宫水三叶的刷题日记

区间 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「区间 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「区间 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「区间 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 **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。

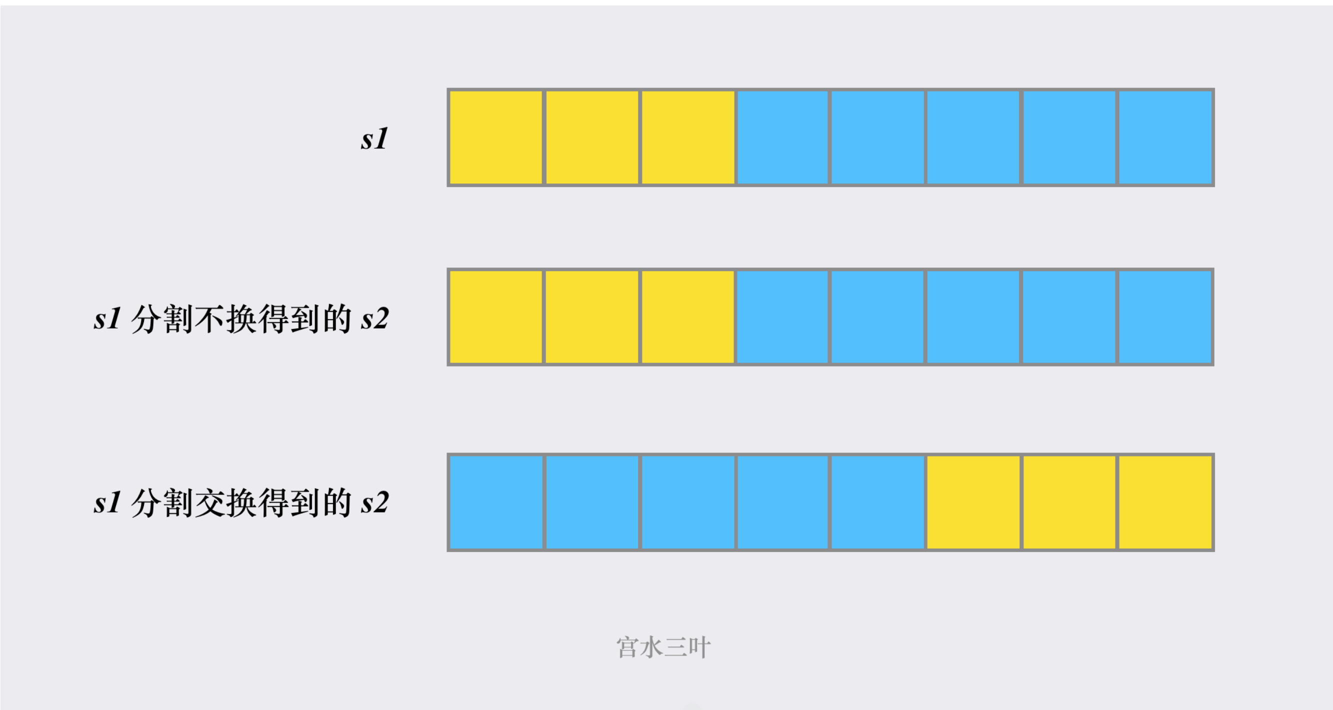
由于题目说了整个生成「扰乱字符串」的过程是通过「递归」来进行。

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：

```
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：

```
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 **516. 最长回文子序列** ，难度为 **中等**。

Tag : 「动态规划」、「区间 DP」

给你一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

示例 1：

```
输入：s = "bbbab"

输出：4

解释：一个可能的最长回文子序列为 "bbbb" 。
```

示例 2：

```
输入：s = "cbbd"

输出：2

解释：一个可能的最长回文子序列为 "bb" 。
```

提示：

- 1 <= s.length <= 1000
- s 仅由小写英文字母组成

动态规划

这是一道经典的区间 DP 题。

之所以可以使用区间 DP 进行求解，是因为在给定一个回文串的基础上，如果在回文串的边缘分

别添加两个新的字符，可以通过判断两字符是否相等来得知新串是否回文。

也就是说，使用小区间的回文状态可以推导出大区间的回文状态值。

从图论意义出发就是，任何一个长度为 len 的回文串，必然由「长度为 $len - 1$」或「长度为 $len - 2$」的回文串转移而来。

两个具有公共回文部分的回文串之间存在拓扑序（存在由「长度较小」回文串指向「长度较大」回文串的有向边）。

通常区间 DP 问题都是，常见的基本流程为：

1. 从小到大枚举区间大小 len
2. 枚举区间左端点 l，同时根据区间大小 len 和左端点计算出区间右端点 $r = l + len - 1$
3. 通过状态转移方程求 $f[l][r]$ 的值

因此，我们 **定义 $f[l][r]$ 为考虑区间 $[l, r]$ 的最长回文子序列长度为多少。**

不失一般性的考虑 $f[l][r]$ 该如何转移。

由于我们的状态定义 **没有限制** 回文串中必须要选 $s[l]$ 或者 $s[r]$。

我们对边界字符 $s[l]$ 和 $s[r]$ 分情况讨论，最终的 $f[l][r]$ 应该在如下几种方案中取 max ：

- 形成的回文串一定不包含 $s[l]$ 和 $s[r]$，即完全不考虑 $s[l]$ 和 $s[r]$：

$$f[l][r] = f[l+1][r-1]$$

- 形成的回文串可能包含 $s[l]$，但一定不包含 $s[r]$：

$$f[l][r] = f[l][r-1]$$

- 形成的回文串可能包含 $s[r]$，但一定不包含 $s[l]$：

$$f[l][r] = f[l+1][r]$$

- 形成的回文串可能包含 $s[l]$，也可能包含 $s[r]$，根据 $s[l]$ 和 $s[r]$ 是否相等：

$$f[l][r] = \begin{cases} f[l+1][r-1] + 2 & s[l] = s[r] \\ f[l+1][r-1] & s[l] \neq s[r] \end{cases}$$

需要说明的是，上述几种情况可以确保我们做到「不漏」，但不能确保「不重」，对于求最值问题，我们只需要确保「不漏」即可，某些状态重复参与比较，不会影响结果的正确性。

> 一些细节：我们需要特判掉长度为 1 和 2 的两种基本情况。当长度为 1 时，必然回文，当长度为 2 时，当且仅当两字符相等时回文。

代码：

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        int[][] f = new int[n][n];
        for (int len = 1; len <= n; len++) {
            for (int l = 0; l + len - 1 < n; l++) {
                int r = l + len - 1;
                if (len == 1) {
                    f[l][r] = 1;
                } else if (len == 2) {
                    f[l][r] = cs[l] == cs[r] ? 2 : 1;
                } else {
                    f[l][r] = Math.max(f[l + 1][r], f[l][r - 1]);
                    f[l][r] = Math.max(f[l][r], f[l + 1][r - 1] + (cs[l] == cs[r] ? 2 : 0)
                }
            }
        }
        return f[0][n - 1];
    }
}
```

- 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 **664. 奇怪的打印机** ，难度为 **困难**。

Tag : 「区间 DP」

有台奇怪的打印机有以下两个特殊要求：

- 打印机每次只能打印由 同一个字符 组成的序列。
- 每次可以在任意起始和结束位置打印新字符，并且会覆盖掉原来已有的字符。

给你一个字符串 s ，你的任务是计算这个打印机打印它需要的最少打印次数。

示例 1：

```
输入：s = "aaabbb"

输出：2

解释：首先打印 "aaa" 然后打印 "bbb"。
```

示例 2：

```
输入：s = "aba"

输出：2

解释：首先打印 "aaa" 然后在第二个位置打印 "b" 覆盖掉原来的字符 'a'。
```

提示：

- 1 <= s.length <= 100
- s 由小写英文字母组成

基本分析

首先，根据题意我们可以分析出一个重要推论：**连续相同的一段字符，必然可以归到同一次打印中，而不会让打印次数变多。注意，这里说的是「归到一次」，而不是说「单独作为一次」。**

怎么理解这句话呢？

举个🌰，对于诸如 `...bbaaabb...` 的样例数据，其中多个连续的 `a` 必然可以归到同一次打印中，但这一次打印可能只是将 `aaa` 作为整体进行打印；也有可能是 `aaa` 与前面或者后面的 `a` 作为整体被打印（然后中间的 `b` 被后来的打印所覆盖）。但无论是何种情况连续一段的

`aaa` 必然是可以「归到同一次打印」中。

我们可以不失一般性证明「连续相同的一段字符，必然可以归到同一次打印中，而不会让打印次数变多」这个推理是否正确：

假设有目标序列 $[..., ai, ..., aj, ...]$ 其中 $[i, j]$ 连续一段字符相同，假如这一段的打印被最后完成（注意最后完成不代表这一段要保留空白，这一段可以此前被打印多次），除了这一段以外所消耗的打印次数为 x，那么根据 $[i, j]$ 不同的打印方案有：

1. 将 $[i, j]$ 单纯划分为多段：总共打印的次数大于 $x + 1$（此方案不会取到打印最小值 $x + 1$，可忽略）
2. 将 $[i, j]$ 归到同一次打印：总共打印的次数等于 $x + 1$
3. 将 $[i, j]$ 结合之前的打印划分为多段，即 $[i, j]$ 一段的两段本身就是「目标字符」，我们本次只需要打印 $[i, j]$ 中间的部分。总共打印的次数等于 $x + 1$

由于同样的地方可以被重复打印，因此我们可以将情况 3 中打印边缘扩展到 i 和 j 处，这样最终打印结果不变，而且总的打印次数没有增加。

到这一步，我们其实已经证明出「连续相同的一段字符，必然可以归到同一次打印中，而不会让打印次数变多」的推论成立了。

但可能会有同学提出疑问：怎么保证 $[i, j]$ 是被最后涂的？怎么保证 $[i, j]$ 不是和其他「不相邻的同样字符」一起打印的？

答案是不用保证，因为不同状态（打印结果）之间相互独立，而有明确的最小转移成本。即从当前打印结果 `a` 变成打印结果 `b`，是具有明确的最小打印次数的（否则本题无解）。因此我们上述的分析可以看做任意两个中间状态转移的“最后一步”，而且不会整体的结果。

对应到本题，题目给定的起始状态是空白字符串 `a`，目标状态是入参字符串 `s`。那么真实最优解中，从 `a` 状态到 `s` 状态中间可能会经过任意个中间状态，假设有两个中间状态 `p` 和 `q`，那么我们上述的分析就可以应用到中间状态 `p` 到 `q` 的转移中，可以令得 `p` 到 `q` 转移所花费的转移成本最低（最优），同时这个转移不会影响「`a` 到 `p` 的转移」和「`q` 到 `s` 的转移」，是相互独立的。

因此这个分析可以推广到真实最优转移路径中的任意一步，是一个具有一般性的结论。

上述分析是第一个切入点，第二个切入点是「重复打印会进行覆盖」，这意味着我们其实不需要确保 $[i, j]$ 这一段在目标字符串中完全相同，而只需要 $s[i] = s[j]$ 相同即可，即后续打印不会从边缘上覆盖 $[i, j]$ 区间的原有打印，否则 $[i, j]$ 这一段的打印就能用范围更小的区间所代替。

这样就引导出我们状态转移的关键：状态转移之间只需要确保首位字符相同。

动态规划

定义 $f[l][r]$ 为将 $[l, r]$ 这一段打印成目标结果所消耗的最小打印次数。

不失一般性考虑 $f[l][r]$ 该如何转移：

- 只染 l 这个位置，此时 $f[l][r] = f[l + 1][r] + 1$
- 不只染 l 这个位置，而是从 l 染到 k（需要确保首位相同 $s[l] = s[k]$）：$f[l][r] = f[l][k - 1] + f[k + 1][r], l < k <= r$

其中状态转移方程中的情况 2 需要说明一下：由于我们只确保 $s[l] = s[k]$，并不确保 $[l, k]$ 之间的字符相同，根据我们基本分析可知，$s[k]$ 这个点可由打印 $s[l]$ 的时候一同打印，因此本身 $s[k]$ 并不独立消耗打印次数，所以这时候 $[l, k]$ 这一段的最小打印次数应该取 $f[l][k - 1]$，而不是 $f[l][k]$。

最终的 $f[l][r]$ 为上述所有方案中取 min。

代码：

```
class Solution {
    public int strangePrinter(String s) {
        int n = s.length();
        int[][] f = new int[n + 1][n + 1];
        for (int len = 1; len <= n; len++) {
            for (int l = 0; l + len - 1 < n; l++) {
                int r = l + len - 1;
                f[l][r] = f[l + 1][r] + 1;
                for (int k = l + 1; k <= r; k++) {
                    if (s.charAt(l) == s.charAt(k)) {
                        f[l][r] = Math.min(f[l][r], f[l][k - 1] + f[k + 1][r]);
                    }
                }
            }
        }
        return f[0][n - 1];
    }
}
```

- 时间复杂度：$O(n^3)$
- 空间复杂度：$O(n^2)$

总结

这道题的原型应该出自 String painter。

如果只是为了把题做出来，难度不算特别大，根据数据范围 10^2，可以猜到是 $O(n^3)$ 做法，通常就是区间 DP 的「枚举长度 + 枚举左端点 + 枚举分割点」的三重循环。

但是要搞懂为啥可以这样做，还是挺难，大家感兴趣的话可以好好想想 ~ 🤣

> **🍭 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 **877. 石子游戏** ，难度为 **中等**。

Tag : 「区间 DP」、「博弈论」

亚历克斯和李用几堆石子在做游戏。偶数堆石子排成一行，每堆都有正整数颗石子 piles[i] 。

游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。

亚历克斯和李轮流进行，亚历克斯先开始。 每回合，玩家从行的开始或结束处取走整堆石头。这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 true ，当李赢得比赛时返回 false 。

示例：

```
输入：[5,3,4,5]
输出：true
解释：
亚历克斯先开始，只能拿前 5 颗或后 5 颗石子 。
假设他取了前 5 颗，这一行就变成了 [3,4,5] 。
如果李拿走前 3 颗，那么剩下的是 [4,5]，亚历克斯拿走后 5 颗赢得 10 分。
如果李拿走后 5 颗，那么剩下的是 [3,4]，亚历克斯拿走后 4 颗赢得 9 分。
这表明，取前 5 颗石子对亚历克斯来说是一个胜利的举动，所以我们返回 true 。
```

提示：

- 2 <= piles.length <= 500
- piles.length 是偶数。
- 1 <= piles[i] <= 500
- sum(piles) 是奇数。

动态规划

定义 $f[l][r]$ 为考虑区间 $[l, r]$，在双方都做最好选择的情况下，先手与后手的最大得分差值为多少。

那么 $f[1][n]$ 为考虑所有石子，先手与后手的得分差值：

- $f[1][n] > 0$，则先手必胜，返回 `True`
- $f[1][n] < 0$，则先手必败，返回 `False`

不失一般性的考虑 $f[l][r]$ 如何转移。根据题意，只能从两端取石子（令 $piles$ 下标从 1 开始），共两种情况：

- 从左端取石子，价值为 $piles[l-1]$；取完石子后，原来的后手变为先手，从 $[l+1, r]$ 区间做最优决策，所得价值为 $f[l+1][r]$。**因此本次先手从左端点取石子的话，双方差值为：**

$$piles[l-1] - f[l+1][r]$$

- 从右端取石子，价值为 $piles[r-1]$；取完石子后，原来的后手变为先手，从 $[l, r-1]$ 区间做最优决策，所得价值为 $f[l][r-1]$。**因此本次先手从右端点取石子的话，双方差值为：**

$$piles[r-1] - f[l][r-1]$$

双方都想赢，都会做最优决策（即使自己与对方分差最大）。因此 $f[l][r]$ 为上述两种情况中的最大值。

根据状态转移方程，我们发现大区间的状态值依赖于小区间的状态值，典型的区间 DP 问题。

按照从小到大「枚举区间长度」和「区间左端点」的常规做法进行求解即可。

代码：

```
class Solution {
    public boolean stoneGame(int[] ps) {
        int n = ps.length;
        int[][] f = new int[n + 2][n + 2];
        for (int len = 1; len <= n; len++) { // 枚举区间长度
            for (int l = 1; l + len - 1 <= n; l++) { // 枚举左端点
                int r = l + len - 1; // 计算右端点
                int a = ps[l - 1] - f[l + 1][r];
                int b = ps[r - 1] - f[l][r - 1];
                f[l][r] = Math.max(a, b);
            }
        }
        return f[1][n] > 0;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

博弈论

事实上，这还是一道很经典的博弈论问题，也是最简单的一类博弈论问题。

为了方便，我们称「石子序列」为石子在原排序中的编号，下标从 1 开始。

由于石子的堆数为偶数，且只能从两端取石子。**因此先手后手所能选择的石子序列，完全取决于先手每一次决定。**

由于石子的堆数为偶数，对于先手而言：**每一次的决策局面，都能「自由地」选择奇数还是偶数**

的序列，从而限制后手下一次「只能」奇数还是偶数石子。

具体的，对于本题，**由于石子堆数为偶数，因此先手的最开始局面必然是** $[奇数, 偶数]$**，即必然是「奇偶性不同的局面」；当先手决策完之后，交到给后手的要么是** $[奇数, 奇数]$ **或者** $[偶数, 偶数]$**，即必然是「奇偶性相同的局面」；后手决策完后，又恢复「奇偶性不同的局面」交回到先手** ...

不难归纳推理，这个边界是可以应用到每一个回合。

因此先手只需要在进行第一次操作前计算原序列中「奇数总和」和「偶数总和」哪个大，然后每一次决策都「限制」对方只能选择「最优奇偶性序列」的对立面即可。

同时又由于所有石子总和为奇数，堆数为偶数，即没有平局，所以先手必胜。

代码：

```
class Solution {
    public boolean stoneGame(int[] piles) {
        return true;
    }
}
```

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

> **🍭 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「区间 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