宫水三叶的刷题日记

回溯算法

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「回溯算法」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「回溯算法」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「回溯算法」；
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 **37. 解数独** ，难度为 **困难**。

Tag：「回溯算法」、「DFS」、「数独问题」

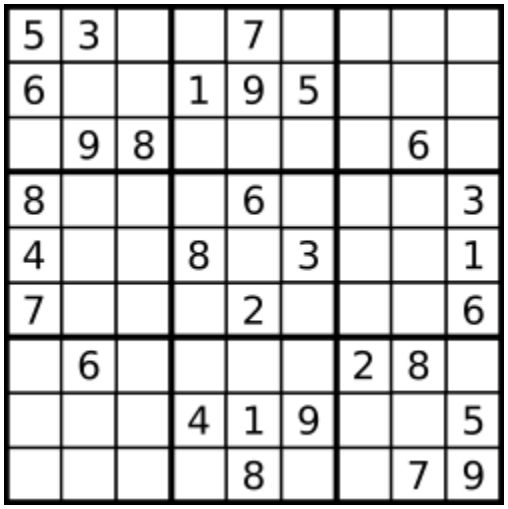
编写一个程序，通过填充空格来解决数独问题。

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：

```
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 999 = 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 **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 **131. 分割回文串** ，难度为 **中等**。

Tag：「回文串」、「回溯算法」、「动态规划」

给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是 回文串 。返回 s 所有可能的分割方案。

回文串 是正着读和反着读都一样的字符串。

示例 1：

```
输入：s = "aab"
输出：[["a","a","b"],["aa","b"]]
```

示例 2：

```
输入：s = "a"
输出：[["a"]]
```

提示：

- 1 <= s.length <= 16
- s 仅由小写英文字母组成

动态规划 + 回溯算法

求所有的分割方案，凡是求所有方案的题基本上都没有什么优化方案，就是「爆搜」。

问题在于，爆搜什么？显然我们可以爆搜每个回文串的起点。如果有连续的一段是回文串，我们再对剩下连续的一段继续爆搜。

为什么能够直接接着剩下一段继续爆搜？

因为任意的子串最终必然能够分割成若干的回文串（最坏的情况下，每个回文串都是一个字

母）。

所以我们每次往下爆搜时，只需要保证自身连续一段是回文串即可。

举个🌰来感受下我们的爆搜过程，假设有样例 `abababa` ，刚开始我们从起点第一个 a 进行爆搜：

1. 发现 `a` 是回文串，先将 `a` 分割出来，再对剩下的 `bababa` 进行爆搜
2. 发现 `aba` 是回文串，先将 `aba` 分割出来，再对剩下的 `baba` 进行爆搜
3. 发现 `ababa` 是回文串，先将 `ababa` 分割出来，再对剩下的 `ba` 进行爆搜
4. 发现 `abababa` 是回文串，先将 `abababa` 分割出来，再对剩下的 `` 进行爆搜

...

然后再对下一个起点（下个字符） `b` 进行爆搜？

不需要。

因为单个字符本身构成了回文串，所以以 `b` 为起点， `b` 之前构成回文串的方案，必然覆盖在我们以第一个字符为起点所展开的爆搜方案内（在这里就是对应了上述的第一步所展开的爆搜方案中）。

因此我们只需要以首个字符为起点，枚举以其开头所有的回文串方案，加入集合，然后对剩下的字符串部分继续爆搜。就能做到以任意字符作为回文串起点进行分割的效果了。

一定要好好理解上面那句话 ~

剩下的问题是，我们如何快速判断连续一段 `[i, j]` 是否为回文串，因为爆搜的过程每个位置都可以作为分割点，复杂度为 $O(2^n)$ 的。

因此我们不可能每次都使用双指针去线性扫描一遍 `[i, j]` 判断是否回文。

一个直观的做法是，我们先预处理除所有的 `f[i][j]` ，`f[i][j]` 代表 `[i, j]` 这一段是否为回文串。

预处理 `f[i][j]` 的过程可以用递推去做。

要想 `f[i][j] == true` ，必须满足以下两个条件：

1. `f[i + 1][j - 1] == true`
2. `s[i] == s[j]`

由于状态 `f[i][j]` 依赖于状态 `f[i + 1][j - 1]`，因此需要我们左端点 `i` 是**从大到小**进行遍历；而右端点 `j` 是**从小到大**进行遍历。

因此，我们的遍历过程可以整理为：**右端点 `j` 一直往右移动（从小到大），在 `j` 固定情况下，左端点 `i` 在 `j` 在左边开始，一直往左移动（从大到小）**

代码：

```java
class Solution {
    public List<List<String>> partition(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        // f[i][j] 代表 [i, j] 这一段是否为回文串
        boolean[][] f = new boolean[n][n];
        for (int j = 0; j < n; j++) {
            for (int i = j; i >= 0; i--) {
                // 当 [i, j] 只有一个字符时，必然是回文串
                if (i == j) {
                    f[i][j] = true;
                } else {
                    // 当 [i, j] 长度为 2 时，满足 cs[i] == cs[j] 即回文串
                    if (j - i + 1 == 2) {
                        f[i][j] = cs[i] == cs[j];

                    // 当 [i, j] 长度大于 2 时，满足 (cs[i] == cs[j] && f[i + 1][j - 1]) 即回
                    } else {
                        f[i][j] = cs[i] == cs[j] && f[i + 1][j - 1];
                    }
                }
            }
        }
        List<List<String>> ans = new ArrayList<>();
        List<String> cur = new ArrayList<>();
        dfs(s, 0, ans, cur, f);
        return ans;
    }
    /**
     * s: 要搜索的字符串
     * u: 以 s 中的那一位作为回文串分割起点
     * ans: 最终结果集
     * cur: 当前结果集
     * f: 快速判断 [i,j] 是否为回文串
     */
    void dfs(String s, int u, List<List<String>> ans, List<String> cur, boolean[][] f) {
        int n = s.length();
        if (u == n) ans.add(new ArrayList<>(cur));
        for (int i = u; i < n; i++) {
            if (f[u][i]) {
                cur.add(s.substring(u, i + 1));
                dfs(s, i + 1, ans, cur, f);
                cur.remove(cur.size() - 1);
            }
        }
    }
```

```
}
```

- 时间复杂度：动态规划预处理的复杂度为 $O(n^2)$；爆搜过程中每个字符都可以作为分割点，并且有分割与不分割两种选择，方案数量为 2^{n-1}，每个字符都需要往后检查剩余字符的分割情况，复杂度为 $O(n)$。整体复杂度为 $O(n * 2^n)$
- 空间复杂度：动态规划部分的复杂度为 $O(n^2)$；方案数量最多为 2^{n-1}，每个方案都是完整字符串 `s` 的分割，复杂度为 $O(n)$，整体复杂度为 $O(n * 2^n)$

总结

对于此类要枚举所有方案的题目，我们都应该先想到「回溯算法」。

「回溯算法」从算法定义上来说，不一定要用 DFS 实现，但通常结合 DFS 来做，难度是最低的。

「回溯算法」根据当前决策有多少种选择，对应了两套模板。

每一次独立的决策只对应 选择 和 不选 两种情况：

1. 确定结束回溯过程的 base case
2. 遍历每个位置，对每个位置进行决策（做选择 -> 递归 -> 撤销选择）

```
void dfs(当前位置, 路径(当前结果), 结果集) {
    if (当前位置 == 结束位置) {
        结果集.add(路径);
        return;
    }

    选择当前位置;
    dfs(下一位置, 路径(当前结果), 结果集);
    撤销选择当前位置;
    dfs(下一位置, 路径(当前结果), 结果集);
}
```

每一次独立的决策都对应了多种选择（通常对应了每次决策能选择什么，或者每次决策能选择多少个 ...）：

1. 确定结束回溯过程的 base case

2. 遍历所有的「选择」
3. 对选择进行决策 (做选择 -> 递归 -> 撤销选择)

```
void dfs(选择列表, 路径(当前结果), 结果集) {
    if (满足结束条件) {
        结果集.add(路径);
        return;
    }

    for (选择 in 选择列表) {
        做选择;
        dfs(路径', 选择列表, 结果集);
        撤销选择;
    }
}
```

拓展

刚好最近在更新「回溯算法」的相关题解，以下题目可以加深你对「回溯」算法的理解和模板的运用：

17. 电话号码的字母组合(中等) : 从一道「回溯算法」经典题与你分享回溯算法的基本套路

39. 组合总和(中等) : DFS + 回溯算法，以及如何确定一道题是否应该使用 DFS + 回溯来求解

40. 组合总和 II(中等) : 【回溯算法】求目标和的组合方案（升级篇）

216. 组合总和 III(中等) : 【回溯算法】借助最后一道「组合总和」问题来总结一下回溯算法

37. 解数独(困难) : 【数独问题】经典面试题：解数独

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

题目描述

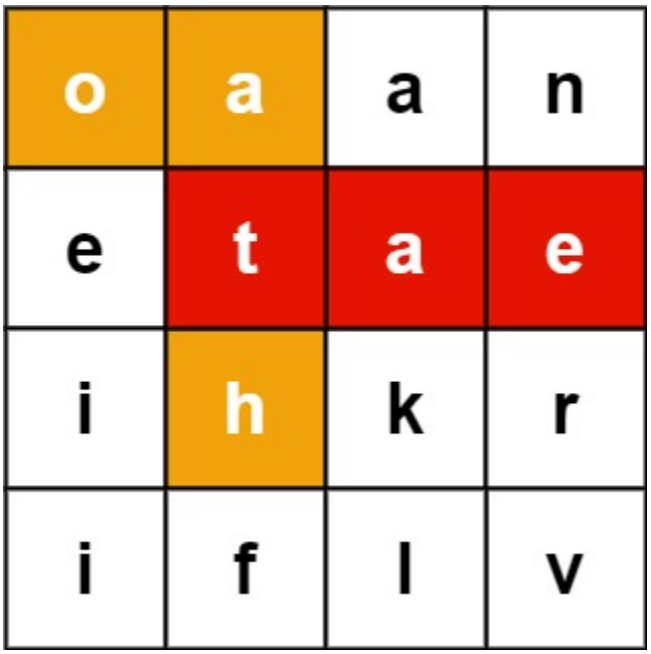
这是 LeetCode 上的 **212. 单词搜索 II** ，难度为 **困难**。

Tag：「回溯算法」、「DFS」、「字典树」

给定一个 m x n 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。

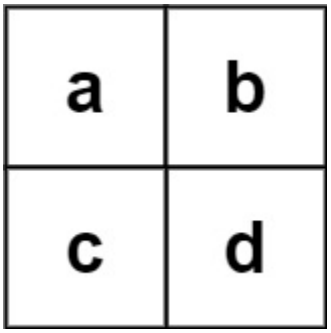
单词必须按照字母顺序，通过 相邻的单元格 内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

示例 1：




```
输入：board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","p
输出：["eat","oath"]
```

示例 2：



```
输入：board = [["a","b"],["c","d"]], words = ["abcb"]

输出：[]
```

提示：

- m == board.length
- n == board[i].length
- 1 <= m, n <= 12
- board[i][j] 是一个小写英文字母
- 1 <= words.length <= 3 * 10^4
- 1 <= words[i].length <= 10
- words[i] 由小写英文字母组成
- words 中的所有字符串互不相同

回溯算法

数据范围只有 12，且 `words` 中出现的单词长度不会超过 10，可以考虑使用「回溯算法」。

起始先将所有 `words` 出现的单词放到 `Set` 结构中，然后以 `board` 中的每个点作为起点进行爆搜（由于题目规定在一个单词中每个格子只能被使用一次，因此还需要一个 `vis` 数组来记录访问过的位置）：

1. 如果当前爆搜到的字符串长度超过 10，直接剪枝；
2. 如果当前搜索到的字符串在 `Set` 中，则添加到答案（同时了防止下一次再搜索到该字符串，需要将该字符串从 `Set` 中移除）。

代码：

```Java
class Solution {
    Set<String> set = new HashSet<>();
    List<String> ans = new ArrayList<>();
    char[][] board;
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    int n, m;
    boolean[][] vis = new boolean[15][15];
    public List<String> findWords(char[][] _board, String[] words) {
        board = _board;
        m = board.length; n = board[0].length;
        for (String w : words) set.add(w);
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                vis[i][j] = true;
                sb.append(board[i][j]);
                dfs(i, j, sb);
                vis[i][j] = false;
                sb.deleteCharAt(sb.length() - 1);
            }
        }
        return ans;
    }
    void dfs(int i, int j, StringBuilder sb) {
        if (sb.length() > 10) return ;
        if (set.contains(sb.toString())) {
            ans.add(sb.toString());
            set.remove(sb.toString());
        }
        for (int[] d : dirs) {
            int dx = i + d[0], dy = j + d[1];
            if (dx < 0 || dx >= m || dy < 0 || dy >= n) continue;
            if (vis[dx][dy]) continue;
            vis[dx][dy] = true;
            sb.append(board[dx][dy]);
            dfs(dx, dy, sb);
            vis[dx][dy] = false;
            sb.deleteCharAt(sb.length() - 1);
        }
    }
}
```

- 时间复杂度：共有 $m * n$ 个起点，每次能往 4 个方向搜索（不考虑重复搜索问题），且搜索的长度不会超过 10。整体复杂度为 $O(m * n * 4^{10})$
- 空间复杂度：$O(\sum_{i=0}^{words.length-1} words[i].length)$

Trie

在「解法一」中，对于任意一个当前位置 (i, j)，我们都不可避免的搜索了四联通的全部方向，这导致了那些无效搜索路径最终只有长度达到 10 才会被剪枝。

要进一步优化我们的搜索过程，需要考虑如何在每一步的搜索中进行剪枝。

我们可以使用 $Trie$ 结构进行建树，对于任意一个当前位置 (i, j) 而言，只有在 $Trie$ 中存在往从字符 a 到 b 的边时，我们才在棋盘上搜索从 a 到 b 的相邻路径。

不了解 $Trie$ 的同学，可以看看这篇题解 (题解) 208. 实现 Trie (前缀树)，里面写了两种实现 $Trie$ 的方式。

对于本题，我们可以使用「TrieNode」的方式进行建 $Trie$。

因为 `words` 里最多有 10^4 个单词，每个单词长度最多为 10，如果开成静态数组的话，不考虑共用行的问题，我们需要开一个大小为 $10^5 * 26$ 的大数组，可能会有 TLE 或 MLE 的风险。

与此同时，我们需要将平时建 $TrieNode$ 中的 `isEnd` 标记属性直接换成记录当前字符 `s`，这样我们在 `DFS` 的过程中则无须额外记录当前搜索字符串。

代码：

```java
class Solution {
    class TrieNode {
        String s;
        TrieNode[] tns = new TrieNode[26];
    }
    void insert(String s) {
        TrieNode p = root;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (p.tns[u] == null) p.tns[u] = new TrieNode();
            p = p.tns[u];
        }
        p.s = s;
    }
    Set<String> set = new HashSet<>();
    char[][] board;
    int n, m;
    TrieNode root = new TrieNode();
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    boolean[][] vis = new boolean[15][15];
    public List<String> findWords(char[][] _board, String[] words) {
        board = _board;
        m = board.length; n = board[0].length;
        for (String w : words) insert(w);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                int u = board[i][j] - 'a';
                if (root.tns[u] != null) {
                    vis[i][j] = true;
                    dfs(i, j, root.tns[u]);
                    vis[i][j] = false;
                }
            }
        }
        List<String> ans = new ArrayList<>();
        for (String s : set) ans.add(s);
        return ans;
    }
    void dfs(int i, int j, TrieNode node) {
        if (node.s != null) set.add(node.s);
        for (int[] d : dirs) {
            int dx = i + d[0], dy = j + d[1];
            if (dx < 0 || dx >= m || dy < 0 || dy >= n) continue;
            if (vis[dx][dy]) continue;
            int u = board[dx][dy] - 'a';
            if (node.tns[u] != null) {
```

```
                vis[dx][dy] = true;
                dfs(dx, dy, node.tns[u]);
                vis[dx][dy] = false;
            }
        }
    }
}
```

- 时间复杂度：共有 $m * n$ 个起点，每次能往 4 个方向搜索（不考虑重复搜索问题），且搜索的长度不会超过 10。整体复杂度为 $O(m * n * 4^{10})$
- 空间复杂度：$O(\sum_{i=0}^{words.length-1} words[i].length * C)$，$C$ 为字符集大小，固定为 26

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 **797. 所有可能的路径** ，难度为 **中等**。

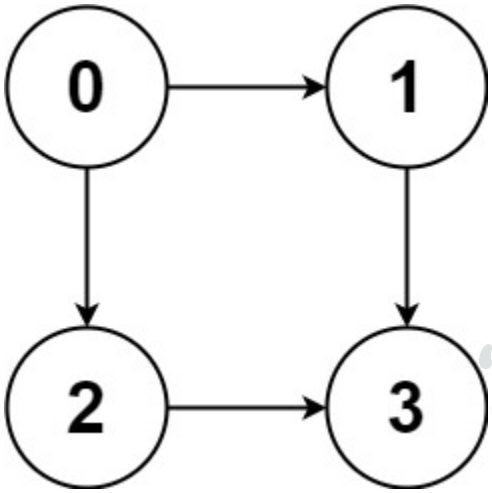
Tag : 「回溯算法」、「DFS」

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：

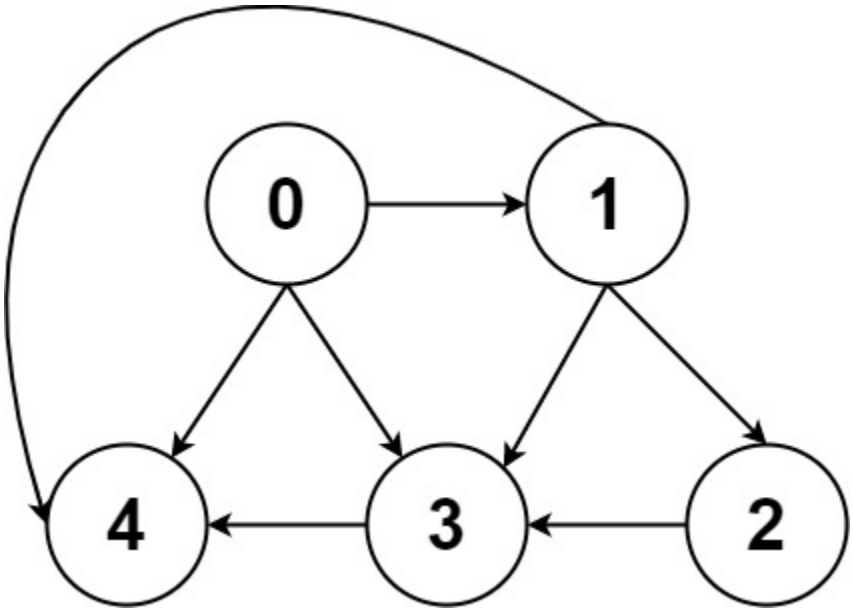


输入：graph = [[1,2],[3],[3],[]]

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 $O(n * 2^n)$

"给作者手机充个电"

YOLO 的赞赏码