宫水三叶的刷题日记

记忆化搜索

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「记忆化搜索」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「记忆化搜索」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「记忆化搜索」；
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 **87. 扰乱字符串** ，难度为 **困难**。

Tag : 「DFS」、「记忆化搜索」、「区间 DP」

使用下面描述的算法可以扰乱字符串 s 得到字符串 t ：

1. 如果字符串的长度为 1 ，算法停止
2. 如果字符串的长度 > 1 ，执行下述步骤：
    - 在一个随机下标处将字符串分割成两个非空的子字符串。即，如果已知字符串 s ，则可以将其分成两个子字符串 x 和 y ，且满足 s = x + y 。
    - 随机 决定是要「交换两个子字符串」还是要「保持这两个子字符串的顺序不变」。即，在执行这一步骤之后，s 可能是 s = x + y 或者 s = y + x 。

- 在 x 和 y 这两个子字符串上继续从步骤 1 开始递归执行此算法。

  给你两个 长度相等 的字符串 s1 和 s2，判断 s2 是否是 s1 的扰乱字符串。如果是，返回 true ；否则，返回 false 。

示例 1：

```
输入：s1 = "great", s2 = "rgeat"

输出：true

解释：s1 上可能发生的一种情形是：
"great" --> "gr/eat" // 在一个随机下标处分割得到两个子字符串
"gr/eat" --> "gr/eat" // 随机决定：「保持这两个子字符串的顺序不变」
"gr/eat" --> "g/r / e/at" // 在子字符串上递归执行此算法。两个子字符串分别在随机下标处进行一轮分割
"g/r / e/at" --> "r/g / e/at" // 随机决定：第一组「交换两个子字符串」，第二组「保持这两个子字符串的顺序不变」
"r/g / e/at" --> "r/g / e/ a/t" // 继续递归执行此算法，将 "at" 分割得到 "a/t"
"r/g / e/ a/t" --> "r/g / e/ a/t" // 随机决定：「保持这两个子字符串的顺序不变」
算法终止，结果字符串和 s2 相同，都是 "rgeat"
这是一种能够扰乱 s1 得到 s2 的情形，可以认为 s2 是 s1 的扰乱字符串，返回 true
```

示例 2：

```
输入：s1 = "abcde", s2 = "caebd"

输出：false
```

示例 3：

```
输入：s1 = "a", s2 = "a"

输出：true
```

提示：

s1.length == s2.length
1 <= s1.length <= 30
s1 和 s2 由小写英文字母组成

朴素解法（TLE）

一个朴素的做法根据「扰乱字符串」的生成规则进行判断。

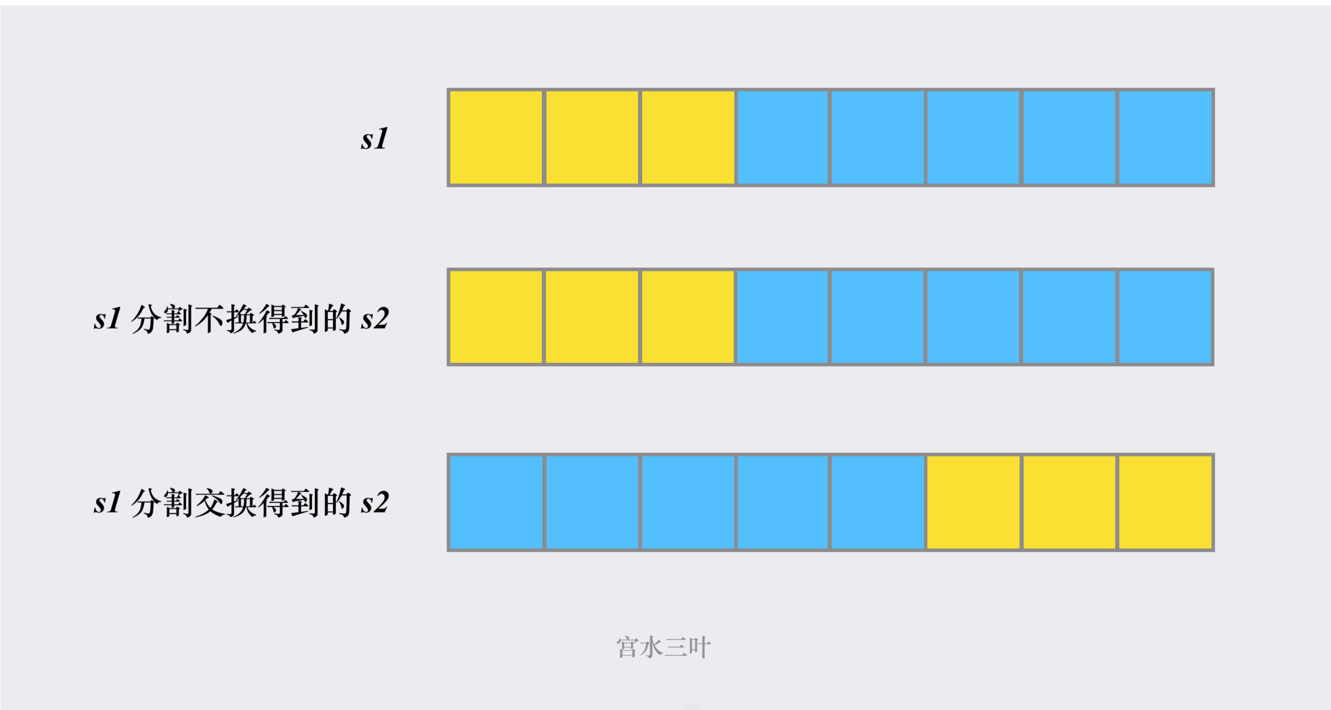
由于题目说了整个生成「扰乱字符串」的过程是通过「递归」来进行。

我们要实现 $isScramble$ 函数的作用是判断 $s1$ 是否可以生成出 $s2$。

这样判断的过程，同样我们可以使用「递归」来做：

假设 $s1$ 的长度为 n， 的第一次分割的分割点为 i，那么 $s1$ 会被分成 $[0, i)$ 和 $[i, n)$ 两部分。

同时由于生成「扰乱字符串」时，可以选交换也可以选不交换。因此我们的 $s2$ 会有两种可能性：



因为对于某个确定的分割点，$s1$ 固定分为两部分，分别为 $[0, i)$ & $[i, n)$。

而 $s2$ 可能会有两种分割方式，分别 $[0, i)$ & $[i, n)$ 和 $[0, n-i)$ & $[n-i, n)$。

我们只需要递归调用 $isScramble$ 检查 $s1$ 的 $[0, i)$ & $[i, n)$ 部分能否与 「$s2$ 的 $[0, i)$ & $[i, n)$」 或者 「$s2$ 的 $[0, n-i)$ & $[n-i, n)$」 匹配即可。

同时，我们将「$s1$ 和 $s2$ 相等」和「$s1$ 和 $s2$ 词频不同」作为「递归」出口。

理解这套做法十分重要，后续的解法都是基于此解法演变过来。

代码：

```java
class Solution {
    public boolean isScramble(String s1, String s2) {
        if (s1.equals(s2)) return true;
        if (!check(s1, s2)) return false;
        int n = s1.length();
        for (int i = 1; i < n; i++) {
            // s1 的 [0,i) 和 [i,n)
            String a = s1.substring(0, i), b = s1.substring(i);
            // s2 的 [0,i) 和 [i,n)
            String c = s2.substring(0, i), d = s2.substring(i);
            if (isScramble(a, c) && isScramble(b, d)) return true;
            // s2 的 [0,n-i) 和 [n-i,n)
            String e = s2.substring(0, n - i), f = s2.substring(n - i);
            if (isScramble(a, f) && isScramble(b, e)) return true;
        }
        return false;
    }
    // 检查 s1 和 s2 词频是否相同
    boolean check(String s1, String s2) {
        if (s1.length() != s2.length()) return false;
        int n = s1.length();
        int[] cnt1 = new int[26], cnt2 = new int[26];
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
        for (int i = 0; i < n; i++) {
            cnt1[cs1[i] - 'a']++;
            cnt2[cs2[i] - 'a']++;
        }
        for (int i = 0; i < 26; i++) {
            if (cnt1[i] != cnt2[i]) return false;
        }
        return true;
    }
}
```

- 时间复杂度：$O(5^n)$
- 空间复杂度：忽略递归与生成子串带来的空间开销，复杂度为 $O(1)$

记忆化搜索

朴素解法卡在了 $286/288$ 个样例。

我们考虑在朴素解法的基础上，增加「记忆化搜索」功能。

我们可以重新设计我们的「爆搜」逻辑：假设 $s1$ 从 i 位置开始，$s2$ 从 j 位置开始，后面的长度为 len 的字符串是否能形成「扰乱字符串」（互为翻转）。

那么在单次处理中，我们可分割的点的范围为 $[1, len)$，然后和「递归」一下，将 $s1$ 分割出来的部分尝试去和 $s2$ 的对应位置匹配。

同样的，我们将「入参对应的子串相等」和「入参对应的子串词频不同」作为「递归」出口。

代码：

```java
class Solution {
    String s1; String s2;
    int n;
    int[][][] cache;
    int N = -1, Y = 1, EMPTY = 0;
    public boolean isScramble(String _s1, String _s2) {
        s1 = _s1; s2 = _s2;
        if (s1.equals(s2)) return true;
        if (s1.length() != s2.length()) return false;
        n = s1.length();
        // cache 的默认值是 EMPTY
        cache = new int[n][n][n + 1];
        return dfs(0, 0, n);
    }
    boolean dfs(int i, int j, int len) {
        if (cache[i][j][len] != EMPTY) return cache[i][j][len] == Y;
        String a = s1.substring(i, i + len), b = s2.substring(j, j + len);
        if (a.equals(b)) {
            cache[i][j][len] = Y;
            return true;
        }
        if (!check(a, b)) {
            cache[i][j][len] = N;
            return false;
        }
        for (int k = 1; k < len; k++) {
            // 对应了「s1 的 [0,i) & [i,n)」匹配「s2 的 [0,i) & [i,n)」
            if (dfs(i, j, k) && dfs(i + k, j + k, len - k)) {
                cache[i][j][len] = Y;
                return true;
            }
            // 对应了「s1 的 [0,i) & [i,n)」匹配「s2 的 [n-i,n) & [0,n-i)」
            if (dfs(i, j + len - k, k) && dfs(i + k, j, len - k)) {
                cache[i][j][len] = Y;
                return true;
            }
        }
        cache[i][j][len] = N;
        return false;
    }
    // 检查 s1 和 s2 词频是否相同
    boolean check(String s1, String s2) {
        if (s1.length() != s2.length()) return false;
        int n = s1.length();
        int[] cnt1 = new int[26], cnt2 = new int[26];
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
```

```java
        for (int i = 0; i < n; i++) {
            cnt1[cs1[i] - 'a']++;
            cnt2[cs2[i] - 'a']++;
        }
        for (int i = 0; i < 26; i++) {
            if (cnt1[i] != cnt2[i]) return false;
        }
        return true;
    }
}
```

- 时间复杂度：$O(n^4)$
- 空间复杂度：$O(n^3)$

动态规划（区间 DP）

其实有了上述「记忆化搜索」方案之后，我们就已经可以直接忽略原问题，将其改成「动态规划」了。

根据「dfs 方法的几个可变入参」作为「状态定义的几个维度」，根据「dfs 方法的返回值」作为「具体的状态值」。

我们可以得到状态定义 $f[i][j][len]$：

$f[i][j][len]$ 代表 $s1$ 从 i 开始，$s2$ 从 j 开始，后面长度为 len 的字符是否能形成「扰乱字符串」（互为翻转）。

状态转移方程其实就是翻译我们「记忆化搜索」中的 dfs 主要逻辑部分：

```java
    // 对应了「s1 的 [0,i) & [i,n)」匹配「s2 的 [0,i) & [i,n)」
    if (dfs(i, j, k) && dfs(i + k, j + k, len - k)) {
        cache[i][j][len] = Y;
        return true;
    }
    // 对应了「s1 的 [0,i) & [i,n)」匹配「s2 的 [n-i,n) & [0,n-i)」
    if (dfs(i, j + len - k, k) && dfs(i + k, j, len - k)) {
        cache[i][j][len] = Y;
        return true;
    }
```

从状态定义上，我们就不难发现这是一个「区间 DP」问题，区间长度大的状态值可以由区间长度小的状态值递推而来。

而且由于本身我们在「记忆化搜索」里面就是从小到大枚举 len，因此这里也需要先将 len 这层循环提前，确保我们转移 $f[i][j][len]$ 时所需要的状态都已经被计算好。

代码：

```java
class Solution {
    public boolean isScramble(String s1, String s2) {
        if (s1.equals(s2)) return true;
        if (s1.length() != s2.length()) return false;
        int n = s1.length();
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
        boolean[][][] f = new boolean[n][n][n + 1];

        // 先处理长度为 1 的情况
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                f[i][j][1] = cs1[i] == cs2[j];
            }
        }

        // 再处理其余长度情况
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                for (int j = 0; j <= n - len; j++) {
                    for (int k = 1; k < len; k++) {
                        boolean a = f[i][j][k] && f[i + k][j + k][len - k];
                        boolean b = f[i][j + len - k][k] && f[i + k][j][len - k];
                        if (a || b) {
                            f[i][j][len] = true;
                        }
                    }
                }
            }
        }
        return f[0][0][n];
    }
}
```

- 时间复杂度：$O(n^4)$
- 空间复杂度：$O(n^3)$

> **🍭 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：

但数据范围为 10^3，直接使用 DFS 肯定会超时。

```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：$O(n^2)$

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

题目描述

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

Tag : 「DFS」、「记忆化搜索」、「背包 DP」、「01 背包」

给你一个整数数组 nums 和一个整数 target 。

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

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

示例 1：

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

示例 2：

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

提示：

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

DFS

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

而 DFS 有「使用全局变量维护」和「接收返回值处理」两种形式。

代码：

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

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

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

记忆化搜索

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

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

以上分析都在 （题解）403. 青蛙过河 完整讲过。

代码：

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

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

动态规划

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

根据记忆化搜索的分析，我们可以定义：

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

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

根据每个数值只能搭配 $+/-$ 使用，可得状态转移方程：

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

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

距离我们完成所有分析还差最后一步。

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

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

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

代码：

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

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

动态规划（优化）

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

即达成某些状态后，不可能再回到我们的「目标状态」。

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

那么我们如何规避掉这些“额外”状态值呢？

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

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

变形得：

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

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

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

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

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

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

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

每个数值有「选」和「不选」两种决策，转移方程为：

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

代码：

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

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

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

题目描述

这是 LeetCode 上的 **552. 学生出勤记录 II** ，难度为 **困难**。

Tag : 「动态规划」、「状态机」、「记忆化搜索」、「矩阵快速幂」、「数学」

可以用字符串表示一个学生的出勤记录，其中的每个字符用来标记当天的出勤情况（缺勤、迟到、到场）。

记录中只含下面三种字符：

- ‘A’：Absent，缺勤
- ‘L’：Late，迟到
- ‘P’：Present，到场

如果学生能够**同时**满足下面两个条件，则可以获得出勤奖励：

- 按**总出勤**计，学生缺勤（‘A’）严格 少于两天。
- 学生**不会**存在 连续 3 天或 连续 3 天以上的迟到（‘L’）记录。

给你一个整数 n ，表示出勤记录的长度（次数）。请你返回记录长度为 n 时，可能获得出勤奖励的记录情况 数量 。

答案可能很大，所以返回对 $10^9 + 7$ **取余**的结果。

示例 1：

```
输入：n = 2

输出：8

解释：
有 8 种长度为 2 的记录将被视为可奖励：
"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"
只有"AA"不会被视为可奖励，因为缺勤次数为 2 次（需要少于 2 次）。
```

示例 2：

```
输入：n = 1

输出：3
```

示例 3：

```
输入：n = 10101

输出：183236316
```

提示：

- 1 <= n <= 10^5

基本分析

根据题意，我们知道一个合法的方案中 `A` 的总出现次数最多为 1 次，`L` 的连续出现次数最多为 2 次。

因此在枚举/统计合法方案的个数时，当我们决策到某一位应该选什么时，我们关心的是当前方案中已经出现了多少个 `A`（以决策当前能否填入 `A`）以及连续出现的 `L` 的次数是多少（以决策当前能否填入 `L`）。

记忆化搜索

枚举所有方案的爆搜 `DFS` 代码不难写，大致的函数签名设计如下：

```
/**
 * @param u 当前还剩下多少位需要决策
 * @param acnt 当前方案中 A 的总出现次数
 * @param lcnt 当前方案中结尾 L 的连续出现次数
 * @param cur 当前方案
 * @param ans 结果集
 */
void dfs(int u, int acnt, int lcnt, String cur, List<String> ans);
```

实际上，我们不需要枚举所有的方案数，因此我们只需要保留函数签名中的前三个参数即可。

同时由于我们在计算某个 $(u, acnt, lcnt)$ 的方案数时，其依赖的状态可能会被重复使用，考虑加入记忆化，将结果缓存起来。

根据题意，n 的取值范围为 $[0, n]$，$acnt$ 取值范围为 $[0, 1]$，$lcnt$ 取值范围为 $[0, 2]$。

代码：

```java
class Solution {
    int mod = (int)1e9+7;
    int[][][] cache;
    public int checkRecord(int n) {
        cache = new int[n + 1][2][3];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    cache[i][j][k] = -1;
                }
            }
        }
        return dfs(n, 0, 0);
    }
    int dfs(int u, int acnt, int lcnt) {
        if (acnt >= 2) return 0;
        if (lcnt >= 3) return 0;
        if (u == 0) return 1;
        if (cache[u][acnt][lcnt] != -1) return cache[u][acnt][lcnt];
        int ans = 0;
        ans = dfs(u - 1, acnt + 1, 0) % mod; // A
        ans = (ans + dfs(u - 1, acnt, lcnt + 1)) % mod; // L
        ans = (ans + dfs(u - 1, acnt, 0)) % mod; // P
        cache[u][acnt][lcnt] = ans;
        return ans;
    }
}
```

- 时间复杂度：有 $O(n * 2 * 3)$ 个状态需要被计算，复杂度为 $O(n)$
- 空间复杂度：$O(n)$

状态机 DP

通过记忆化搜索的分析我们发现，当我们在决策下一位是什么的时候，依赖于前面已经填入的 `A` 的个数以及当前结尾处的 `L` 的连续出现次数。

也就是说，状态 $f[u][acnt][lcnt]$ 必然被某些特定状态所更新，或者说由 $f[u][[acnt][lcnt]$ 出发，所能更新的状态是固定的。

因此这其实是一个状态机模型的 DP 问题。

根据「更新 $f[u][acnt][lcnt]$ 需要哪些状态值」还是「从 $f[u][acnt][lcnt]$ 出发，能够更新哪些状态」，我们能够写出两种方式（方向）的 DP 代码：

一类是从 $f[u][acnt][lcnt]$ 往回找所依赖的状态；一类是从 $f[u][acnt][lcnt]$ 出发往前去更新所能更新的状态值。

无论是何种方式（方向）的 DP 实现都只需搞清楚「当前位的选择对 $acnt$ 和 $lcnt$ 的影响」即可。

代码：

```java
// 从 f[u][acnt][lcnt] 往回找所依赖的状态
class Solution {
    int mod = (int)1e9+7;
    public int checkRecord(int n) {
        int[][][] f = new int[n + 1][2][3];
        f[0][0][0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    if (j == 1 && k == 0) { // A
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j - 1][0]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j - 1][1]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j - 1][2]) % mod;
                    }
                    if (k != 0) { // L
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][k - 1]) % mod;
                    }
                    if (k == 0) { // P
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][0]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][1]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][2]) % mod;
                    }
                }
            }
        }
        int ans = 0;
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 3; k++) {
                ans += f[n][j][k];
                ans %= mod;
            }
        }
        return ans;
    }
}
```

```java
// 从 f[u][acnt][lcnt] 出发往前去更新所能更新的状态值
class Solution {
    int mod = (int)1e9+7;
    public int checkRecord(int n) {
        int[][][] f = new int[n + 1][2][3];
        f[0][0][0] = 1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    if (j != 1) f[i + 1][j + 1][0] = (f[i + 1][j + 1][0] + f[i][j][k]) % m
                    if (k != 2) f[i + 1][j][k + 1] = (f[i + 1][j][k + 1] + f[i][j][k]) % m
                    f[i + 1][j][0] = (f[i + 1][j][0] + f[i][j][k]) % mod; // P
                }
            }
        }
        int ans = 0;
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 3; k++) {
                ans += f[n][j][k];
                ans %= mod;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

矩阵快速幂

之所以在动态规划解法中强调更新状态的方式（方向）是「往回」还是「往前」，是因为对于存在线性关系（同时又具有结合律）的递推式，我们能够通过「矩阵快速幂」来进行加速。

矩阵快速幂的基本分析之前在 (题解) 1137. 第 N 个泰波那契数 详细讲过。

由于 $acnt$ 和 $lcnt$ 的取值范围都很小，其组合的状态只有 $2 * 3 = 6$ 种，我们使用 $idx = acnt * 3 + lcnt$ 来代指组合（通用的二维转一维方式）：

- $idx = 0$ ：$acnt = 0$、$lcnt = 0$；
- $idx = 1$ ：$acnt = 1$、$lcnt = 0$；

...
- $idx = 5$ ：$acnt = 1$、$lcnt = 2$；

最终答案为 $ans = \sum_{idx=0}^{5} f[n][idx]$，将答案依赖的状态整理成列向量：

$$g[n] = \begin{bmatrix} f[n][0] \\ f[n][1] \\ f[n][2] \\ f[n][3] \\ f[n][4] \\ f[n][5] \end{bmatrix}$$

根据状态机逻辑，可得：

$$g[n] = \begin{bmatrix} f[n][0] \\ f[n][1] \\ f[n][2] \\ f[n][3] \\ f[n][4] \\ f[n][5] \end{bmatrix} = \begin{bmatrix} f[n-1][0]*1 + f[n-1][1]*1 + f[n-1][2]*1 + f[n-1][3]*0 + f \\ f[n-1][0]*1 + f[n-1][1]*0 + f[n-1][2]*0 + f[n-1][3]*0 + f \\ f[n-1][0]*0 + f[n-1][1]*1 + f[n-1][2]*0 + f[n-1][3]*0 + f \\ f[n-1][0]*1 + f[n-1][1]*1 + f[n-1][2]*1 + f[n-1][3]*1 + f \\ f[n-1][0]*0 + f[n-1][1]*0 + f[n-1][2]*0 + f[n-1][3]*1 + f \\ f[n-1][0]*0 + f[n-1][1]*0 + f[n-1][2]*0 + f[n-1][3]*0 + f \end{bmatrix}$$

我们令：

$$mat = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

根据「矩阵乘法」即有：

$$g[n] = mat * g[n-1]$$

起始时，我们只有 $g[0]$，根据递推式得：

$$g[n] = mat * mat * ... * mat * g[0]$$

再根据矩阵乘法具有「结合律」，最终可得：

$$g[n] = mat^n * g[0]$$

计算 mat^n 可以套用「快速幂」进行求解。

代码：

```java
class Solution {
    int N = 6;
    int mod = (int)1e9+7;
    long[][] mul(long[][] a, long[][] b) {
        int r = a.length, c = b[0].length, z = b.length;
        long[][] ans = new long[r][c];
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                for (int k = 0; k < z; k++) {
                    ans[i][j] += a[i][k] * b[k][j];
                    ans[i][j] %= mod;
                }
            }
        }
        return ans;
    }
    public int checkRecord(int n) {
        long[][] ans = new long[][]{
            {1}, {0}, {0}, {0}, {0}, {0}
        };
        long[][] mat = new long[][]{
            {1, 1, 1, 0, 0, 0},
            {1, 0, 0, 0, 0, 0},
            {0, 1, 0, 0, 0, 0},
            {1, 1, 1, 1, 1, 1},
            {0, 0, 0, 1, 0, 0},
            {0, 0, 0, 0, 1, 0}
        };
        while (n != 0) {
            if ((n & 1) != 0) ans = mul(mat, ans);
            mat = mul(mat, mat);
            n >>= 1;
        }
        int res = 0;
        for (int i = 0; i < N; i++) {
            res += ans[i][0];
            res %= mod;
        }
        return res;
    }
}
```

- 时间复杂度：$O(\log n)$
- 空间复杂度：$O(1)$

> **🍭 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 **576. 出界的路径数** ，难度为 **中等**。

Tag : 「路径 DP」、「动态规划」、「记忆化搜索」

给你一个大小为 m x n 的网格和一个球。球的起始坐标为 [startRow, startColumn] 。

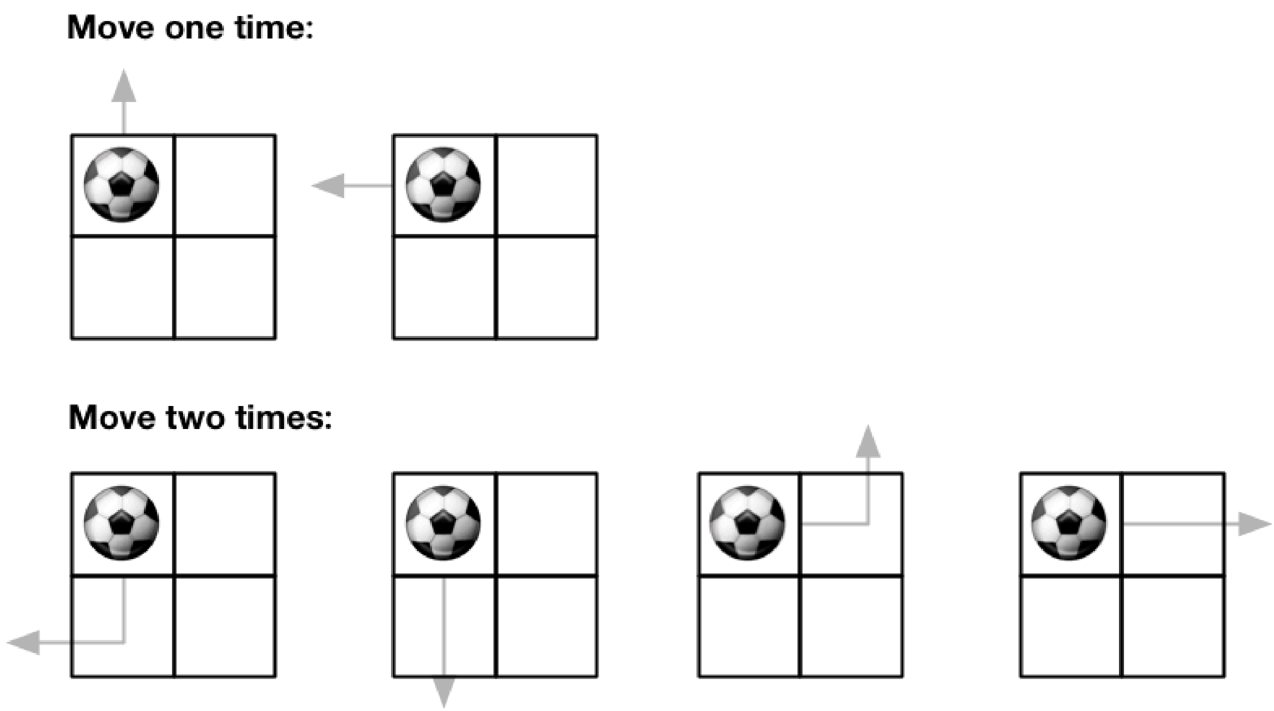
你可以将球移到在四个方向上相邻的单元格内（可以穿过网格边界到达网格之外）。

你最多可以移动 maxMove 次球。

给你五个整数 m、n、maxMove、startRow 以及 startColumn ，找出并返回可以将球移出边界的路径数量。

因为答案可能非常大，返回对 $10^9 + 7$ 取余 后的结果。

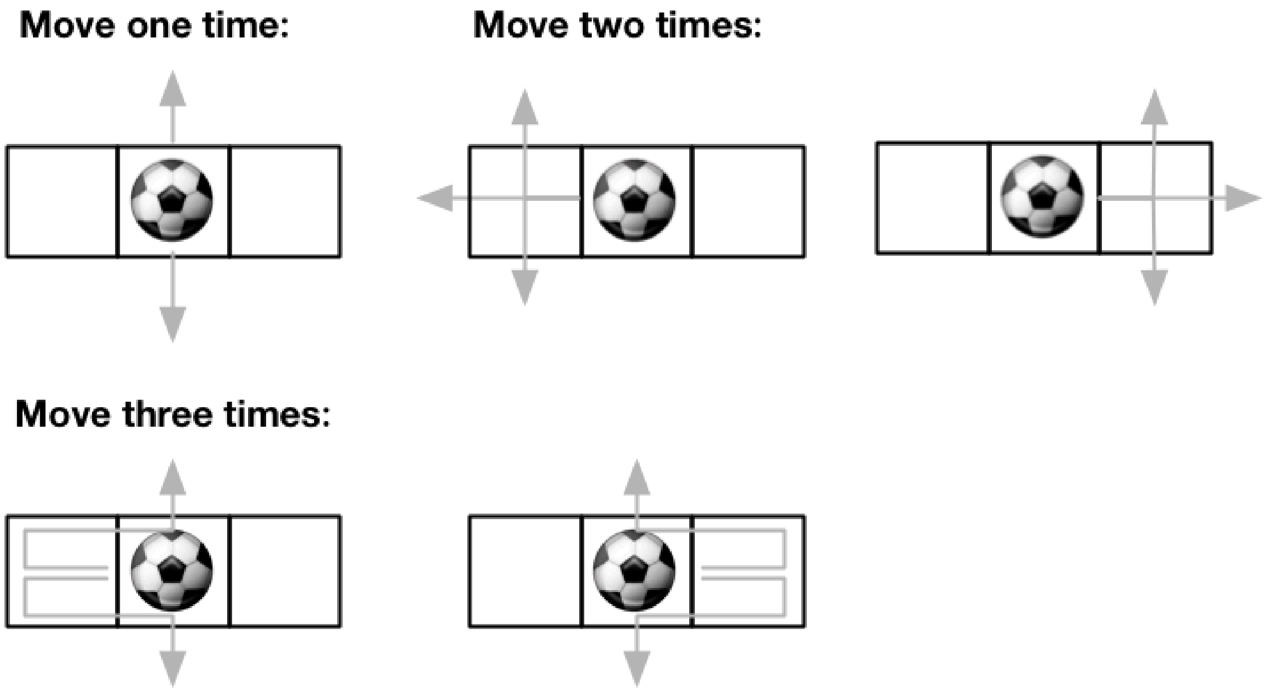
示例 1：




```
输入：m = 2, n = 2, maxMove = 2, startRow = 0, startColumn = 0

输出：6
```

示例 2：




```
输入：m = 1, n = 3, maxMove = 3, startRow = 0, startColumn = 1

输出：12
```

提示：

- 1 <= m, n <= 50
- 0 <= maxMove <= 50
- 0 <= startRow < m
- 0 <= startColumn < n

基本分析

通常来说，朴素的路径 DP 问题之所以能够使用常规 DP 方式进行求解，是因为只能往某一个方向（一维棋盘的路径问题）或者只能往某两个方向（二维棋盘的路径问题）移动。

这样的移动规则意味着，我们不会重复进入同一个格子。

从图论的意义出发：将每个格子视为点的话，如果能够根据移动规则从 `a` 位置一步到达 `b` 位

置，则说明存在一条由 `a` 指向 `b` 的有向边。

也就是说，在朴素的路径 DP 问题中，“单向”的移动规则注定了我们的图不存在环，是一个存在拓扑序的有向无环图，因此我们能够使用常规 DP 手段来求解。

回到本题，移动规则是四联通，并不是“单向”的，在某条出界的路径中，我们是有可能重复进入某个格子，即存在环。

因此我们需要换一种 DP 思路进行求解。

记忆化搜索

通常在直接 DP 不好入手的情况下，我们可以先尝试写一个「记忆化搜索」的版本。

那么如果是让你设计一个 DFS 函数来解决本题，你会如何设计？

我大概会这样设计：

```
int dfs(int x, int y, int k) {}
```

重点放在几个「可变参数」与「返回值」上：(x, y) 代表当前所在的位置，k 代表最多使用多少步，返回值代表路径数量。

根据 DP-动态规划 第八讲 的学习中，我们可以确定递归出口为：

1. 当前到达了棋盘外的位置，说明找到了一条出界路径，返回 1；
2. 在条件 1 不满足的前提下，当剩余步数为 0（不能再走下一步），说明没有找到一条合法的出界路径，返回 0。

主逻辑则是根据四联通规则进行移动即可，最终答案为 `dfs(startRow, startColumn, maxMove)`。

代码：

```java
class Solution {
    int MOD = (int)1e9+7;
    int m, n, max;
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    int[][][] cache;
    public int findPaths(int _m, int _n, int _max, int r, int c) {
        m = _m; n = _n; max = _max;
        cache = new int[m][n][max + 1];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k <= max; k++) {
                    cache[i][j][k] = -1;
                }
            }
        }
        return dfs(r, c, max);
    }
    int dfs(int x, int y, int k) {
        if (x < 0 || x >= m || y < 0 || y >= n) return 1;
        if (k == 0) return 0;
        if (cache[x][y][k] != -1) return cache[x][y][k];
        int ans = 0;
        for (int[] d : dirs) {
            int nx = x + d[0], ny = y + d[1];
            ans += dfs(nx, ny, k - 1);
            ans %= MOD;
        }
        cache[x][y][k] = ans;
        return ans;
    }
}
```

动态规划

根据我们的「记忆化搜索」，我们可以设计一个二维数组 $f[][]$ 作为我们的 dp 数组：

- **第一维代表 DFS 可变参数中的 (x, y) 所对应 $index$。取值范围为 $[0, m * n)$**
- **第二维代表 DFS 可变参数中的 k。取值范围为 $[0, max]$**

dp 数组中存储的就是我们 `DFS` 的返回值：路径数量。

根据 dp 数组中的维度设计和存储目标值，我们可以得知「状态定义」为：

$f[i][j]$ 代表从位置 i 出发，可用步数不超过 j 时的路径数量。

至此，我们只是根据「记忆化搜索」中的 `DFS` 函数的签名，就已经得出我们的「状态定义」了，接下来需要考虑「转移方程」。

当有了「状态定义」之后，我们需要从「最后一步」来推导出「转移方程」：

由于题目允许往四个方向进行移动，因此我们的最后一步也要统计四个相邻的方向。

由此可得我们的状态转移方程：

$$f[(x,y)][step] = f[(x-1,y)][step-1] + f[(x+1,y)][step-1] + f[(x,y-1)][step-1] +$$

注意，转移方程中 dp 数组的第一维存储的是 (x,y) 对应的 idx。

从转移方程中我们发现，更新 $f[i][j]$ 依赖于 $f[x][j-1]$，因此我们转移过程中需要将最大移动步数进行「从小到大」枚举。

至此，我们已经完成求解「路径规划」问题的两大步骤：「状态定义」&「转移方程」。

但这还不是所有，我们还需要一些 **有效值** 来滚动下去。

其实就是需要一些「有效值」作为初始化状态。

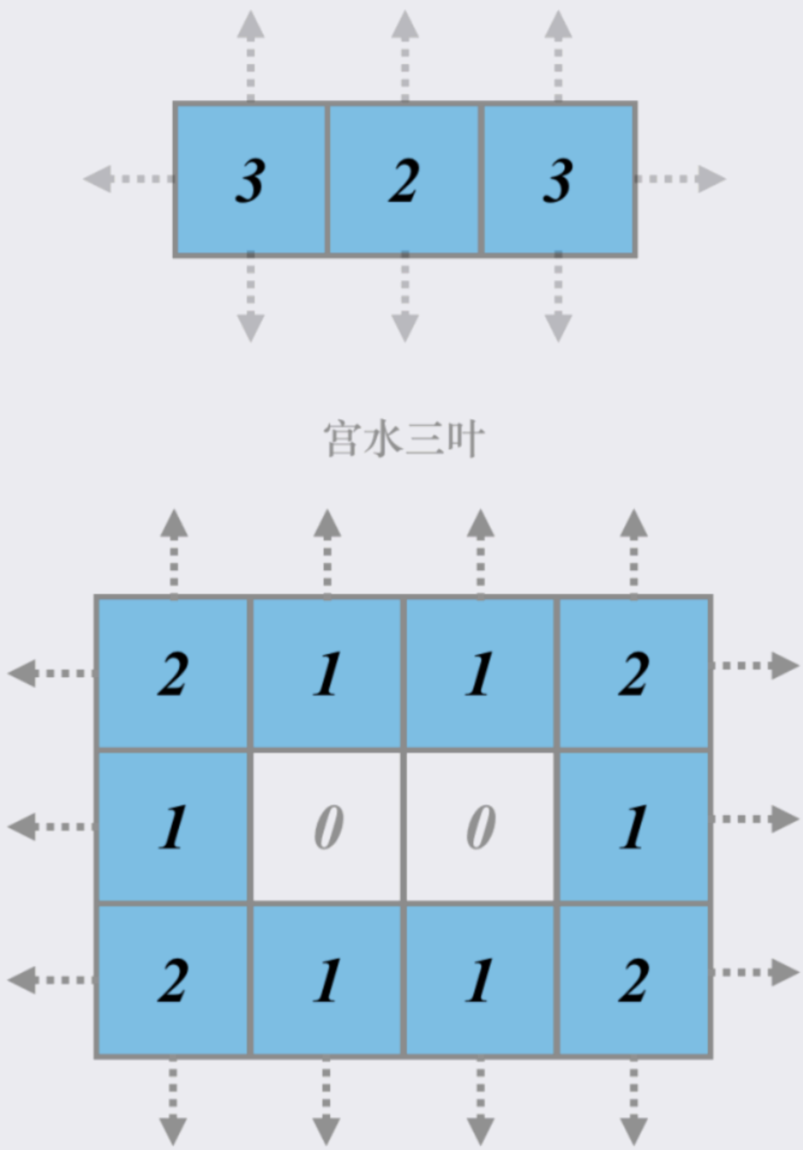
观察我们的「转移方程」可以发现，整个转移过程是一个累加过程，如果没有一些有效的状态（非零值）进行初始化的话，整个递推过程并没有意义。

那么哪些值可以作为成为初始化状态呢？

显然，当我们已经位于矩阵边缘的时候，我们可以一步跨出矩阵，这算作一条路径。

同时，由于我们能够往四个方向进行移动，因此不同的边缘格子会有不同数量的路径。

目的是为了我们整个 DP 过程可以有效的递推下去。



每个格子内的数字代表从该位置走出矩阵的路径数量

换句话说，我们需要先对边缘格子进行初始化操作，预处理每个边缘格子直接走出矩阵的路径数量。

目的是为了我们整个 DP 过程可以有效的递推下去。

可以发现，动态规划的实现，本质是将问题进行反向：原问题是让我们求从棋盘的特定位置出

发，出界的路径数量。实现时，我们则是从边缘在状态出发，逐步推导回起点的出界路径数量为多少。

代码：

```java
class Solution {
    int MOD = (int)1e9+7;
    int m, n, max;
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    public int findPaths(int _m, int _n, int _max, int r, int c) {
        m = _m; n = _n; max = _max;
        int[][] f = new int[m * n][max + 1];
        // 初始化边缘格子的路径数量
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i == 0) add(i, j, f);
                if (j == 0) add(i, j, f);
                if (i == m - 1) add(i, j, f);
                if (j == n - 1) add(i, j, f);
            }
        }
        // 从小到大枚举「可移动步数」
        for (int k = 1; k <= max; k++) {
            // 枚举所有的「位置」
            for (int idx = 0; idx < m * n; idx++) {
                int[] info = parseIdx(idx);
                int x = info[0], y = info[1];
                for (int[] d : dirs) {
                    int nx = x + d[0], ny = y + d[1];
                    if (nx < 0 || nx >= m || ny < 0 || ny >= n) continue;
                    int nidx = getIdx(nx, ny);
                    f[idx][k] += f[nidx][k - 1];
                    f[idx][k] %= MOD;
                }
            }
        }
        return f[getIdx(r, c)][max];
    }
    void add(int x, int y, int[][] f) {
        for (int k = 1; k <= max; k++) {
            f[getIdx(x, y)][k]++;
        }
    }
    int getIdx(int x, int y) {
        return x * n + y;
    }
    int[] parseIdx(int idx) {
        return new int[]{idx / n, idx % n};
    }
}
```

- 时间复杂度：共有 $m * n * max$ 个状态需要转移，复杂度为 $O(m * n * max)$
- 空间复杂度：$O(m * n * max)$

> **🍭 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 **1137. 第 N 个泰波那契数** ，难度为 **简单**。

Tag : 「动态规划」、「递归」、「递推」、「矩阵快速幂」、「打表」

泰波那契序列 Tn 定义如下：

T0 = 0, T1 = 1, T2 = 1, 且在 n >= 0 的条件下 Tn+3 = Tn + Tn+1 + Tn+2

给你整数 n，请返回第 n 个泰波那契数 T_n 的值。

示例 1：

```
输入：n = 4

输出：4

解释：
T_3 = 0 + 1 + 1 = 2
T_4 = 1 + 1 + 2 = 4
```

示例 2：

```
输入：n = 25

输出：1389537
```

提示：

- 0 <= n <= 37
- 答案保证是一个 32 位整数，即 answer <= 2^{31} - 1。

迭代实现动态规划

都直接给出状态转移方程了，其实就是道模拟题。

使用三个变量，从前往后算一遍即可。

代码：

```
class Solution {
    public int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        int a = 0, b = 1, c = 1;
        for (int i = 3; i <= n; i++) {
            int d = a + b + c;
            a = b;
            b = c;
            c = d;
        }
        return c;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

递归实现动态规划

也就是记忆化搜索，创建一个 `cache` 数组用于防止重复计算。

代码：

```java
class Solution {
    int[] cache = new int[40];
    public int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        if (cache[n] != 0) return cache[n];
        cache[n] = tribonacci(n - 1) + tribonacci(n - 2) + tribonacci(n - 3);
        return cache[n];
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

矩阵快速幂

这还是一道「矩阵快速幂」的板子题。

首先你要对「快速幂」和「矩阵乘法」概念有所了解。

矩阵快速幂用于求解一般性问题：给定大小为 $n * n$ 的矩阵 M，求答案矩阵 M^k，并对答案矩阵中的每位元素对 P 取模。

在上述两种解法中，当我们要求解 $f[i]$ 时，需要将 $f[0]$ 到 $f[n-1]$ 都算一遍，因此需要线性的复杂度。

对于此类的「数列递推」问题，我们可以使用「矩阵快速幂」来进行加速（比如要递归一个长度为 $1e9$ 的数列，线性复杂度会被卡）。

使用矩阵快速幂，我们只需要 $O(\log n)$ 的复杂度。

根据题目的递推关系（$i >= 3$）：

$$f(i) = f(i-1) + f(i-2) + f(i-3)$$

我们发现要求解 $f(i)$，其依赖的是 $f(i-1)$、$f(i-2)$ 和 $f(i-3)$。

我们可以将其存成一个列向量：

$$\begin{bmatrix} f(i-1) \\ f(i-2) \\ f(i-3) \end{bmatrix}$$

当我们整理出依赖的列向量之后，不难发现，我们想求的 $f(i)$ 所在的列向量是这样的：

$$\begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix}$$

利用题目给定的依赖关系，对目标矩阵元素进行展开：

$$\begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix} = \begin{bmatrix} f(i-1) * 1 + f(i-2) * 1 + f(i-3) * 1 \\ f(i-1) * 1 + f(i-2) * 0 + f(i-3) * 0 \\ f(i-1) * 0 + f(i-2) * 1 + f(i-3) * 0 \end{bmatrix}$$

那么根据矩阵乘法，即有：

$$\begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} f(i-1) \\ f(i-2) \\ f(i-3) \end{bmatrix}$$

我们令

$$Mat = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

然后发现，利用 Mat 我们也能实现数列递推（公式太难敲了，随便列两项吧）：

$$Mat * \begin{bmatrix} f(i-1) \\ f(i-2) \\ f(i-3) \end{bmatrix} = \begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix}$$

$$Mat * \begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i) \\ f(i-1) \end{bmatrix}$$

再根据矩阵运算的结合律，最终有：

$$\begin{bmatrix} f(n) \\ f(n-1) \\ f(n-2) \end{bmatrix} = Mat^{n-2} * \begin{bmatrix} f(2) \\ f(1) \\ f(0) \end{bmatrix}$$

从而将问题转化为求解 Mat^{n-2} ，这时候可以套用「矩阵快速幂」解决方案。

代码：

```
class Solution {
    int N = 3;
    int[][] mul(int[][] a, int[][] b) {
        int[][] c = new int[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j] + a[i][2] * b[2][j];
            }
        }
        return c;
    }
    public int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        int[][] ans = new int[][]{
            {1,0,0},
            {0,1,0},
            {0,0,1}
        };
        int[][] mat = new int[][]{
            {1,1,1},
            {1,0,0},
            {0,1,0}
        };
        int k = n - 2;
        while (k != 0) {
            if ((k & 1) != 0) ans = mul(ans, mat);
            mat = mul(mat, mat);
            k >>= 1;
        }
        return ans[0][0] + ans[0][1];
    }
}
```

- 时间复杂度：$O(\log n)$
- 空间复杂度：$O(1)$

打表

当然，我们也可以将数据范围内的所有答案进行打表预处理，然后在询问时直接查表返回。

但对这种题目进行打表带来的收益没有平常打表题的大，因为打表内容不是作为算法必须的一个环节，而直接是作为该询问的答案，但测试样例是不会相同的，即不会有两个测试数据都是 $n = 37$。

这时候打表节省的计算量是不同测试数据之间的相同前缀计算量，例如 $n = 36$ 和 $n = 37$，其 35 之前的计算量只会被计算一次。

因此直接为「解法二」的 `cache` 添加 `static` 修饰其实是更好的方式：代码更短，同时也能起到同样的节省运算量的效果。

代码：

```
class Solution {
    static int[] cache = new int[40];
    static {
        cache[0] = 0;
        cache[1] = 1;
        cache[2] = 1;
        for (int i = 3; i < cache.length; i++) {
            cache[i] = cache[i - 1] + cache[i - 2] + cache[i - 3];
        }
    }
    public int tribonacci(int n) {
        return cache[n];
    }
}
```

- 时间复杂度：将打表逻辑交给 OJ，复杂度为 $O(C)$，C 固定为 40。将打表逻辑放到本地进行，复杂度为 $O(1)$
- 空间复杂度：$O(n)$

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

💡**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「记忆化搜索」获取下载链接。

觉得专题不错，可以请作者吃糖🍭🍭🍭：

版权声明：任何形式的转载请保留出处 Wiki。