLast(poll.right != null ? poll.right : emptyNode);
            }
        }
        return sb.toString();
    }

    public TreeNode deserialize(String data) {
        if (data.equals("")) return null;

        String[] ss = data.split("_");
        int n = ss.length;
        TreeNode root = new TreeNode(Integer.parseInt(ss[0]));
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        for (int i = 1; i < n - 1; i += 2) {
            TreeNode poll = d.pollFirst();
            int a = Integer.parseInt(ss[i]), b = Integer.parseInt(ss[i + 1]);
            if (a != INF) {
                poll.left = new TreeNode(a);
                d.addLast(poll.left);
            }
            if (b != INF) {
                poll.right = new TreeNode(b);
                d.addLast(poll.right);
            }
        }
        return root;
    }
}
```

- 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 **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 **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

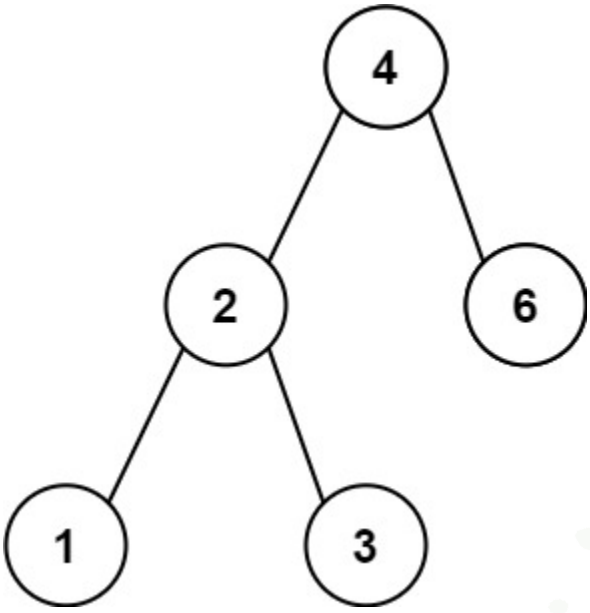
这是 LeetCode 上的 **783. 二叉搜索树节点最小距离** ，难度为 **简单**。

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

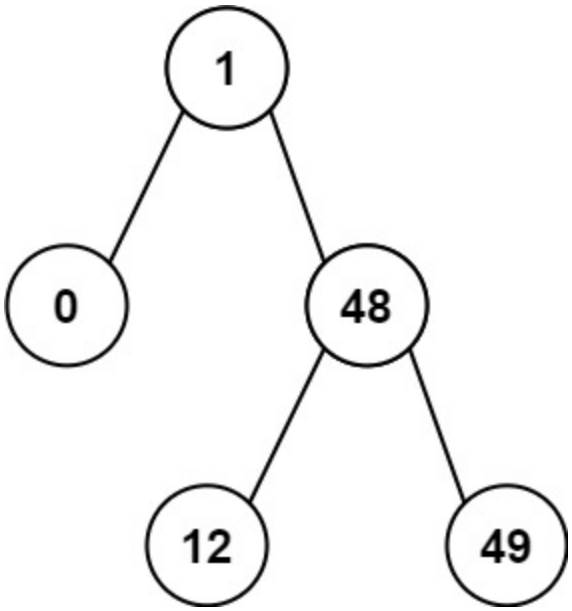
示例 1：




```
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 **

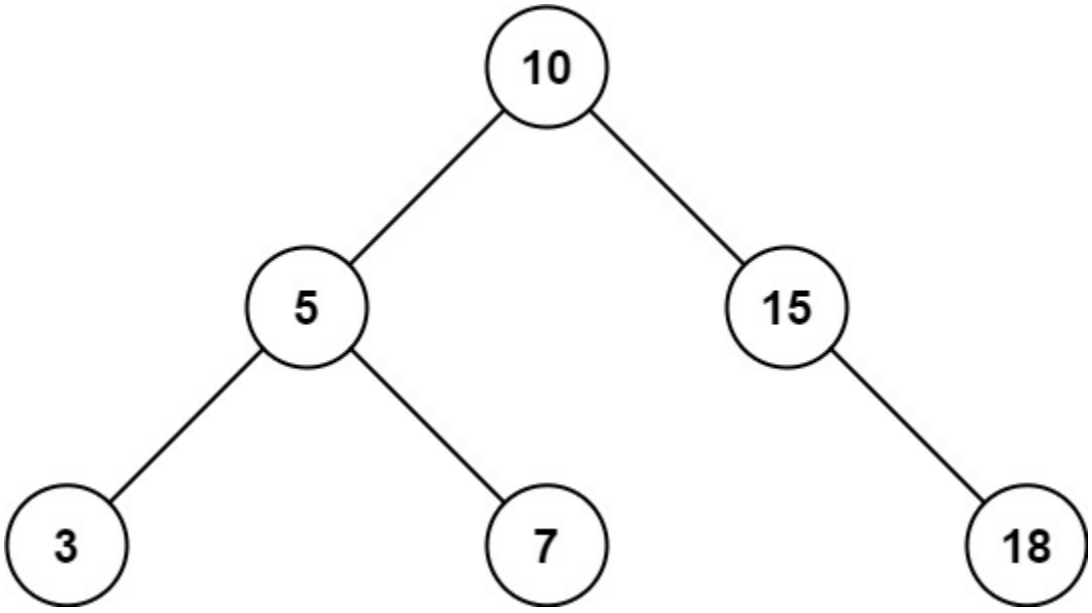
题目描述

这是 LeetCode 上的 **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。

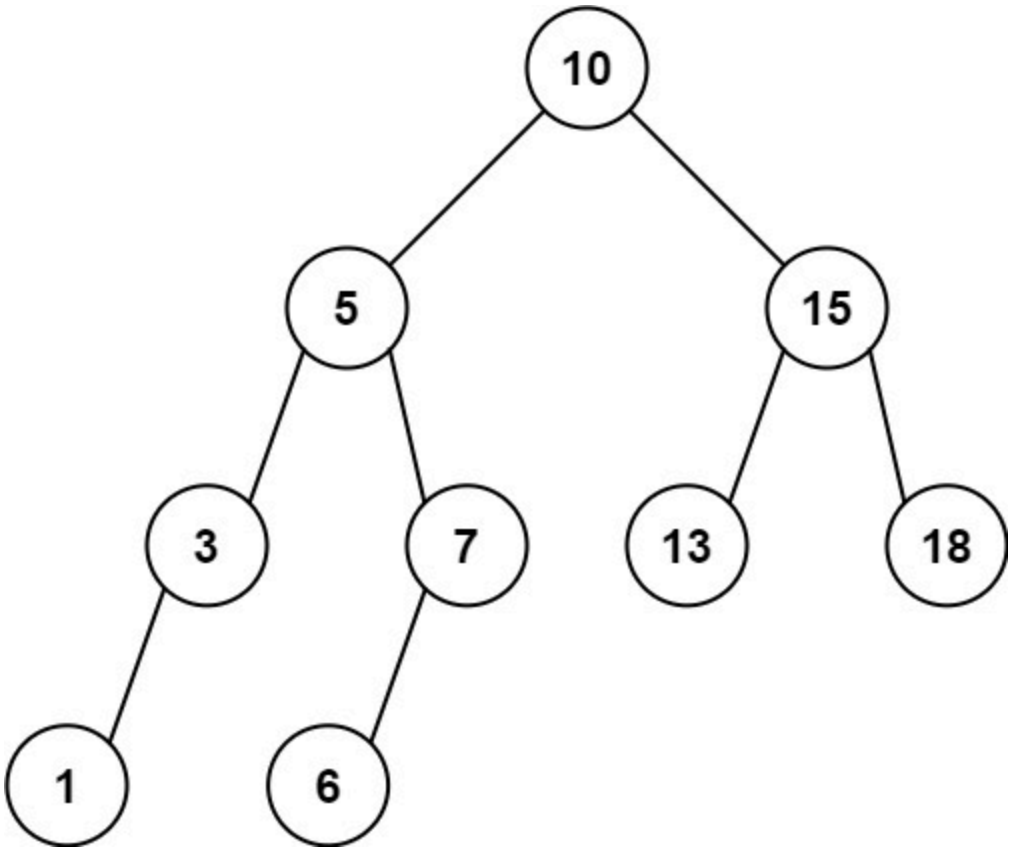
示例 1：




```
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：

```
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 **993. 二叉树的堂兄弟节点** ，难度为 **简单**。

Tag : 「树的搜索」、「BFS」、「DFS」

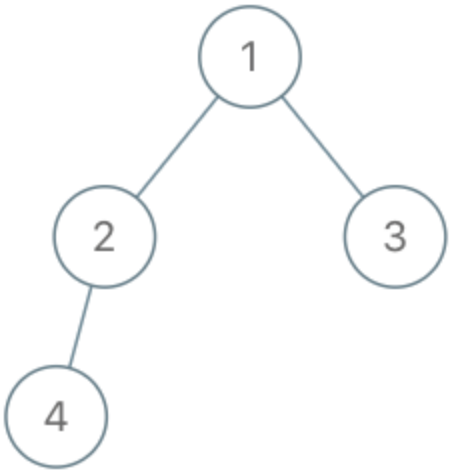
在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 k+1 处。

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。

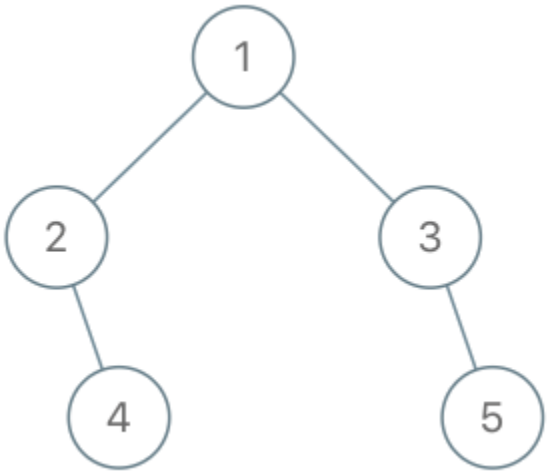
示例 1：




```
输入：root = [1,2,3,4], x = 4, y = 3

输出：false
```

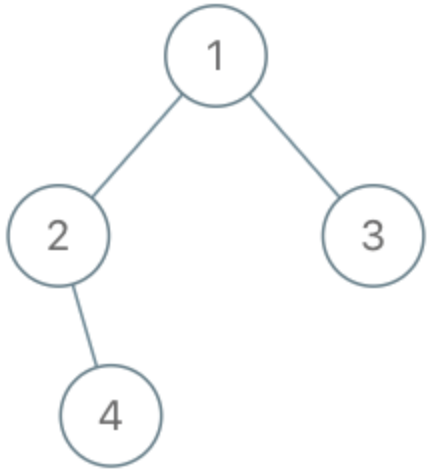
示例 2：




```
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」：

```Java
/**
 * 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：

```Java
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};
    }
}
```

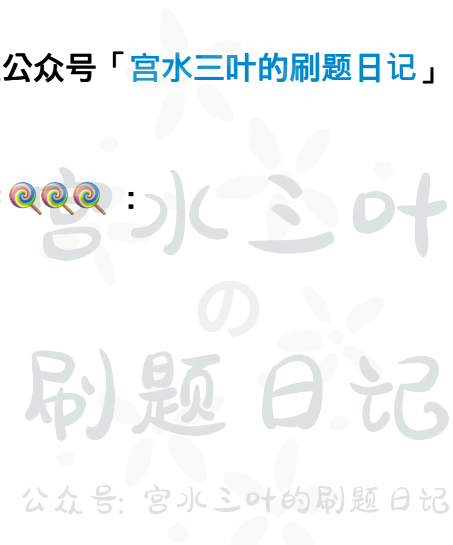
- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍭 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「BFS」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