宫水三叶的刷题日记

启发式搜索

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「启发式搜索」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「启发式搜索」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「启发式搜索」；
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 **127. 单词接龙** ，难度为 **困难**。

Tag : 「双向 BFS」

字典 wordList 中从单词 beginWord 和 endWord 的 转换序列 是一个按下述规格形成的序列：

- 序列中第一个单词是 beginWord 。
- 序列中最后一个单词是 endWord 。
- 每次转换只能改变一个字母。
- 转换过程中的中间单词必须是字典 wordList 中的单词。

给你两个单词 beginWord 和 endWord 和一个字典 wordList ，找到从 beginWord 到 endWord 的 最短转换序列 中的 单词数目 。如果不存在这样的转换序列，返回 0。

示例 1：

```
输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
输出：5
解释：一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog", 返回它的长度 5。
```

示例 2：

```
输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
输出：0
解释：endWord "cog" 不在字典中，所以无法进行转换。
```

提示：

- 1 <= beginWord.length <= 10
- endWord.length == beginWord.length
- 1 <= wordList.length <= 5000
- wordList[i].length == beginWord.length
- beginWord、endWord 和 wordList[i] 由小写英文字母组成
- beginWord != endWord
- wordList 中的所有字符串 互不相同

基本分析

根据题意，每次只能替换一个字符，且每次产生的新单词必须在 `wordList` 出现过。

一个朴素的实现方法是，使用 BFS 的方式求解。

从 `beginWord` 出发，枚举所有替换一个字符的方案，如果方案存在于 `wordList` 中，则加入队列中，这样队列中就存在所有替换次数为 1 的单词。然后从队列中取出元素，继续这个过程，直到遇到 `endWord` 或者队列为空为止。

同时为了「防止重复枚举到某个中间结果」和「记录每个中间结果是经过多少次转换而来」，我们需要建立一个「哈希表」进行记录。

哈希表的 KV 形式为 `{ 单词 : 由多少次转换得到 }`。

当枚举到新单词 `str` 时，需要先检查是否已经存在与「哈希表」中，如果不存在则更新「哈希表」并将新单词放入队列中。

这样的做法可以确保「枚举到所有由 `beginWord` 到 `endWord` 的转换路径」，并且由 `beginWord` 到 `endWord` 的「最短转换路径」必然会最先被枚举到。

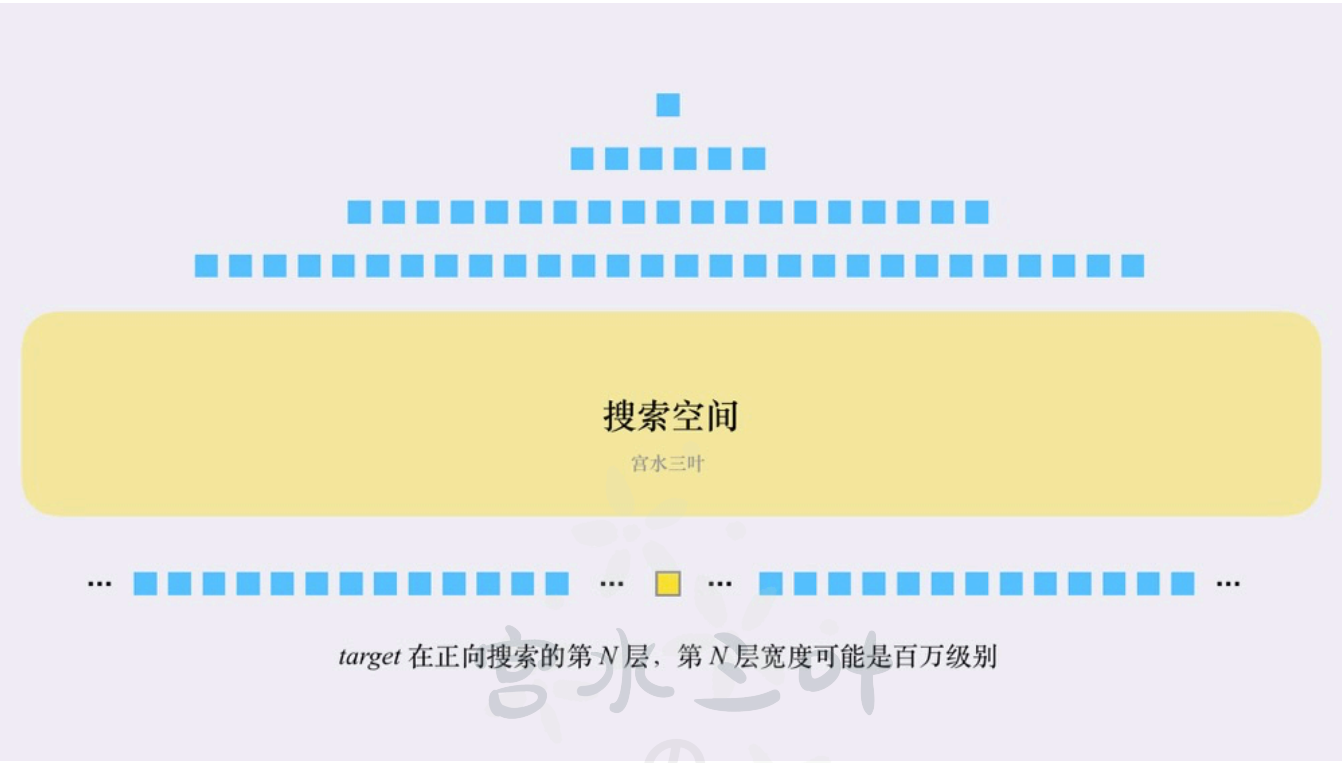
双向 BFS

经过分析，BFS 确实可以做，但本题的数据范围较大：`1 <= beginWord.length <= 10`

朴素的 BFS 可能会带来「搜索空间爆炸」的情况。

想象一下，如果我们的 `wordList` 足够丰富（包含了所有单词），对于一个长度为 10 的 `beginWord` 替换一次字符可以产生 $10 * 25$ 个新单词（每个替换点可以替换另外 25 个小写字母），第一层就会产生 250 个单词；第二层会产生超过 $6 * 10^4$ 个新单词 ...

随着层数的加深，这个数字的增速越快，这就是「搜索空间爆炸」问题。

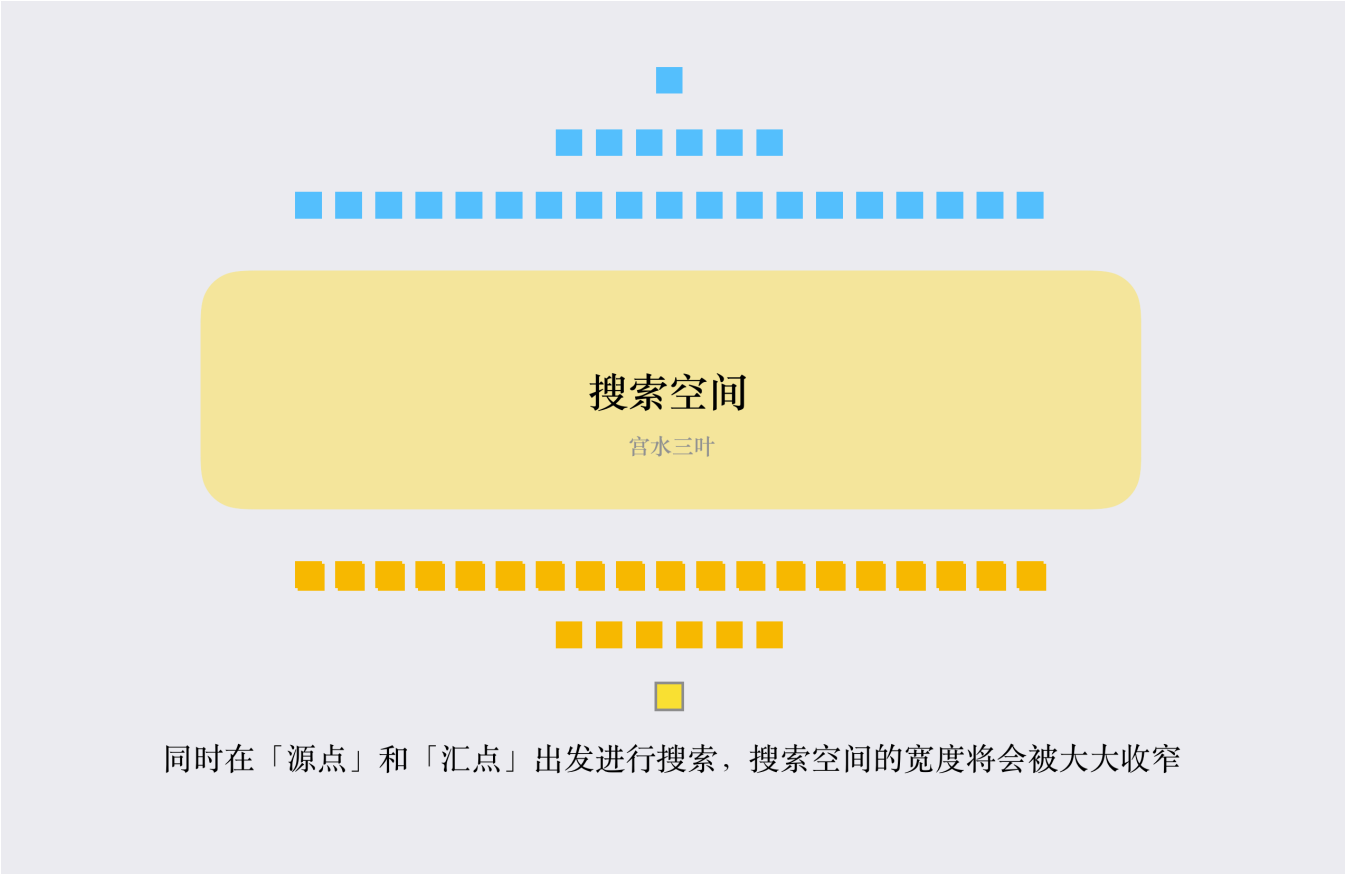


在朴素的 BFS 实现中，空间的瓶颈主要取决于搜索空间中的最大宽度。

那么有没有办法让我们不使用这么宽的搜索空间，同时又能保证搜索到目标结果呢？

「双向 BFS」可以很好的解决这个问题：

同时从两个方向开始搜索，一旦搜索到相同的值，意味着找到了一条联通起点和终点的最短路径。



同时在「源点」和「汇点」出发进行搜索，搜索空间的宽度将会被大大收窄

「双向 BFS」的基本实现思路如下：

1. 创建「两个队列」分别用于两个方向的搜索；
2. 创建「两个哈希表」用于「解决相同节点重复搜索」和「记录转换次数」；
3. 为了尽可能让两个搜索方向“平均”，每次从队列中取值进行扩展时，先判断哪个队列容量较少；
4. 如果在搜索过程中「搜索到对方搜索过的节点」，说明找到了最短路径。

「双向 BFS」基本思路对应的伪代码大致如下：

```
d1、d2 为两个方向的队列
m1、m2 为两个方向的哈希表，记录每个节点距离起点的

// 只有两个队列都不空，才有必要继续往下搜索
// 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
while(!d1.isEmpty() && !d2.isEmpty()) {
    if (d1.size() < d2.size()) {
        update(d1, m1, m2);
    } else {
        update(d2, m2, m1);
    }
}

// update 为从队列 d 中取出一个元素进行「一次完整扩展」的逻辑
void update(Deque d, Map cur, Map other) {}
```

回到本题，我们看看如何使用「双向 BFS」进行求解。

估计不少同学是第一次接触「双向 BFS」，因此这次我写了大量注释。

建议大家带着对「双向 BFS」的基本理解去阅读。

代码：

```java
class Solution {
    String s, e;
    Set<String> set = new HashSet<>();
    public int ladderLength(String _s, String _e, List<String> ws) {
        s = _s;
        e = _e;
        // 将所有 word 存入 set，如果目标单词不在 set 中，说明无解
        for (String w : ws) set.add(w);
        if (!set.contains(e)) return 0;
        int ans = bfs();
        return ans == -1 ? 0 : ans + 1;
    }

    int bfs() {
        // d1 代表从起点 beginWord 开始搜索（正向）
        // d2 代表从结尾 endWord 开始搜索（反向）
        Deque<String> d1 = new ArrayDeque<>(), d2 = new ArrayDeque();

        /*
         * m1 和 m2 分别记录两个方向出现的单词是经过多少次转换而来
         * e.g.
         * m1 = {"abc":1} 代表 abc 由 beginWord 替换 1 次字符而来
         * m2 = {"xyz":3} 代表 xyz 由 endWord 替换 3 次字符而来
         */
        Map<String, Integer> m1 = new HashMap<>(), m2 = new HashMap<>();
        d1.add(s);
        m1.put(s, 0);
        d2.add(e);
        m2.put(e, 0);

        /*
         * 只有两个队列都不空，才有必要继续往下搜索
         * 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
         * e.g.
         * 例如，如果 d1 为空了，说明从 beginWord 搜索到底都搜索不到 endWord，反向搜索也没必要进行了
         */
        while (!d1.isEmpty() && !d2.isEmpty()) {
            int t = -1;
            // 为了让两个方向的搜索尽可能平均，优先拓展队列内元素少的方向
            if (d1.size() <= d2.size()) {
                t = update(d1, m1, m2);
            } else {
                t = update(d2, m2, m1);
            }
            if (t != -1) return t;
        }
```

```
        return -1;
    }

    // update 代表从 deque 中取出一个单词进行扩展，
    // cur 为当前方向的距离字典；other 为另外一个方向的距离字典
    int update(Deque<String> deque, Map<String, Integer> cur, Map<String, Integer> other)
        // 获取当前需要扩展的原字符串
        String poll = deque.pollFirst();
        int n = poll.length();

        // 枚举替换原字符串的哪个字符 i
        for (int i = 0; i < n; i++) {
            // 枚举将 i 替换成哪个小写字母
            for (int j = 0; j < 26; j++) {
                // 替换后的字符串
                String sub = poll.substring(0, i) + String.valueOf((char)('a' + j)) + poll
                if (set.contains(sub)) {
                    // 如果该字符串在「当前方向」被记录过（拓展过），跳过即可
                    if (cur.containsKey(sub)) continue;

                    // 如果该字符串在「另一方向」出现过，说明找到了联通两个方向的最短路
                    if (other.containsKey(sub)) {
                        return cur.get(poll) + 1 + other.get(sub);
                    } else {
                        // 否则加入 deque 队列
                        deque.addLast(sub);
                        cur.put(sub, cur.get(poll) + 1);
                    }
                }
            }
        }
        return -1;
    }
}
```

- 时间复杂度：令 `wordList` 长度为 n，`beginWord` 字符串长度为 m。由于所有的搜索结果必须都在 `wordList` 出现过，因此算上起点最多有 $n+1$ 节点，最坏情况下，所有节点都联通，搜索完整张图复杂度为 $O(n^2)$；从 `beginWord` 出发进行字符替换，替换时进行逐字符检查，复杂度为 $O(m)$。整体复杂度为 $O(m*n^2)$
- 空间复杂度：同等空间大小。$O(m*n^2)$

总结

这本质其实是一个「所有边权均为 `1` 」最短路问题：将 `beginWord` 和所有在 `wordList` 出现过的字符串看做是一个点。每一次转换操作看作产生边权为 `1` 的边。问题求以 `beginWord` 为源点，以 `endWord` 为汇点的最短路径。

借助这个题，我向你介绍了「双向 BFS」，「双向 BFS」可以有效解决「搜索空间爆炸」问题。

对于那些搜索节点随着层数增加呈倍数或指数增长的搜索问题，可以使用「双向 BFS」进行求解。

【补充】启发式搜索 AStar

可以直接根据本题规则来设计 A* 的「启发式函数」。

比如对于两个字符串 `a` `b` 直接使用它们不同字符的数量来充当估值距离，我觉得是合适的。

因为不同字符数量的差值可以确保不会超过真实距离（是一个理论最小替换次数）。

注意：本题数据比较弱，用 A* 过了，但通常我们需要「确保有解」，A* 的启发搜索才会发挥真正价值。而本题，除非 `endWord` 本身就不在 `wordList` 中，其余情况我们无法很好提前判断「是否有解」。这时候 A* 将不能带来「搜索空间的优化」，效果不如「双向 BFS」。

代码：

```Java
class Solution {
    class Node {
        String str;
        int val;
        Node (String _str, int _val) {
            str = _str;
            val = _val;
        }
    }
    String s, e;
    int INF = 0x3f3f3f3f;
    Set<String> set = new HashSet<>();
    public int ladderLength(String _s, String _e, List<String> ws) {
        s = _s;
        e = _e;
        for (String w : ws) set.add(w);
        if (!set.contains(e)) return 0;
        int ans = astar();
        return ans == -1 ? 0 : ans + 1;
    }
    int astar() {
        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->a.val-b.val);
        Map<String, Integer> dist = new HashMap<>();
        dist.put(s, 0);
        q.add(new Node(s, f(s)));

        while (!q.isEmpty()) {
            Node poll = q.poll();
            String str = poll.str;
            int distance = dist.get(str);
            if (str.equals(e)) {
                break;
            }
            int n = str.length();
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < 26; j++) {
                    String sub = str.substring(0, i) + String.valueOf((char)('a' + j)) + s
                    if (!set.contains(sub)) continue;
                    if (!dist.containsKey(sub) || dist.get(sub) > distance + 1) {
                        dist.put(sub, distance + 1);
                        q.add(new Node(sub, dist.get(sub) + f(sub)));
                    }
                }
            }
        }
        return dist.containsKey(e) ? dist.get(e) : -1;
```

```
    }
    int f(String str) {
        if (str.length() != e.length()) return INF;
        int n = str.length();
        int ans = 0;
        for (int i = 0; i < n; i++) {
            ans += str.charAt(i) == e.charAt(i) ? 0 : 1;
        }
        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 **752. 打开转盘锁** ，难度为 **中等**。

Tag : 「双向 BFS」、「启发式搜索」、「AStar 算法」、「IDAStar 算法」

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’ 。每个拨轮可以自由旋转：例如把 ‘9’ 变为 ‘0’，‘0’ 变为 ‘9’ 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 ‘0000’ ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回 -1 。

示例 1:

```
输入：deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出：6
解释：
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
因为当拨动到 "0102" 时这个锁就会被锁定。
```

示例 2:

```
输入: deadends = ["8888"], target = "0009"
输出：1
解释：
把最后一位反向旋转一次即可 "0000" -> "0009"。
```

示例 3:

```
输入: deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"
输出：-1
解释：
无法旋转到目标数字且不被锁定。
```

示例 4:

```
输入: deadends = ["0000"], target = "8888"
输出：-1
```

提示：

- 1 <= deadends.length <= 500
- deadends[i].length == 4
- target.length == 4
- target 不在 deadends 之中
- target 和 deadends[i] 仅由若干位数字组成

基本分析

首先，我建议你先做「127. 单词接龙」，然后再回过头将本题作为「练习题」。

「127. 单词接龙」定位困难，而本题定位中等。主要体现在数据范围上，思维难度上「127. 单词接龙」并不比本题难，大胆做。

「127. 单词接龙」原题链接在 这里，相关题解在 这里。

回到本题，根据题意，可以确定这是一个「最短路/最小步数」问题。

此类问题，通常我们会使用「BFS」求解，但朴素的 BFS 通常会带来搜索空间爆炸问题。

因此我们可以使用与 (题解)127. 单词接龙 类似的思路进行求解。

双向 BFS

我们知道，递归树的展开形式是一棵多阶树。

使用朴素 BFS 进行求解时，队列中最多会存在“两层”的搜索节点。

因此搜索空间的上界取决于 **目标节点所在的搜索层次的深度所对应的宽度**。

下图展示了朴素 BFS 可能面临的搜索空间爆炸问题：

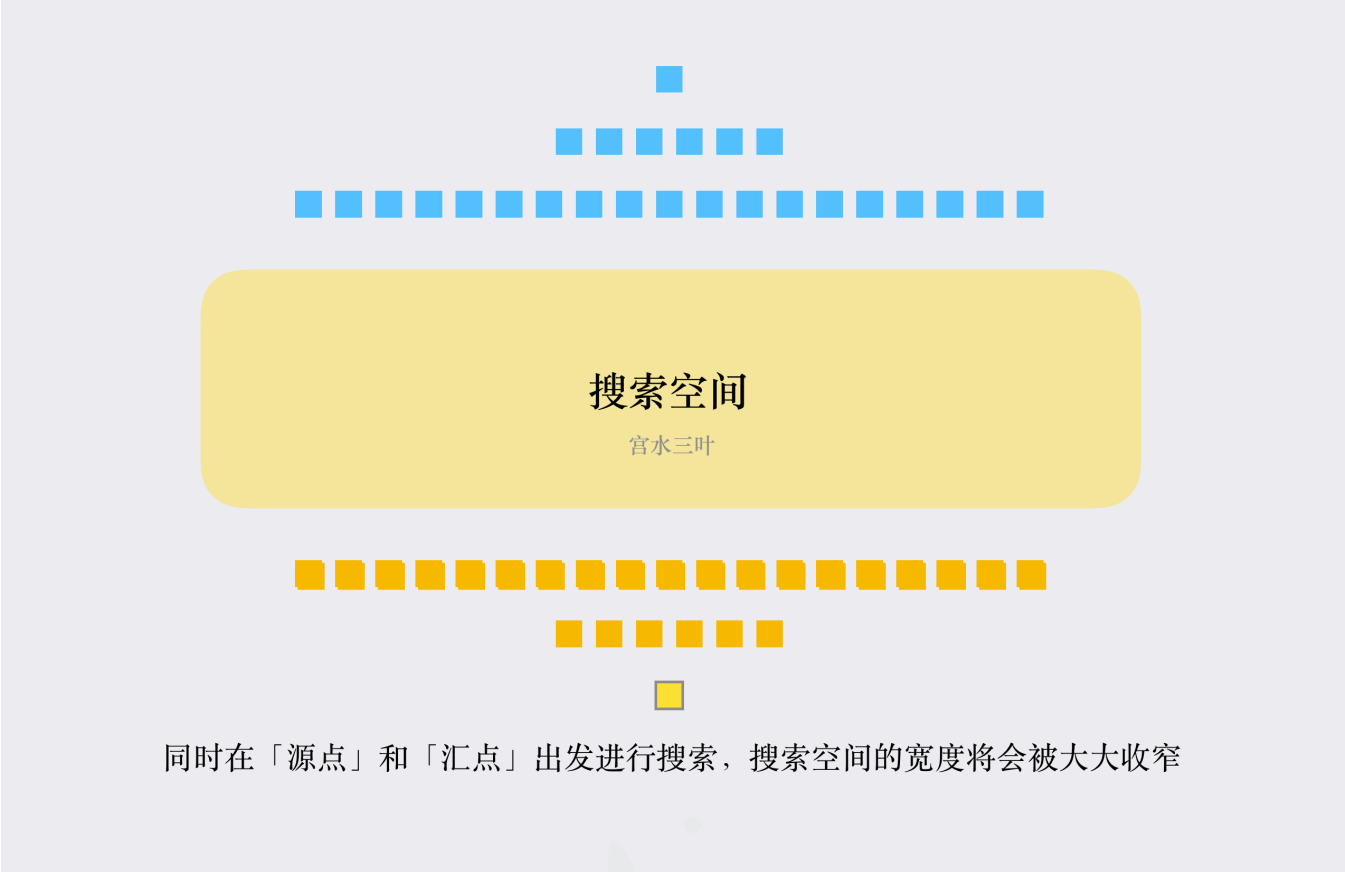
在朴素的 BFS 实现中，空间的瓶颈主要取决于搜索空间中的最大宽度。

那么有没有办法让我们不使用这么宽的搜索空间，同时又能保证搜索到目标结果呢？

「双向 BFS」 可以很好的解决这个问题：

同时从两个方向开始搜索，一旦搜索到相同的值，意味着找到了一条联通起点和终点的最短路径。

对于「有解」、「有一定数据范围」同时「层级节点数量以倍数或者指数级别增长」的情况，「双向 BFS」的搜索空间通常只有「朴素 BFS」的空间消耗的几百分之一，甚至几千分之一。



同时在「源点」和「汇点」出发进行搜索，搜索空间的宽度将会被大大收窄

「双向 BFS」的基本实现思路如下：

1. 创建「两个队列」分别用于两个方向的搜索；
2. 创建「两个哈希表」用于「解决相同节点重复搜索」和「记录转换次数」；
3. 为了尽可能让两个搜索方向“平均”，每次从队列中取值进行扩展时，先判断哪个队列容量较少；
4. 如果在搜索过程中「搜索到对方搜索过的节点」，说明找到了最短路径。

「双向 BFS」基本思路对应的伪代码大致如下：

```
d1、d2 为两个方向的队列
m1、m2 为两个方向的哈希表，记录每个节点距离起点的

// 只有两个队列都不空，才有必要继续往下搜索
// 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
while(!d1.isEmpty() && !d2.isEmpty()) {
    if (d1.size() < d2.size()) {
        update(d1, m1, m2);
    } else {
        update(d2, m2, m1);
    }
}

// update 为从队列 d 中取出一个元素进行「一次完整扩展」的逻辑
void update(Deque d, Map cur, Map other) {}
```

回到本题，我们看看如何使用「双向 BFS」进行求解。

估计不少同学是第一次接触「双向 BFS」，因此这次我写了大量注释。

建议大家带着对「双向 BFS」的基本理解去阅读。

代码：

```java
class Solution {
    String t, s;
    Set<String> set = new HashSet<>();
    public int openLock(String[] _ds, String _t) {
        s = "0000";
        t = _t;
        if (s.equals(t)) return 0;
        for (String d : _ds) set.add(d);
        if (set.contains(s)) return -1;
        int ans = bfs();
        return ans;
    }
    int bfs() {
        // d1 代表从起点 s 开始搜索（正向）
        // d2 代表从结尾 t 开始搜索（反向）
        Deque<String> d1 = new ArrayDeque<>(), d2 = new ArrayDeque<>();
        /*
         * m1 和 m2 分别记录两个方向出现的状态是经过多少次转换而来
         * e.g
         * m1 = {"1000":1} 代表 "1000" 由 s="0000" 替换 1 次字符而来
         * m2 = {"9999":3} 代表 "9999" 由 t="9996" 替换 3 次字符而来
         */
        Map<String, Integer> m1 = new HashMap<>(), m2 = new HashMap<>();
        d1.addLast(s);
        m1.put(s, 0);
        d2.addLast(t);
        m2.put(t, 0);

        /*
         * 只有两个队列都不空，才有必要继续往下搜索
         * 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
         * e.g.
         * 例如，如果 d1 为空了，说明从 s 搜索到底都搜索不到 t，反向搜索也没必要进行了
         */
        while (!d1.isEmpty() && !d2.isEmpty()) {
            int t = -1;
            if (d1.size() <= d2.size()) {
                t = update(d1, m1, m2);
            } else {
                t = update(d2, m2, m1);
            }
            if (t != -1) return t;
        }
        return -1;
    }
    int update(Deque<String> deque, Map<String, Integer> cur, Map<String, Integer> other)
```

```
            String poll = deque.pollFirst();
            char[] pcs = poll.toCharArray();
            int step = cur.get(poll);
            // 枚举替换哪个字符
            for (int i = 0; i < 4; i++) {
                // 能「正向转」也能「反向转」，这里直接枚举偏移量 [-1,1] 然后跳过 0
                for (int j = -1; j <= 1; j++) {
                    if (j == 0) continue;

                    // 求得替换字符串 str
                    int origin = pcs[i] - '0';
                    int next = (origin + j) % 10;
                    if (next == -1) next = 9;

                    char[] clone = pcs.clone();
                    clone[i] = (char)(next + '0');
                    String str = String.valueOf(clone);

                    if (set.contains(str)) continue;
                    if (cur.containsKey(str)) continue;

                    // 如果在「另一方向」找到过，说明找到了最短路，否则加入队列
                    if (other.containsKey(str)) {
                        return step + 1 + other.get(str);
                    } else {
                        deque.addLast(str);
                        cur.put(str, step + 1);
                    }
                }
            }
            return -1;
        }
    }
```

AStar 算法

可以直接根据本题规则来设计 A* 的「启发式函数」。

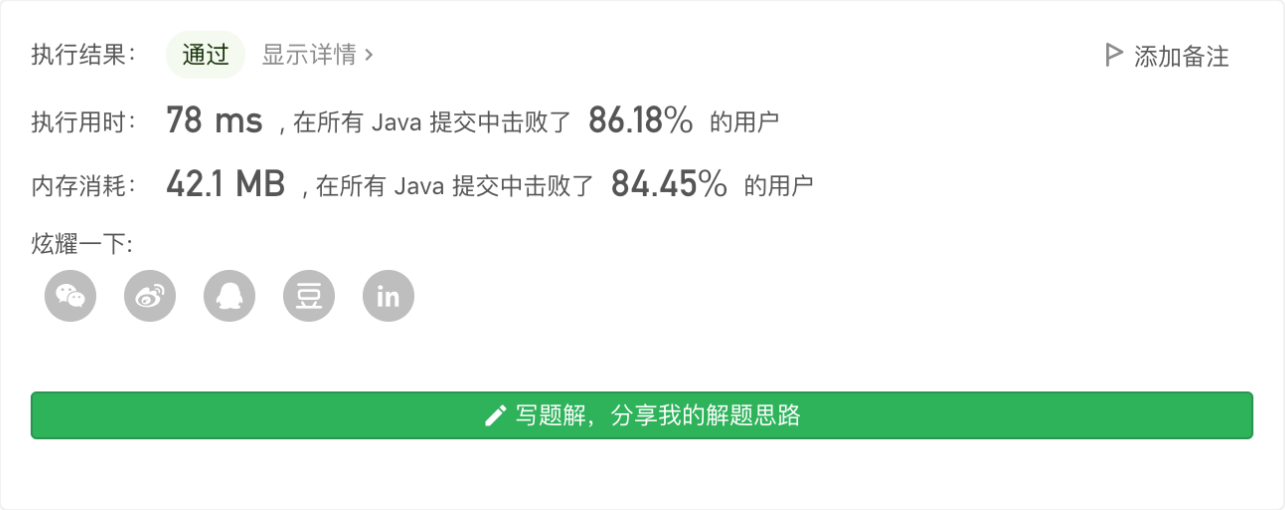
比如对于两个状态 `a` 和 `b` 可直接计算出「理论最小转换次数」：**不同字符的转换成本之和** 。

需要注意的是：由于我们衡量某个字符 `str` 的估值是以目标字符串 `target` 为基准，因此我们只能确保 `target` 出队时为「距离最短」，而不能确保中间节点出队时「距离最短」，因此

我们不能单纯根据某个节点是否「曾经入队」而决定是否入队，还要结合当前节点的「最小距离」是否被更新而决定是否入队。

这一点十分关键，在代码层面上体现在 `map.get(str).step > poll.step + 1` 的判断上。

注意：本题用 A* 过了，但通常我们需要先「确保有解」，A* 的启发搜索才会发挥真正价值。而本题，除非 `t` 本身在 `deadends` 中，其余情况我们无法很好提前判断「是否有解」。对于无解的情况 A* 效果不如「双向 BFS」。



代码：

```java
class Solution {
    class Node {
        String str;
        int val, step;
        /**
         *  str : 对应字符串
         *  val : 估值（与目标字符串 target 的最小转换成本）
         *  step: 对应字符串是经过多少步转换而来
         */
        Node(String _str, int _val, int _step) {
            str = _str;
            val = _val;
            step = _step;
        }
    }
    int f(String str) {
        int ans = 0;
        for (int i = 0; i < 4; i++) {
            int cur = str.charAt(i) - '0', target = t.charAt(i) - '0';
            int a = Math.min(cur, target), b = Math.max(cur, target);
            // 在「正向转」和「反向转」之间取 min
            int min = Math.min(b - a, a + 10 - b);
            ans += min;
        }
        return ans;
    }
    String s, t;
    Set<String> set = new HashSet<>();
    public int openLock(String[] ds, String _t) {
        s = "0000";
        t = _t;
        if (s.equals(t)) return 0;
        for (String d : ds) set.add(d);
        if (set.contains(s)) return -1;

        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->a.val-b.val);
        Map<String, Node> map = new HashMap<>();
        Node root = new Node(s, f(s), 0);
        q.add(root);
        map.put(s, root);
        while (!q.isEmpty()) {
            Node poll = q.poll();
            char[] pcs = poll.str.toCharArray();
            int step = poll.step;
            if (poll.str.equals(t)) return step;
            for (int i = 0; i < 4; i++) {
```

```
                for (int j = -1; j <= 1; j++) {
                    if (j == 0) continue;
                    int cur = pcs[i] - '0';
                    int next = (cur + j) % 10;
                    if (next == -1) next = 9;

                    char[] clone = pcs.clone();
                    clone[i] = (char)(next + '0');
                    String str = String.valueOf(clone);

                    if (set.contains(str)) continue;
                    // 如果 str 还没搜索过，或者 str 的「最短距离」被更新，则入队
                    if (!map.containsKey(str) || map.get(str).step > step + 1) {
                        Node node = new Node(str, step + 1 + f(str), step + 1);
                        map.put(str, node);
                        q.add(node);
                    }
                }
            }
        }
        return -1;
    }
}
```

IDA* 算法

同样我们可以使用基于 `DFS` 的启发式 IDA* 算法：

- 仍然使用 `f()` 作为估值函数
- 利用旋转次数有限：总旋转次数不会超过某个阈值 $\max$。
- 利用「迭代加深」的思路找到最短距离

理想情况下，由于存在正向旋转和反向旋转，每一位转轮从任意数字开始到达任意数字，消耗次数不会超过 5 次，因此理想情况下可以设定 $\max = 5 * 4$ 。

但考虑 `deadends` 的存在，我们需要将 $\max$ 定义得更加保守一些：$\max = 10 * 4$ 。

但这样的阈值设定，加上 IDA* 算法每次会重复遍历「距离小于与目标节点距离」的所有节点，会有很大的 TLE 风险。

因此我们需要使用动态阈值：不再使用固定的阈值，而是利用 `target` 计算出「最大的转移成

本」作为我们的「最深数量级」。

PS. 上述的阈值分析是科学做法。对于本题可以利用数据弱，直接使用 $\max = 5 * 4$ 也可以通过，并且效果不错。

但必须清楚 $\max = 5 * 4$ 可能是一个错误的阈值，本题起点为 `0000`，考虑将所有正向转换的状态都放入 `deadends` 中，`target` 为 `2222`。这时候我们可以只限定 `0000` 先变为 `9999` 再往回变为 `2222` 的通路不在 `deadends` 中。

这时候使用 $\max = 5 * 4$ 就不对，但本题数据弱，可以通过（想提交错误数据拿积分吗？别试了，我已经提交了 🤣

代码：

```java
class Solution {
    String s, t;
    String cur;
    Set<String> set = new HashSet<>();
    Map<String, Integer> map = new HashMap<>();
    public int openLock(String[] ds, String _t) {
        s = "0000";
        t = _t;
        if (s.equals(t)) return 0;
        for (String d : ds) set.add(d);
        if (set.contains(s)) return -1;

        int depth = 0, max = getMax();
        cur = s;
        map.put(cur, 0);
        while (depth <= max && !dfs(0, depth)) {
            map.clear();
            cur = s;
            map.put(cur, 0);
            depth++;
        }
        return depth > max ? -1 : depth;
    }
    int getMax() {
        int ans = 0;
        for (int i = 0; i < 4; i++) {
            int origin = s.charAt(i) - '0', next = t.charAt(i) - '0';
            int a = Math.min(origin, next), b = Math.max(origin, next);
            int max = Math.max(b - a, a + 10 - b);
            ans += max;
        }
        return ans;
    }
    int f() {
        int ans = 0;
        for (int i = 0; i < 4; i++) {
            int origin = cur.charAt(i) - '0', next = t.charAt(i) - '0';
            int a = Math.min(origin, next), b = Math.max(origin, next);
            int min = Math.min(b - a, a + 10 - b);
            ans += min;
        }
        return ans;
    }
    boolean dfs(int u, int max) {
        if (u + f() > max) return false;
        if (f() == 0) return true;
```

```
        String backup = cur;
        char[] cs = cur.toCharArray();
        for (int i = 0; i < 4; i++) {
            for (int j = -1; j <= 1; j++) {
                if (j == 0) continue;
                int origin = cs[i] - '0';
                int next = (origin + j) % 10;
                if (next == -1) next = 9;
                char[] clone = cs.clone();
                clone[i] = (char)(next + '0');
                String str = String.valueOf(clone);
                if (set.contains(str)) continue;
                if (!map.containsKey(str) || map.get(str) > u + 1) {
                    cur = str;
                    map.put(str, u + 1);
                    if (dfs(u + 1, max)) return true;
                    cur = backup;
                }
            }
        }
        return false;
    }
}
```

> **🍭 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 **773. 滑动谜题** ，难度为 **困难**。

Tag : 「BFS」、「最小步数」、「AStar 算法」、「启发式搜索」

在一个 2 x 3 的板上（board）有 5 块砖瓦，用数字 1~5 来表示, 以及一块空缺用 0 来表示.

一次移动定义为选择 0 与一个相邻的数字（上下左右）进行交换.

最终当板 board 的结果是 1,2,3],[4,5,0 谜板被解开。

给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1 。

示例：

```
输入：board = [[1,2,3],[4,0,5]]
输出：1
解释：交换 0 和 5 ，1 步完成
```

```
输入：board = [[1,2,3],[5,4,0]]
输出：-1
解释：没有办法完成谜板
```

```
输入：board = [[4,1,2],[5,0,3]]
输出：5
解释：
最少完成谜板的最少移动次数是 5 ，
一种移动路径:
尚未移动: [[4,1,2],[5,0,3]]
移动 1 次: [[4,1,2],[0,5,3]]
移动 2 次: [[0,1,2],[4,5,3]]
移动 3 次: [[1,0,2],[4,5,3]]
移动 4 次: [[1,2,0],[4,5,3]]
移动 5 次: [[1,2,3],[4,5,0]]
```

```
输入：board = [[3,2,4],[1,5,0]]
输出：14
```

提示：

- board 是一个如上所述的 2 x 3 的数组.
- board[i][j] 是一个 [0, 1, 2, 3, 4, 5] 的排列.

基本分析

这是八数码问题的简化版：将 $3 * 3$ 变为 $2 * 3$，同时将「输出路径」变为「求最小步数」。

通常此类问题可以使用「BFS」、「AStar 算法」、「康拓展开」进行求解。

由于问题简化到了 $2 * 3$，我们使用前两种解法即可。

BFS

为了方便，将原来的二维矩阵转成字符串（一维矩阵）进行处理。

这样带来的好处直接可以作为哈希 `Key` 使用，也可以很方便进行「二维坐标」与「一维下标」的转换。

由于固定是 $2 * 3$ 的格子，因此任意的合法二维坐标 (x, y) 和对应一维下标 idx 可通过以下转换：

- $idx = x * 3 + y$
- $x = idx / 3, y = idx \% 3$

其余的就是常规的 `BFS` 过程了。

代码：

```Java
class Solution {
    class Node {
        String str;
        int x, y;
        Node(String _str, int _x, int _y) {
            str = _str; x = _x; y = _y;
        }
    }
    int n = 2, m = 3;
    String s, e;
    int x, y;
    public int slidingPuzzle(int[][] board) {
        s = "";
        e = "123450";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                s += board[i][j];
                if (board[i][j] == 0) {
                    x = i; y = j;
                }
            }
        }
        int ans = bfs();
        return ans;
    }
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    int bfs() {
        Deque<Node> d = new ArrayDeque<>();
        Map<String, Integer> map = new HashMap<>();
        Node root = new Node(s, x, y);
        d.addLast(root);
        map.put(s, 0);
        while (!d.isEmpty()) {
            Node poll = d.pollFirst();
            int step = map.get(poll.str);
            if (poll.str.equals(e)) return step;
            int dx = poll.x, dy = poll.y;
            for (int[] di : dirs) {
                int nx = dx + di[0], ny = dy + di[1];
                if (nx < 0 || nx >= n || ny < 0 || ny >= m) continue;
                String nStr = update(poll.str, dx, dy, nx, ny);
                if (map.containsKey(nStr)) continue;
                Node next = new Node(nStr, nx, ny);
                d.addLast(next);
                map.put(nStr, step + 1);
            }
```

```
        }
        return -1;
    }
    String update(String cur, int i, int j, int p, int q) {
        char[] cs = cur.toCharArray();
        char tmp = cs[i * m + j];
        cs[i * m + j] = cs[p * m + q];
        cs[p * m + q] = tmp;
        return String.valueOf(cs);
    }
}
```

A* 算法

可以直接根据本题规则来设计 A* 的「启发式函数」。

比如对于两个状态 `a` 和 `b` 可直接计算出「理论最小转换次数」：**所有位置的数值「所在位置」与「目标位置」的曼哈顿距离之和（即横纵坐标绝对值之和）**。

注意，我们只需要计算「非空格」位置的曼哈顿距离即可，因为空格的位置会由其余数字占掉哪些位置而唯一确定。

A* 求最短路的正确性问题：由于我们衡量某个状态 `str` 的估值是以目标字符串 `e=123450` 为基准，因此我们只能确保 `e` 出队时为「距离最短」，而不能确保中间节点出队时「距离最短」，因此我们不能单纯根据某个节点是否「曾经入队」而决定是否入队，还要结合当前节点的「最小距离」是否被更新而决定是否入队。

这一点十分关键，在代码层面上体现在 `map.get(nStr) > step + 1` 的判断上。

我们知道，A* 在有解的情况下，才会发挥「启发式搜索」的最大价值，因此如果我们能够提前判断无解的情况，对 A* 算法来说会是巨大的提升。

而对于通用的 $N * N$ 数码问题，判定有解的一个充要条件是：**「逆序对」数量为偶数，如果不满足，必然无解，直接返回 -1 即可。**

对该结论的充分性证明和必要性证明完全不在一个难度上，所以建议记住这个结论即可。

代码：

```Java
class Solution {
    class Node {
        String str;
        int x, y;
        int val;
        Node(String _str, int _x, int _y, int _val) {
            str = _str; x = _x; y = _y; val = _val;
        }
    }
    int f(String str) {
        int ans = 0;
        char[] cs1 = str.toCharArray(), cs2 = e.toCharArray();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                // 跳过「空格」，计算其余数值的曼哈顿距离
                if (cs1[i * m + j] == '0' || cs2[i * m + j] == '0') continue;
                int cur = cs1[i * m + j], next = cs2[i * m + j];
                int xd = Math.abs((cur - 1) / 3 - (next - 1) / 3);
                int yd = Math.abs((cur - 1) % 3 - (next - 1) % 3);
                ans += (xd + yd);
            }
        }
        return ans;
    }
    int n = 2, m = 3;
    String s, e;
    int x, y;
    public int slidingPuzzle(int[][] board) {
        s = "";
        e = "123450";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                s += board[i][j];
                if (board[i][j] == 0) {
                    x = i; y = j;
                }
            }
        }

        // 提前判断无解情况
        if (!check(s)) return -1;

        int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
        Node root = new Node(s, x, y, f(s));
        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->a.val-b.val);
        Map<String, Integer> map = new HashMap<>();
```

```
        q.add(root);
        map.put(s, 0);
        while (!q.isEmpty()) {
            Node poll = q.poll();
            int step = map.get(poll.str);
            if (poll.str.equals(e)) return step;
            int dx = poll.x, dy = poll.y;
            for (int[] di : dirs) {
                int nx = dx + di[0], ny = dy + di[1];
                if (nx < 0 || nx >= n || ny < 0 || ny >= m) continue;
                String nStr = update(poll.str, dx, dy, nx, ny);
                if (!map.containsKey(nStr) || map.get(nStr) > step + 1) {
                    Node next = new Node(nStr, nx, ny, step + 1 + f(nStr));
                    q.add(next);
                    map.put(nStr, step + 1);
                }
            }
        }
        return 0x3f3f3f3f; // never
    }
    String update(String cur, int i, int j, int p, int q) {
        char[] cs = cur.toCharArray();
        char tmp = cs[i * m + j];
        cs[i * m + j] = cs[p * m + q];
        cs[p * m + q] = tmp;
        return String.valueOf(cs);
    }
    boolean check(String str) {
        char[] cs = str.toCharArray();
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n * m; i++) {
            if (cs[i] != '0') list.add(cs[i] - '0');
        }
        int cnt = 0;
        for (int i = 0; i < list.size(); i++) {
            for (int j = i + 1; j < list.size(); j++) {
                if (list.get(i) < list.get(j)) cnt++;
            }
        }
        return cnt % 2 == 0;
    }
}
```

> **🍥 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 **847. 访问所有节点的最短路径** ，难度为 **困难**。

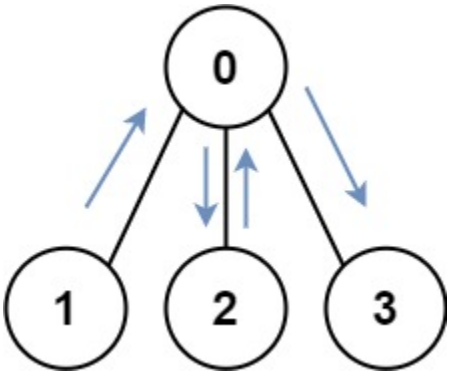
Tag：「图」、「图论 BFS」、「动态规划」、「状态压缩」

存在一个由 n 个节点组成的无向连通图，图中的节点按从 0 到 n - 1 编号。

给你一个数组 graph 表示这个图。其中，graph[i] 是一个列表，由所有与节点 i 直接相连的节点组成。

返回能够访问所有节点的最短路径的长度。你可以在任一节点开始和停止，也可以多次重访节点，并且可以重用边。

示例 1：

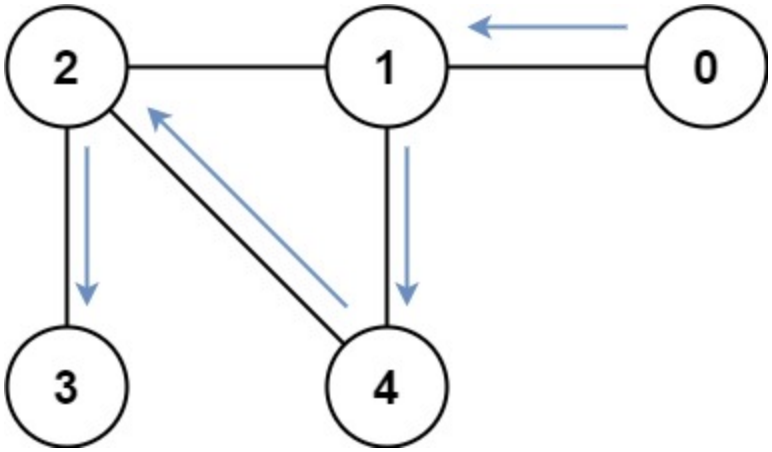



```
输入：graph = [[1,2,3],[0],[0],[0]]

输出：4

解释：一种可能的路径为 [1,0,2,0,3]
```

示例 2：




```
输入：graph = [[1],[0,2,4],[1,3,4],[2],[1,2]]

输出：4

解释：一种可能的路径为 [0,1,4,2,3]
```

提示：

- n == graph.length
- 1 <= n <= 12
- 0 <= graph[i].length < n
- graph[i] 不包含 i
- 如果 graph[a] 包含 b ，那么 graph[b] 也包含 a
- 输入的图总是连通图

基本分析

为了方便，令点的数量为 n，边的数量为 m。

这是一个等权无向图，题目要我们求从「**一个点都没访问过**」到「**所有点都被访问**」的最短路径。

同时 n 只有 12，容易想到使用「状态压缩」来代表「当前点的访问状态」：**使用二进制表示长度为 32 的 `int` 的低 12 来代指点是否被访问过。**

我们可以通过一个具体的样例，来感受下「状态压缩」是什么意思：

例如 $(000...0101)_2$ 代表编号为 0 和编号为 2 的节点已经被访问过，而编号为 1 的节点尚未被访问。

然后再来看看使用「状态压缩」的话，一些基本的操作该如何进行：

假设变量 $state$ 存放了「当前点的访问状态」，当我们需要检查编号为 x 的点是否被访问过时，可以使用位运算 `a = (state >> x) & 1`，来获取 $state$ 中第 x 位的二进制表示，如果 `a` 为 1 代表编号为 x 的节点已被访问，如果为 0 则未被访问。

同理，当我们需要将标记编号为 x 的节点已经被访问的话，可以使用位运算 `state | (1 << x)` 来实现标记。

状态压缩 + BFS

因为是等权图，求从某个状态到另一状态的最短路，容易想到 `BFS`。

同时我们需要知道下一步能往哪些点进行移动，因此除了记录当前的点访问状态 $state$ 以外，还需要记录最后一步是在哪个点 u，因此我们需要使用二元组进行记录 $(state, u)$，同时使用 $dist$ 来记录到达 $(state, u)$ 使用的步长是多少。

> 一些细节：由于点的数量较少，使用「邻接表」或者「邻接矩阵」来存图都可以。对于本题，由于已经给出了 $graph$ 数组，因此可以直接充当「邻接表」来使用，而无须做额外的存图操作。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int shortestPathLength(int[][] graph) {
        int n = graph.length;
        int mask = 1 << n;

        // 初始化所有的 (state, u) 距离为正无穷
        int[][] dist = new int[mask][n];
        for (int i = 0; i < mask; i++) Arrays.fill(dist[i], INF);

        // 因为可以从任意起点出发，先将起始的起点状态入队，并设起点距离为 0
        Deque<int[]> d = new ArrayDeque<>(); // state, u
        for (int i = 0; i < n; i++) {
            dist[1 << i][i] = 0;
            d.addLast(new int[]{1 << i, i});
        }

        // BFS 过程，如果从点 u 能够到达点 i，则更新距离并进行入队
        while (!d.isEmpty()) {
            int[] poll = d.pollFirst();
            int state = poll[0], u = poll[1], step = dist[state][u];
            if (state == mask - 1) return step;
            for (int i : graph[u]) {
                if (dist[state | (1 << i)][i] == INF) {
                    dist[state | (1 << i)][i] = step + 1;
                    d.addLast(new int[]{state | (1 << i), i});
                }
            }
        }
        return -1; // never
    }
}
```

- 时间复杂度：点（状态）数量为 $n * 2^n$，边的数量为 $n^2 * 2^n$，`BFS` 复杂度上界为点数加边数，整体复杂度为 $O(n^2 * 2^n)$
- 空间复杂度：$O(n * 2^n)$

Floyd + 状压 DP

其实在上述方法中，我们已经使用了与 DP 状态定义分析很像的思路了。甚至我们的元祖设计 $(state, u)$ 也很像状态定义的两个维度。

那么为什么我们不使用 $f[state][u]$ 为从「没有点被访问过」到「访问过的点状态为 $state$」，并最后一步落在点 u 的状态定义，然后跑一遍 DP 来做呢？

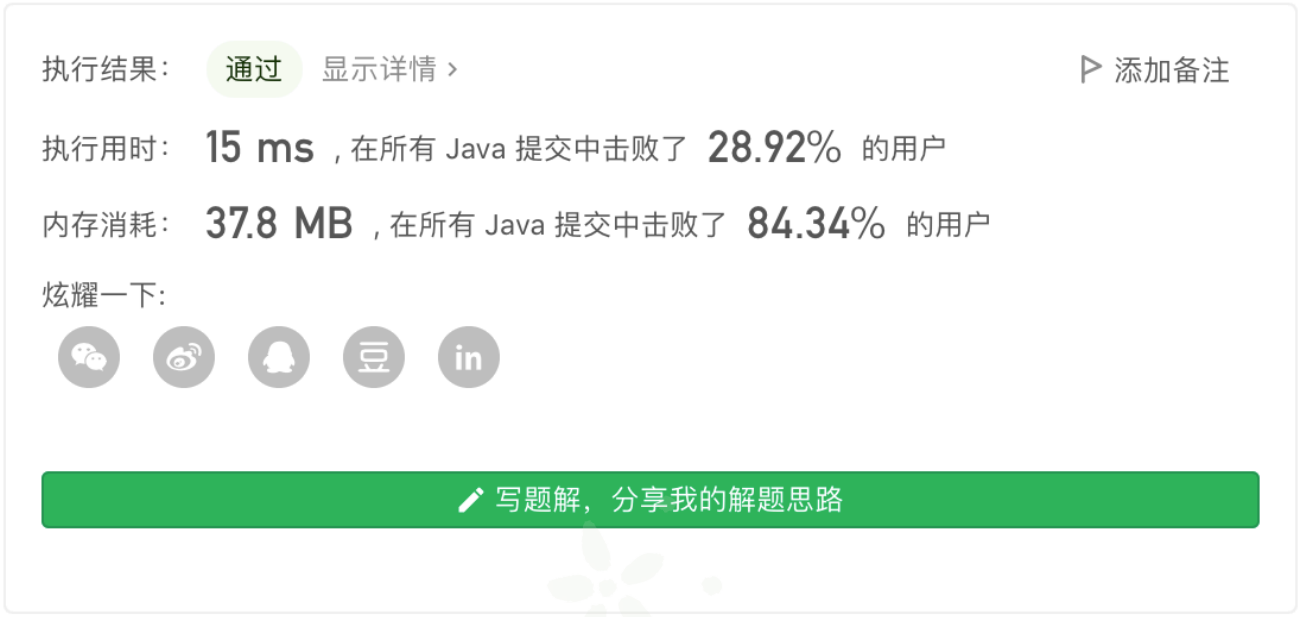
是因为如果从「常规的 DP 转移思路」出发，状态之间不存在拓扑序（有环），这就导致了我们在计算某个 $f[state][u]$ 时，它所依赖的状态并不确保已经被计算/更新完成，所以我们无法使用常规的 DP 手段来求解。

> 这里说的常规 DP 手段是指：枚举所有与 u 相连的节点 v，用 $f[state'][v]$ 来更新 $f[state][u]$ 的转移方式。

常规的 DP 转移方式状态间不存在拓扑序，我们需要换一个思路进行转移。

对于某个 $state$ 而言，我们可以枚举其最后一个点 i 是哪一个，充当其达到 $state$ 的最后一步，然后再枚举下一个点 j 是哪一个，充当移动的下一步（当然前提是满足 $state$ 的第 i 位为 1，而第 j 位为 0）。

求解任意两点最短路径，可以使用 Floyd 算法，复杂度为 $O(n^3)$。



代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int shortestPathLength(int[][] graph) {
        int n = graph.length;
        int mask = 1 << n;

        // Floyd 求两点的最短路径
        int[][] dist = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = INF;
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j : graph[i]) dist[i][j] = 1;
        }
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }

        // DP 过程，如果从 i 能够到 j 的话，使用 i 到 j 的最短距离（步长）来转移
        int[][] f = new int[mask][n];
        // 起始时，让所有状态的最短距离（步长）为正无穷
        for (int i = 0; i < mask; i++) Arrays.fill(f[i], INF);
        // 由于可以将任意点作为起点出发，可以将这些起点的最短距离（步长）设置为 0
        for (int i = 0; i < n; i++) f[1 << i][i] = 0;

        // 枚举所有的 state
        for (int state = 0; state < mask; state++) {
            // 枚举 state 中已经被访问过的点
            for (int i = 0; i < n; i++) {
                if (((state >> i) & 1) == 0) continue;
                // 枚举 state 中尚未被访问过的点
                for (int j = 0; j < n; j++) {
                    if (((state >> j) & 1) == 1) continue;
                    f[state | (1 << j)][j] = Math.min(f[state | (1 << j)][j], f[state][i]
                }
            }
        }

        int ans = INF;
        for (int i = 0; i < n; i++) ans = Math.min(ans, f[mask - 1][i]);
```

```
        return ans;
    }
}
```

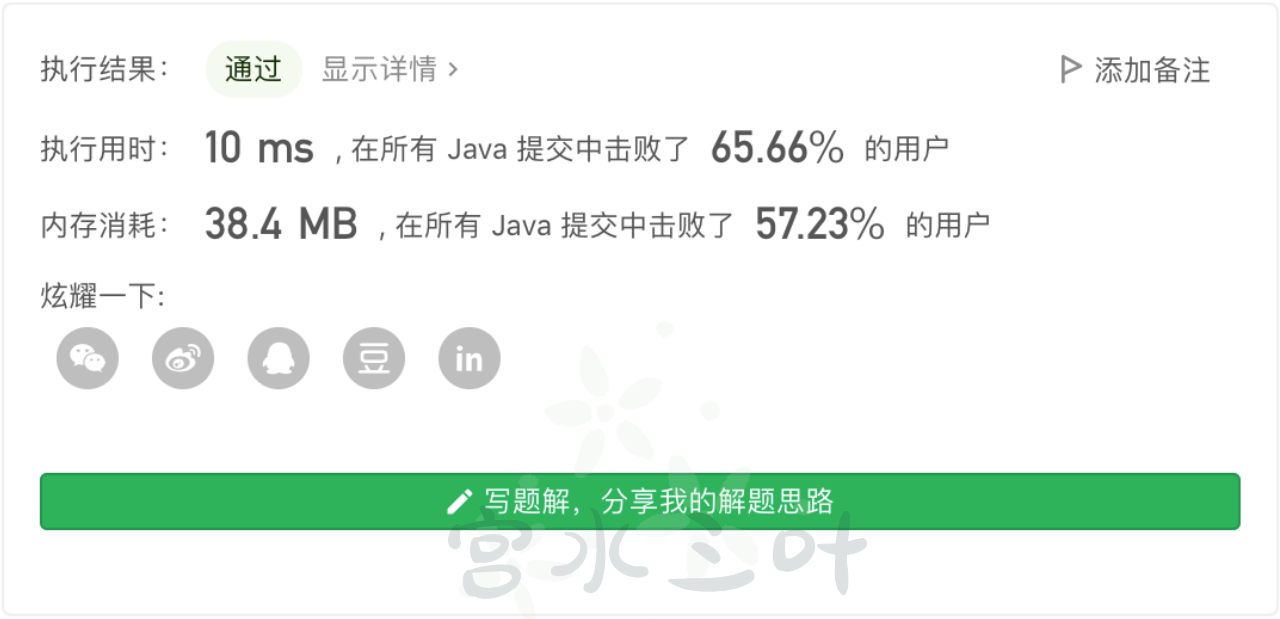
- 时间复杂度：Floyd 复杂度为 $O(n^3)$；DP 共有 $n * 2^n$ 个状态需要被转移，每次转移复杂度为 $O(n)$，总的复杂度为 $O(n^2 * 2^n)$。整体复杂度为 $O(\max(n^3, n^2 * 2^n))$
- 空间复杂度：$O(n * 2^n)$

AStar

显然，从 $state$ 到 $state'$ 的「理论最小修改成本」为两者二进制表示中不同位数的个数。

同时，当且仅当在 $state$ 中 1 的位置与 $state'$ 中 0 存在边，才有可能取到这个「理论最小修改成本」。

因此直接使用当前状态 $state$ 与最终目标状态 `1 << n` 两者二进制表示中不同位数的个数作为启发预估值是合适的。



代码：

```Java
class Solution {
    int INF = 0x3f3f3f3f;
    int n;
    int f(int state) {
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (((state >> i) & 1) == 0) ans++;
        }
        return ans;
    }
    public int shortestPathLength(int[][] g) {
        n = g.length;
        int mask = 1 << n;
        int[][] dist = new int[mask][n];
        for (int i = 0; i < mask; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = INF;
            }
        }
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->a[2]-b[2]); // state, u, val
        for (int i = 0; i < n; i++) {
            dist[1 << i][i] = 0;
            q.add(new int[]{1<< i, i, f(i << 1)});
        }
        while (!q.isEmpty()) {
            int[] poll = q.poll();
            int state = poll[0], u = poll[1], step = dist[state][u];
            if (state == mask - 1) return step;
            for (int i : g[u]) {
                int nState = state | (1 << i);
                if (dist[nState][i] > step + 1) {
                    dist[nState][i] = step + 1;
                    q.add(new int[]{nState, i, step + 1 + f(nState)});
                }
            }
        }
        return -1; // never
    }
}
```

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？

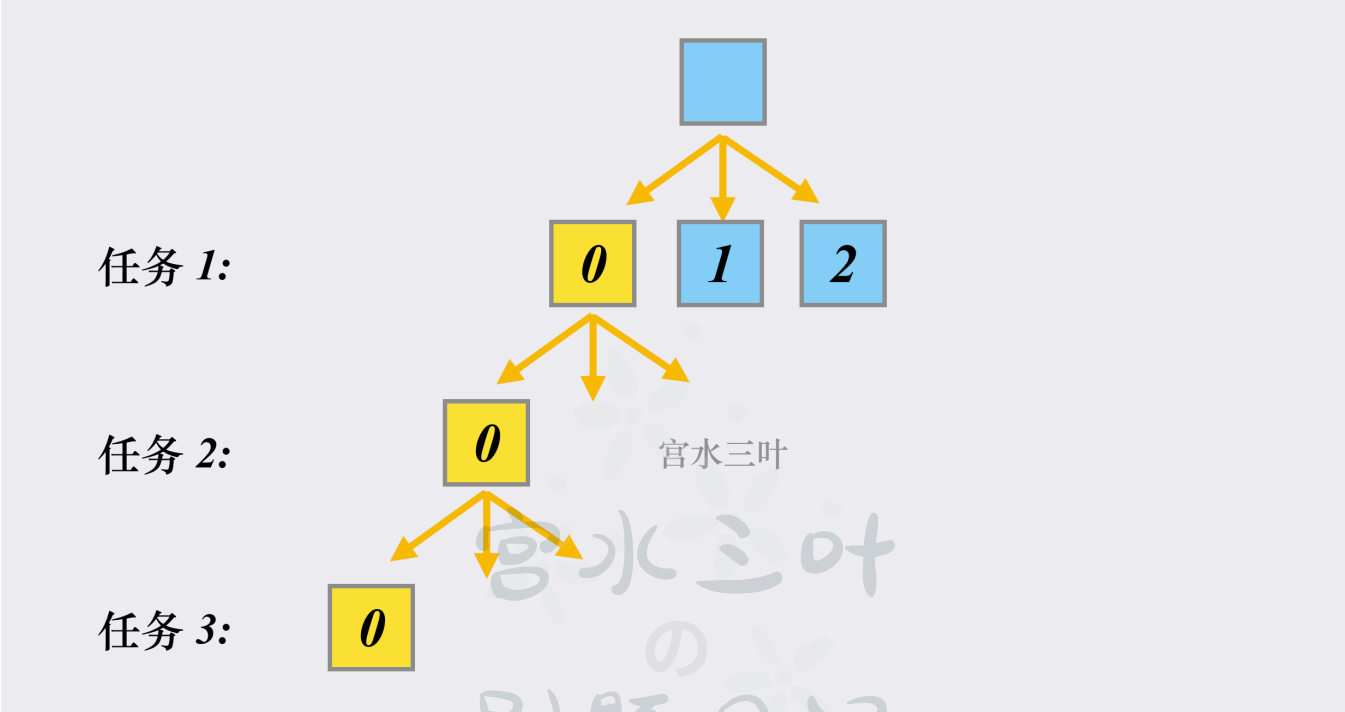
我们可以重新审视一下这道题。

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。

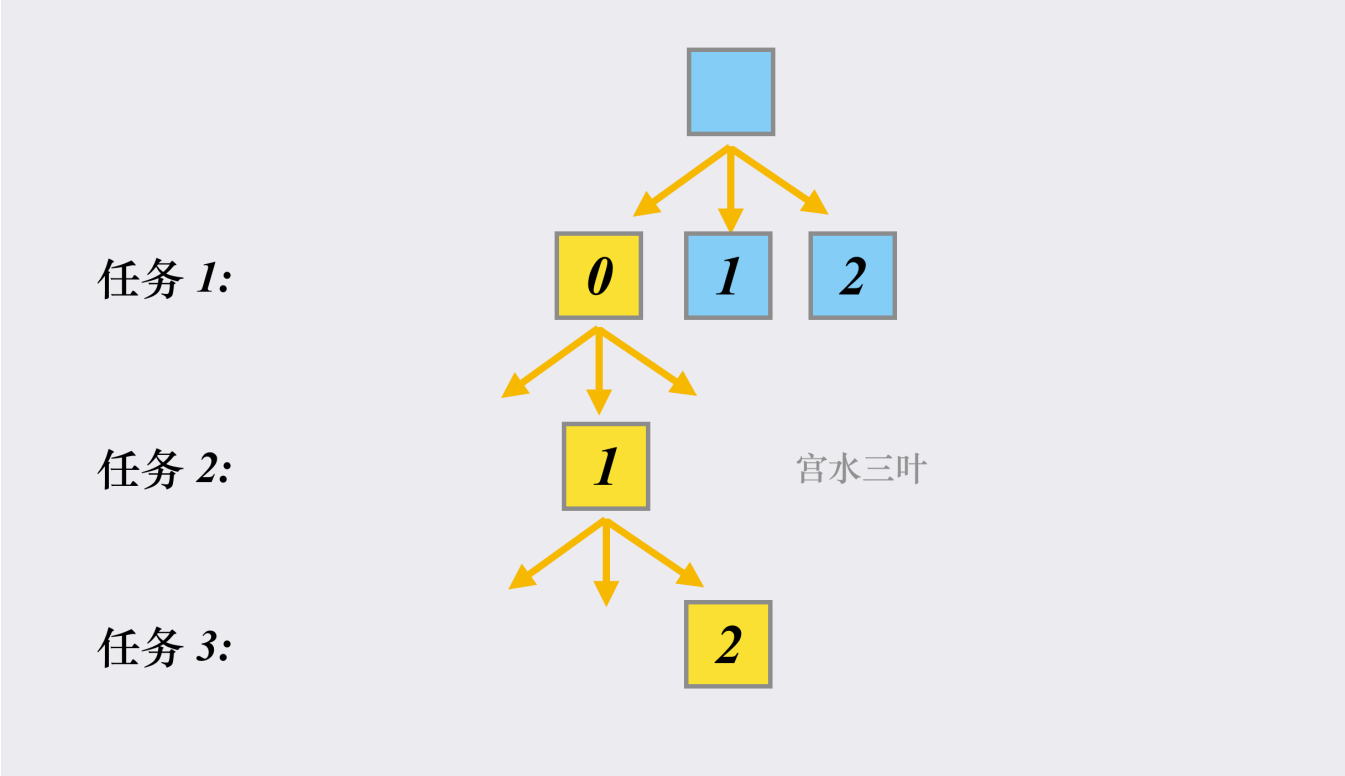
对应的递归树其实是一颗高度为 n 的 k 阶树。

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？

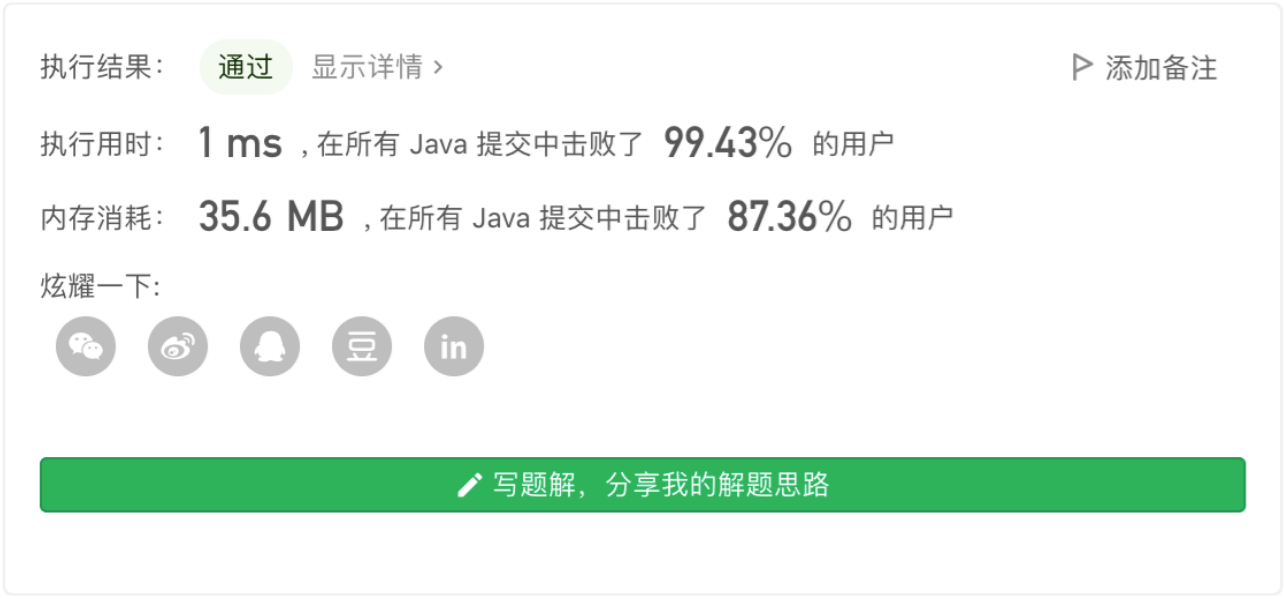
根据经验猜的，然后提交。根据结果是 `WA` 还是 `TLE` 来决定之后的调参方向。如果是 `WA` 说明部分数据落到了「局部最优」或者尚未达到「全局最优」。

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 **

💡**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「启发式搜索」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