宫水三叶的刷题日记

序列 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「序列 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「序列 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「序列 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 **354. 俄罗斯套娃信封问题** ，难度为 **困难**。

Tag : 「二分」、「序列 DP」

给你一个二维整数数组 envelopes ，其中 envelopes[i] = [wi, hi] ，表示第 i 个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算「最多能有多少个」信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

```
输入：envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出：3
解释：最多信封的个数为 3, 组合为: [2,3] => [5,4] => [6,7]。
```

示例 2：

```
输入：envelopes = [[1,1],[1,1],[1,1]]
输出：1
```

提示：

- 1 <= envelopes.length <= 5000
- envelopes[i].length == 2
- 1 <= wi, hi <= 10^4

动态规划

这是一道经典的 DP 模型题目：最长上升子序列（LIS）。

首先我们先对 `envelopes` 进行排序，确保信封是从小到大进行排序。

问题就转化为我们从这个序列中选择 `k` 个信封形成新的序列，使得新序列中的每个信封都能严格覆盖前面的信封（宽高都严格大于）。

我们可以**定义状态 $f[i]$ 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值。**

对于每个 $f[i]$ 而言，最小值为 1，代表只选择自己一个信封。

那么对于一般的 $f[i]$ 该如何求解呢？因为第 i 件物品是必须选择的。我们可以枚举前面的 $i - 1$ 件物品，哪一件可以作为第 i 件物品的上一件物品。

在前 $i - 1$ 件物品中只要有符合条件的，我们就使用 $max(f[i], f[j] + 1)$ 更新 $f[i]$。

然后在所有方案中取一个 `max` 即是答案。

代码：

```
class Solution {
    public int maxEnvelopes(int[][] es) {
        int n = es.length;
        if (n == 0) return n;
        // 因为我们在找第 i 件物品的前一件物品时，会对前面的 i - 1 件物品都遍历一遍，因此第二维（高度）
        Arrays.sort(es, (a, b)->a[0]-b[0]);
        int[] f = new int[n]; // f(i) 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值
        int ans = 1;
        for (int i = 0; i < n; i++) {
            // 对于每个 f[i] 都满足最小值为 1
            f[i] = 1;
            // 枚举第 i 件物品的前一件物品，
            for (int j = i - 1; j >= 0; j--) {
                // 只要有满足条件的前一件物品，我们就尝试使用 f[j] + 1 更新 f[i]
                if (check(es, j, i)) {
                    f[i] = Math.max(f[i], f[j] + 1);
                }
            }
            // 在所有的 f[i] 中取 max 作为 ans
            ans = Math.max(ans, f[i]);
        }
        return ans;
    }
    boolean check(int[][] es, int mid, int i) {
        return es[mid][0] < es[i][0] && es[mid][1] < es[i][1];
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

二分 + 动态规划

上述方案其实算是一个朴素方案，复杂度是 $O(n^2)$ 的，也是我最先想到思路，但是题目没有给出数据范围，也不知道能不能过。

唯唯诺诺交了一个居然过了。

下面讲下其他优化解法。

首先还是和之前一样，我们可以通过复杂度分析来想优化方向。

指数算法往下优化就是对数解法或者线性解法。

仔细观察朴素解法，其实可优化的地方主要就是找第 i 件物品的前一件物品的过程。

如果想要加快这个查找过程，我们需要使用某种数据结构进行记录。

并且是边迭代边更新数据结构里面的内容。

首先因为我们对 `w` 进行了排序（从小到大），然后迭代也是从前往后进行，因此我们只需要保证迭代过程中，对于 `w` 相同的数据不更新，就能保证 `g` 中只会出现满足 `w` 条件的信封。

到这一步，还需要用到的东西有两个：一个是 `h`，因为只有 `w` 和 `h` 都同时满足，我们才能加入上升序列中；一个是信封所对应的上升序列长度，这是我们加速查找的核心。

我们使用数组 `g` 来记录，$g[i]$ 表示长度为 i 的最长上升子序列的中的最小「信封高度」，同时需要使用 `len` 记录当前记录到的最大长度。

还是不理解？没关系，我们可以直接看看代码，我把基本逻辑写在了注释当中（你的重点应该落在对 $g[]$ 数组的理解上）。

代码：

```java
class Solution {
    public int maxEnvelopes(int[][] es) {
        int n = es.length;
        if (n == 0) return n;
        // 由于我们使用了 g 记录高度，因此这里只需将 w 从小到达排序即可
        Arrays.sort(es, (a, b)->a[0] - b[0]);
        // f(i) 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值
        int[] f = new int[n];
        // g(i) 记录的是长度为 i 的最长上升子序列的最小「信封高度」
        int[] g = new int[n];
        // 因为要取 min，用一个足够大（不可能）的高度初始化
        Arrays.fill(g, Integer.MAX_VALUE);
        g[0] = 0;
        int ans = 1;
        for (int i = 0, j = 0, len = 1; i < n; i++) {
            // 对于 w 相同的数据，不更新 g 数组
            if (es[i][0] != es[j][0]) {
                // 限制 j 不能越过 i，确保 g 数组中只会出现第 i 个信封前的「历史信封」
                while (j < i) {
                    int prev = f[j], cur = es[j][1];
                    if (prev == len) {
                        // 与当前长度一致了，说明上升序列多增加一位
                        g[len++] = cur;
                    } else {
                        // 始终保留最小的「信封高度」，这样可以确保有更多的信封可以与其行程上升序列
                        // 举例：同样是上升长度为 5 的序列，保留最小高度为 5 记录（而不是保留任意的，
                        g[prev] = Math.min(g[prev], cur);
                    }
                    j++;
                }
            }

            // 二分过程
            // g[i] 代表的是上升子序列长度为 i 的「最小信封高度」
            int l = 0, r = len;
            while (l < r) {
                int mid = l + r >> 1;
                // 令 check 条件为 es[i][1] <= g[mid]（代表 w 和 h 都严格小于当前信封）
                // 这样我们找到的就是满足条件，最靠近数组中心点的数据（也就是满足 check 条件的最大下标
                // 对应回 g[] 数组的含义，其实就是找到 w 和 h 都满足条件的最大上升长度
                if (es[i][1] <= g[mid]) {
                    r = mid;
                } else {
                    l = mid + 1;
                }
            }
```

```
                // 更新 f[i] 与答案
                f[i] = r;
                ans = Math.max(ans, f[i]);
            }
            return ans;
        }
}
```

- 时间复杂度：对于每件物品都是通过「二分」找到其前一件物品。复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

证明

我们可以这样做的前提是 `g` 数组具有二段性，可以通过证明其具有「单调性」来实现。

当然这里指的是 `g` 被使用的部分，也就是 $[0, len-1]$ 的部分。

我们再回顾一下 `g[]` 数组的定义：$g[i]$ **表示长度为 i 的最长上升子序列的中的最小「信封高度」**

例如 $g[] = [0, 3, 4, 5]$ 代表的含义是：

- 上升序列长度为 0 的最小历史信封高度为 0
- 上升序列长度为 1 的最小历史信封高度为 3
- 上升序列长度为 2 的最小历史信封高度为 4
- 上升序列长度为 3 的最小历史信封高度为 5

可以通过反证法来证明其单调性:

假设 $g[]$ 不具有单调性，即至少有 $g[i] > g[j]$ ($i < j$，令 $a = g[i], b = g[j]$)

显然与我们的处理逻辑冲突。因为如果考虑一个「最小高度」为 `b` 的信封能够凑出长度为 `j` 的上升序列，自然也能凑出比 `j` 短的上升序列，对吧？

举个🌰，我们有信封：1,1],[2,2],[3,3],[4,4],[5,5，我们能凑出很多种长度为 2 的上升序列方案，其中最小的方案是高度最小的方案是 1,1],[2,2。因此这时候 g[2] = 2，代表能凑出长度为 2 的上升序列所 **必须使用的信封** 的最小高度为 2。

这时候反过来考虑，如果使用 [2,2] 能够凑出长度为 2 的上升序列，必然也能凑出长度为 1 的上升序列（删除前面的其他信封即可）。

推而广之，如果我们有 $g[j] = b$，也就是凑成长度为 `j` 必须使用的最小信封高度为 b。那么我必然能够保留高度为 `b` 的信封，删掉上升序列中的一些信封，凑成任意长度比 `j` 小的上升序列。

综上，$g[i] > g[j]$（$i < j$）与处理逻辑冲突，$g[]$ 数组为严格单调上升数组。

既然 $g[]$ 具有单调性，我们可以通过「二分」找到恰满足 check 条件的最大下标（最大下标达标表示最长上升序列长度）。

树状数组 + 动态规划

在「二分 + 动态规划」的解法中，我们通过「二分」来优化找第 i 个文件的前一个文件过程。

这个过程同样能通过「树状数组」来实现。

首先仍然是对 `w` 进行排序，然后使用「树状数组」来维护 h 维度的前缀最大值。

对于 `h` 的高度，我们只关心多个信封之间的大小关系，而不关心具体相差多少，我们需要对 `h` 进行离散化。

通常使用「树状数组」都需要进行离散化，尤其是这里我们本身就要使用 $O(n)$ 的空间来存储 dp 值。

代码：

```
class Solution {
    int[] tree;
    int lowbit(int x) {
        return x & -x;
    }

    public int maxEnvelopes(int[][] es) {
        int n = es.length;
        if (n == 0) return n;

        // 由于我们使用了 g 记录高度，因此这里只需将 w 从小到达排序即可
        Arrays.sort(es, (a, b)->a[0] - b[0]);

        // 先将所有的 h 进行离散化
        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < n; i++) set.add(es[i][1]);
        int cnt = set.size();
        int[] hs = new int[cnt];
        int idx = 0;
        for (int i : set) hs[idx++] = i;
        Arrays.sort(hs);
        for (int i = 0; i < n; i++) es[i][1] = Arrays.binarySearch(hs, es[i][1]) + 1;

        // 创建树状数组
        tree = new int[cnt + 1];

        // f(i) 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值
        int[] f = new int[n];
        int ans = 1;
        for (int i = 0, j = 0; i < n; i++) {
            // 对于 w 相同的数据，不更新 tree 数组
            if (es[i][0] != es[j][0]) {
                // 限制 j 不能越过 i，确保 tree 数组中只会出现第 i 个信封前的「历史信封」
                while (j < i) {
                    for (int u = es[j][1]; u <= cnt; u += lowbit(u)) {
                        tree[u] = Math.max(tree[u], f[j]);
                    }
                    j++;
                }
            }
            f[i] = 1;
            for (int u = es[i][1] - 1; u > 0; u -= lowbit(u)) {
                f[i] = Math.max(f[i], tree[u] + 1);
            }
            ans = Math.max(ans, f[i]);
        }
```

```
        return ans;
    }
}
```

- 时间复杂度：处理每个物品时更新「树状数组」复杂度为$O(\log n)$。整体复杂度为 $O(n \log n)$
- 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 **368. 最大整除子集** ，难度为 **中等**。

Tag : 「序列 DP」

给你一个由 无重复 正整数组成的集合 nums ，请你找出并返回其中最大的整除子集 answer ，子集中每一元素对 (answer[i], answer[j]) 都应当满足：

- answer[i] % answer[j] == 0 ，或
- answer[j] % answer[i] == 0

如果存在多个有效解子集，返回其中任何一个均可。

示例 1：

```
输入：nums = [1,2,3]

输出：[1,2]

解释：[1,3] 也会被视为正确答案。
```

示例 2：

```
输入：nums = [1,2,4,8]

输出：[1,2,4,8]
```

提示：

- 1 <= nums.length <= 1000
- 1 <= nums[i] <= 2 * 10^9
- nums 中的所有整数 互不相同

基本分析

根据题意：**对于符合要求的「整除子集」中的任意两个值，必然满足「较大数」是「较小数」的倍数。**

数据范围是 10^3，我们不可能采取获取所有子集，再检查子集是否合法的爆搜解法。

通常「递归」做不了，我们就往「递推」方向去考虑。

由于存在「整除子集」中任意两个值必然存在倍数/约数关系的性质，我们自然会想到对 `nums` 进行排序，然后从集合 `nums` 中从大到小进行取数，每次取数只考虑当前决策的数是否与「整除子集」中的最后一个数成倍数关系即可。

这时候你可能会想枚举每个数作为「整除子集」的起点，然后从前往后遍历一遍，每次都将符合「与当前子集最后一个元素成倍数」关系的数加入答案。

举个🌰，假设有原数组 `[1,2,4,8]`，“或许”我们期望的决策过程是：

1. 遍历到数字 `1`，此时「整除子集」为空，加到「整除子集」中；
2. 遍历到数字 `2`，与「整除子集」的最后一个元素（`1`）成倍数关系，加到「整除子集」中；
3. 遍历到数字 `4`，与「整除子集」的最后一个元素（`2`）成倍数关系，自然也与 `2` 之前的元素成倍数关系，加到「整除子集」中；
4. 遍历到数字 `8`，与「整除子集」的最后一个元素（`4`）成倍数关系，自然也与 `4` 之前的元素成倍数关系，加到「整除子集」中。

但这样的做法只能够确保得到「合法解」，无法确保得到的是「最长整除子集」。

当时担心本题数据太弱，上述错误的解法也能够通过，所以还特意实现了一下，还好被卡住了（🤣

同时也得到这个反例：`[9,18,54,90,108,180,360,540,720]`，如果按照我们上述逻辑，我们得到的是 `[9,18,54,108,540]` 答案（长度为 5），但事实上存在更长的「整除子集」：`[9,18,90,180,360,720]`（长度为 6）。

其本质是因为同一个数的不同倍数之间不存在必然的「倍数/约数关系」，而只存在「具有公约数」的性质，这会导致我们「模拟解法」错过最优解。

比如上述🌰，`54` & `90` 和 `18` 存在倍数关系，但两者本身不存在倍数关系。

因此当我们决策到某一个数 `nums[i]` 时（`nums` 已排好序），我们无法直接将 `nums[i]` 直接接在符合「约数关系」的、最靠近位置 `i` 的数后面，而是要**检查位置 `i` 前面的所有符合「约数关系」的位置，找一个已经形成「整除子集」长度最大的数**。

换句话说，当我们对 `nums` 排好序并从前往后处理时，在处理到 `nums[i]` 时，我们希望知道位置 `i` 之前的下标已经形成的「整除子集」长度是多少，然后从中选一个最长的「整除子集」，将 `nums[i]` 接在后面（前提是符合「倍数关系」）。

动态规划

基于上述分析，我们不难发现这其实是一个序列 DP 问题：**某个状态的转移依赖于与前一个状态的关系。即 `nums[i]` 能否接在 `nums[j]` 后面，取决于是否满足 `nums[i] % nums[j] == 0` 条件。**

可看做是「最长上升子序列」问题的变形题。

定义 $f[i]$ 为考虑前 `i` 个数字，且以第 `i` 个数为结尾的最长「整除子集」长度。

我们不失一般性的考虑任意位置 `i`，存在两种情况：

- 如果在 `i` 之前找不到符合条件 `nums[i] % nums[j] == 0` 的位置 `j`，那么 `nums[i]` 不能接在位置 `i` 之前的任何数的后面，只能自己独立作为「整除子集」的第一个数，此时状态转移方程为 $f[i] = 1$；
- 如果在 `i` 之前能够找到符合条件的位置 `j`，则取所有符合条件的 `f[j]` 的最大值，代表如果希望找到以 `nums[i]` 为结尾的最长「整除子集」，需要将 `nums[i]` 接到符合条件的最长的 `nums[j]` 后面，此时状态转移方程为 $f[i] = f[j] + 1$。

同时由于我们需要输出具体方案，需要额外使用 `g[]` 数组来记录每个状态是由哪个状态转移而

来。

定义 $g[i]$ 为记录 $f[i]$ 是由哪个下标的状态转移而来，如果 $f[i] = f[j] + 1$, 则有 $g[i] = j$。

对于求方案数的题目，多开一个数组来记录状态从何转移而来是最常见的手段。

当我们求得所有的状态值之后，可以对 `f[]` 数组进行遍历，取得具体的最长「整除子集」长度和对应下标，然后使用 `g[]` 数组进行回溯，取得答案。

代码：

```java
class Solution {
    public List<Integer> largestDivisibleSubset(int[] nums) {
        Arrays.sort(nums);
        int n = nums.length;
        int[] f = new int[n];
        int[] g = new int[n];
        for (int i = 0; i < n; i++) {
            // 至少包含自身一个数，因此起始长度为 1，由自身转移而来
            int len = 1, prev = i;
            for (int j = 0; j < i; j++) {
                if (nums[i] % nums[j] == 0) {
                    // 如果能接在更长的序列后面，则更新「最大长度」&「从何转移而来」
                    if (f[j] + 1 > len) {
                        len = f[j] + 1;
                        prev = j;
                    }
                }
            }
            // 记录「最终长度」&「从何转移而来」
            f[i] = len;
            g[i] = prev;
        }

        // 遍历所有的 f[i]，取得「最大长度」和「对应下标」
        int max = -1, idx = -1;
        for (int i = 0; i < n; i++) {
            if (f[i] > max) {
                idx = i;
                max = f[i];
            }
        }

        // 使用 g[] 数组回溯出具体方案
        List<Integer> ans = new ArrayList<>();
        while (ans.size() != max) {
            ans.add(nums[idx]);
            idx = g[idx];
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

证明

之所以上述解法能够成立，问题能够转化为「最长上升子序列（LIS）」问题进行求解，本质是利用了「全序关系」中的「可传递性」。

在 LIS 问题中，我们是利用了「关系运算符 $\geqslant$ 」的传递性，因此当我们某个数 `a` 能够接在 `b` 后面，只需要确保 $a \geqslant b$ 成立，即可确保 `a` 大于等于 `b` 之前的所有值。

那么同理，如果我们想要上述解法成立，我们还需要证明如下内容：

- **「倍数/约数关系」具有传递性**

由于我们将 `nums[i]` 往某个数字后面接时（假设为 `nums[j]` ），只检查了其与 `nums[j]` 的关系，并没有去检查 `nums[i]` 与 `nums[j]` 之前的数值是否具有「倍数/约数关系」。

换句话说，我们只确保了最终答案 `[a1, a2, a3, ..., an]` 相邻两数值之间具有「倍数/约数关系」，并不明确任意两值之间具有「倍数/约数关系」。

因此需要证得由 $a|b$ 和 $b|c$，可推导出 $a|c$ 的传递性：

由 $a|b$ 可得 $b = x * a$
由 $b|c$ 可得 $c = y * b$

最终有 $c = y * b = y * x * a$，由于 x 和 y 都是整数，因此可得 $a|c$。

得证「倍数/约数关系」具有传递性。

> **🍭 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 **446. 等差数列划分 II - 子序列** ，难度为 **困难**。

Tag : 「动态规划」、「序列 DP」、「容斥原理」、「数学」

给你一个整数数组 nums ，返回 nums 中所有 等差子序列 的数目。

如果一个序列中 至少有三个元素 ，并且任意两个相邻元素之差相同，则称该序列为等差序列。

- 例如，[1, 3, 5, 7, 9]、[7, 7, 7, 7] 和 [3, -1, -5, -9] 都是等差序列。
- 再例如，[1, 1, 2, 5, 7] 不是等差序列。

数组中的子序列是从数组中删除一些元素（也可能不删除）得到的一个序列。

- 例如，[2,5,10] 是 [1,2,1,2,4,1,5,10] 的一个子序列。

题目数据保证答案是一个 32-bit 整数。

示例 1：

```
输入：nums = [2,4,6,8,10]

输出：7

解释：所有的等差子序列为：
[2,4,6]
[4,6,8]
[6,8,10]
[2,4,6,8]
[4,6,8,10]
[2,4,6,8,10]
[2,6,10]
```

示例 2：

```
输入：nums = [7,7,7,7,7]

输出：16

解释：数组中的任意子序列都是等差子序列。
```

提示：

- 1 <= nums.length <= 1000
- -2^{31} <= nums[i] <= 2^{31} - 1

基本分析

从题目描述来看，我们可以确定这是一个「序列 DP」问题，通常「序列 DP」需要 $O(n^2)$ 的时

间复杂度，而某些具有特殊性质的「序列 DP」问题，例如 LIS 问题，能够配合贪心思路 + 二分做到 $O(n \log n)$ 复杂度。再看一眼数据范围为 10^3，基本可以确定这是一道复杂度为 $O(n^2)$ 的「序列 DP」问题。

动态规划 + 容斥原理

既然分析出是序列 DP 问题，我们可以先猜想一个基本的状态定义，看是否能够「不重不漏」的将状态通过转移计算出来。如果不行，我们再考虑引入更多的维度来进行求解。

先从最朴素的猜想出发，定义 $f[i]$ 为考虑下标不超过 i 的所有数，并且以 $nums[i]$ 为结尾的等差序列的个数。

不失一般性的 $f[i]$ 该如何转移，不难发现我们需要枚举 $[0, i-1]$ 范围内的所有数，假设当前我们枚举到 $[0, i-1]$ 中的位置 j，我们可以直接算出两个位置的差值 $d = nums[i] - nums[j]$，但我们不知道 $f[j]$ 存储的子序列数量是差值为多少的。

同时，根据题目我们要求的是所有的等差序列的个数，而不是求差值为某个具体值 x 的等差序列的个数。换句话说，我们需要记录下所有差值的子序列个数，并求和才是答案。

因此我们的 $f[i]$ 不能是一个数，而应该是一个「集合」，该集合记录下了所有以 $nums[i]$ 为结尾，差值为所有情况的子序列的个数。

我们可以设置 $f[i] = g$，其中 g 为一个「集合」数据结构，我们期望在 $O(1)$ 的复杂度内查的某个差值 d 的子序列个数是多少。

这样 $f[i][j]$ 就代表了以 $nums[i]$ 为结尾，并且差值为 j 的子序列个数是多少。

当我们多引入一维进行这样的状态定义后，我们再分析一下能否「不重不漏」的通过转移计算出所有的动规值。

不失一般性的考虑 $f[i][j]$ 该如何转移，显然序列 DP 问题我们还是要枚举区间 $[0, i-1]$ 的所有数。

和其他的「序列 DP」问题一样，枚举当前位置前面的所有位置的目的，是为了找到当前位置的数，能够接在哪一个位置的后面，形成序列。

对于本题，枚举区间 $[0, i-1]$ 的所有数的含义是：枚举以 $nums[i]$ 为子序列结尾时，它的前

一个值是什么，也就是 $nums[i]$ 接在哪个数的后面，形成等差子序列。

这样必然是可以「不重不漏」的处理到所有以 $nums[i]$ 为子序列结尾的情况的。

至于具体的状态转移方程，我们令差值 $d = nums[i] - nums[j]$，显然有（先不考虑长度至少为 3 的限制）：

$$f[i][d] = \sum_{j=0}^{i-1}(f[j][d] + 1)$$

含义为：**在原本以 $nums[j]$ 为结尾的，且差值为 d 的子序列的基础上接上 $nums[i]$，再加上新的子序列 $(nums[j], nums[i])$，共 $f[j][d] + 1$ 个子序列。**

最后对所有的哈希表的「值」对进行累加计数，就是以任意位置为结尾，长度大于 1 的等差子序列的数量 ans。

这时候再看一眼数据范围 $-2^{31} <= nums[i] <= 2^{31} - 1$，如果从数据范围出发，使用「数组」充当集合的话，我们需要将数组开得很大，必然会爆内存。

但同时有 $1 <= nums.length <= 1000$，也就是说「最小差值」和「最大差值」之间可能相差很大，但是差值的数量是有限的，不会超过 n^2 个。

为了不引入复杂的「离散化」操作，我们可以直接使用「哈希表」来充当「集合」。

每一个 $f[i]$ 为一个哈希表，哈希表的以 `{d:cnt}` 的形式进行存储，`d` 为子序列差值，`cnt` 为子序列数量。

虽然相比使用数组，哈希表常数更大，但是经过上述分析，我们的复杂度为 $O(n^2)$，计算量为 10^6，距离计算量上界 10^7 还保有一段距离，因此直接使用哈希表十分安全。

到这里，我们解决了不考虑「长度为至少为 3」限制的原问题。

那么需要考虑「长度为至少为 3」限制怎么办？

显然，我们计算的 ans 为统计所有的「长度大于 1」的等差子序列数量，由于长度必然为正整数，也就是统计的是「长度大于等于 2」的等差子序列的数量。

因此，如果我们能够求出长度为 2 的子序列的个数的话，从 ans 中减去，得到的就是「长度为至少为 3」子序列的数量。

长度为 2 的等差子序列，由于没有第三个数的差值限制，因此任意的数对 (j, i) 都是一个合法的长度为 2 的等差子序列。

而求长度为 n 的数组的所有数对，其实就是求 **首项为 0，末项为 $n-1$，公差为 1，长度为 n 的等差数列之和**，直接使用「等差数列求和」公式求解即可。

代码：

```
class Solution {
    public int numberOfArithmeticSlices(int[] nums) {
        int n = nums.length;
        // 每个 f[i] 均为哈希表，哈希表键值对为 {d : cnt}
        // d : 子序列差值
        // cnt : 以 nums[i] 为结尾，且差值为 d 的子序列数量
        List<Map<Long, Integer>> f = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            Map<Long, Integer> cur = new HashMap<>();
            for (int j = 0; j < i; j++) {
                Long d = nums[i] * 1L - nums[j];
                Map<Long, Integer> prev = f.get(j);
                int cnt = cur.getOrDefault(d, 0);
                cnt += prev.getOrDefault(d, 0);
                cnt ++;
                cur.put(d, cnt);
            }
            f.add(cur);
        }
        int ans = 0;
        for (int i = 0; i < n; i++) {
            Map<Long, Integer> cur = f.get(i);
            for (Long key : cur.keySet()) ans += cur.get(key);
        }
        int a1 = 0, an = n - 1;
        int cnt = (a1 + an) * n / 2;
        return ans - cnt;
    }
}
```

- 时间复杂度：DP 过程的复杂度为 $O(n^2)$，遍历所有的哈希表的复杂度上界不会超过 $O(n^2)$。整体复杂度为 $O(n^2)$
- 空间复杂度：所有哈希表存储的复杂度上界不会超过 $O(n^2)$

> **🍭 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 **583. 两个字符串的删除操作** ，难度为 **中等**。

Tag :「最长公共子序列」、「序列 DP」

给定两个单词 word1 和 word2，找到使得 word1 和 word2 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

示例：

```
输入: "sea", "eat"

输出: 2

解释: 第一步将"sea"变为"ea"，第二步将"eat"变为"ea"
```

提示：

- 给定单词的长度不超过500。
- 给定单词中的字符只含有小写字母。

转换为 LCS 问题

首先，给定两字符 `s1` 和 `s2`，求经过多少次删除操作，可使得两个相等字符串。

该问题等价于求解两字符的「最长公共子序列」，若两者长度分别为 n 和 m，而最长公共子序列长度为 max，则 $n - max + m - max$ 即为答案。

对「最长公共子序列（LCS）」不熟悉的同学，可以看 (题解) 1143. 最长公共子序列。

$f[i][j]$ 代表考虑 $s1$ 的前 i 个字符、考虑 $s2$ 的前 j 个字符（但最长公共子序列中不一定包含 $s1[i]$ 或者 $s2[j]$）时形成的「最长公共子序列（LCS）」长度。

当有了「状态定义」之后，基本上「转移方程」就是呼之欲出：

- `s1[i]==s2[j]` : $f[i][j] = f[i-1][j-1] + 1$。代表 **必然使用 $s1[i]$ 与 $s2[j]$**

时 LCS 的长度。

- `s1[i]!=s2[j]`：$f[i][j] = max(f[i-1][j], f[i][j-1])$。代表 **必然不使用** $s1[i]$ **（但可能使用**$s2[j]$**）** 时 和 **必然不使用** $s2[j]$ **（但可能使用**$s1[i]$**）** 时 LCS 的长度。

可以发现，上述两种讨论已经包含了「不使用 $s1[i]$ 和 $s2[j]$」、「仅使用 $s1[i]$」、「仅使用 $s2[j]$」和「使用 $s1[i]$ 和 $s2[j]$」四种情况。

虽然「不使用 $s1[i]$ 和 $s2[j]$」会被 $f[i-1][j]$ 和 $f[i][j-1]$ 重复包含，但对于求最值问题，重复比较并不想影响答案正确性。

因此最终的 $f[i][j]$ 为上述两种讨论中的最大值。

一些编码细节：

通常会习惯性往字符串头部追加一个空格，以减少边界判断（使下标从 1 开始，并很容易构造出可滚动的「有效值」）。但实现上，不用真的往字符串中最佳空格，只需在初始化动规值时假定存在首部空格，以及对最后的 LCS 长度进行减一操作即可。

代码：

```
class Solution {
    public int minDistance(String s1, String s2) {
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
        int n = s1.length(), m = s2.length();
        int[][] f = new int[n + 1][m + 1];
        // 假定存在哨兵空格，初始化 f[0][x] 和 f[x][0]
        for (int i = 0; i <= n; i++) f[i][0] = 1;
        for (int j = 0; j <= m; j++) f[0][j] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                f[i][j] = Math.max(f[i - 1][j], f[i][j - 1]);
                if (cs1[i - 1] == cs2[j - 1]) f[i][j] = Math.max(f[i][j], f[i - 1][j - 1]
            }
        }
        int max = f[n][m] - 1; // 减去哨兵空格
        return n - max + m - max;
    }
}
```

- 时间复杂度：$O(n*m)$
- 空间复杂度：$O(n*m)$

序列 DP

上述解决方案是套用了「最长公共子序列（LCS）」进行求解，最后再根据 LCS 长度计算答案。

而更加契合题意的状态定义是根据「最长公共子序列（LCS）」的原始状态定义进行微调：**定义 $f[i][j]$ 代表考虑 $s1$ 的前 i 个字符、考虑 $s2$ 的前 j 个字符（最终字符串不一定包含 $s1[i]$ 或 $s2[j]$）时形成相同字符串的最小删除次数。**

同理，不失一般性的考虑 $f[i][j]$ 该如何计算：

- `s1[i]==s2[j]`：$f[i][j] = f[i-1][j-1]$，代表可以不用必然删掉 $s1[i]$ 和 $s2[j]$ 形成相同字符串；
- `s1[i]!=s2[j]`：$f[i][j] = \min(f[i-1][j]+1, f[i][j-1]+1)$，代表至少一个删除 $s1[i]$ 和 $s2[j]$ 中的其中一个。

$f[i][j]$ 为上述方案中的最小值，最终答案为 $f[n][m]$。

代码：

```
class Solution {
    public int minDistance(String s1, String s2) {
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
        int n = s1.length(), m = s2.length();
        int[][] f = new int[n + 1][m + 1];
        for (int i = 0; i <= n; i++) f[i][0] = i;
        for (int j = 0; j <= m; j++) f[0][j] = j;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                f[i][j] = Math.min(f[i - 1][j] + 1, f[i][j - 1] + 1);
                if (cs1[i - 1] == cs2[j - 1]) f[i][j] = Math.min(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：$O(n*m)$

🍭 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 **673. 最长递增子序列的个数** ，难度为 **中等**。

Tag : 「动态规划」、「序列 DP」、「树状数组」、「最长上升子序列」

给定一个未排序的整数数组，找到最长递增子序列的个数。

示例 1:

```
输入: [1,3,5,4,7]

输出: 2

解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。
```

示例 2:

```
输入: [2,2,2,2,2]

输出: 5

解释: 最长递增子序列的长度是1，并且存在5个子序列的长度为1，因此输出5。
注意: 给定的数组长度不超过 2000 并且结果一定是32位有符号整数。
```

序列 DP

与朴素的 LIS 问题（问长度）相比，本题问的是最长上升子序列的个数。

我们只需要在朴素 LIS 问题的基础上通过「记录额外信息」来进行求解即可。

在朴素的 LIS 问题中，我们定义 $f[i]$ **为考虑以** $nums[i]$ **为结尾的最长上升子序列的长度**。 最终答案为所有 $f[0...(n-1)]$ 中的最大值。

不失一般性地考虑 $f[i]$ 该如何转移：

- 由于每个数都能独自一个成为子序列，因此起始必然有 $f[i] = 1$；
- 枚举区间 $[0, i)$ 的所有数 $nums[j]$，如果满足 $nums[j] < nums[i]$，说明 $nums[i]$ 可以接在 $nums[j]$ 后面形成上升子序列，此时使用 $f[j]$ 更新 $f[i]$，即有 $f[i] = f[j] + 1$。

回到本题，由于我们需要求解的是最长上升子序列的个数，因此需要额外定义 $g[i]$ **为考虑以 $nums[i]$ 结尾的最长上升子序列的个数。**

结合 $f[i]$ 的转移过程，不失一般性地考虑 $g[i]$ 该如何转移：

- 同理，由于每个数都能独自一个成为子序列，因此起始必然有 $g[i] = 1$；
- 枚举区间 $[0, i)$ 的所有数 $nums[j]$，如果满足 $nums[j] < nums[i]$，说明 $nums[i]$ 可以接在 $nums[j]$ 后面形成上升子序列，这时候对 $f[i]$ 和 $f[j] + 1$ 的大小关系进行分情况讨论：
    - 满足 $f[i] < f[j] + 1$：说明 $f[i]$ 会被 $f[j] + 1$ 直接更新，此时同步直接更新 $g[i] = g[j]$ 即可；
    - 满足 $f[i] = f[j] + 1$：说明找到了一个新的符合条件的前驱，此时将值继续累加到方案数当中，即有 $g[i] += g[j]$。

在转移过程，我们可以同时记录全局最长上升子序列的最大长度 max，最终答案为所有满足 $f[i] = max$ 的 $g[i]$ 的累加值。

代码：

```java
class Solution {
    public int findNumberOfLIS(int[] nums) {
        int n = nums.length;
        int[] f = new int[n], g = new int[n];
        int max = 1;
        for (int i = 0; i < n; i++) {
            f[i] = g[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[j] < nums[i]) {
                    if (f[i] < f[j] + 1) {
                        f[i] = f[j] + 1;
                        g[i] = g[j];
                    } else if (f[i] == f[j] + 1) {
                        g[i] += g[j];
                    }
                }
            }
            max = Math.max(max, f[i]);
        }
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (f[i] == max) ans += g[i];
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

LIS 问题的贪心解 + 树状数组

我们知道，对于朴素的 LIS 问题存在贪心解法，能够在 $O(n \log n)$ 复杂度内求解 LIS 问题。

在贪心解中，我们会多开一个贪心数组 q，用来记录长度为 len 的最长上升子序列的「最小结尾元素」为何值：$q[len] = x$ **代表长度为 len 的最长上升子序列的最小结尾元素为 x**。

可以证明 q 存在单调性，因此每次确定 $nums[i]$ 可以接在哪个 $nums[j]$ 后面会形成最长上升子序列时，可以通过「二分」来找到满足 $nums[j] < nums[i]$ 的最大下标来实现。

对于本题，由于我们需要求最长上升子序列的个数，单纯使用一维的贪心数组记录最小结尾元素

并不足以。

考虑对其进行扩展，期望能取到「最大长度」的同时，能够知道这个「最大长度」对应多少个子序列数量，同时期望该操作复杂度为 $O(\log n)$。

我们可以使用「树状数组」维护二元组 (len, cnt) 信息：

1. 因为数据范围较大（$-10^6 <= nums[i] <= 10^6$），但数的个数为 2000，因此第一步先对 $nums$ 进行离散化操作；
2. 在遍历 $nums$ 时，每次从树状数组中查询值严格小于 $nums[i]$ 离散值（利用 $nums[i]$ 离散化后的值仍为正整数，我们可以直接查询小于等于 $nums[i]$ 离散值 -1 的值）的最大长度，及最大长度对应的数量；
3. 对于流程 2 中查得的 (len, cnt)，由于 $nums[i]$ 可以接在其后，因此首先长度加一，同时数量将 cnt 累加到该离散值中。

代码：

```java
class Solution {
    int n;
    int[][] tr = new int[2010][2];
    int lowbit(int x) {
        return x & -x;
    }
    int[] query(int x) {
        int len = 0, cnt = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            if (len == tr[i][0]) {
                cnt += tr[i][1];
            } else if (len < tr[i][0]) {
                len = tr[i][0];
                cnt = tr[i][1];
            }
        }
        return new int[]{len, cnt};
    }
    void add(int x, int[] info) {
        for (int i = x; i <= n; i += lowbit(i)) {
            int len = tr[i][0], cnt = tr[i][1];
            if (len == info[0]) {
                cnt += info[1];
            } else if (len < info[0]) {
                len = info[0];
                cnt = info[1];
            }
            tr[i][0] = len; tr[i][1] = cnt;
        }
    }
    public int findNumberOfLIS(int[] nums) {
        n = nums.length;
        // 离散化
        int[] tmp = nums.clone();
        Arrays.sort(tmp);
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0, idx = 1; i < n; i++) {
            if (!map.containsKey(tmp[i])) map.put(tmp[i], idx++);
        }
        // 树状数组维护 (len, cnt) 信息
        for (int i = 0; i < n; i++) {
            int x = map.get(nums[i]);
            int[] info = query(x - 1);
            int len = info[0], cnt = info[1];
            add(x, new int[]{len + 1, Math.max(cnt, 1)});
        }
```

```java
        int[] ans = query(n);
        return ans[1];
    }
}
```

* 时间复杂度：$O(n \log n)$
* 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 **740. 删除并获得点数** ，难度为 **中等**。

Tag : 「序列 DP」

给你一个整数数组 nums ，你可以对它进行一些操作。

每次操作中，选择任意一个 nums[i] ，删除它并获得 nums[i] 的点数。之后，你必须删除 所有 等于 nums[i] - 1 和 nums[i] + 1 的元素。

开始你拥有 0 个点数。返回你能通过这些操作获得的最大点数。

示例 1：

```
输入：nums = [3,4,2]

输出：6

解释：
删除 4 获得 4 个点数，因此 3 也被删除。
之后，删除 2 获得 2 个点数。总共获得 6 个点数。
```

示例 2：

```
输入：nums = [2,2,3,3,3,4]

输出：9

解释：
删除 3 获得 3 个点数，接着要删除两个 2 和 4 。
之后，再次删除 3 获得 3 个点数，再次删除 3 获得 3 个点数。
总共获得 9 个点数。
```

提示：

- 1 <= nums.length <= 2 * 10^4
- 1 <= nums[i] <= 10^4

动态规划

根据题意，当我们选择 $nums[i]$ 的时候，比 $nums[i]$ 大/小 一个单位的数都不能被选择。

如果我们将数组排好序，从前往后处理，其实只需要考虑“当前数”与“前一个数”的「大小 & 选择」关系即可，这样处理完，显然每个数的「前一位/后一位」都会被考虑到。

这样我们将问题转化为一个「序列 DP」问题（选择某个数，需要考虑前一个数的「大小/选择」状态）。

定义 $f[i][0]$ 代表数值为 i 的数字「不选择」的最大价值；$f[i][1]$ 代表数值为 i 的数字「选择」的最大价值。

为了方便，我们可以先对 $nums$ 中出现的所有数值进行计数，而且由于数据范围只有 10^4，我们可以直接使用数组 $cnts[]$ 进行计数：$cnts[x] = i$ 代表数值 x 出现了 i 次。

然后分别考虑一般性的 $f[i][0]$ 和 $f[i][1]$ 该如何计算：

- **$f[i][0]$：当数值 i 不被选择，那么前一个数「可选/可不选」，在两者中取 max 即可。转移方程为 $f[i][0] = \max(f[i-1][0], f[i-1][1])$**
- **$f[i][1]$：当数值 i 被选，那么前一个数只能「不选」，同时为了总和最大数值 i 要选就全部选完。转移方程为 $f[i][1] = f[i-1][0] + i * cnts[i]$**

代码：

```
class Solution {
    int[] cnts = new int[10009];
    public int deleteAndEarn(int[] nums) {
        int n = nums.length;
        int max = 0;
        for (int x : nums) {
            cnts[x]++;
            max = Math.max(max, x);
        }
        // f[i][0] 代表「不选」数值 i；f[i][1] 代表「选择」数值 i
        int[][] f = new int[max + 1][2];
        for (int i = 1; i <= max; i++) {
            f[i][1] = f[i - 1][0] + i * cnts[i];
            f[i][0] = Math.max(f[i - 1][1], f[i - 1][0]);
        }
        return Math.max(f[max][0], f[max][1]);
    }
}
```

- 时间复杂度：遍历 $nums$ 进行计数和取最大值 max，复杂度为 $O(n)$；共有 $max * 2$ 个状态需要被转移，每个状态转移的复杂度为 $O(1)$。整体复杂度为 $O(n + max)$。
- 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 **978. 最长湍流子数组** ，难度为 **中等**。

Tag : 「序列 DP」

当 A 的子数组 A[i], A[i+1], ..., A[j] 满足下列条件时，我们称其为湍流子数组：

- 若 i <= k < j，当 k 为奇数时， A[k] > A[k+1]，且当 k 为偶数时，A[k] < A[k+1]；
- 或 若 i <= k < j，当 k 为偶数时，A[k] > A[k+1] ，且当 k 为奇数时， A[k] < A[k+1]。

也就是说，如果比较符号在子数组中的每个相邻元素对之间翻转，则该子数组是湍流子数组。

返回 A 的最大湍流子数组的长度。

示例 1：

```
输入：[9,4,2,10,7,8,8,1,9]
输出：5
解释：(A[1] > A[2] < A[3] > A[4] < A[5])
```

示例 2：

```
输入：[4,8,12,16]
输出：2
```

示例 3：

```
输入：[100]
输出：1
```

提示：

- 1 <= A.length <= 40000
- 0 <= A[i] <= 10^9

基本分析思路

本题其实是要我们求最长一段呈 `↗ ↘ ↗ ↘` 或者 `↘ ↗ ↘ ↗` 形状的数组长度。

看一眼数据范围，有 40000，那么枚举起点和终点，然后对划分出来的子数组检查是否为「湍流子数组」的朴素解法就不能过了。

朴素解法的复杂度为 $O(n^3)$ ，直接放弃朴素解法。

复杂度往下优化，其实就 $O(n)$ 的 DP 解法了。

动态规划

至于 DP 如何分析，通过我们会先考虑一维 DP 能否求解，不行再考虑二维 DP …

对于本题，**由于每个位置而言，能否「接着」上一个位置**形成「湍流」，取决于上一位置是由什么形状而来。

举个例子，对于样例 `[3,4,2]`，从 4 -> 2 已经确定是 `↘` 状态，那么对于 2 这个位置能否「接着」4 形成「湍流」，要求 4 必须是由 `↗` 而来。

因此我们还需要记录某一位是如何来的（`↗` 还是 `↘`），需要使二维 DP 来求解 ~

我们定义 f(i,j) 代表以位置 i 为结尾，而结尾状态为 j 的最长湍流子数组长度（0：上升状态 / 1:下降状态）

PS. 这里的状态定义我是猜的，这其实是个技巧。通常我们做 DP 题，都是先猜一个定义，然后看看这个定义是否能分析出状态转移方程帮助我们「不重不漏」的枚举所有的方案。一般我是直接根据答案来猜定义，这里是求最长子数组长度，所以我猜一个 f(i,j) 代表最长湍流子数组长度

那么 `f[i][j]` 该如何求解呢？

我们知道位置 `i` 是如何来是唯一确定的（取决于 `arr[i]` 和 `arr[i - 1]` 的大小关系），而只有三种可能性：

- `arr[i - 1] < arr[i]`：该点是由上升而来，能够「接着」的条件是 `i - 1` 是由下降而来。则有：`f[i][0] = f[i - 1][1] + 1`
- `arr[i - 1] > arr[i]`：改点是由下降而来，能够「接着」的条件是 `i - 1` 是由上升而来。则有：`f[i][1] = f[i - 1][0] + 1`
- `arr[i - 1] = arr[i]`：不考虑，不符合「湍流」的定义

代码：

```java
class Solution {
    public int maxTurbulenceSize(int[] arr) {
        int n = arr.length;
        int ans = 1;
        int[][] f = new int[n][2];
        for (int i = 0; i < 2; i++) f[0][i] = 1;
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < 2; j++) f[i][j] = 1;
            if (arr[i - 1] < arr[i]) f[i][0] = f[i - 1][1] + 1;
            if (arr[i - 1] > arr[i]) f[i][1] = f[i - 1][0] + 1;
            for (int j = 0; j < 2; j++) ans = Math.max(ans, f[i][j]);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

动态规划（空间优化：奇偶滚动）

我们发现对于 `f[i][j]` 状态的更新只依赖于 `f[i - 1][j]` 的状态。

因此我们可以使用「奇偶滚动」方式来将第一维从 `n` 优化到 2 。

修改的方式也十分机械，只需要改为「奇偶滚动」的维度直接修改成 2 ，然后该维度的所有访问方式增加 `%2` 即可：

```java
class Solution {
    public int maxTurbulenceSize(int[] arr) {
        int n = arr.length;
        int ans = 1;
        int[][] f = new int[2][2];
        for (int i = 0; i < 2; i++) f[0][i] = 1;
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < 2; j++) f[i % 2][j] = 1;
            if (arr[i - 1] < arr[i]) f[i % 2][0] = f[(i - 1) % 2][1] + 1;
            if (arr[i - 1] > arr[i]) f[i % 2][1] = f[(i - 1) % 2][0] + 1;
            for (int j = 0; j < 2; j++) ans = Math.max(ans, f[i % 2][j]);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：使用固定 `2 * 2` 的数组空间。复杂度为 $O(1)$

动态规划（空间优化：维度消除）

既然只需要记录上一行状态，能否直接将行的维度消除呢？

答案是可以的，当我们要转移第 `i` 行的时候，`f` 数组装的就已经是 `i - 1` 行的结果。

这也是著名「背包问题」的一维通用优手段。

但相比于「奇偶滚动」的空间优化，这种优化手段只是常数级别的优化（空间复杂度与「奇偶滚动」相同），而且优化通常涉及代码改动。

```java
class Solution {
    public int maxTurbulenceSize(int[] arr) {
        int n = arr.length;
        int ans = 1;
        int[] f = new int[2];
        for (int i = 0; i < 2; i++) f[i] = 1;
        for (int i = 1; i < n; i++) {
            int dp0 = f[0], dp1 = f[1];
            if (arr[i - 1] < arr[i]) {
                f[0] = dp1 + 1;
            } else {
                f[0] = 1;
            }
            if (arr[i - 1] > arr[i]) {
                f[1] = dp0 + 1;
            } else {
                f[1] = 1;
            }
            for (int j = 0; j < 2; j++) ans = Math.max(ans, f[j]);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

其他

如果你对「猜 DP 的状态定义」还没感觉，这里有道类似的题目可以瞧一眼：45. 跳跃游戏 II

> **🍭 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 **1035. 不相交的线** ，难度为 **中等**。

Tag : 「最长公共子序列」、「序列 DP」、「LCS」

在两条独立的水平线上按给定的顺序写下 nums1 和 nums2 中的整数。

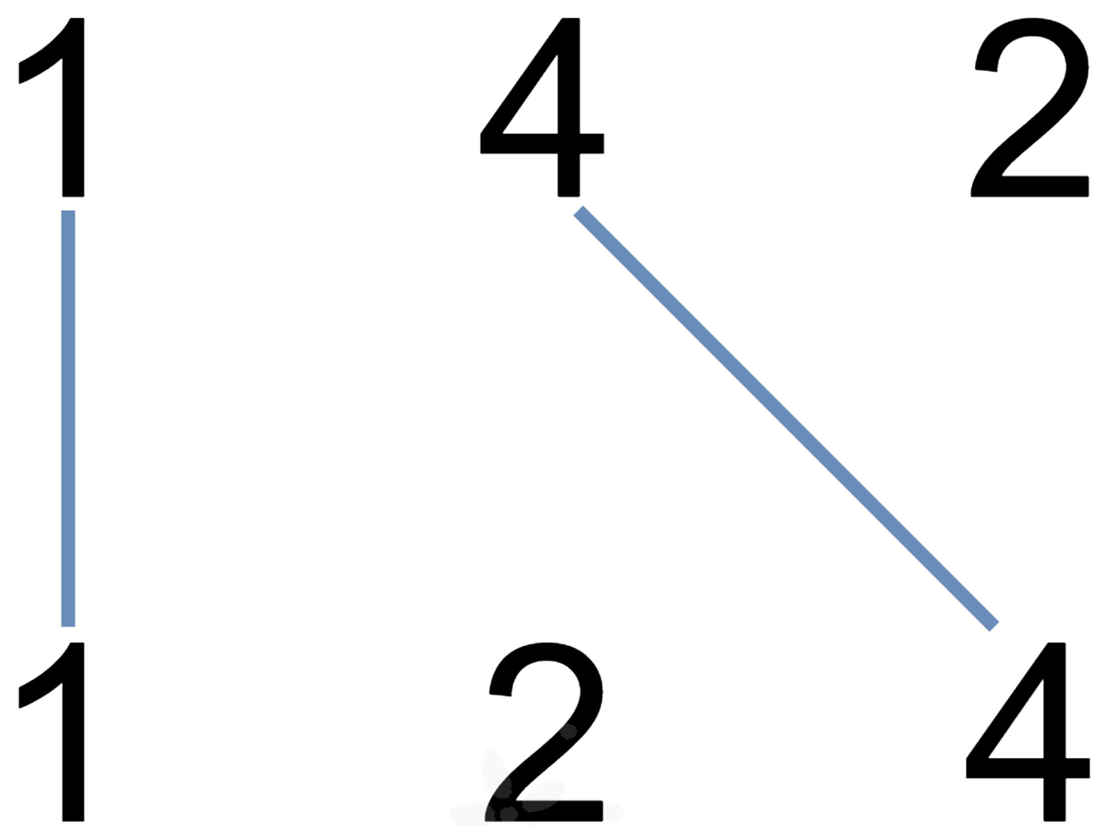
现在，可以绘制一些连接两个数字 nums1[i] 和 nums2[j] 的直线，这些直线需要同时满足满足：

- nums1[i] == nums2[j]
- 且绘制的直线不与任何其他连线（非水平线）相交。

请注意，连线即使在端点也不能相交：每个数字只能属于一条连线。

以这种方法绘制线条，并返回可以绘制的最大连线数。

示例 1：




```
输入：nums1 = [1,4,2], nums2 = [1,2,4]

输出：2

解释：可以画出两条不交叉的线，如上图所示。
但无法画出第三条不相交的直线，因为从 nums1[1]=4 到 nums2[2]=4 的直线将与从 nums1[2]=2 到 nums2[1]=2 的直线相交。
```

示例 2：

```
输入：nums1 = [2,5,1,2,5], nums2 = [10,5,2,1,5,2]

输出：3
```

示例 3：

```
输入：nums1 = [1,3,7,1,7,5], nums2 = [1,9,2,5,1]

输出：2
```

提示：

- 1 <= nums1.length <= 500
- 1 <= nums2.length <= 500
- 1 <= nums1[i], nums2[i] <= 2000

动态规划

这是一道「最长公共子序列（LCS）」的轻度变形题。

为了让你更好的与「最长公共子序列（LCS）」裸题进行对比，我们使用 $s1$ 代指 $nums1$，$s2$ 代指 $nums2$。

对于这类题都使用如下「状态定义」即可：

$f[i][j]$ 代表考虑 $s1$ 的前 i 个字符、考虑 $s2$ 的前 j 的字符，形成的最长公共子序列长度。

然后不失一般性的考虑 $f[i][j]$ 如何转移。

由于我们的「状态定义」只是说「考虑前 i 个和考虑前 j 个字符」，并没有说「一定要包含第 i 个或者第 j 个字符」（这也是「最长公共子序列 LCS」与「最长上升子序列 LIS」状态定义上的最大不同）。

我们需要考虑「不包含 $s1[i]$，不包含 $s2[j]$」、「不包含 $s1[i]$，包含 $s2[j]$」「包含 $s1[i]$，不包含 $s2[j]$」、「包含 $s1[i]$，包含 $s2[j]$」四种情况：

- 不包含 $s1[i]$，不包含 $s2[j]$：结合状态定义，可以使用 $f[i-1][j-1]$ 进行精确表示。
- 包含 $s1[i]$，包含 $s2[j]$：前提是 $s1[i] = s2[j]$，可以使用 $f[i-1][j-1]+1$ 进行精确表示。
- 不包含 $s1[i]$，包含 $s2[j]$：结合状态定义，我们无法直接将该情况表示出来。
  注意 $f[i-1][j]$ 只是表示「必然不包含 $s1[i]$，但可能包含$s2[j]$」的情况，也就是说 $f[i-1][j]$ 其实是该情况与情况 1 的合集。
  但是由于我们求的是「最大值」，只需要确保「不漏」即可保证答案的正确（某些情况被重复参与比较不影响正确性），因此这里直接使用 $f[i-1][j]$ 进行表示没有问题。
- 包含 $s1[i]$，不包含 $s2[j]$：与情况 3 同理，直接使用 $f[i][j-1]$ 表示没有问题。

$f[i][j]$ 就是在上述所有情况中取 max 而来，由于情况 1 被 情况 3 和 情况 4 所包含，因此我们只需要考虑 $f[i-1][j]$、$f[i][j-1]$ 和 $f[i-1][j-1]+1$ 三种状态即可，其中最后一种状态需要满足 $s1[i] = s2[j]$ 前提条件。

因此我们最后的状态转移方程为：

$$f[i][j] = \begin{cases} \max(f[i-1][j], f[i][j-1], f[i-1][j-1]+1) & s1[i] = s2[j] \\ \max(f[i-1][j], f[i][j-1]) & s1[i] \neq s2[j] \end{cases}$$

上述分析过程建议加深理解，估计很多同学能 AC 但其实并不知道 LCS 问题的状态转移是包含了「重复状态比较」的。

代码：

```java
class Solution {
    public int maxUncrossedLines(int[] s1, int[] s2) {
        int n = s1.length, m = s2.length;
        int[][] f = new int[n + 1][m + 1];
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                f[i][j] = Math.max(f[i - 1][j], f[i][j - 1]);
                if (s1[i - 1] == s2[j - 1]) {
                    f[i][j] = Math.max(f[i][j], f[i - 1][j - 1] + 1);
                }
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：$O(n * m)$
- 空间复杂度：$O(n * m)$

> **🍭 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 **1143. 最长公共子序列** ，难度为 **中等**。

Tag : 「最长公共子序列」、「LCS」、「序列 DP」

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

- 例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

示例 1：

```
输入：text1 = "abcde", text2 = "ace"

输出：3

解释：最长公共子序列是 "ace" ，它的长度为 3 。
```

示例 2：

```
输入：text1 = "abc", text2 = "abc"

输出：3

解释：最长公共子序列是 "abc" ，它的长度为 3 。
```

示例 3：

```
输入：text1 = "abc", text2 = "def"

输出：0

解释：两个字符串没有公共子序列，返回 0 。
```

提示：

- 1 <= text1.length, text2.length <= 1000
- text1 和 text2 仅由小写英文字符组成。

动态规划【空格技巧】

这是一道「最长公共子序列（LCS）」的裸题。

对于这类题的都使用如下「状态定义」即可：

$f[i][j]$ 代表考虑 $s1$ 的前 i 个字符、考虑 $s2$ 的前 j 的字符，形成的最长公共子序列长度。

当有了「状态定义」之后，基本上「转移方程」就是呼之欲出：

- `s1[i]==s2[j]`：$f[i][j] = f[i-1][j-1] + 1$。代表**必然使用 $s1[i]$ 与 $s2[j]$ 时** LCS 的长度。

- `s1[i]!=s2[j]` : $f[i][j] = max(f[i-1][j], f[i][j-1])$。代表**必然不使用** $s1[i]$ **（但可能使用**$s2[j]$**）时** 和 **必然不使用** $s2[j]$ **（但可能使用**$s1[i]$**）时** LCS 的长度。

一些编码细节：

通常我会习惯性往字符串头部追加一个空格，以减少边界判断（使下标从 1 开始，并很容易构造出可滚动的「有效值」）。

代码：

```java
class Solution {
    public int longestCommonSubsequence(String s1, String s2) {
        int n = s1.length(), m = s2.length();
        s1 = " " + s1; s2 = " " + s2;
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
        int[][] f = new int[n + 1][m + 1];

        // 因为有了追加的空格，我们有了显然的初始化值（以下两种初始化方式均可）
        // for (int i = 0; i <= n; i++) Arrays.fill(f[i], 1);
        for (int i = 0; i <= n; i++) f[i][0] = 1;
        for (int j = 0; j <= m; j++) f[0][j] = 1;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (cs1[i] == cs2[j]) {
                    f[i][j] = f[i -1][j - 1] + 1;
                } else {
                    f[i][j] = Math.max(f[i - 1][j], f[i][j - 1]);
                }
            }
        }

        // 减去最开始追加的空格
        return f[n][m] - 1;
    }
}
```

- 时间复杂度：$O(n*m)$
- 空间复杂度：$O(n*m)$

动态规划【利用偏移】

上述「追加空格」的做法是我比较习惯的做法 🤣

事实上，我们也可以通过修改「状态定义」来实现递推：

$f[i][j]$ 代表考虑 $s1$ 的前 $i-1$ 个字符、考虑 $s2$ 的前 $j-1$ 的字符，形成的最长公共子序列长度。

那么最终的 $f[n][m]$ 就是我们的答案，$f[0][0]$ 当做无效值，不处理即可。

- `s1[i-1]==s2[j-1]` : $f[i][j] = f[i-1][j-1] + 1$。代表使用 $s1[i-1]$ 与 $s2[j-1]$形成最长公共子序列的长度。
- `s1[i-1]!=s2[j-1]` : $f[i][j] = max(f[i-1][j], f[i][j-1])$。代表不使用 $s1[i-1]$ 形成最长公共子序列的长度、不使用 $s2[j-1]$ 形成最长公共子序列的长度。这两种情况中的最大值。

代码：

```
class Solution {
    public int longestCommonSubsequence(String s1, String s2) {
        int n = s1.length(), m = s2.length();
        char[] cs1 = s1.toCharArray(), cs2 = s2.toCharArray();
        int[][] f = new int[n + 1][m + 1];
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (cs1[i - 1] == cs2[j - 1]) {
                    f[i][j] = f[i - 1][j - 1] + 1;
                } else {
                    f[i][j] = Math.max(f[i - 1][j], f[i][j - 1]);
                }
            }
        }
        return f[n][m];
    }
}
```

- 时间复杂度：$O(n*m)$
- 空间复杂度：$O(n*m)$

> **🍭 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 **1473. 粉刷房子 III** ，难度为 **困难**。

Tag : 「动态规划」、「序列 DP」

在一个小城市里，有 m 个房子排成一排，你需要给每个房子涂上 n 种颜色之一（颜色编号为 1 到 n ）。

有的房子去年夏天已经涂过颜色了，所以这些房子不可以被重新涂色。

我们将连续相同颜色尽可能多的房子称为一个街区。

（比方说 houses = [1,2,2,3,3,2,1,1] ，它包含 5 个街区 [{1}, {2,2}, {3,3}, {2}, {1,1}] 。）

给你一个数组 houses ，一个 m * n 的矩阵 cost 和一个整数 target ，其中：

- `houses[i]`：是第 i 个房子的颜色，0 表示这个房子还没有被涂色。
- `cost[i][j]`：是将第 i 个房子涂成颜色 j+1 的花费。

请你返回房子涂色方案的最小总花费，使得每个房子都被涂色后，恰好组成 target 个街区。

如果没有可用的涂色方案，请返回 -1 。

示例 1：

```
输入：houses = [0,0,0,0,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]], m = 5, n = 2, target = 3

输出：9

解释：房子涂色方案为 [1,2,2,1,1]
此方案包含 target = 3 个街区，分别是 [{1}, {2,2}, {1,1}]。
涂色的总花费为 (1 + 1 + 1 + 1 + 5) = 9。
```

示例 2：

```
输入：houses = [0,2,1,2,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]], m = 5, n = 2, target = 3

输出：11

解释：有的房子已经被涂色了，在此基础上涂色方案为 [2,2,1,2,2]
此方案包含 target = 3 个街区，分别是 [{2,2}, {1}, {2,2}]。
给第一个和最后一个房子涂色的花费为 (10 + 1) = 11。
```

示例 3：

```
输入：houses = [0,0,0,0,0], cost = [[1,10],[10,1],[1,10],[10,1],[1,10]], m = 5, n = 2, target = 5

输出：5
```

示例 4：

```
输入：houses = [3,1,2,3], cost = [[1,1,1],[1,1,1],[1,1,1],[1,1,1]], m = 4, n = 3, target = 3

输出：-1

解释：房子已经被涂色并组成了 4 个街区，分别是 [{3},{1},{2},{3}] ，无法形成 target = 3 个街区。
```

提示：

- m == houses.length == cost.length
- n == cost[i].length
- 1 <= m <= 100
- 1 <= n <= 20
- 1 <= target <= m
- 0 <= houses[i] <= n
- 1 <= cost[i][j] <= 10^4

动态规划

定义 $f[i][j][k]$ 为考虑前 i 间房子，且第 i 间房子的颜色编号为 j，前 i 间房子形成的分区数量为 k 的所有方案中的「最小上色成本」。

我们不失一般性的考虑 $f[i][j][k]$ 该如何转移，由于某些房子本身就已经上色，上色的房子是不

允许被粉刷的。

我们可以根据第 i 间房子是否已经被上色，进行分情况讨论：

- 第 i 间房子已经被上色，即 $houses[i] != 0$，此时只有满足 $j == houses[i]$ 的状态才是有意义的，其余状态均为 `INF`。
  同时根据「第 i 间房子的颜色 j」与「第 $i-1$ 间房子的颜色 p」是否相同，会决定第 i 间房子是否形成一个新的分区。这同样需要进行分情况讨论。
  整理后的转移方程为：

$$f[i][j][k] = \begin{cases} min(f[i-1][j][k], f[i-1][p][k-1]) & j == houses[i], p != j \\ INF & j != houses[i] \end{cases}$$

- 第 i 间房子尚未被上色，即 $houses[i] == 0$，此时房子可以被粉刷成任意颜色。不会有无效状态的情况。
  同样，根据「第 i 间房子的颜色 j」与「第 $i-1$ 间房子的颜色 p」是否相同，会决定第 i 间房子是否形成一个新的分区。
  转移方程为：

$$f[i][j][k] = min(f[i-1][j][k], f[i-1][p][k-1]) + cost[i][j-1], p != j$$

一些编码细节：

- 下标转换：这是个人习惯，无论做什么题，我都喜欢将下标转换为从 1 开始，目的是为了「节省负值下标的分情况讨论」、「将无效状态限制在 0 下标内」或者「充当哨兵」等等。
- 将 `0x3f3f3f3f` 作为 `INF`：因为目标是求最小值，我们应当使用一个较大值充当正无穷，来关联无效状态。同时为了确保不会出现「在正无穷基础上累加导致丢失正无穷含义」的歧义情况，我们可以使用一个有「累加空间」的值作为「正无穷」（这个问题刚好最近在 这里 专门讲过）。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int minCost(int[] hs, int[][] cost, int m, int n, int t) {
        int[][][] f = new int[m + 1][n + 1][t + 1];

        // 不存在分区数量为 0 的状态
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                f[i][j][0] = INF;
            }
        }

        for (int i = 1; i <= m; i++) {
            int color = hs[i - 1];
            for (int j = 1; j <= n; j++) {
                for (int k = 1; k <= t; k++) {
                    // 形成分区数量大于房子数量，状态无效
                    if (k > i) {
                        f[i][j][k] = INF;
                        continue;
                    }

                    // 第 i 间房间已经上色
                    if (color != 0) {
                        if (j == color) { // 只有与「本来的颜色」相同的状态才允许被转移
                            int tmp = INF;
                            // 先从所有「第 i 间房形成新分区」方案中选最优（即与上一房间颜色不同）
                            for (int p = 1; p <= n; p++) {
                                if (p != j) {
                                    tmp = Math.min(tmp, f[i - 1][p][k - 1]);
                                }
                            }
                            // 再结合「第 i 间房不形成新分区」方案中选最优（即与上一房间颜色相同）
                            f[i][j][k] = Math.min(f[i - 1][j][k], tmp);

                        } else { // 其余状态无效
                            f[i][j][k] = INF;
                        }

                    // 第 i 间房间尚未上色
                    } else {
                        int u = cost[i - 1][j - 1];
                        int tmp = INF;
                        // 先从所有「第 i 间房形成新分区」方案中选最优（即与上一房间颜色不同）
                        for (int p = 1; p <= n; p++) {
                            if (p != j) {
```

```java
                        tmp = Math.min(tmp, f[i - 1][p][k - 1]);
                    }
                }
                // 再结合「第 i 间房不形成新分区」方案中选最优（即与上一房间颜色相同）
                // 并将「上色成本」添加进去
                f[i][j][k] = Math.min(tmp, f[i - 1][j][k]) + u;
            }
        }
    }
}

// 从「考虑所有房间，并且形成分区数量为 t」的所有方案中找答案
int ans = INF;
for (int i = 1; i <= n; i++) {
    ans = Math.min(ans, f[m][i][t]);
}
return ans == INF ? -1 : ans;
}
}
```

- 时间复杂度：共有 $m * n * t$ 个状态需要被转移，每次转移需要枚举「所依赖的状态」的颜色，复杂度为 $O(n)$。整体复杂度为 $O(m * n^2 * t)$
- 空间复杂度：$O(m * n * t)$

状态定义的由来

对于有一定 DP 刷题量的同学来说，上述的「状态定义」应该很好理解。

根据经验，我们可以很容易确定「房间编号维度 i」和「分区数量维度 k」需要纳入考虑，同时为了在转移过程中，我们能够清楚知道从哪些状态转移过来需要增加「分区数量」，哪些状态转移过来不需要增加，因此需要多引入「最后一间房间颜色 j」维度。

至于对 DP 不熟悉的同学，可以从写「爆搜」开始入手。

这里的“写”不一定真的要动手去实现一个完整的「爆搜」方案，只需要合理设计出来 `DFS` 函数签名即可。

但为了更好理解，我写了一个完整版的供你参考。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    int ans = INF;
    int[] hs;
    int[][] cost;
    int m, n, t;
    public int minCost(int[] _hs, int[][] _cost, int _m, int _n, int _t) {
        m = _m; n = _n; t = _t;
        hs = _hs;
        cost = _cost;
        dfs(0, -1, 0, 0);
        return ans == INF ? -1 : ans;
    }
    // u : 当前处理到的房间编号
    // last : 当前处理的房间颜色
    // cnt : 当前形成的分区数量
    // sum : 当前的上色成本
    void dfs(int u, int last, int cnt, int sum) {
        if (sum >= ans || cnt > t) return;
        if (u == m) {
            if (cnt == t) {
                ans = Math.min(ans, sum);
            }
            return;
        }
        int color = hs[u];
        if (color == 0) {
            for (int i = 1; i <= n; i++) {
                int nCnt = u - 1 < 0 ? 1 : last == i ? cnt : cnt + 1;
                dfs(u + 1, i, nCnt, sum + cost[u][i - 1]);
            }
        } else {
            int nCnt = u - 1 < 0 ? 1 : last == color ? cnt : cnt + 1;
            dfs(u + 1, color, nCnt, sum);
        }
    }
}
```

- 时间复杂度：`n` 为颜色数量，`m` 为房间数量。不考虑剪枝效果，每个房间都可以粉刷 `n` 种颜色，复杂度为指数级别的 $O(n^m)$
- 空间复杂度：忽略递归带来的额外空间开销。复杂度为 $O(1)$

可以发现，`DFS` 的可变参数有四个，其中 `sum` 是用于更新最终答案 `ans` 的，其应该作为动规值，其余三个参数，作为动规数组的三个维度。

至此，我们可以确定动态规划的「状态定义」，关于如何利用这种「技巧」来得到一个可靠的「状态定义」最早在 这里 讲过。

记忆化搜索

看到评论区有同学贴了「记忆化搜索」的版本，那么就作为补充增加到题解吧 ~

注意记忆化容器应当与我们的「动规数组」结构保存一致。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    int m, n, t;
    int[] hs;
    int[][] cost;
    boolean[][][] vis;
    int[][][] cache;
    public int minCost(int[] _hs, int[][] _cost, int _m, int _n, int _t) {
        m = _m; n = _n; t = _t;
        hs = _hs;
        cost = _cost;
        vis = new boolean[m + 1][n + 1][t + 1];
        cache = new int[m + 1][n + 1][t + 1];
        int ans = dfs(0, 0, 0, 0);
        return ans == INF ? -1 : ans;
    }
    int dfs(int u, int last, int cnt, int sum){
        if(cnt > t) return INF;
        if(vis[u][last][cnt]) return cache[u][last][cnt];
        if (u == m) return cnt == t ? 0 : INF;
        int ans = INF;
        int color = hs[u];
        if(color == 0){
            for(int i = 1; i <= n; i++){
                int nCnt = u == 0 ? 1 : last == i ? cnt : cnt + 1;
                int cur = dfs(u + 1, i, nCnt, sum + cost[u][i - 1]);
                ans = Math.min(ans, cur + cost[u][i - 1]);
            }
        } else{
            int nCnt = u == 0 ? 1 : last == color ? cnt : cnt + 1;
            int cur = dfs(u + 1, color, nCnt, sum);
            ans = Math.min(ans, cur);
        }
        vis[u][last][cnt] = true;
        cache[u][last][cnt] = ans;
        return ans;
    }
}
```

- 时间复杂度：共有 $m * n * t$ 个状态需要被转移，每次转移需要枚举「所依赖的状态」的颜色，复杂度为 $O(n)$。整体复杂度为 $O(m * n^2 * t)$
- 空间复杂度：$O(m * n * t)$

> **🍭 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 **1713. 得到子序列的最少操作次数** ，难度为 **困难**。

Tag : 「最长公共子序列」、「最长上升子序列」、「贪心」、「二分」

给你一个数组 target ，包含若干 互不相同 的整数，以及另一个整数数组 arr ，arr 可能 包含重复元素。

每一次操作中，你可以在 arr 的任意位置插入任一整数。比方说，如果 arr = [1,4,1,2] ，那么你可以在中间添加 3 得到 [1,4,3,1,2] 。你可以在数组最开始或最后面添加整数。

请你返回 最少 操作次数，使得 target 成为 arr 的一个子序列。

一个数组的 子序列 指的是删除原数组的某些元素（可能一个元素都不删除），同时不改变其余元素的相对顺序得到的数组。比方说，[2,7,4] 是 [4,2,3,7,2,1,4] 的子序列（加粗元素），但 [2,4,2] 不是子序列。

示例 1：

```
输入：target = [5,1,3], arr = [9,4,2,3,4]

输出：2

解释：你可以添加 5 和 1 ，使得 arr 变为 [5,9,4,1,2,3,4] ，target 为 arr 的子序列。
```

示例 2：

```
输入：target = [6,4,8,1,3,2], arr = [4,7,6,2,3,8,6,1]

输出：3
```

提示：

- 1 <= target.length, arr.length <= 10^5
- 1 <= target[i], arr[i] <= 10^9
- target 不包含任何重复元素。

基本分析

为了方便，我们令 $target$ 长度为 n，arr 长度为 m，$target$ 和 arr 的最长公共子序列长度为 max，不难发现最终答案为 $n - max$。

因此从题面来说，这是一道最长公共子序列问题（LCS）。

但朴素求解 LCS 问题复杂度为 $O(n * m)$，使用状态定义「**$f[i][j]$ 为考虑 `a` 数组的前 i 个元素和 `b` 数组的前 j 个元素的最长公共子序列长度为多少**」进行求解。

而本题的数据范围为 10^5，使用朴素求解 LCS 的做法必然超时。

一个很显眼的切入点是 $target$ 数组元素各不相同，当 LCS 问题增加某些条件限制之后，会存在一些很有趣的性质。

其中一个经典的性质就是：**当其中一个数组元素各不相同时，最长公共子序列问题（LCS）可以转换为最长上升子序列问题（LIS）进行求解。同时最长上升子序列问题（LIS）存在使用「维护单调序列 + 二分」的贪心解法，复杂度为 $O(n \log n)$。**

因此本题可以通过「抽象成 LCS 问题」->「利用 $target$ 数组元素各不相同，转换为 LIS 问题」->「使用 LIS 的贪心解法」，做到 $O(n \log n)$ 的复杂度。

基本方向确定后，我们证明一下第 2 步和第 3 步的合理性与正确性。

证明

1. 为何其中一个数组元素各不相同，LCS 问题可以转换为 LIS 问题？

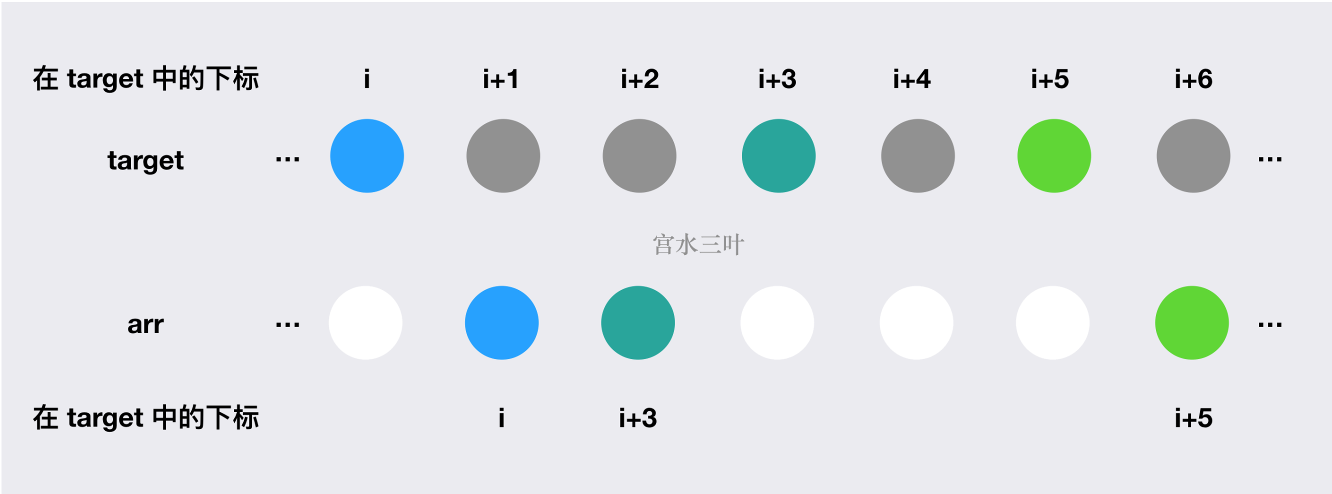
本质是利用「当其中一个数组元素各不相同时，这时候每一个“公共子序列”都对应一个不重复元素数组的下标数组“上升子序列”，反之亦然」。

我们可以使用题目给定的两个数组（$target$ 和 arr）理解上面的话。

由于 $target$ 元素各不相同，那么首先 $target$ 元素和其对应下标，具有唯一的映射关系。

然后我们可以将重点放在两者的公共元素上（忽略非公共元素），**每一个“公共子序列”自然对应了一个下标数组“上升子序列”，反之亦然。**

注意：下图只画出了两个数组的某个片段，不要错误理解为两数组等长。



如果存在某个“公共子序列”，根据“子序列”的定义，那么对应下标序列必然递增，也就是对应了一个“上升子序列”。

反过来，对于下标数组的某个“上升子序列”，首先意味着元素在 $target$ 出现过，并且出现顺序递增，符合“公共子序列”定义，即对应了一个“公共子序列”。

至此，我们将原问题 LCS 转换为了 LIS 问题。

2. 贪心求解 LIS 问题的正确性证明？

朴素的 LIS 问题求解，我们需要定义一个 $f[i]$ 数组代表以 $nums[i]$ 为结尾的最长上升子序列的长度为多少。

对于某个 $f[i]$ 而言，我们需要往回检查 $[0, i-1]$ 区间内，所有可以将 $nums[i]$ 接到后面的位置 j，在所有的 $f[j] + 1$ 中取最大值更新 $f[i]$。因此朴素的 LIS 问题复杂度是 $O(n^2)$ 的。

LIS 的贪心解法则是维护一个额外 g 数组，$g[len] = x$ 代表上升子序列长度为 len 的上升子序列的「最小结尾元素」为 x。

整理一下，我们总共有两个数组：

- f 动规数组：与朴素 LIS 解法的动规数组含义一致。$f[i]$ 代表以 $nums[i]$ 为结尾的上升子序列的最大长度；
- g 贪心数组：$g[len] = x$ 代表上升子序列长度为 len 的上升子序列的「最小结尾元素」为 x。

由于我们计算 $f[i]$ 时，需要找到满足 $nums[j] < nums[i]$，同时取得最大 $f[j]$ 的位置 j。

我们期望通过 g 数组代替线性遍历。

显然，如果 g 数组具有「单调递增」特性的话，我们可以通过「二分」找到符合 $g[idx] < nums[i]$ 分割点 idx（下标最大），即利用 $O(\log n)$ 复杂度找到最佳转移位置。

我们可以很容易 **通过反证法结合 g 数组的定义来证明 g 数组具有「单调递增」特性。**

假设存在某个位置 i 和 j，且 $i < j$，不满足「单调递增」，即如下两种可能：

- $g[i] = g[j] = x$：这意味着某个值 x 既能作为长度 i 的上升子序列的最后一位，也能作为长度为 j 的上升子序列的最后一位。
  根据我们对 g 数组的定义，$g[i] = x$ 意味在所有长度为 i 上升子序列中「最小结尾元素」为 x，但同时由于 $g[j] = x$，而且「上升子序列」必然是「严格单调」，因此我们可以通过删除长度为 j 的子序列后面的元素（调整出一个长度为 i 的子序列）来找到一个比 $g[i]$ 小的合法值。
  也就是我们找到了一个长度为 i 的上升子序列，且最后一位元素必然严格小于 x。
  因此 $g[i] = g[j] = x$ 恒不成立；
- $g[i] > g[j] = x$：同理，如果存在一个长度为 j 的合法上升子序列的「最小结尾元素」为 x 的话，那么必然能够找到一个比 x 小的值来更新 $g[i]$。即 $g[i] > g[j]$ 恒不成立。

根据全序关系，在证明 $g[i] = g[j]$ 和 $g[i] > g[j]$ 恒不成立后，可得 $g[i] < g[j]$ 恒成立。

至此，我们证明了 g 数组具有单调性，从而证明了每一个 $f[i]$ 均与朴素 LIS 解法得到的值相同，即贪心解是正确的。

动态规划 + 贪心 + 二分

根据「基本分析 & 证明」，通过维护一个贪心数组 g，来更新动规数组 f，在求得「最长上升子序列」长度之后，利用「“公共子序列”和“上升子序列”」的一一对应关系，可以得出“最长公共子序列”长度，从而求解出答案。

代码：

```java
class Solution {
    public int minOperations(int[] t, int[] arr) {
        int n = t.length, m = arr.length;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            map.put(t[i], i);
        }
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < m; i++) {
            int x = arr[i];
            if (map.containsKey(x)) list.add(map.get(x));
        }
        int len = list.size();
        int[] f = new int[len], g = new int[len + 1];
        Arrays.fill(g, Integer.MAX_VALUE);
        int max = 0;
        for (int i = 0; i < len; i++) {
            int l = 0, r = len;
            while (l < r) {
                int mid = l + r + 1 >> 1;
                if (g[mid] < list.get(i)) l = mid;
                else r = mid - 1;
            }
            int clen = r + 1;
            f[i] = clen;
            g[clen] = Math.min(g[clen], list.get(i));
            max = Math.max(max, clen);
        }
        return n - max;
    }
}
```

- 时间复杂度：通过 $O(n)$ 复杂度得到 $target$ 的下标映射关系；通过 $O(m)$ 复杂度得到映射数组 $list$；贪心求解 LIS 的复杂度为 $O(m\log m)$。整体复杂度为 $O(n+m\log m)$
- 空间复杂度：$O(n+m)$

> **🍭 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「序列 DP」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。