宫水三叶的刷题日记

线性 DP

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「线性 DP」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「线性 DP」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「线性 DP」；
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 **10. 正则表达式匹配** ，难度为 **困难**。

Tag : 「动态规划」

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 ‘.’ 和 ‘*’ 的正则表达式匹配。

- ‘.’ 匹配任意单个字符
- ‘*’ 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

示例 1：

```
输入：s = "aa" p = "a"
输出：false
解释："a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入：s = "aa" p = "a*"
输出：true
解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次
```

示例 3：

```
输入：s = "ab" p = ".*"
输出：true
解释：".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
```

示例 4：

```
输入：s = "aab" p = "c*a*b"
输出：true
解释：因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。
```

示例 5：

```
输入：s = "mississippi" p = "mis*is*p*."
输出：false
```

提示：

- 0 <= s.length <= 20
- 0 <= p.length <= 30
- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 . 和 *。
- 保证每次出现字符 * 时，前面都匹配到有效的字符

动态规划

整理一下题意，对于字符串 `p` 而言，有三种字符：

- 普通字符：需要和 `s` 中同一位置的字符完全匹配
- `'.'`：能够匹配 `s` 中同一位置的任意字符
- `'*'`：不能够单独使用 `'*'`，必须和前一个字符同时搭配使用，数据保证了 `'*'` 能够找到前面一个字符。能够匹配 `s` 中同一位置字符任意次。

所以本题关键是分析当出现 `a*` 这种字符时，是匹配 0 个 a、还是 1 个 a、还是 2 个 a ...

本题可以使用动态规划进行求解：

- 状态定义：`f(i,j)` 代表考虑 `s` 中以 `i` 为结尾的子串和 `p` 中的 `j` 为结尾的子串是否匹配。即最终我们要求的结果为 `f[n][m]` 。
- 状态转移：也就是我们要考虑 `f(i,j)` 如何求得，前面说到了 `p` 有三种字符，所以这里的状态转移也要分三种情况讨论：
    1. `p[j]` 为普通字符：匹配的条件是前面的字符匹配，同时 `s` 中的第 `i` 个字符和 `p` 中的第 `j` 位相同。 即 `f(i,j) = f(i - 1, j - 1) && s[i] == p[j]` 。
    2. `p[j]` 为 `'.'`：匹配的条件是前面的字符匹配， `s` 中的第 `i` 个字符可以是任意字符。即 `f(i,j) = f(i - 1, j - 1) && p[j] == '.'` 。
    3. `p[j]` 为 `'*'`：读得 `p[j - 1]` 的字符，例如为字符 a。 然后根据 `a*` 实际匹配 `s` 中 `a` 的个数是 0 个、1 个、2 个 ...
    3.1. 当匹配为 0 个：`f(i,j) = f(i, j - 2)`
    3.2. 当匹配为 1 个：`f(i,j) = f(i - 1, j - 2) && (s[i] == p[j - 1] || p[j - 1] == '.')`
    3.3. 当匹配为 2 个：`f(i,j) = f(i - 2, j - 2) && ((s[i] == p[j - 1] && s[i - 1] == p[j -`
    ...

我们知道，通过「枚举」来确定 `*` 到底匹配多少个 `a` 这样的做法，算法复杂度是很高的。

我们需要挖掘一些「性质」来简化这个过程。

表达式展开可得：

f(i, j) = f(i, j - 2) || (f(i - 1, j - 2) && s[i] 匹配 p[j - 1]) || (f(i - 2, j - 2) && s[i - 1:i] 匹配 p[j - 1])...

将 i = i - 1 代入表达式可得：

f(i - 1, j) = f(i - 1, j - 2) || (f(i - 2, j - 2) && s[i - 1] 匹配 p[j - 1]) || (f(i - 3, j - 2) && s[i - 2:i] 匹配 p[j - 1])...

每个 *item* 都相差了 *s[i] 匹配 p[j - 1]* ，也就是 *f(i - 1, j)* 和 *f(i,j)* 整体相差了*s[i] 匹配 p[j - 1]*

f(i, j) = f(i, j - 2) || (f(i - 1, j) && s[i] 匹配 p[j - 1])

f(i, j) = f(i, j - 2) || (f(i - 1, j) && (s[i] == p[j - 1] || p[j - 1] == '.'))

代码：

```Java
class Solution {
    public boolean isMatch(String ss, String pp) {
        // 技巧：往原字符头部插入空格，这样得到 char 数组是从 1 开始，而且可以使得 f[0][0] = true，可
        int n = ss.length(), m = pp.length();
        ss = " " + ss;
        pp = " " + pp;
        char[] s = ss.toCharArray();
        char[] p = pp.toCharArray();
        // f(i,j) 代表考虑 s 中的 1~i 字符和 p 中的 1~j 字符 是否匹配
        boolean[][] f = new boolean[n + 1][m + 1];
        f[0][0] = true;
        for (int i = 0; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                // 如果下一个字符是 '*'，则代表当前字符不能被单独使用，跳过
                if (j + 1 <= m && p[j + 1] == '*') continue;

                // 对应了 p[j] 为普通字符和 '.' 的两种情况
                if (i - 1 >= 0 && p[j] != '*') {
                    f[i][j] = f[i - 1][j - 1] && (s[i] == p[j] || p[j] == '.');
                }

                // 对应了 p[j] 为 '*' 的情况
                else if (p[j] == '*') {
                    f[i][j] = (j - 2 >= 0 && f[i][j - 2]) || (i - 1 >= 0 && f[i - 1][j] &&
                }
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：n 表示 s 的长度，m 表示 p 的长度，总共 $n * m$ 个状态。复杂度为 $O(n * m)$
- 空间复杂度：使用了二维数组记录结果。复杂度为 $O(n * m)$

动态规划本质上是枚举（不重复的暴力枚举），因此其复杂度很好分析，有多少个状态就要被计算多少次，复杂度就为多少。

题目描述

这是 LeetCode 上的 **44. 通配符匹配** ，难度为 **困难**。

Tag : 「线性 DP」

给定一个字符串 (s) 和一个字符模式 § ，实现一个支持 ‘?’ 和 ‘*’ 的通配符匹配。

- ‘?’ 可以匹配任何单个字符。
- ‘*’ 可以匹配任意字符串（包括空字符串）。
  两个字符串完全匹配才算匹配成功。

说明:

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 *。

示例 1:

```
输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入:
s = "aa"
p = "*"
输出: true
解释: '*' 可以匹配任意字符串。
```

示例 3:

```
输入:
s = "cb"
p = "?a"
输出: false
解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。
```

示例 4:

```
输入:
s = "adceb"
p = "*a*b"
输出: true
解释:
第一个 '*' 可以匹配空字符串,
第二个 '*' 可以匹配字符串 "dce".
```

示例 5:

```
输入:
s = "acdcb"
p = "a*c?b"
输出: false
```

动态规划

这道题与 10. 正则表达式匹配 的分析思路是类似的。

但和第 10 题相比，本题要简单一些。

整理一下题意，对于字符串 `p` 而言，有三种字符：

- 普通字符：需要和 `s` 中同一位置的字符完全匹配
- `'?'`：能够匹配 `s` 中同一位置的任意字符
- `'*'`：能够匹配任意字符串

所以本题关键是分析当出现 `'*'` 这种字符时，是匹配 0 个字符、还是 1 个字符、还是 2 个字符 ...

本题可以使用动态规划进行求解：

- 状态定义：`f(i,j)` 代表考虑 `s` 中以 `i` 为结尾的子串和 `p` 中的 `j` 为结尾的子串是否匹配。即最终我们要求的结果为 `f[n][m]` 。
- 状态转移：也就是我们要考虑 `f(i,j)` 如何求得，前面说到了 `p` 有三种字符，所以这里的状态转移也要分三种情况讨论：

1. `p[j]` 为普通字符：匹配的条件是前面的字符匹配，同时 `s` 中的第 `i` 个字符和 `p` 中的第 `j` 位相同。 即 `f(i,j) = f(i - 1, j - 1) && s[i] == p[j]` 。
2. `p[j]` 为 `'.'`：匹配的条件是前面的字符匹配，`s` 中的第 `i` 个字符可以是任意字符。即 `f(i,j) = f(i - 1, j - 1) && p[j] == '.'`。
3. `p[j]` 为 `'*'`：可匹配任意长度的字符，可以匹配 0 个字符、匹配 1 个字符、匹配 2 个字符
   3.1. 当匹配为 0 个：`f(i,j) = f(i, j - 1)`
   3.2. 当匹配为 1 个：`f(i,j) = f(i - 1, j - 1)`
   3.3. 当匹配为 2 个：`f(i,j) = f(i - 2, j - 1)`
   ...
   3.k. 当匹配为 k 个：`f(i,j) = f(i - k, j - 1)`

因此对于 `p[j] = '*'` 的情况，想要 `f(i, j) = true` ，只需要其中一种情况为 `true` 即可。也就是状态之间是「或」的关系：

$$f[i][j] = f[i][j-1] || f[i-1][j-1] || ... || f[i-k][j-1] (i >= k)$$

这意味着我们要对 `k` 种情况进行枚举检查吗？

其实并不用，对于这类问题，我们通常可以通过「代数」进简化，将 `i - 1` 代入上述的式子：

$$f[i-1][j] = f[i-1][j-1] || f[i-2][j-1] || ... || f[i-k][j-1] (i >= k)$$

可以发现，`f[i - 1][j]` 与 `f[i][j]` 中的 `f[i][j - 1]` 开始的后半部分是一样的，因此有：

$$f[i][j] = f[i][j-1] || f[i-1][j] (i >= 1)$$

PS. 其实类似的推导，我在 10. 正则表达式匹配 也做过，第 10 题的推导过程还涉及等差概念，我十分推荐你去回顾一下。如果你能搞懂第 10 题整个过程，这题其实就是小 Case。

编码细节：

1. 通过上述的推导过程，你会发现设计不少的「回退检查」操作（即遍历到 `i` 位，要回头检查 `i - 1` 等），因此我们可以将「哨兵技巧」应用到本题，往两个字符串的头部插入哨兵
2. 对于 `p[j] = '.'` 和 `p[j] = 普通字符` 的情况，想要为 `true` ，其实有共同的条件 `f[i - 1][j - 1] == true` ，因此可以合到一起来做

代码：

```
class Solution {
    public boolean isMatch(String ss, String pp) {
        int n = ss.length(), m = pp.length();
        // 技巧：往原字符头部插入空格，这样得到 char 数组是从 1 开始，而且可以使得 f[0][0] = true，可
        ss = " " + ss;
        pp = " " + pp;
        char[] s = ss.toCharArray();
        char[] p = pp.toCharArray();
        // f(i,j) 代表考虑 s 中的 1~i 字符和 p 中的 1~j 字符 是否匹配
        boolean[][] f = new boolean[n + 1][m + 1];
        f[0][0] = true;
        for (int i = 0; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (p[j] == '*') {
                    f[i][j] = f[i][j - 1] || (i - 1 >= 0 && f[i - 1][j]);
                } else {
                    f[i][j] = i - 1 >= 0 && f[i - 1][j - 1] && (s[i] == p[j] || p[j] == '?
                }
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：`n` 表示 `s` 的长度，`m` 表示 `p` 的长度，总共 `n * m` 个状态。复杂度为 $O(n*m)$
- 空间复杂度：使用了二维数组记录结果。复杂度为 $O(n*m)$

再次强调，动态规划本质上是枚举（不重复的暴力枚举），因此其复杂度很好分析，有多少个状态就要被计算多少次，复杂度就为多少。

题目描述

这是 LeetCode 上的 **45. 跳跃游戏 II** ，难度为 **中等**。

Tag：「贪心」、「线性 DP」、「双指针」

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。
     从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
```

示例 2:

```
输入: [2,3,0,1,4]

输出: 2
```

提示:

- 1 <= nums.length <= 1000
- 0 <= nums[i] <= 10^5

BFS

对于这一类问题，我们一般都是使用 BFS 进行求解。

本题的 BFS 解法的复杂度是 $O(n^2)$，数据范围为 10^3，可以过。

代码：

```java
class Solution {
    public int jump(int[] nums) {
        int n = nums.length;
        int ans = 0;
        boolean[] st = new boolean[n];
        Deque<Integer> d = new ArrayDeque<>();
        st[0] = true;
        d.addLast(0);
        while (!d.isEmpty()) {
            int size = d.size();
            while (size-- > 0) {
                int idx = d.pollFirst();
                if (idx == n - 1) return ans;
                for (int i = idx + 1; i <= idx + nums[idx] && i < n; i++) {
                    if (!st[i]) {
                        st[i] = true;
                        d.addLast(i);
                    }
                }
            }
            ans++;
        }
        return ans;
    }
}
```

- 时间复杂度：如果每个点跳跃的距离足够长的话，每次都会将当前点「后面的所有点」进行循环入队操作（由于 st 的存在，不一定都能入队，但是每个点都需要被循环一下）。复杂度为 $O(n^2)$
- 空间复杂度：队列中最多有 $n-1$ 个元素。复杂度为 $O(n)$

双指针 + 贪心 + 动态规划

本题数据范围只有 10^3，所以 $O(n^2)$ 勉强能过。

如果面试官要将数据范围出到 10^6，又该如何求解呢？

我们需要考虑 $O(n)$ 的做法。

其实通过 10^6 这个数据范围，就已经可以大概猜到是道 DP 题。

我们定义 $f[i]$ 为到达第 i 个位置所需要的最少步数，那么答案是 $f[n-1]$。

学习过 路径 DP 专题 的同学应该知道，通常确定 DP 的「状态定义」有两种方法。

- 一种是根据经验猜一个状态定义，会结合题目给定的维度，和要求的答案去猜。
- 另外一种则是通过设计一个合理的 `DFS` 方法签名来确定状态定义。

这里我是采用第一种方法。

至于如何确定「状态定义」是否可靠，关键是看使用这个状态定义能否推导出合理的「状态转移方程」，来覆盖我们所有的状态。

不失一般性的考虑 $f[n-1]$ 该如何转移：

我们知道最后一个点前面可能会有很多个点能够一步到达最后一个点。

宫水三叶

也就是有 $f[n-1] = min(f[n-k], ..., f[n-3], f[n-2]) + 1$。

然后我们再来考虑集合 $f[n-k], ..., f[n-3], f[n-2]$ 有何特性。

不然发现其实必然有 $f[n-k] <= ... <= f[n-3] <= f[n-2]$。

推而广之，不止是经过一步能够到达最后一个点的集合，其实任意连续的区间都有这个性质。

举个🌰，比如我经过至少 5 步到达第 i 个点，那么必然不可能出现使用步数少于 5 步就能达到第 $i+1$ 个点的情况。到达第 $i+1$ 个点的至少步数必然是 5 步或者 6 步。

搞清楚性质之后，再回头看我们的状态定义：$f[i]$ *为到达第 i 个位置所需要的最少步数。*

因此当我们要求某一个 $f[i]$ 的时候，我们需要找到最早能够经过一步到达 i 点的 j 点。

即有状态转移方程：$f[i] = f[j] + 1$。

也就是我们每次都贪心的取离 i 点最远的点 j 来更新 $f[i]$。

而这个找 j 的过程可以使用双指针来找。

因此这个思路其实是一个「双指针 + 贪心 + 动态规划」的一个解法。

代码：

```
class Solution {
    public int jump(int[] nums) {
        int n = nums.length;
        int[] f = new int[n];
        for (int i = 1, j = 0; i < n; i++) {
            while (j + nums[j] < i) j++;
            f[i] = f[j] + 1;
        }
        return f[n - 1];
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

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

题目描述

这是 LeetCode 上的 **91. 解码方法** ，难度为 **中等**。

Tag：「线性 DP」

一条包含字母 A-Z 的消息通过以下映射进行了 编码 ：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

要 解码 已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，“11106” 可以映射为：

- “AAJF” ，将消息分组为 (1 1 10 6)
- “KJF” ，将消息分组为 (11 10 6)

注意，消息不能分组为 (1 11 06) ，因为 “06” 不能映射为 “F” ，这是由于 “6” 和 “06” 在映射中并不等价。

给你一个只含数字的 非空 字符串 s ，请计算并返回 解码 方法的 总数 。

题目数据保证答案肯定是一个 32 位 的整数。

示例 1：

```
输入：s = "12"

输出：2

解释：它可以解码为 "AB"（1 2）或者 "L"（12）。
```

示例 2：

```
输入：s = "226"

输出：3

解释：它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。
```

示例 3：

输入：s = "0"

输出：0

解释：没有字符映射到以 0 开头的数字。
含有 0 的有效映射是 'J' -> "10" 和 'T'-> "20" 。
由于没有字符，因此没有有效的方法对此进行解码，因为所有数字都需要映射。

示例 4：

输入：s = "06"

输出：0

解释："06" 不能映射到 "F" ，因为字符串含有前导 0（"6" 和 "06" 在映射中并不等价）。

提示：

- 1 <= s.length <= 100
- s 只包含数字，并且可能包含前导零。

基本分析

我们称一个解码内容为一个 `item`。

为根据题意，每个 `item` 可以由一个数字组成，也可以由两个数字组成。

数据范围为 100，很具有迷惑性，可能会有不少同学会想使用 DFS 进行爆搜。

我们可以大致分析一下这样的做法是否可行：不失一般性的考虑字符串 `s` 中的任意位置 `i`，位置 `i` 既可以作为一个独立 `item`，也可以与上一位置组成新 `item`，那么相当于每个位置都有两种分割选择（先不考虑分割结果的合法性问题），这样做法的复杂度是 $O(2^n)$ 的，当 `n` 范围是 100 时，远超我们计算机单秒运算量（10^7）。即使我们将「判断分割结果是否合法」的操作放到爆搜过程中做剪枝，也与我们的单秒最大运算量相差很远。

递归的方法不可行，我们需要考虑递推的解法。

动态规划

这其实是一道字符串类的动态规划题，不难发现对于字符串 `s` 的某个位置 `i` 而言，我们只关心「位置 `i` 自己能否形成独立 `item` 」和「位置 `i` 能够与上一位置（ `i-1` ）能否形成 `item` 」，而不关心 `i-1` 之前的位置。

有了以上分析，我们可以从前往后处理字符串 `s` ，使用一个数组记录以字符串 `s` 的每一位作为结尾的解码方案数。即定义 $f[i]$ 为考虑前 i 个字符的解码方案数。

对于字符串 `s` 的任意位置 `i` 而言，其存在三种情况：

- 只能由位置 `i` 的单独作为一个 `item` ，设为 `a` ，转移的前提是 `a` 的数值范围为 $[1, 9]$，转移逻辑为 $f[i] = f[i - 1]$。
- 只能由位置 `i` 的与前一位置（ `i-1` ）共同作为一个 `item` ，设为 `b` ，转移的前提是 `b` 的数值范围为 $[10, 26]$，转移逻辑为 $f[i] = f[i - 2]$。
- 位置 `i` 既能作为独立 `item` 也能与上一位置形成 `item` ，转移逻辑为 $f[i] = f[i - 1] + f[i - 2]$。

因此，我们有如下转移方程：

$$
\begin{cases}

f[i] = f[i - 1], 1 \leqslant a \leq 9 \\

f[i] = f[i - 2], 10 \leqslant b \leqslant 26 \\

f[i] = f[i - 1] + f[i - 2], 1 \leqslant a \leq 9, 10 \leqslant b \leqslant 26 \\

\end{cases}
$$

其他细节：由于题目存在前导零，而前导零属于无效 `item` 。可以进行特判，但个人习惯往字符串头部追加空格作为哨兵，追加空格既可以避免讨论前导零，也能使下标从 1 开始，简化 `f[i-1]` 等负数下标的判断。

代码：

```Java
class Solution {
    public int numDecodings(String s) {
        int n = s.length();
        s = " " + s;
        char[] cs = s.toCharArray();
        int[] f = new int[n + 1];
        f[0] = 1;
        for (int i = 1; i <= n; i++) {
            // a : 代表「当前位置」单独形成 item
            // b : 代表「当前位置」与「前一位置」共同形成 item
            int a = cs[i] - '0', b = (cs[i - 1] - '0') * 10 + (cs[i] - '0');
            // 如果 a 属于有效值，那么 f[i] 可以由 f[i - 1] 转移过来
            if (1 <= a && a <= 9) f[i] = f[i - 1];
            // 如果 b 属于有效值，那么 f[i] 可以由 f[i - 2] 或者 f[i - 1] & f[i - 2] 转移过来
            if (10 <= b && b <= 26) f[i] += f[i - 2];
        }
        return f[n];
    }
}
```

- 时间复杂度：共有 `n` 个状态需要被转移。复杂度为 $O(n)$。
- 空间复杂度：$O(n)$。

空间优化

不难发现，我们转移 `f[i]` 时只依赖 `f[i-1]` 和 `f[i-2]` 两个状态。

因此我们可以采用与「滚动数组」类似的思路，只创建长度为 3 的数组，通过取余的方式来复用不再需要的下标。

代码：

```java
class Solution {
    public int numDecodings(String s) {
        int n = s.length();
        s = " " + s;
        char[] cs = s.toCharArray();
        int[] f = new int[3];
        f[0] = 1;
        for (int i = 1; i <= n; i++) {
            f[i % 3] = 0;
            int a = cs[i] - '0', b = (cs[i - 1] - '0') * 10 + (cs[i] - '0');
            if (1 <= a && a <= 9) f[i % 3] = f[(i - 1) % 3];
            if (10 <= b && b <= 26) f[i % 3] += f[(i - 2) % 3];
        }
        return f[n % 3];
    }
}
```

- 时间复杂度：共有 `n` 个状态需要被转移。复杂度为 $O(n)$。
- 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 **115. 不同的子序列** ，难度为 **困难**。

Tag：「线性 DP」

给定一个字符串 s 和一个字符串 t ，计算在 s 的子序列中 t 出现的个数。

字符串的一个 子序列 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，“ACE” 是 “ABCDE” 的一个子序列，而 “AEC” 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

```
输入：s = "rabbbit", t = "rabbit"
输出：3
解释：
如下图所示, 有 3 种可以从 s 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)
rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
```

示例 2：

```
输入：s = "babgbag", t = "bag"
输出：5
解释：
如下图所示, 有 5 种可以从 s 中得到 "bag" 的方案。
(上箭头符号 ^ 表示选取的字母)
babgbag
^^ ^
babgbag
^^    ^
babgbag
^    ^^
babgbag
  ^  ^^
babgbag
    ^^^
```

提示：

- 0 <= s.length, t.length <= 1000
- s 和 t 由英文字母组成

基本思路

有两个字符串 `s` 和 `t`，长度数量级都为 10^3。

一个朴素的想法是，找出所有 `s` 的子序列，与 `t` 进行比较，找所有子序列的复杂度是 $O(2^n)$，肯定会超时。

因此，我们放弃这种朴素思路。

字符串匹配也不具有二段性质，不可能有 log 级别的算法，那么复杂度再往下优化就是 $O(n * m)$ 的递推 DP 做法了。

动态规划

DP 的状态定义猜测通常是一门经验学科。

但是，对于两个字符串匹配，一个非常通用的状态定义如下：

定义 $f[i][j]$ 为考虑 `s` 中 $[0,i]$ 个字符，`t` 中 $[0,j]$ 个字符的匹配个数。

那么显然对于某个 $f[i][j]$ 而言，从「最后一步」的匹配进行分析，包含两类决策：

- 不让 `s[i]` 参与匹配，也就是需要让 `s` 中 $[0,i-1]$ 个字符去匹配 `t` 中的 $[0,j]$ 字符。此时匹配值为 $f[i-1][j]$
- 让 `s[i]` 参与匹配，这时候只需要让 `s` 中 $[0,i-1]$ 个字符去匹配 `t` 中的 $[0,j-1]$ 字符即可，同时满足 `s[i]=t[j]`。此时匹配值为 $f[i-1][j-1]$

最终 $f[i][j]$ 就是两者之和。

代码：

```java
class Solution {
    public int numDistinct(String s, String t) {
        // 技巧：往原字符头部插入空格，这样得到 char 数组是从 1 开始
        // 同时由于往头部插入相同的（不存在的）字符，不会对结果造成影响，而且可以使得 f[i][0] = 1，可以
        int n = s.length(), m = t.length();
        s = " " + s;
        t = " " + t;
        char[] cs = s.toCharArray(), ct = t.toCharArray();
        // f(i,j) 代表考虑「s 中的下标为 0~i 字符」和「t 中下标为 0~j 字符」是否匹配
        int[][] f = new int[n + 1][m + 1];
        // 原字符只有小写字符，当往两个字符插入空格之后，f[i][0] = 1 是一个显而易见的初始化条件
        for (int i = 0; i <= n; i++) f[i][0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                // 包含两种决策：
                // 不使用 cs[i] 进行匹配，则有 f[i][j] = f[i - 1][j]
                f[i][j] = f[i - 1][j];
                // 使用 cs[i] 进行匹配，则要求 cs[i] == ct[j]，然后有  f[i][j] += f[i - 1][j -
                if (cs[i] == ct[j]) {
                    f[i][j] += f[i - 1][j - 1];
                }
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：$O(n * m)$
- 空间复杂度：$O(n * m)$

PS. 需要说明的是，由于中间结果会溢出，CPP 中必须使用 long long，而 Java 不用。由于 Java 中 int 的存储机制，只要在运算过程中只要不涉及取 min、取 max 或者其他比较操作的话，中间结果溢出不会影响最终结果。

总结

1. 关于字符串匹配，通常有两种（你也可以理解为一种）通用的状态定义：

- $f[i][j]$ 表示「第一个字符 `s` 中 $[0, i]$ 个字符」与「第二个字符 `t` 中 $[0, j]$ 个字符」的匹配结果
- $f[i][j]$ 表示「第一个字符 `s` 中 $[0, i]$ 个字符」与「第二个字符 `t` 中 $[0, j]$ 个字

符」且「最后一个字符为 `t[j]` 」的匹配结果

2. 往两个字符串的头部追加「不存在」的字符，目的是为了能够构造出可以滚动（被累加）下去的初始化值

进阶

事实上，关于字符串匹配问题，还有一道稍稍不同的变形的题目。

也是利用了类似的「通用思路」(状态定义) &「技巧」，然后对匹配过程中的字符进行分情况讨论，学有余力的同学可以看看：

10. 正则表达式匹配 : 如何利用的「等差」性质降低「正则字符串匹配」算法复杂度 ...

> **🍭 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 **119. 杨辉三角 II** ，难度为 **简单**。

Tag : 「数学」、「线性 DP」

给定一个非负索引 k，其中 k ≤ 33，返回杨辉三角的第 k 行。

在杨辉三角中，每个数是它左上方和右上方的数的和。

示例:

```
输入: 3
输出: [1,3,3,1]
```

进阶：

- 你可以优化你的算法到 O(k) 空间复杂度吗？

动态规划

```
class Solution {
    public List<Integer> getRow(int idx) {
        int[][] f = new int[idx + 1][idx + 1];
        f[0][0] = 1;
        for (int i = 1; i < idx + 1; i++) {
            for (int j = 0; j < i + 1; j++) {
                f[i][j] = f[i - 1][j];
                if (j - 1 >= 0) f[i][j] += f[i - 1][j - 1];
                if (f[i][j] == 0) f[i][j] = 1;
            }
        }
        List<Integer> ans = new ArrayList<>();
        for (int i = 0; i < idx + 1; i++) ans.add(f[idx][i]);
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

滚动数组

滚动数组优化十分机械，直接将滚动的维度从 `i` 改造为 `i % 2` 或 `i & 1` 即可。

`i & 1` 相比于 `i % 2` 在不同架构的机器上，效率会更稳定些 ~

```Java
class Solution {
    public List<Integer> getRow(int idx) {
        int[][] f = new int[2][idx + 1];
        f[0][0] = 1;
        for (int i = 1; i < idx + 1; i++) {
            for (int j = 0; j < i + 1; j++) {
                f[i & 1][j] = f[(i - 1) & 1][j];
                if (j - 1 >= 0) f[i & 1][j] += f[(i - 1) & 1][j - 1];
                if (f[i & 1][j] == 0) f[i & 1][j] = 1;
            }
        }
        List<Integer> ans = new ArrayList<>();
        for (int i = 0; i < idx + 1; i++) ans.add(f[idx & 1][i]);
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

一维优化

只有第 `i` 行的更新只依赖于 `i - 1` 行，因此可以直接消除行的维度：

```Java
class Solution {
    public List<Integer> getRow(int idx) {
        int[] f = new int[idx + 1];
        f[0] = 1;
        for (int i = 1; i < idx + 1; i++) {
            for (int j = i; j >= 0; j--) {
                if (j - 1 >= 0) f[j] += f[j - 1];
                if (f[j] == 0) f[j] = 1;
            }
        }
        List<Integer> ans = new ArrayList<>();
        for (int i = 0; i < idx + 1; i++) ans.add(f[i]);
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$

- 空间复杂度：$O(n)$

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

题目描述

这是 LeetCode 上的 **213. 打家劫舍 II** ，难度为 **中等**。

Tag : 「线性 DP」

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。

这个地方所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。

同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下 ，今晚能够偷窃到的最高金额。

示例 1：

```
输入：nums = [2,3,2]

输出：3

解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2）, 因为他们是相邻的。
```

示例 2：

```
输入：nums = [1,2,3,1]

输出：4

解释：你可以先偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

示例 3：

```
输入：nums = [0]

输出：0
```

提示：

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 1000

动态规划

在 198. 打家劫舍 中，并没有「第一间」和「最后一间」不能同时选择的限制，因此我们从头到尾做一遍 DP 即可。

在 198. 打家劫舍 中，我们可以将状态定义为两维：

$f[i][j]$ 代表考虑前 i 个房间，当前 i 房间的现在状态为 j 的最大价值。

- $f[i][0]$ 代表考虑前 i 个房间，并且「不选」第 i 个房间的最大价值。由于已经明确了第 i 个房间不选，因此 $f[i][0]$ 可以直接由 $max(f[i-1][0], f[i-1][1])$ 转移而来。
- $f[i][1]$ 代表考虑前 i 个房间，并且「选」第 i 个房间的最大价值。由于已经明确了第 i 个房间被选，因此 $f[i][1]$ 直接由 $f[i-1][0] + nums[i]$ 转移过来。

到这里，你已经解决了 198. 打家劫舍 了。

对于本题，由于只是增加了「第一间」和「最后一间」不能同时选择的限制。

通常，对于一些明显不是「增加维度」的新限制条件，我们应当考虑直接将其拎出讨论，而不是多增加一维进行状态记录。

我们可以把「第一间」&「最后一间」单独拎出来讨论：

- 明确「不选」第一间：
    1. 初始化 $f[0][0]$ 和 $f[0][1]$，均为 0。
    2. 先从「第二间」开始递推到「倒数第二间」的最大价值。
    3. 再处理「最后一间」的情况：由于明确了「不选第一间」，则最后的最

大价值为 $max(f[n-2][1], f[n-2][0] + nums[n-1])$。

- 允许「选」第一间：
    1. 初始化 $f[0][0]$ 和 $f[0][1]$，分别为 0 和 $nums[0]$。
    2. 先从「第二间」开始递推到「倒数第二间」的最大价值。
    3. 再处理「最后一间」的情况：由于明确了「选第一间」，则最后的最大价值为 $max(f[n-2][0], f[n-2][1])$。

走完两遍 DP 后，再从两种情况的最大价值中再取一个 max 即是答案。

代码：

```
class Solution {
    public int rob(int[] nums) {
        int n = nums.length;
        if (n == 0) return 0;
        if (n == 1) return nums[0];

        // 第一间「必然不选」的情况
        int[][] f = new int[n][2];
        for (int i = 1; i < n - 1; i++) {
            f[i][0] = Math.max(f[i - 1][0], f[i - 1][1]);
            f[i][1] = f[i - 1][0] + nums[i];
        }
        int a = Math.max(f[n - 2][1], f[n - 2][0] + nums[n - 1]);

        // 第一间「允许选」的情况
        f[0][0] = 0; f[0][1] = nums[0];
        for (int i = 1; i < n - 1; i++) {
            f[i][0] = Math.max(f[i - 1][0], f[i - 1][1]);
            f[i][1] = f[i - 1][0] + nums[i];
        }
        int b = Math.max(f[n - 2][0], f[n - 2][1]);

        return Math.max(a, b);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

空间优化

不难发现，我们状态转移最多依赖到前面的 1 行，因此可以通过很机械的「滚动数组」方式将空间修改到 $O(1)$。

代码：

```
class Solution {
    public int rob(int[] nums) {
        int n = nums.length;
        if (n == 0) return 0;
        if (n == 1) return nums[0];

        // 第一间「必然不选」的情况
        int[][] f = new int[2][2];
        for (int i = 1; i < n - 1; i++) {
            f[i%2][0] = Math.max(f[(i - 1)%2][0], f[(i - 1)%2][1]);
            f[i%2][1] = f[(i - 1)%2][0] + nums[i];
        }
        int a = Math.max(f[(n - 2)%2][1], f[(n - 2)%2][0] + nums[n - 1]);

        // 第一间「允许选」的情况
        f[0][0] = 0; f[0][1] = nums[0];
        for (int i = 1; i < n - 1; i++) {
            f[i%2][0] = Math.max(f[(i - 1)%2][0], f[(i - 1)%2][1]);
            f[i%2][1] = f[(i - 1)%2][0] + nums[i];
        }
        int b = Math.max(f[(n - 2)%2][0], f[(n - 2)%2][1]);

        return Math.max(a, b);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

题目描述

这是 LeetCode 上的 **403. 青蛙过河** ，难度为 **困难**。

Tag：「DFS」、「BFS」、「记忆化搜索」、「线性 DP」

一只青蛙想要过河。 假定河流被等分为若干个单元格，并且在每一个单元格内都有可能放有一块石子（也有可能没有）。 青蛙可以跳上石子，但是不可以跳入水中。

给你石子的位置列表 stones（用单元格序号 升序 表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一块石子上）。

开始时， 青蛙默认已站在第一块石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格 1 跳至单元格 2 ）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1 个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

示例 1：

```
输入：stones = [0,1,3,5,6,8,12,17]

输出：true

解释：青蛙可以成功过河，按照如下方案跳跃：跳 1 个单位到第 2 块石子, 然后跳 2 个单位到第 3 块石子, 接着 跳 2 个单位到第 4 块
```

示例 2：

```
输入：stones = [0,1,2,3,4,8,9,11]

输出：false

解释：这是因为第 5 和第 6 个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。
```

提示：

- 2 <= stones.length <= 2000
- 0 <= stones[i] <= 2^{31} - 1
- stones[0] == 0

DFS（TLE）

根据题意，我们可以使用 `DFS` 来模拟/爆搜一遍，检查所有的可能性中是否有能到达最后一块石子的。

通常设计 `DFS` 函数时，我们只需要不失一般性的考虑完成第 i 块石子的跳跃需要些什么信息即可：

- 需要知道当前所在位置在哪，也就是需要知道当前石子所在列表中的下标 u。
- 需要知道当前所在位置是经过多少步而来的，也就是需要知道上一步的跳跃步长 k。

代码：

```Java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        // 将石子信息存入哈希表
        // 为了快速判断是否存在某块石子，以及快速查找某块石子所在下标
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        // 根据题意，第一步是固定经过步长 1 到达第一块石子（下标为 1）
        if (!map.containsKey(1)) return false;
        return dfs(ss, ss.length, 1, 1);
    }

    /**
     * 判定是否能够跳到最后一块石子
     * @param ss 石子列表【不变】
     * @param n  石子列表长度【不变】
     * @param u  当前所在的石子的下标
     * @param k  上一次是经过多少步跳到当前位置的
     * @return 是否能跳到最后一块石子
     */
    boolean dfs(int[] ss, int n, int u, int k) {
        if (u == n - 1) return true;
        for (int i = -1; i <= 1; i++) {
            // 如果是原地踏步的话，直接跳过
            if (k + i == 0) continue;
            // 下一步的石子理论编号
            int next = ss[u] + k + i;
            // 如果存在下一步的石子，则跳转到下一步石子，并 DFS 下去
            if (map.containsKey(next)) {
                boolean cur = dfs(ss, n, map.get(next), k + i);
                if (cur) return true;
            }
        }
        return false;
    }
}
```

- 时间复杂度：$O(3^n)$
- 空间复杂度：$O(3^n)$

但数据范围为 10^3，直接使用 DFS 肯定会超时。

我们需要考虑加入「记忆化」功能，或者改为使用带标记的 `BFS`。

记忆化搜索

在考虑加入「记忆化」时，我们只需要将 `DFS` 方法签名中的【可变】参数作为维度，`DFS` 方法中的返回值作为存储值即可。

通常我们会使用「数组」来作为我们缓存中间结果的容器，

对应到本题，就是需要一个 `boolean[石子列表下标][跳跃步数]` 这样的数组，但使用布尔数组作为记忆化容器往往无法区分「状态尚未计算」和「状态已经计算，并且结果为 `false`」两种情况。

因此我们需要转为使用 `int[石子列表下标][跳跃步数]`，默认值 0 代表状态尚未计算，-1 代表计算状态为 `false`，1 代表计算状态为 `true`。

接下来需要估算数组的容量，可以从「数据范围」入手分析。

根据 `2 <= stones.length <= 2000`，我们可以确定第一维（数组下标）的长度为 2009，而另外一维（跳跃步数）是与跳转过程相关的，无法直接确定一个精确边界，但是一个显而易见的事实是，跳到最后一块石子之后的位置是没有意义的，因此我们不会有「跳跃步长」大于「石子列表长度」的情况，因此也可以定为 2009（这里是利用了由下标为 i 的位置发起的跳跃不会超过 $i + 1$ 的性质）。

至此，我们定下来了记忆化容器为 `int[][] cache = new int[2009][2009]`。

但是可以看出，上述确定容器大小的过程还是需要一点点分析 & 经验的。

那么是否有思维难度再低点的方法呢？

答案是有的，直接使用「哈希表」作为记忆化容器。「哈希表」本身属于非定长容器集合，我们不需要分析两个维度的上限到底是多少。

另外，当容器维度较多且上界较大时（例如上述的 `int[2009][2009]`），直接使用「哈希表」可以有效降低「爆空间/时间」的风险（不需要每跑一个样例都创建一个百万级的数组）。

代码：

```java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    // int[][] cache = new int[2009][2009];
    Map<String, Boolean> cache = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        if (!map.containsKey(1)) return false;
        return dfs(ss, ss.length, 1, 1);
    }
    boolean dfs(int[] ss, int n, int u, int k) {
        String key = u + "_" + k;
        // if (cache[u][k] != 0) return cache[u][k] == 1;
        if (cache.containsKey(key)) return cache.get(key);
        if (u == n - 1) return true;
        for (int i = -1; i <= 1; i++) {
            if (k + i == 0) continue;
            int next = ss[u] + k + i;
            if (map.containsKey(next)) {
                boolean cur = dfs(ss, n, map.get(next), k + i);
                // cache[u][k] = cur ? 1 : -1;
                cache.put(key, cur);
                if (cur) return true;
            }
        }
        // cache[u][k] = -1;
        cache.put(key, false);
        return false;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

动态规划

有了「记忆化搜索」的基础，要写写出来动态规划就变得相对简单了。

我们可以从 `DFS` 函数出发，写出「动态规划」解法。

我们的 `DFS` 函数签名为：

```
boolean dfs(int[] ss, int n, int u, int k);
```

其中前两个参数为不变参数，后两个为可变参数，返回值是我们的答案。

因此可以设定为 $f[][]$ 作为动规数组：

1. 第一维为可变参数 u，代表石子列表的下标，范围为数组 `stones` 长度；
2. 第二维为可变参数 k，代表上一步的的跳跃步长，前面也分析过了，最多不超过数组 `stones` 长度。

这样的「状态定义」所代表的含义：当前在第 i 个位置，并且是以步长 k 跳到位置 i 时，是否到达最后一块石子。

那么对于 $f[i][k]$ 是否为真，则取决于上一位置 j 的状态值，结合每次步长的变化为 `[-1,0,1]` 可知：

- 可从 $f[j][k-1]$ 状态而来：先是经过 $k-1$ 的跳跃到达位置 j，再在原步长的基础上 `+1`，跳到了位置 i。
- 可从 $f[j][k]$ 状态而来：先是经过 k 的跳跃到达位置 j，维持原步长不变，跳到了位置 i。
- 可从 $f[j][k+1]$ 状态而来：先是经过 $k+1$ 的跳跃到达位置 j，再在原步长的基础上 `-1`，跳到了位置 i。

只要上述三种情况其中一种为真，则 $f[i][j]$ 为真。

至此，我们解决了动态规划的「状态定义」&「状态转移方程」部分。

但这就结束了吗？还没有。

我们还缺少可让状态递推下去的「有效值」，或者说缺少初始化环节。

因为我们的 $f[i][k]$ 依赖于之前的状态进行“或运算”而来，转移方程本身不会产生 $true$ 值。因此为了让整个「递推」过程可滚动，我们需要先有一个为 $true$ 的状态值。

这时候再回看我们的状态定义：**当前在第 i 个位置，并且是以步长 k 跳到位置 i 时，是否到达最后一块石子。**

显然，我们事先是不可能知道经过「多大的步长」跳到「哪些位置」，最终可以到达最后一块石

子。

这时候需要利用「对偶性」将跳跃过程「翻转」过来分析：

我们知道起始状态是「经过步长为 1」的跳跃到达「位置 1」，如果从起始状态出发，存在一种方案到达最后一块石子的话，那么必然存在一条反向路径，它是以从「最后一块石子」开始，并以「某个步长 k」开始跳跃，最终以回到位置 1。

因此我们可以设 $f[1][1] = true$，作为我们的起始值。

这里本质是利用「路径可逆」的性质，将问题进行了「等效对偶」。表面上我们是进行「正向递推」，但事实上我们是在验证是否存在某条「反向路径」到达位置 1。

建议大家加强理解 ~

代码：

```
class Solution {
    public boolean canCross(int[] ss) {
        int n = ss.length;
        // check first step
        if (ss[1] != 1) return false;
        boolean[][] f = new boolean[n + 1][n + 1];
        f[1][1] = true;
        for (int i = 2; i < n; i++) {
            for (int j = 1; j < i; j++) {
                int k = ss[i] - ss[j];
                // 我们知道从位置 j 到位置 i 是需要步长为 k 的跳跃

                // 而从位置 j 发起的跳跃最多不超过 j + 1
                // 因为每次跳跃，下标至少增加 1，而步长最多增加 1
                if (k <= j + 1) {
                    f[i][k] = f[j][k - 1] || f[j][k] || f[j][k + 1];
                }
            }
        }
        for (int i = 1; i < n; i++) {
            if (f[n - 1][i]) return true;
        }
        return false;
    }
}
```

- 时间复杂度：$O(n^2)$

- 空间复杂度：$O(n^2)$

BFS

事实上，前面我们也说到，解决超时 `DFS` 问题，除了增加「记忆化」功能以外，还能使用带标记的 `BFS`。

因为两者都能解决 `DFS` 的超时原因：大量的重复计算。

但为了「记忆化搜索」&「动态规划」能够更好的衔接，所以我把 `BFS` 放到最后。

如果你能够看到这里，那么这里的 `BFS` 应该看起来会相对轻松。

它更多是作为「记忆化搜索」的另外一种实现形式。

代码：

```Java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        if (!map.containsKey(1)) return false;

        boolean[][] vis = new boolean[n][n];
        Deque<int[]> d = new ArrayDeque<>();
        vis[1][1] = true;
        d.addLast(new int[]{1, 1});

        while (!d.isEmpty()) {
            int[] poll = d.pollFirst();
            int idx = poll[0], k = poll[1];
            if (idx == n - 1) return true;
            for (int i = -1; i <= 1; i++) {
                if (k + i == 0) continue;
                int next = ss[idx] + k + i;
                if (map.containsKey(next)) {
                    int nIdx = map.get(next), nK = k + i;
                    if (nIdx == n - 1) return true;
                    if (!vis[nIdx][nK]) {
                        vis[nIdx][nK] = true;
                        d.addLast(new int[]{nIdx, nK});
                    }
                }
            }
        }

        return false;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

题目描述

这是 LeetCode 上的 **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] 。

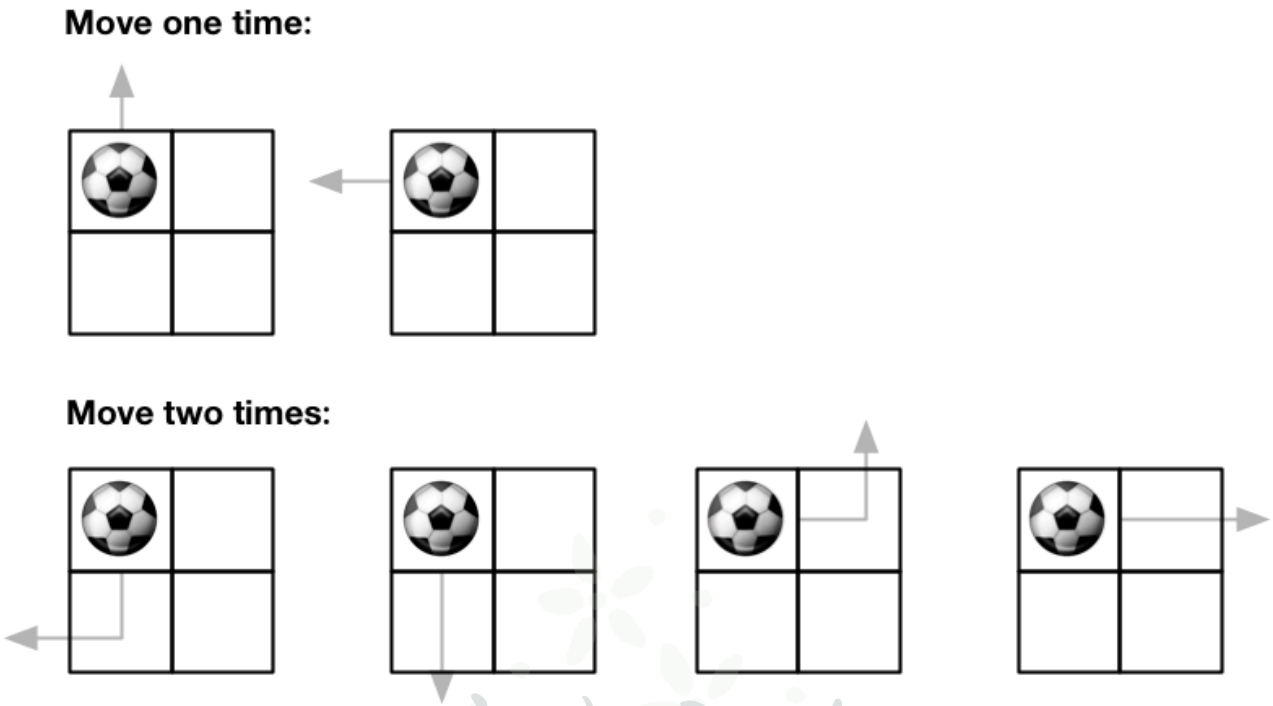
你可以将球移到在四个方向上相邻的单元格内（可以穿过网格边界到达网格之外）。

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。

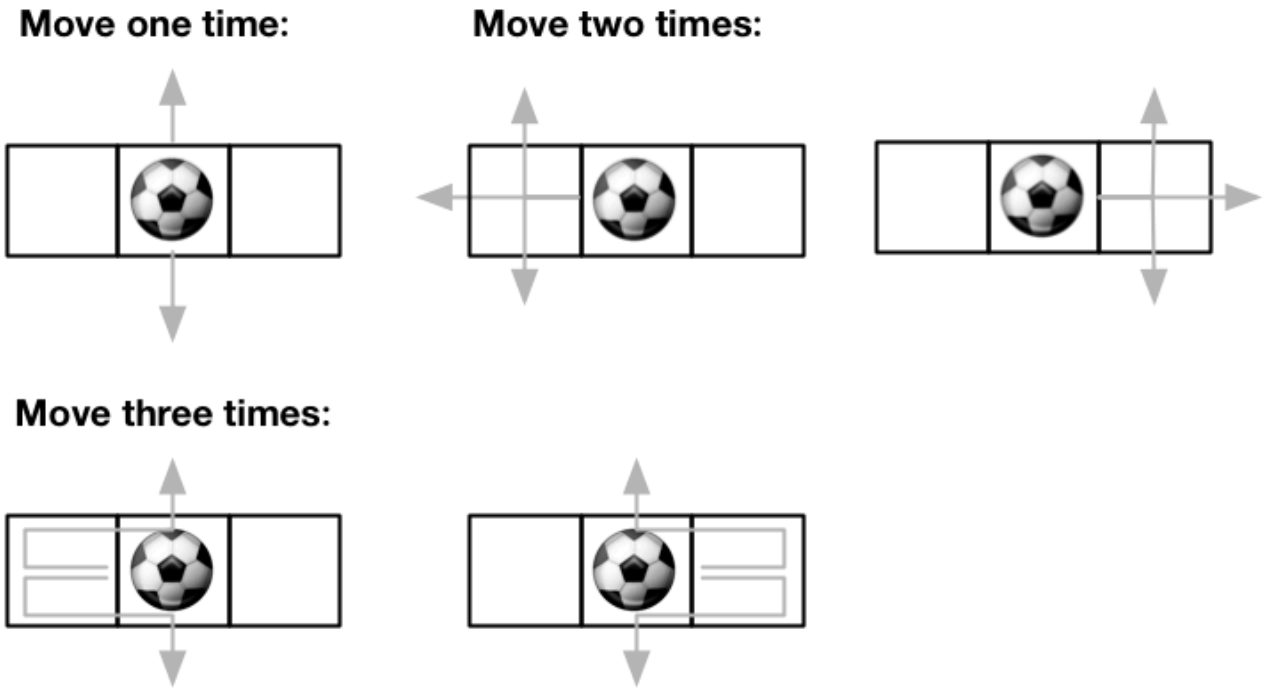
示例 1：




```
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。

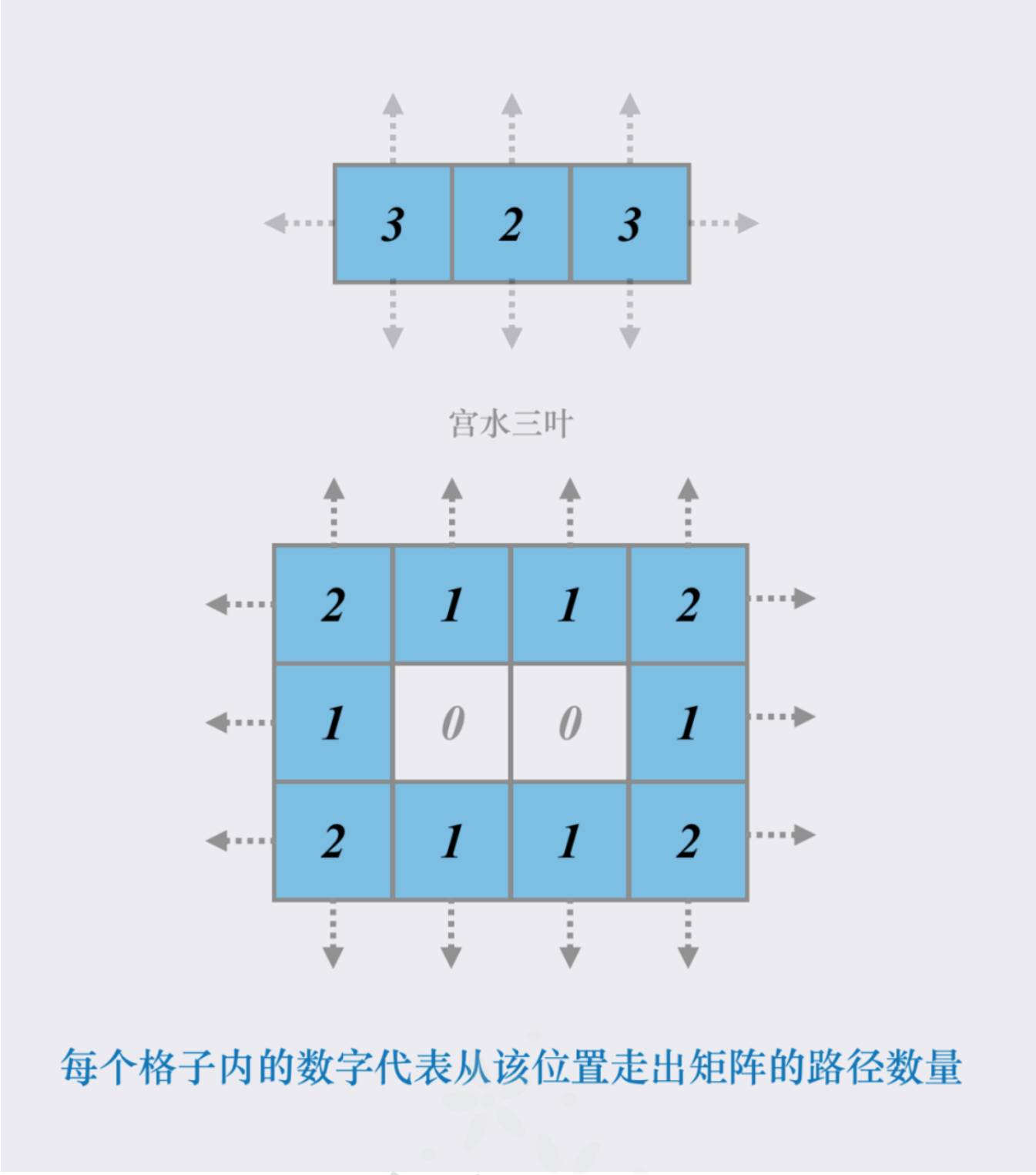
其实就是需要一些「有效值」作为初始化状态。

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。



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：$O(m * n * max)$

题目描述

这是 LeetCode 上的 **639. 解码方法 II** ，难度为 **困难**。

Tag：「线性 DP」、「枚举」

一条包含字母 A-Z 的消息通过以下的方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

要 解码 一条已编码的消息，所有的数字都必须分组，然后按原来的编码方案反向映射回字母（可能存在多种方式）。例如，“11106” 可以映射为：

- “AAJF” 对应分组 (1 1 10 6)
- “KJF” 对应分组 (11 10 6)

注意，像 (1 11 06) 这样的分组是无效的，因为 “06” 不可以映射为 ‘F’ ，因为 “6” 与 “06” 不同。

除了 上面描述的数字字母映射方案，编码消息中可能包含 `'*'` 字符，可以表示从 ‘1’ 到 ‘9’ 的任一数字（不包括 ‘0’）。例如，编码字符串 `"1*"` 可以表示 “11”、“12”、“13”、“14”、“15”、“16”、“17”、“18” 或 “19” 中的任意一条消息。对 “1*” 进行解码，相当于解码该字符串可以表示的任何编码消息。

给你一个字符串 s ，由数字和 ‘*’ 字符组成，返回 解码 该字符串的方法 数目 。

由于答案数目可能非常大，返回对 $10^9 + 7$ 取余 的结果。

示例 1：

输入：s = "*"

输出：9

解释：这一条编码消息可以表示 "1"、"2"、"3"、"4"、"5"、"6"、"7"、"8" 或 "9" 中的任意一条。
可以分别解码成字符串 "A"、"B"、"C"、"D"、"E"、"F"、"G"、"H" 和 "I" 。
因此，"*" 总共有 9 种解码方法。

示例 2：

输入：s = "1*"

输出：18

解释：这一条编码消息可以表示 "11"、"12"、"13"、"14"、"15"、"16"、"17"、"18" 或 "19" 中的任意一条。
每种消息都可以由 2 种方法解码（例如，"11" 可以解码成 "AA" 或 "K"）。
因此，"1*" 共有 9 * 2 = 18 种解码方法。

示例 3：

输入：s = "2*"

输出：15

解释：这一条编码消息可以表示 "21"、"22"、"23"、"24"、"25"、"26"、"27"、"28" 或 "29" 中的任意一条。
"21"、"22"、"23"、"24"、"25" 和 "26" 由 2 种解码方法，但 "27"、"28" 和 "29" 仅有 1 种解码方法。
因此，"2*" 共有 (6 * 2) + (3 * 1) = 12 + 3 = 15 种解码方法。

提示：

- 1 <= s.length <= 10^5
- s[i] 是 `0-9` 中的一位数字或字符 ‘*’

分情况讨论 DP

这是一道普通的线性 DP 题（之所以说普通，是因为状态定义比较容易想到），也是 （题解）91. 解码方法 的进阶题。

我们称一个解码内容为一个 `item`。

定义 $f[i]$ 为考虑以 $s[i]$ 为结尾的字符串，共有多少种解码方案。

那么最终答案为 $f[n-1]$，同时我们有显而易见的起始状态 `f[0] = s[0] == '*' ? 9 : (s[0] != '0' ? 1 : 0)`.

不失一般性考虑 $f[i]$ 该如何转移，$s[i]$ 要么是 `*`，要么是数字，对应一个分情况讨论过程：

- 当 $s[i]$ 为 `*`：此时考虑 $s[i]$ 是单独作为一个 `item`，还是与上一个字符共同作为一个 `item`:
    - $s[i]$ 单独作为一个 `item`：由于 `*` 可以代指数字 `1-9`，因此有 $f[i] = f[i-1] * 9$；
    - $s[i]$ 与上一个字符共同作为一个 `item`：此时需要对上一个字符 $s[j]$ 进行讨论：
        - $s[j]$ 为数字 `1`：此时 $s[i]$ 可以代指 `1-9`，对应了 `item` 为 `11-19` 这 9 种情况，此时有 $f[i] = f[i-2] * 9$（**如果 $f[i-2]$ 取不到，则使用 1 代指，下面同理**）；
        - $s[j]$ 为数字 `2`：此时 $s[i]$ 可以代指 `1-6`，对应了 `item` 为 `21-26` 这 6 种情况，此时有 $f[i] = f[i-2] * 6$；
        - $s[j]$ 为字符 `*`：此时两个 `*` 对应了合法方案为 $11-19$ 和 $21-26$ 共 15 种方案，此时有 $f[i] = f[i-2] * 15$；
- 当 $s[i]$ 为数字：此时可以从「前一字符 $s[j]$ 为何种字符」和「当前 $s[i]$ 是否为 0」出发进行讨论:
    - $s[j]$ 为字符 `*`，根据当前 $s[i]$ 是否为 0 讨论：
        - $s[i]$ 为数字 0：此时 $s[i]$ 无法独自作为一个 `item`，只能与 $s[j]$ 组合，对应了 10 和 20 两种情况，此时有 $f[i] = f[i-2] * 2$；
        - $s[i]$ 为数字 `1-9`，此时首先有 $s[i]$ 可以作为一个独立 `item` 的情况，即有 $f[i] = f[i-1]$，然后对 $s[i]$ 的数值大小进一步分情况讨论：
            - $s[i]$ 为数字 `1-6`，此时 $s[j]$ 可以代指 1 和 2，对应了方案 $1x$ 和 $2x$，此时有 $f[i] = f[i-2] * 2$；
            - $s[i]$ 为数字 `7-9`，此时 $s[j]$ 可以代指 1，对应方案 $1x$，此时有 $f[i] = f[i-2]$；
    - $s[j]$ 为数字类型，此时从「当前 $s[i]$ 是否为 0」出发进行讨论：

- $s[i]$ 为数字 0：此时 $s[j]$ 只有为 1 和 2 时，才是合法方案，则有 $f[i] = f[i-2]$；
- $s[i]$ 为数字 `1-9`：此时首先有 $s[i]$ 可以作为一个独立 `item` 的情况，即有 $f[i] = f[i-1]$，然后再考虑能够与 $s[j]$ 组成合法 `item` 的情况：
  - $s[j]$ 为数值 1：此时有 $f[i] = f[i-2]$；
  - $s[j]$ 为数值 2，且 $s[i]$ 为数值 `1-6`：此时有 $f[i] = f[i-2]$。

由于是求方案数，因此最终的 $f[i]$ 为上述所有的合法的分情况讨论的累加值，并对 $1e9+7$ 取模。

> 一些细节：实现上了避免大量对 $f[i-2]$ 是否可以取得的讨论，我们可以对 `s` 前追加一个空格作为哨兵（无须真正插入），以简化代码，同时由于 $f[i]$ 只依赖于 $f[i-1]$ 和 $f[i-1]$，可以使用「滚动数组」的形式进行空间空间优化（见 $P2$）。

> 另外，对于「滚动数组」的空间优化方式，还需要说明两点：转移前先使用变量保存 `(i-1)%3` 和 `(i-2)%3` 的计算结果，防止大量的重复计算；不能再偷懒使用 `toCharArray`，只能使用 `charAt`，因为 Java 为了遵循字符串不变的原则，会在调用 `toCharArray` 时返回新数组，这样复杂度就还是 $O(n)$ 的。
> 诸如此类的「滚动数组」优化方式，最早在 这里 讲过。

代码：

```java
class Solution {
    int mod = (int)1e9+7;
    public int numDecodings(String s) {
        char[] cs = s.toCharArray();
        int n = cs.length;
        long[] f = new long[n];
        f[0] = cs[0] == '*' ? 9 : (cs[0] != '0' ? 1 : 0);
        for (int i = 1; i < n; i++) {
            char c = cs[i], prev = cs[i - 1];
            if (c == '*') {
                // cs[i] 单独作为一个 item
                f[i] += f[i - 1] * 9;
                // cs[i] 与前一个字符共同作为一个 item
                if (prev == '*') {
                    // 11 - 19 & 21 - 26
                    f[i] += (i - 2 >= 0 ? f[i - 2] : 1) * 15;
                } else {
                    int u = (int)(prev - '0');
                    if (u == 1) {
                        f[i] += (i - 2 >= 0 ? f[i - 2] : 1) * 9;
                    } else if (u == 2) {
                        f[i] += (i - 2 >= 0 ? f[i - 2] : 1) * 6;
                    }
                }
            } else {
                int t = (int)(c - '0');
                if (prev == '*') {
                    if (t == 0) {
                        f[i] += (i - 2 >= 0 ? f[i - 2] : 1) * 2;
                    } else {
                        // cs[i] 单独作为一个 item
                        f[i] += f[i - 1];
                        // cs[i] 与前一个字符共同作为一个 item
                        if (t <= 6) {
                            f[i] += (i - 2 >= 0 ? f[i - 2] : 1) * 2;
                        } else {
                            f[i] += i - 2 >= 0 ? f[i - 2] : 1;
                        }
                    }
                } else {
                    int u = (int)(prev - '0');
                    if (t == 0) {
                        if (u == 1 || u == 2) {
                            f[i] += i - 2 >= 0 ? f[i - 2] : 1;
                        }
                    } else {
```

```
                        // cs[i] 单独作为一个 item
                        f[i] += (f[i - 1]);
                        // cs[i] 与前一个字符共同作为一个 item
                        if (u == 1) {
                            f[i] += i - 2 >= 0 ? f[i - 2] : 1;
                        } else if (u == 2 && t <= 6) {
                            f[i] += i - 2 >= 0 ? f[i - 2] : 1;
                        }
                    }
                }
            }
            f[i] %= mod;
        }
        return (int)(f[n - 1]);
    }
}
```

```java
class Solution {
    int mod = (int)1e9+7;
    public int numDecodings(String s) {
        int n = s.length() + 1;
        long[] f = new long[3];
        f[0] = 1;
        f[1] = s.charAt(0) == '*' ? 9 : (s.charAt(0) != '0' ? 1 : 0);
        for (int i = 2; i < n; i++) {
            char c = s.charAt(i - 1), prev = s.charAt(i - 2);
            int p1 = (i - 1) % 3, p2 = (i - 2) % 3;
            long cnt = 0;
            if (c == '*') {
                // cs[i] 单独作为一个 item
                cnt += f[p1] * 9;
                // cs[i] 与前一个字符共同作为一个 item
                if (prev == '*') {
                    cnt += f[p2] * 15;
                } else {
                    int u = (int)(prev - '0');
                    if (u == 1) cnt += f[p2] * 9;
                    else if (u == 2) cnt += f[p2] * 6;
                }
            } else {
                int t = (int)(c - '0');
                if (prev == '*') {
                    if (t == 0) {
                        cnt += f[p2]* 2;
                    } else {
                        // cs[i] 单独作为一个 item
                        cnt += f[p1];
                        // cs[i] 与前一个字符共同作为一个 item
                        if (t <= 6) cnt += f[p2] * 2;
                        else cnt += f[p2];
                    }
                } else {
                    int u = (int)(prev - '0');
                    if (t == 0) {
                        if (u == 1 || u == 2) cnt += f[p2];
                    } else {
                        // cs[i] 单独作为一个 item
                        cnt += f[p1];
                        // cs[i] 与前一个字符共同作为一个 item
                        if (u == 1) cnt += f[p2];
                        else if (u == 2 && t <= 6) cnt += f[p2];
                    }
                }
            }
```

```
            }
            f[i % 3] = cnt % mod;
        }
        return (int)(f[(n - 1) % 3]);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：使用「滚动数组」进行优化，复杂度为 $O(1)$，否则为 $O(n)$

枚举 DP

上述解法之所以复杂，是因为不仅仅要对当前字符 $s[i]$ 分情况讨论，还需要对上一个字符 $s[j]$ 分情况讨论。

事实上，我们可以利用解码对象只有 `A-Z` 来进行枚举。

在从前往后处理字符串 `s` 时，枚举 $s[i]$ 参与构成的解码内容 `item` 是字母 `A-Z` 中哪一个，从而将分情况讨论转变成对应位的字符对比。

代码：

```java
class Solution {
    int mod = (int)1e9+7;
    public int numDecodings(String s) {
        int n = s.length();
        long[] f = new long[3];
        f[0] = 1;
        for (int i = 1; i <= n; i++) {
            char c = s.charAt(i - 1);
            int t = c - '0';
            long cnt = 0;
            int p1 = (i - 1) % 3, p2 = (i - 2) % 3;
            // 枚举组成什么 item（A -> 1; B -> 2 ...）
            for (int item = 1; item <= 26; item++) {
                if (item < 10) { // 该 item 由一个字符组成
                    if (c == '*' || t == item) cnt += f[p1];
                } else { // 该 item 由两个字符组成
                    if (i - 2 < 0) break;
                    char prev = s.charAt(i - 2);
                    int u = prev - '0';
                    int a = item / 10, b = item % 10;
                    if ((prev == '*' || u == a) && (t == b || (c == '*' && b != 0))) cnt +
                }
            }
            f[i % 3] = cnt % mod;
        }
        return (int)(f[n % 3]);
    }
}
```

- 时间复杂度：$O(n * C)$，其中 C 为解码内容字符集大小，固定为 26
- 空间复杂度：$O(1)$

题目描述

这是 LeetCode 上的 **650. 只有两个键的键盘** ，难度为 **中等**。

Tag：「动态规划」、「线性 DP」、「数学」、「打表」

最初记事本上只有一个字符 'A'。你每次可以对这个记事本进行两种操作：

- Copy All（复制全部）：复制这个记事本中的所有字符（不允许仅复制部分字符）。
- Paste（粘贴）：粘贴 上一次 复制的字符。

给你一个数字 n ，你需要使用最少的操作次数，在记事本上输出 恰好 n 个 ‘A’ 。返回能够打印出 n 个 ‘A’ 的最少操作次数。

示例 1：

```
输入：3

输出：3

解释：
最初, 只有一个字符 'A'。
第 1 步, 使用 Copy All 操作。
第 2 步, 使用 Paste 操作来获得 'AA'。
第 3 步, 使用 Paste 操作来获得 'AAA'。
```

示例 2：

```
输入：n = 1

输出：0
```

提示：

- 1 <= n <= 1000

动态规划

定义 $f[i][j]$ 为经过最后一次操作后，当前记事本上有 i 个字符，粘贴板上有 j 个字符的最小操作次数。

由于我们粘贴板的字符必然是经过 `Copy All` 操作而来，因此对于一个合法的 $f[i][j]$ 而言，必然有 $j <= i$。

不失一般性地考虑 $f[i][j]$ 该如何转移：

- 最后一次操作是 `Paste` 操作：此时粘贴板的字符数量不会发生变化，即有 $f[i][j] = f[i-j][j] + 1$；
- 最后一次操作是 `Copy All` 操作：那么此时的粘贴板的字符数与记事本上的字符数相等（满足 $i = j$），此时的 $f[i][j] = \min(f[i][x] + 1), 0 \le x < i$。

我们发现最后一个合法的 $f[i][j]$（满足 $i = j$）依赖与前面 $f[i][j]$（满足 $j < i$）。

因此实现上，我们可以使用一个变量 min 保存前面转移的最小值，用来更新最后的 $f[i][j]$。

再进一步，我们发现如果 $f[i][j]$ 的最后一次操作是由 `Paste` 而来，原来粘贴板的字符数不会超过 $i/2$，因此在转移 $f[i][j]$（满足 $j < i$）时，其实只需要枚举 $[0, i/2]$ 即可。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int minSteps(int n) {
        int[][] f = new int[n + 1][n + 1];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= n; j++) {
                f[i][j] = INF;
            }
        }
        f[1][0] = 0; f[1][1] = 1;
        for (int i = 2; i <= n; i++) {
            int min = INF;
            for (int j = 0; j <= i / 2; j++) {
                f[i][j] = f[i - j][j] + 1;
                min = Math.min(min, f[i][j]);
            }
            f[i][i] = min + 1;
        }
        int ans = INF;
        for (int i = 0; i <= n; i++) ans = Math.min(ans, f[n][i]);
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

数学

如果我们将「1 次 `Copy All` + x 次 `Paste`」看做一次“动作”的话。

那么 一次“动作”所产生的效果就是将原来的字符串变为原来的 $x + 1$ 倍。

最终的最小操作次数方案可以等价以下操作流程：

1. 起始对长度为 1 的记事本字符进行 1 次 `Copy All` + $k_1 - 1$ 次 `Paste` 操作（消耗次数为 k_1，得到长度为 k_1 的记事本长度）；
2. 对长度为为 k_1 的记事本字符进行 1 次 `Copy All` + $k_2 - 1$ 次 `Paste` 操作（消耗次数为 $k_1 + k_2$，得到长度为 $k_1 * k_2$ 的记事本长度）

   ...

最终经过 k 次“动作”之后，得到长度为 n 的记事本长度，即有：

$$n = k_1 * k_2 * ... * k_x$$

问题转化为：如何对 n 进行拆分，可以使得 $k_1 + k_2 + ... + k_x$ 最小。

对于任意一个 k_i（合数）而言，根据定理 $a * b >= a + b$ 可知进一步的拆分必然不会导致结果变差。

因此，我们只需要使用「试除法」对 n 执行分解质因数操作，累加所有的操作次数，即可得到答案。

代码：

```Java
class Solution {
    public int minSteps(int n) {
        int ans = 0;
        for (int i = 2; i * i <= n; i++) {
            while (n % i == 0) {
                ans += i;
                n /= i;
            }
        }
        if (n != 1) ans += n;
        return ans;
    }
}
```

- 时间复杂度：$O(\sqrt{n})$
- 空间复杂度：$O(1)$

打表

我们发现，对于某个 $minSteps(i)$ 而言为定值，且数据范围只有 1000，因此考虑使用打表来做。

代码：

```Java
class Solution {
    static int N = 1010;
    static int[] g = new int[N];
    static {
        for (int k = 2; k < N; k++) {
            int cnt = 0, n = k;
            for (int i = 2; i * i <= n; i++) {
                while (n % i == 0) {
                    cnt += i;
                    n /= i;
                }
            }
            if (n != 1) cnt += n;
            g[k] = cnt;
        }
        // System.out.println(Arrays.toString(g)); // 输出打表结果
    }
    public int minSteps(int n) {
        return g[n];
    }
}
```

```Java
class Solution {
    static int[] g = new int[]{0, 0, 2, 3, 4, 5, 5, 7, 6, 6, 7, 11, 7, 13, 9, 8, 8, 17, 8,
    public int minSteps(int n) {
        return g[n];
    }
}
```

- 时间复杂度：将打表逻辑配合 `static` 交给 OJ 执行，复杂度为 $O(C * \sqrt{C})$，C 为常数，固定为 1010；将打表逻辑放到本地执行，复杂度为 $O(1)$
- 空间复杂度：$O(C)$

> **🍭 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 **678. 有效的括号字符串** ，难度为 **中等**。

Tag：「有效括号问题」、「动态规划」、「模拟」

给定一个只包含三种字符的字符串：`(` ，`)` 和 `*`，写一个函数来检验这个字符串是否为有效字符串。

有效字符串具有如下规则：

- 任何左括号 `(` 必须有相应的右括号 `)`。
- 任何右括号 `)` 必须有相应的左括号 `(` 。
- 左括号 `(` 必须在对应的右括号之前 `)`。
- 可以被视为单个右括号 `)` ，或单个左括号 `(` ，或一个空字符串。
- 一个空字符串也被视为有效字符串。

示例 1:

```
输入: "()"

输出: True
```

示例 2:

```
输入: "(*)"

输出: True
```

示例 3:

```
输入: "(*))"

输出: True
```

注意:

- 字符串大小将在 [1，100] 范围内。

动态规划

定义 $f[i][j]$ 为考虑前 i 个字符（字符下标从 1 开始），能否与 j 个右括号形成合法括号序列。

起始时只有 $f[0][0]$ 为 $true$，最终答案为 $f[n][0]$。

不失一般性的考虑 $f[i][j]$ 该如何转移：

- 当前字符为 `(`：如果 $f[i][j]$ 为 $true$，必然有 $f[i-1][j-1]$ 为 $true$，反之亦然。即有 $f[i][j] = f[i-1][j-1]$；
- 当前字符为 `)`：如果 $f[i][j]$ 为 $true$，必然有 $f[i-1][j+1]$ 为 $true$，反之亦然。即有 $f[i][j] = f[i-1][j+1]$；
- 当前字符为 `*`：根据 `*` 代指的符号不同，分为三种情况，只有有一种情况为 $true$ 即可。即有 $f[i][j] = f[i-1][j-1] \vee f[i-1][j+1] \vee f[i-1][j]$。

代码：

```
class Solution {
    public boolean checkValidString(String s) {
        int n = s.length();
        boolean[][] f = new boolean[n + 1][n + 1];
        f[0][0] = true;
        for (int i = 1; i <= n; i++) {
            char c = s.charAt(i - 1);
            for (int j = 0; j <= i; j++) {
                if (c == '(') {
                    if (j - 1 >= 0) f[i][j] = f[i - 1][j - 1];
                } else if (c == ')') {
                    if (j + 1 <= i) f[i][j] = f[i - 1][j + 1];
                } else {
                    f[i][j] = f[i - 1][j];
                    if (j - 1 >= 0) f[i][j] |= f[i - 1][j - 1];
                    if (j + 1 <= i) f[i][j] |= f[i - 1][j + 1];
                }
            }
        }
        return f[n][0];
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

模拟

通过解法一，我们进一步发现，对于某个 $f[i][x]$ 而言（即动规数组中的某一行），值为 $true$

的必然为连续一段。

即 **由于存在可变化的 `*` 符号，因此考虑在考虑前 i 个字符，其能与消耗的左括号的数量具有明确的「上界与下界」。且当前上界与下界的变化，仅取决于「当前为何种字符」，以及「处理上一个字符时上界与下界为多少」。**

但直接记录所能消耗的左括号上限和下限需要处理较多的边界问题。

我们可以使用与（题解）301. 删除无效的括号 类似的思路：

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。

同时由于本题存在 `*`，因此我们需要记录得分的区间区间是多少，而不仅是一个具体的得分。

具体的，使用两个变量 `l` 和 `r` 分别表示「最低得分」和「最高得分」。

根据当前处理到的字符进行分情况讨论：

- 当前字符为 `(`：`l` 和 `r` 同时加一；
- 当前字符为 `)`：`l` 和 `r` 同时减一；
- 当前字符为 `*`：如果 `*` 代指成 `(` 的话，`l` 和 `r` 都进行加一；如果 `*` 代指成 `)` 的话，`l` 和 `r` 都进行减一；如果 `*` 不变的话，`l` 和 `r` 均不发生变化。因此总的 `l` 的变化为减一，总的 `r` 的变化为加一。

需要注意的是，在匹配过程中如果 `l` 为负数，需要重置为 0，因为如果当前序列本身为不合法括号序列的话，增加 `(` 必然还是不合法。同时，当出现 `l > r` 说明上界为负数，即右括号过多，必然为非合法方案，返回 $false$。

代码：

```java
class Solution {
    public boolean checkValidString(String s) {
        int l = 0, r = 0;
        for (char c : s.toCharArray()) {
            if (c == '(') {
                l++; r++;
            } else if (c == ')') {
                l--; r--;
            } else {
                l--; r++;
            }
            l = Math.max(l, 0);
            if (l > r) return false;
        }
        return l == 0;
    }
}
```

- 时间复杂度：$O(n)$
- 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 **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：

```java
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：

```Java
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 **

题目描述

这是 LeetCode 上的 **1751. 最多可以参加的会议数目 II**，难度为 **困难**。

Tag：「二分」、「线性 DP」

给你一个 events 数组，其中 events[i] = [startDayi, endDayi, valuei] ，表示第 i 个会议

在 startDayi 天开始，第 endDayi 天结束，如果你参加这个会议，你能得到价值 valuei 。

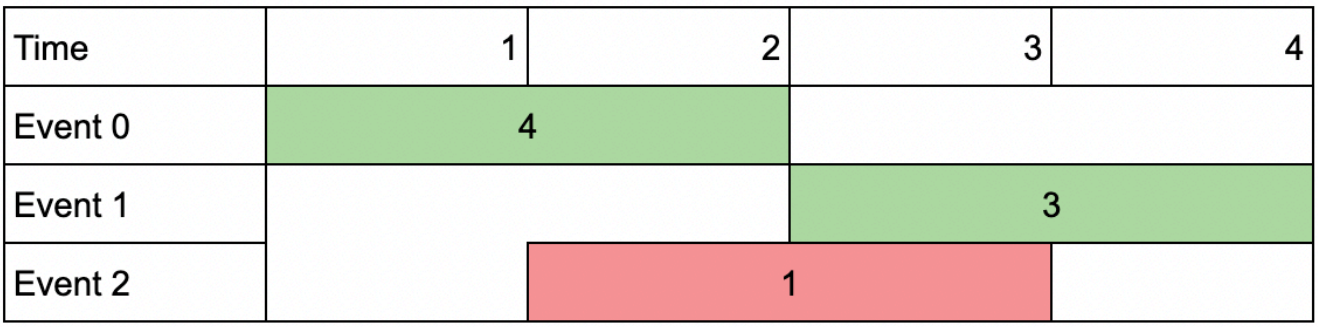
同时给你一个整数 k 表示你能参加的最多会议数目。

你同一时间只能参加一个会议。如果你选择参加某个会议，那么你必须 完整 地参加完这个会议。

会议结束日期是包含在会议内的，也就是说你不能同时参加一个开始日期与另一个结束日期相同的两个会议。

请你返回能得到的会议价值 最大和 。

示例 1：

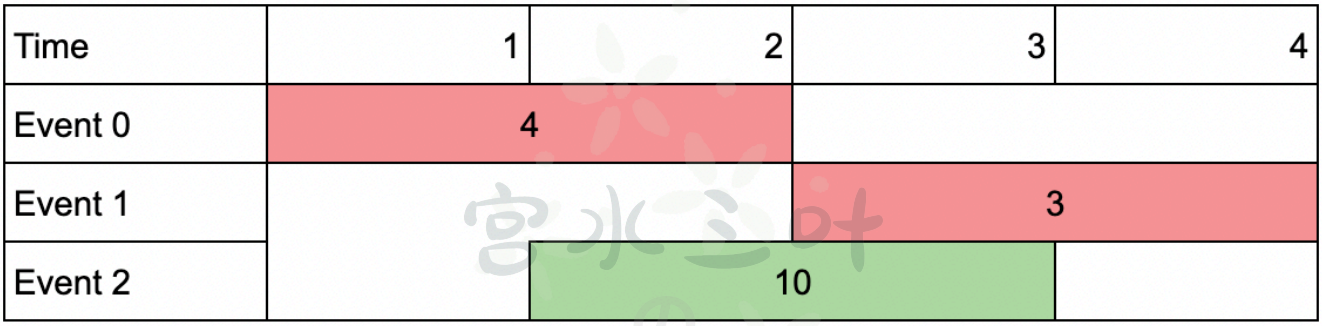


输入：events = [[1,2,4],[3,4,3],[2,3,1]], k = 2

输出：7

解释：选择绿色的活动会议 0 和 1，得到总价值和为 4 + 3 = 7 。

示例 2：

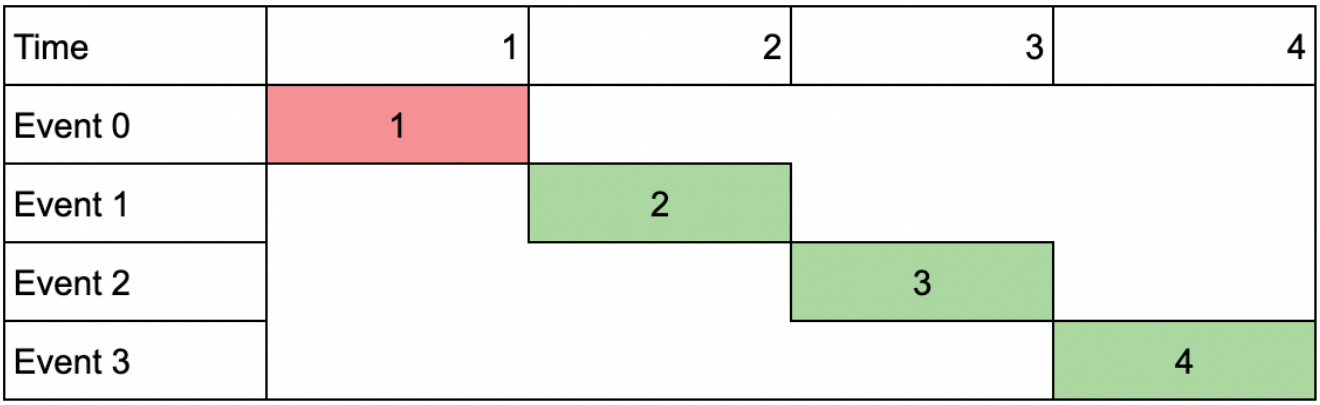


输入：events = [[1,2,4],[3,4,3],[2,3,10]], k = 2

输出：10

解释：参加会议 2 ，得到价值和为 10 。
你没法再参加别的会议了，因为跟会议 2 有重叠。你 不 需要参加满 k 个会议。

示例 3：



输入：events = [[1,1,1],[2,2,2],[3,3,3],[4,4,4]], k = 3

输出：9

解释：尽管会议互不重叠，你只能参加 3 个会议，所以选择价值最大的 3 个会议。

提示：

- 1 <= k <= events.length
- 1 <= k * events.length <= 10^6
- 1 <= startDayi <= endDayi <= 10^9
- 1 <= valuei <= 10^6

基本思路

定义 `f[i][j]` 为考虑前 `i` 个时间，选择不超过 `j` 的最大价值

对于每件时间，都有选择与不选两种选择，不选有 `f[i][j] = f[i - 1][j]`。

选择的话，则要找到第 i 件事件之前，与第 i 件事件不冲突的事件，记为 `last`，则有 `f[i][j] = f[last][j - 1]`。

两者取一个 max，则是 `f[i][j]` 的值。

分析到这里，因为我们要找 `last`，我们需要先对 `events` 的结束时间排序，然后找从右往左找，找到第一个满足 `结束时间 小于 当前事件的开始时间` 的事件，就是 `last`

而找 `last` 的过程，可以直接循环找，也可以通过二分来找，都能过。

动态规划

不通过「二分」来找 `last` 的 DP 解法。

代码：

```java
class Solution {
    public int maxValue(int[][] es, int k) {
        int n = es.length;
        Arrays.sort(es, (a, b)->a[1]-b[1]);
        int[][] f = new int[n + 1][k + 1]; // f[i][j] 代表考虑前 i 个事件，选择不超过 j 的最大价
        for (int i = 1; i <= n; i++) {
            int[] ev = es[i - 1];
            int s = ev[0], e = ev[1], v = ev[2];

            // 找到第 i 件事件之前，与第 i 件事件不冲突的事件
            // 对于当前事件而言，冲突与否，与 j 无关
            int last = 0;
            for (int p = i - 1; p >= 1; p--) {
                int[] prev = es[p - 1];
                if (prev[1] < s) {
                    last = p;
                    break;
                }
            }

            for (int j = 1; j <= k; j++) {
                f[i][j] = Math.max(f[i - 1][j], f[last][j - 1] + v);
            }
        }
        return f[n][k];
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$，循环 `n` 个事件，每次循环需要往回找一个事件，复杂度为 $O(n)$，并更新 `k` 个状态，复杂度为 $O(k)$，因此转移的复杂度为 $O(n * (n + k))$；总的复杂度为 $O(n * (n + k + \log n))$
- 空间复杂度：$O(n * k)$

二分 + 动态规划

通过「二分」来找 `last` 的 DP 解法。

代码：

```java
class Solution {
    public int maxValue(int[][] es, int k) {
        int n = es.length;
        Arrays.sort(es, (a, b)->a[1]-b[1]);
        int[][] f = new int[n + 1][k + 1]; // f[i][j] 代表考虑前 i 个事件，选择不超过 j 的最大价
        for (int i = 1; i <= n; i++) {
            int[] ev = es[i - 1];
            int s = ev[0], e = ev[1], v = ev[2];

            // 通过「二分」，找到第 i 件事件之前，与第 i 件事件不冲突的事件
            // 对于当前事件而言，冲突与否，与 j 无关
            int l = 1, r = i - 1;
            while (l < r) {
                int mid = l + r + 1 >> 1;
                int[] prev = es[mid - 1];
                if (prev[1] < s) {
                    l = mid;
                } else {
                    r = mid - 1;
                }
            }
            int last = (r > 0 && es[r - 1][1] < s) ? r : 0;

            for (int j = 1; j <= k; j++) {
                f[i][j] = Math.max(f[i - 1][j], f[last][j - 1] + v);
            }
        }
        return f[n][k];
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$，循环 `n` 个事件，每次循环需要往回找一个事件，复杂度为 $O(\log n)$，并更新 `k` 个状态，复杂度为 $O(k)$，因此转移的复杂度为 $O(n * (\log n + k))$；总的复杂度为 $O(n * (k + \log n))$
- 空间复杂度：$O(n * k)$

题目描述

这是 LeetCode 上的 **1787. 使所有区间的异或结果为零** ，难度为 **困难**。

Tag：「线性 DP」、「异或」、「数学」

给你一个整数数组 `nums` 和一个整数 `k` 。
区间 `[left, right]``（left <= right）`的 异或结果 是对下标位于 `left` 和 `right`（包括 `left` 和 `right` ）之间所有元素进行 XOR 运算的结果：`nums[left] XOR nums[left+1] XOR ... XOR nums[right]` 。

返回数组中**要更改的最小元素数** ，以使所有长度为 `k` 的区间异或结果等于零。

示例 1：

```
输入：nums = [1,2,0,3,0], k = 1

输出：3

解释：将数组 [1,2,0,3,0] 修改为 [0,0,0,0,0]
```

示例 2：

```
输入：nums = [3,4,5,2,1,7,3,4,7], k = 3

输出：3

解释：将数组 [3,4,5,2,1,7,3,4,7] 修改为 [3,4,7,3,4,7,3,4,7]
```

示例 3：

```
输入：nums = [1,2,4,1,2,5,1,2,6], k = 3

输出：3

解释：将数组[1,2,4,1,2,5,1,2,6] 修改为 [1,2,3,1,2,3,1,2,3]
```

提示：

- 1 <= k <= nums.length <= 2000
- 0 <= nums[i] < 2^{10}

基本分析

题目示例所包含的提示过于明显了，估计很多同学光看三个样例就猜出来了：**答案数组必然是每 k 个一组进行重复的。**

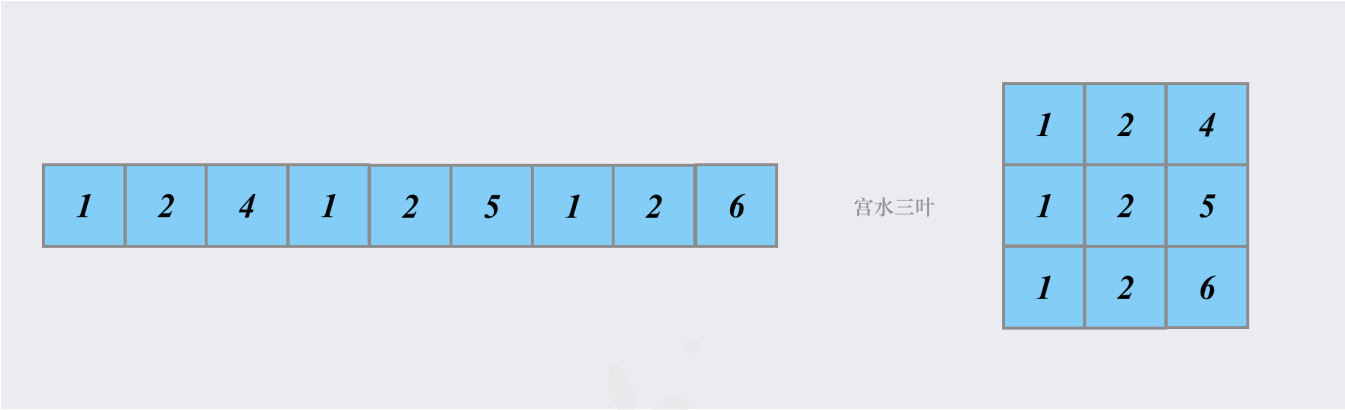
这样的性质是可由「答案数组中所有长度为 k 的区间异或结果为 0」推导出来的：

- 假设区间 $[i, j]$ 长度为 k，其异或结果为 0。即 $nums[i] \oplus nums[i+1] \oplus ... \oplus nums[j] = 0$
- 长度不变，将区间整体往后移动一位 $[i+1, j+1]$，其异或结果为 0。即 $nums[i+1] \oplus ... \oplus nums[j] \oplus nums[j+1] = 0$
- 两式结合，中间 $[i+1, j]$ 区间的数值出现两次，抵消掉最终剩下 $nums[i] \oplus nums[j+1] = 0$，即推出 $nums[i]$ 必然等于 $num[j+1]$。

即答案数组必然每 k 个一组进行重复。

也可以从滑动窗口的角度分析：窗口每滑动一格，会新增和删除一个值。对于异或而言，新增和删除都是对值本身做异或，因此新增值和删除值相等（保持窗口内异或值为 0）。

因此我们将这个一维的输入看成二维，从而将问题转化为：**使得最终每列相等，同时「首行」的异或值为 0 的最小修改次数。**



当然 n 和 k 未必成倍数关系，这时候最后一行可能为不足 k 个。这也是为什么上面没有说「每行异或结果为 0」，而是说「首行异或结果为 0」的原因：

动态规划

定义 $f[i][xor]$ 为考虑前 i 列，且首行前 i 列异或值为 xor 的最小修改次数，最终答案为 $f[k-1][0]$。

第一维的范围为 $[0, k)$，由输入参数给出；第二维为 $[0, 2^{10})$，根据题目给定的数据范围 `0 <= nums[i] < 2^10` 可得（异或为不进位加法，因此最大异或结果不会超过 2^{10}）。

为了方便，我们需要使用哈希表 map 记录每一列的数字分别出现了多少次，使用变量 cnt 统计该列有多少数字（因为最后一行可能不足 k 个）。

不失一般性的考虑 $f[i][xor]$ 如何转移:

- 当前处于第 0 列：由于没有前一列，这时候只需要考虑怎么把该列变成 xor 即可：

$$f[0][xor] = cnt - map.get(xor)$$

- 当前处于其他列：需要考虑与前面列的关系。
  我们知道最终的 $f[i][xor]$ 由两部分组成：一部分是前 $i-1$ 列的修改次数，一部分是当前列的修改次数。
  这时候需要分情况讨论：
  - **仅修改当列的部分数**：这时候需要从哈希表中遍历已存在的数，在所有方案中取 min：

$$f[i][xor] = f[i-1][xor \oplus cur] + cnt - map.get(cur)$$

  - **对整列进行修改替换**：此时当前列的修改成本固定为 cnt，只需要取前 $i-1$ 列的最小修改次数过来即可：

$$f[i][xor] = f[i-1][xor'] + cnt$$

最终 $f[i][xor]$ 在所有上述方案中取 min。为了加速「取前 $i-1$ 列的最小修改次数」的过程，我们可以多开一个 $g[]$ 数组来记录前一列的最小状态值。

代码：

```
class Solution {
    public int minChanges(int[] nums, int k) {
        int n = nums.length;
        int max = 1024;
        int[][] f = new int[k][max];
        int[] g = new int[k];
        for (int i = 0; i < k; i++) {
            Arrays.fill(f[i], 0x3f3f3f3f);
            g[i] = 0x3f3f3f3f;
        }
        for (int i = 0, cnt = 0; i < k; i++, cnt = 0) {
            // 使用 map 和 cnt 分别统计当前列的「每个数的出现次数」和「有多少个数」
            Map<Integer, Integer> map = new HashMap<>();
            for (int j = i; j < n; j += k) {
                map.put(nums[j], map.getOrDefault(nums[j], 0) + 1);
                cnt++;
            }
            if (i == 0) { // 第 0 列：只需要考虑如何将该列变为 xor 即可
                for (int xor = 0; xor < max; xor++) {
                    f[0][xor] = Math.min(f[0][xor], cnt - map.getOrDefault(xor, 0));
                    g[0] = Math.min(g[0], f[0][xor]);
                }
            } else { // 其他列：考虑与前面列的关系
                for (int xor = 0; xor < max; xor++) {
                    f[i][xor] = g[i - 1] + cnt; // 整列替换
                    for (int cur : map.keySet()) { // 部分替换
                        f[i][xor] = Math.min(f[i][xor], f[i - 1][xor ^ cur] + cnt - map.ge
                    }
                    g[i] = Math.min(g[i], f[i][xor]);
                }
            }
        }
        return f[k - 1][0];
    }
}
```

- 时间复杂度：共有 $O(C * k)$ 个状态需要被转移（其中 C 固定为 2^{10}），每个状态的转移需要遍历哈希表，最多有 $\frac{n}{k}$ 个数，复杂度为 $O(\frac{n}{k})$。整体复杂度为 $O(C *$

$n)$

- 空间复杂度：$O(C * k)$，其中 C 固定为 $2^{10} + 1$

> **🍭 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「线性 DP」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