宫水三叶的刷题日记

# 图论 DFS

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

**噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「图论 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「图论 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「图论 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 **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 **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
        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，当某一条边别应用之后，恰好使用得「起点」和「结点」联通
```

```java
        // 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 **797. 所有可能的路径** ，难度为 **中等**。

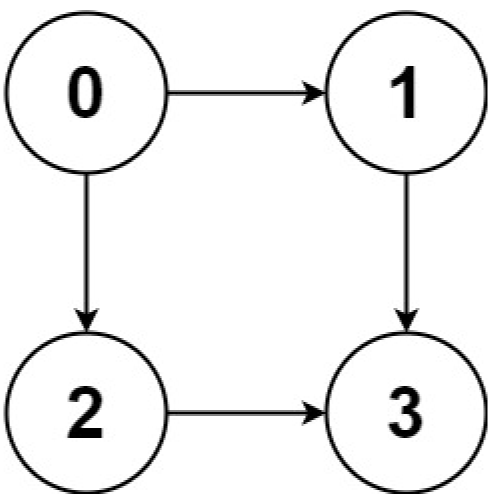
Tag : 「回溯算法」、「DFS」

给你一个有 n 个节点的 有向无环图（DAG），请你找出所有从节点 0 到节点 n-1 的路径并输出（不要求按特定顺序）

二维数组的第 i 个数组中的单元都表示有向图中 i 号节点所能到达的下一些节点，空就是没有下一个结点了。

译者注：有向图是有方向的，即规定了 a→b 你就不能从 b→a 。

示例 1：

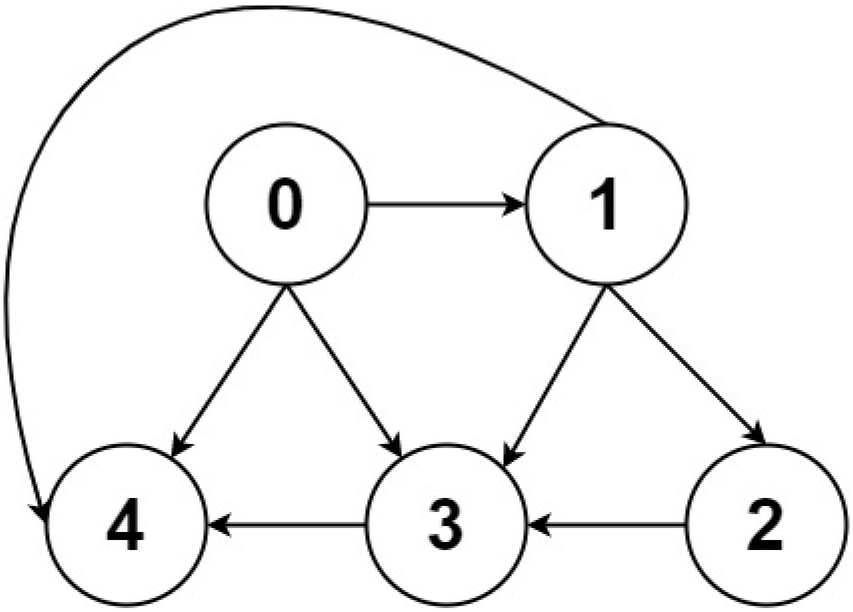



```
输入：graph = [[1,2],[3],[3],[]]

输出：[[0,1,3],[0,2,3]]

解释：有两条路径 0 -> 1 -> 3 和 0 -> 2 -> 3
```

示例 2：




```
输入：graph = [[4,3,1],[3,2,4],[3],[4],[]]

输出：[[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]
```

示例 3：

```
输入：graph = [[1],[]]

输出：[[0,1]]
```

示例 4：

```
输入：graph = [[1,2,3],[2],[3],[]]

输出：[[0,1,2,3],[0,2,3],[0,3]]
```

示例 5：

```
输入：graph = [[1,3],[2],[3],[]]

输出：[[0,1,2,3],[0,3]]
```

**提示：**

- n == graph.length
- 2 <= n <= 15
- 0 <= graph[i][j] < n
- graph[i][j] != i（即，不存在自环）
- graph[i] 中的所有元素 互不相同
- 保证输入为 有向无环图（DAG）

---

## DFS

$n$ 只有 $15$，且要求输出所有方案，因此最直观的解决方案是使用 `DFS` 进行爆搜。

起始将 $0$ 进行加入当前答案，当 $n - 1$ 被添加到当前答案时，说明找到了一条从 $0$ 到 $n - 1$ 的路径，将当前答案加入结果集。

当我们决策到第 $x$ 位（非零）时，该位置所能放入的数值由第 $x - 1$ 位已经填入的数所决定，同时由于给定的 $graph$ 为有向无环图（拓扑图），因此按照第 $x - 1$ 位置的值去决策第 $x$ 位的内容，必然不会决策到已经在当前答案的数值，否则会与 $graph$ 为有向无环图（拓扑图）的先决条件冲突。

换句话说，与一般的爆搜不同的是，我们不再需要 $vis$ 数组来记录某个点是否已经在当前答案中。

代码：

```java
class Solution {
    int[][] g;
    int n;
    List<List<Integer>> ans = new ArrayList<>();
    List<Integer> cur = new ArrayList<>();
    public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
        g = graph;
        n = g.length;
        cur.add(0);
        dfs(0);
        return ans;
    }
    void dfs(int u) {
        if (u == n - 1) {
            ans.add(new ArrayList<>(cur));
            return ;
        }
        for (int next : g[u]) {
            cur.add(next);
            dfs(next);
            cur.remove(cur.size() - 1);
        }
    }
}
```

- 时间复杂度：共有 $n$ 个节点，每个节点有选和不选两种决策，总的方案数最多为 $2^n$，对于每个方案最坏情况需要 $O(n)$ 的复杂度进行拷贝并添加到结果集。整体复杂度为 $O(n * 2^n)$
- 空间复杂度：最多有 $2^n$ 种方案，每个方案最多有 $n$ 个元素。整体复杂度为 $O(n * 2^n)$

---

> **🍭 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 **863. 二叉树中所有距离为 K 的结点** ，难度为 **中等**。

Tag : 「图论 BFS」、「图论 DFS」、「二叉树」

给定一个二叉树（具有根结点 root），一个目标结点 target，和一个整数值 K 。

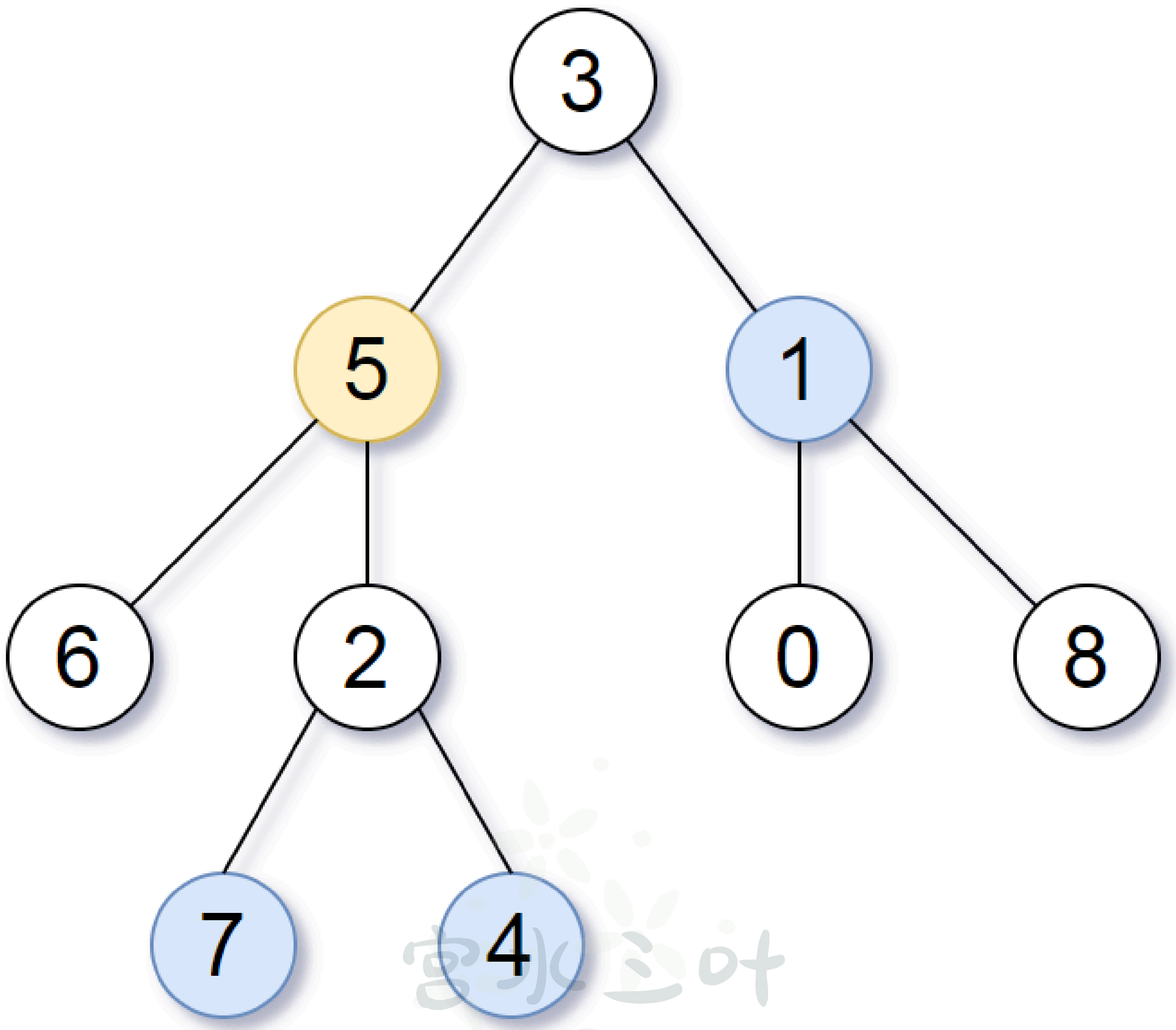
返回到目标结点 target 距离为 K 的所有结点的值的列表。 答案可以以任何顺序返回。

示例 1：

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

输出：[7,4,1]

解释：
所求结点为与目标结点（值为 5）距离为 2 的结点，
值分别为 7，4，以及 1
```




注意，输入的 “root” 和 “target” 实际上是树上的结点。
上面的输入仅仅是对这些对象进行了序列化描述。

提示：

- 给定的树是非空的。
- 树上的每个结点都具有唯一的值 0 <= node.val <= 500 。
- 目标结点 target 是树上的结点。
- 0 <= K <= 1000.

---

## 基本分析

显然，如果题目是以图的形式给出的话，我们可以很容易通过「`BFS` / 迭代加深」找到距离为 $k$ 的节点集。

而树是一类特殊的图，我们可以通过将二叉树转换为图的形式，再进行「`BFS` / 迭代加深」。

由于二叉树每个点最多有 $2$ 个子节点，点和边的数量接近，属于稀疏图，因此我们可以使用「邻接表」的形式进行存储。

建图方式为：对于二叉树中相互连通的节点（`root` 与 `root.left`、`root` 和 `root.right`），建立一条无向边。

建图需要遍历整棵树，使用 `DFS` 或者 `BFS` 均可。

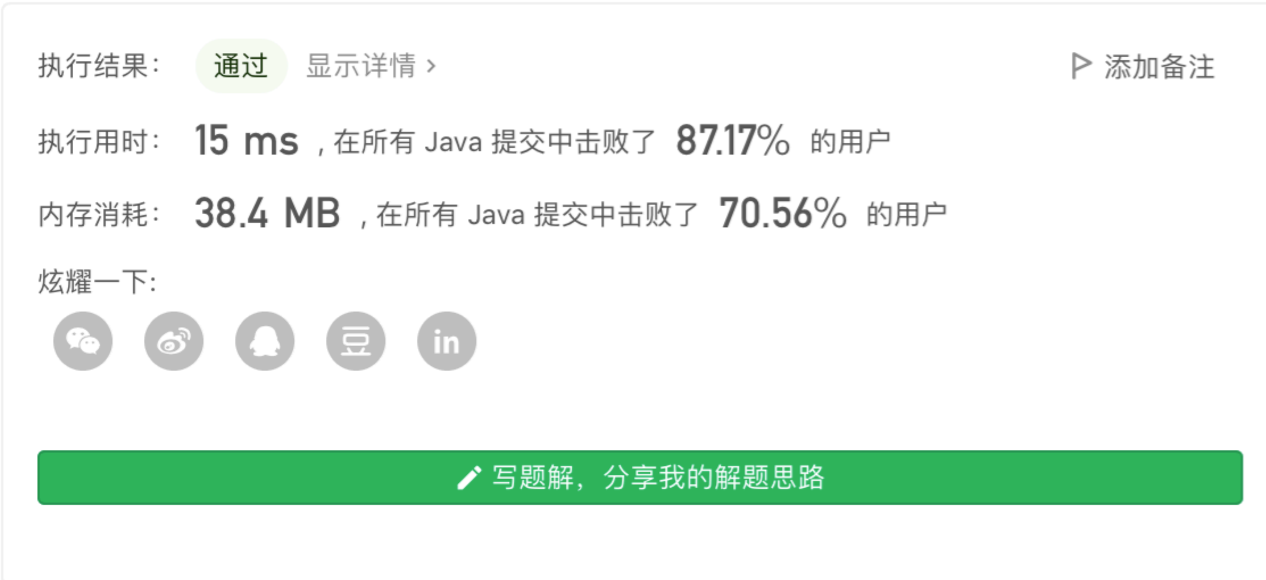
由于所有边的权重均为 $1$，我们可以使用 「`BFS` / 迭代加深」 找到从目标节点 `target` 出发，与目标节点距离为 $k$ 的节点，然后将其添加到答案中。

> 一些细节：利用每个节点具有唯一的值，我们可以直接使用节点值进行建图和搜索。

---

## 建图 + `BFS`

由「基本分析」，可写出「建图 + `BFS`」的实现。


执行结果： 通过 显示详情 ›
添加备注
执行用时： 15 ms ，在所有 Java 提交中击败了 87.17% 的用户
内存消耗： 38.4 MB ，在所有 Java 提交中击败了 70.56% 的用户
炫耀一下：
写题解，分享我的解题思路

代码：

```java
class Solution {
    int N = 1010, M = N * 2;
    int[] he = new int[N], e = new int[M], ne = new int[M];
    int idx;
    void add(int a, int b) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx++;
    }
    boolean[] vis = new boolean[N];
    public List<Integer> distanceK(TreeNode root, TreeNode t, int k) {
        List<Integer> ans = new ArrayList<>();
        Arrays.fill(he, -1);
        dfs(root);
        Deque<Integer> d = new ArrayDeque<>();
        d.addLast(t.val);
        vis[t.val] = true;
        while (!d.isEmpty() && k >= 0) {
            int size = d.size();
            while (size-- > 0) {
                int poll = d.pollFirst();
                if (k == 0) {
                    ans.add(poll);
                    continue;
                }
                for (int i = he[poll]; i != -1 ; i = ne[i]) {
                    int j = e[i];
                    if (!vis[j]) {
                        d.addLast(j);
                        vis[j] = true;
                    }
                }
            }
            k--;
        }
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        if (root.left != null) {
            add(root.val, root.left.val);
            add(root.left.val, root.val);
            dfs(root.left);
        }
        if (root.right != null) {
            add(root.val, root.right.val);
```

```
            add(root.right.val, root.val);
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：通过 `DFS` 进行建图的复杂度为 $O(n)$；通过 `BFS` 找到距离 $target$ 为 $k$ 的节点，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

---

## 建图 + 迭代加深

由「基本分析」，可写出「建图 + 迭代加深」的实现。

迭代加深的形式，我们只需要结合题意，搜索深度为 $k$ 的这一层即可。


代码：

```java
class Solution {
    int N = 1010, M = N * 2;
    int[] he = new int[N], e = new int[M], ne = new int[M];
    int idx;
    void add(int a, int b) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx++;
    }
    boolean[] vis = new boolean[N];
    public List<Integer> distanceK(TreeNode root, TreeNode t, int k) {
        List<Integer> ans = new ArrayList<>();
        Arrays.fill(he, -1);
        dfs(root);
        vis[t.val] = true;
        find(t.val, k, 0, ans);
        return ans;
    }
    void find(int root, int max, int cur, List<Integer> ans) {
        if (cur == max) {
            ans.add(root);
            return ;
        }
        for (int i = he[root]; i != -1; i = ne[i]) {
            int j = e[i];
            if (!vis[j]) {
                vis[j] = true;
                find(j, max, cur + 1, ans);
            }
        }
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        if (root.left != null) {
            add(root.val, root.left.val);
            add(root.left.val, root.val);
            dfs(root.left);
        }
        if (root.right != null) {
            add(root.val, root.right.val);
            add(root.right.val, root.val);
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：通过 `DFS` 进行建图的复杂度为 $O(n)$；通过迭代加深找到距离 $target$ 为 $k$ 的节点，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

---

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

# 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：

```java
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？

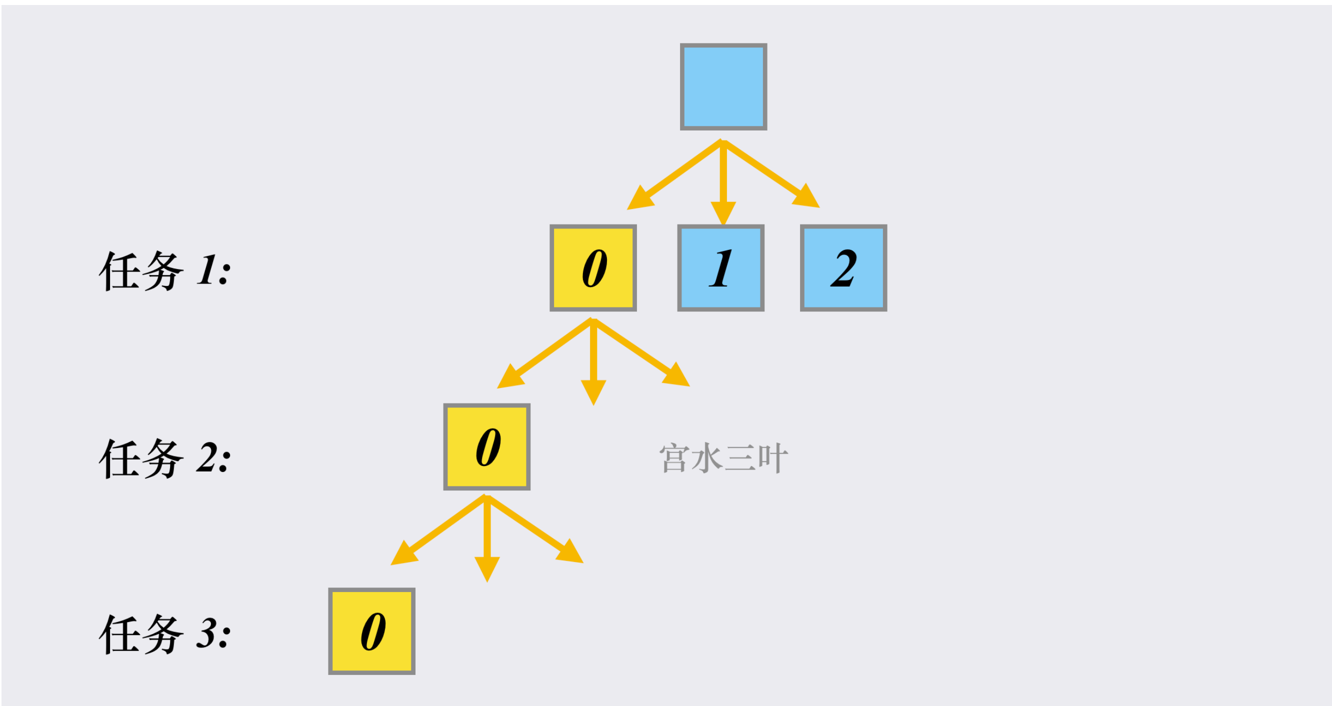
我们可以重新审视一下这道题。

**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。

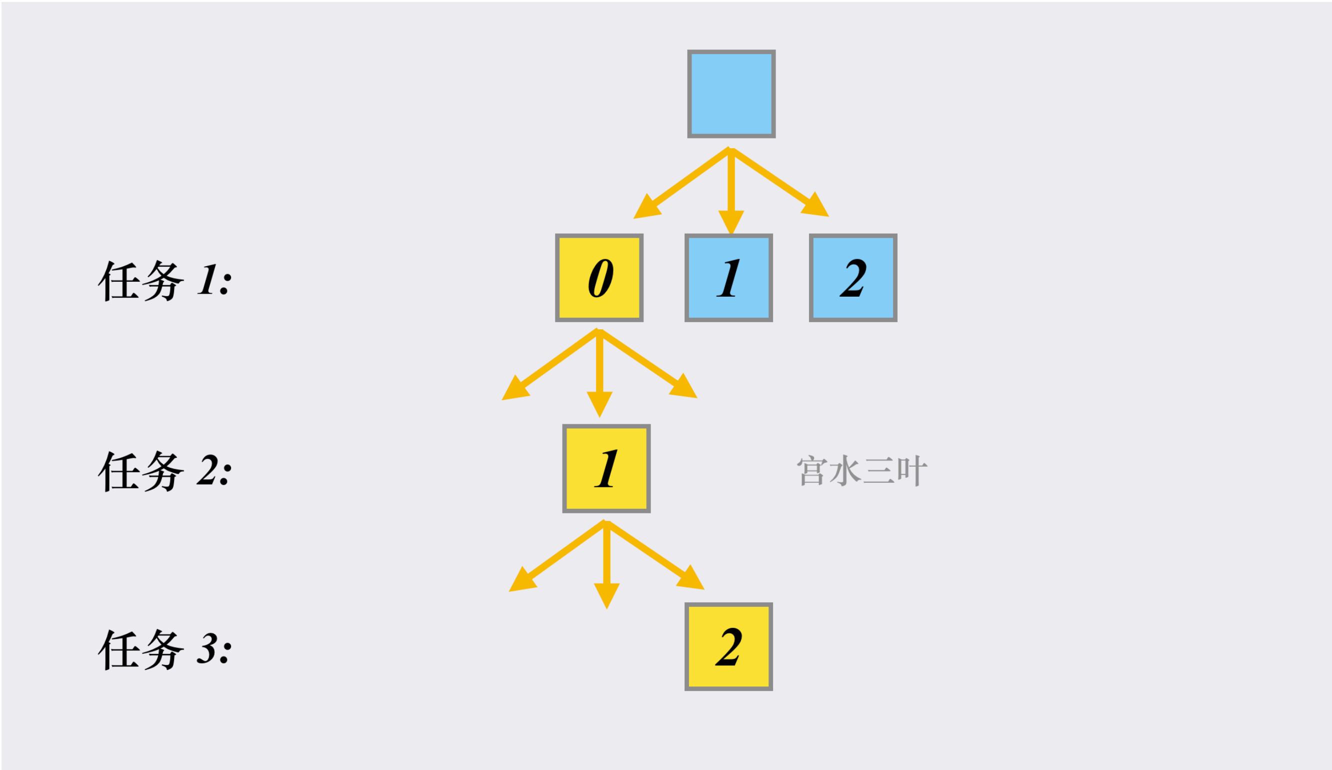
**对应的递归树其实是一颗高度为 $n$ 的 $k$ 阶树。**

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：

```java
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）
    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

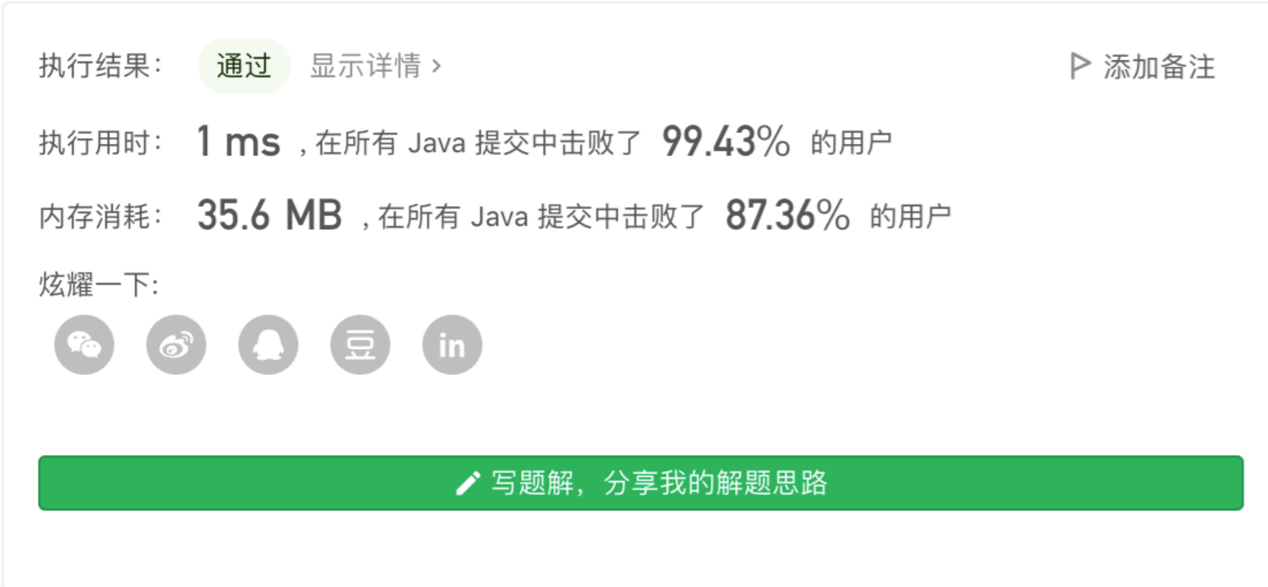
**Q0. 模拟退火有何风险？**
随机算法，会面临 `WA` 和 `TLE` 风险。

**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。

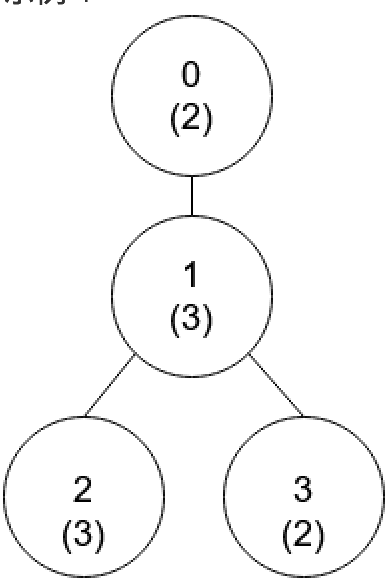
给你一个整数数组 nums 和一个二维数组 edges 来表示这棵树。nums[i] 表示第 i 个点的值，edges[j] = [uj, vj] 表示节点 uj 和节点 vj 在树中有一条边。

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。

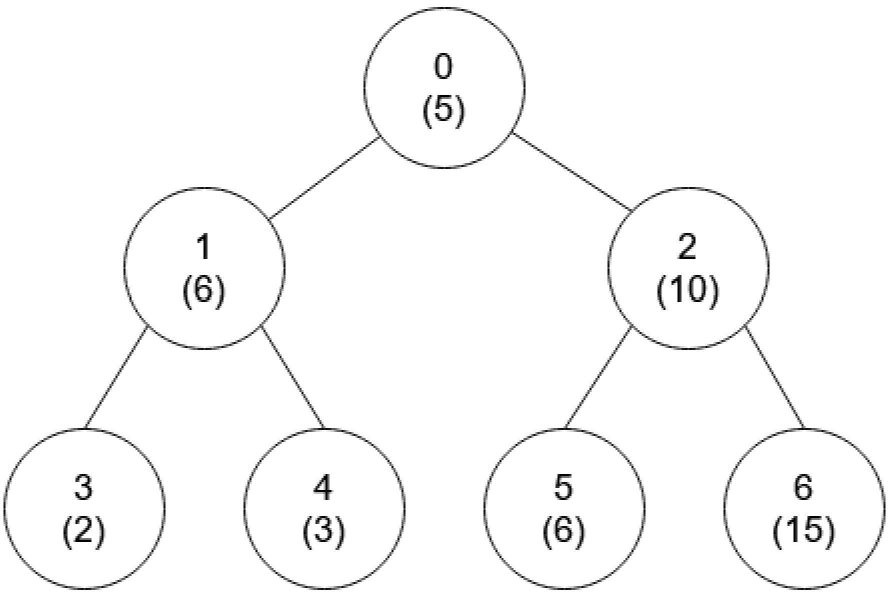
请你返回一个大小为 n 的数组 ans ，其中 ans[i]是离节点 i 最近的祖先节点且满足 nums[i] 和 nums[ans[i]] 是 互质的 ，如果不存在这样的祖先节点，ans[i] 为 -1 。

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
```

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：

```
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「图论 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