宫水三叶的刷题日记

贪心算法

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「贪心算法」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「贪心算法」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「贪心算法」；
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 **11. 盛最多水的容器** ，难度为 **中等**。

Tag：「双指针」、「贪心」

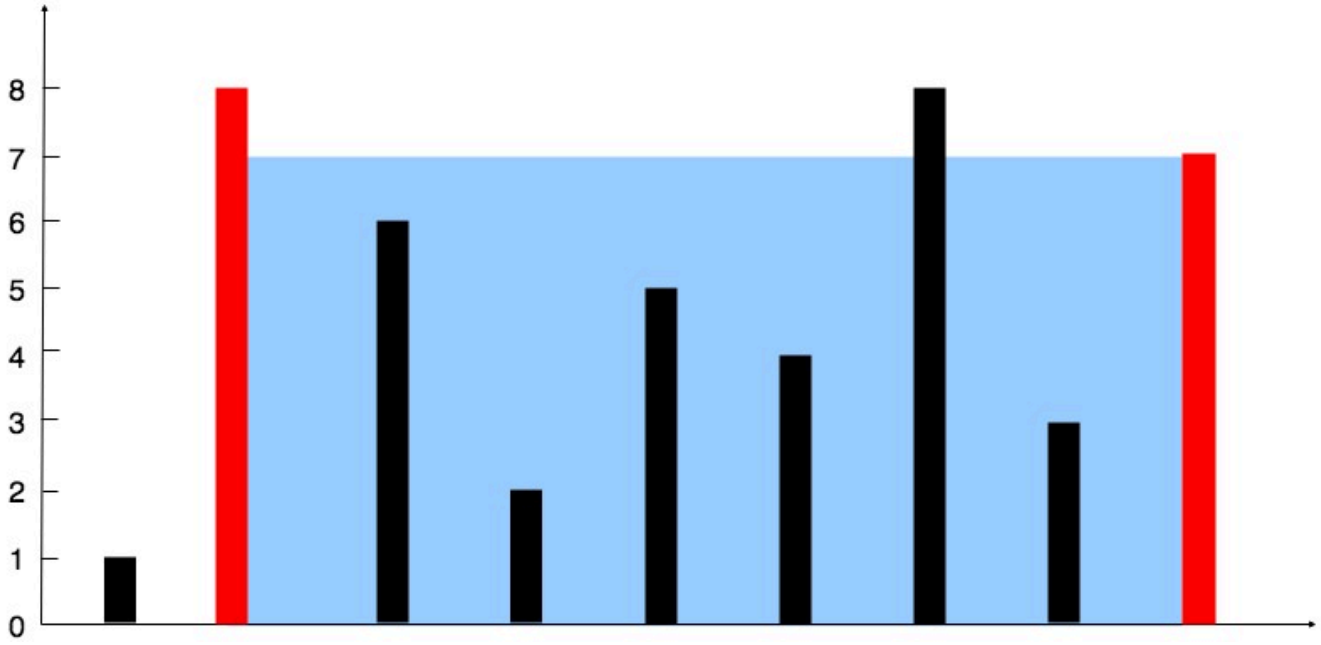
给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。

在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0) 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器。

示例 1：



输入：[1,8,6,2,5,4,8,3,7]
输出：49
解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例 2：

输入：height = [1,1]
输出：1

示例 3：

输入：height = [4,3,2,1,4]
输出：16

示例 4：

输入：height = [1,2,1]
输出：2

提示：

- n = height.length
- 2 <= n <= 3 * 10^4
- 0 <= height[i] <= 3 * 10^4

朴素解法

我们可以直接枚举所有的情况：先枚举确定左边界，再往右枚举确定右边界。

然后再记录枚举过程中的最大面积即可：

代码：

```
class Solution {
    public int maxArea(int[] height) {
        int n = height.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int w = j - i;
                int h = Math.min(height[i], height[j]);
                ans = Math.max(w * h, ans);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(1)$

双指针 + 贪心

先用两个指针 `i` 和 `j` 指向左右边界，然后考虑指针应该怎么移动。

由于构成矩形的面积，取决于 `i` 和 `j` 之间的距离（记为 `w`） 和 `i` 和 `j` 下标对应的高度的最小值（记为 `h`）。

首先无论是 `i` 指针往右移动还是 `j` 指针往左移动都会导致 `w` 变小，所以想要能够枚举到更大的面积，我们应该让 `h` 在指针移动后变大。

不妨假设当前情况是 `height[i] < heigth[j]`（此时矩形的高度为 `height[i]`），然后分情况讨论：

- 让 `i` 和 `j` 两者高度小的指针移动，即 `i` 往右移动：
    - 移动后，i 指针对应的高度变小，即 `height[i] > height[i + 1]`：`w` 和 `h` 都变小了，面积一定变小
    - 移动后，i 指针对应的高度不变，即 `height[i] = height[i + 1]`：`w` 变小，`h` 不变，面积一定变小
    - 移动后，i 指针对应的高度变大，即 `height[i] < height[i + 1]`：`w` 变小，`h` 变大，面积可能会变大
- 让 `i` 和 `j` 两者高度大的指针移动，即 `j` 往左移动：
    - 移动后，j 指针对应的高度变小，即 `height[j] > height[j - 1]`：`w` 变小，`h` 可能不变或者变小（当 `height[j - 1] >= height[i]` 时，`h` 不变；当 `height[j - 1] < height[i]` 时，`h` 变小），面积一定变小
    - 移动后，j 指针对应的高度不变，即 `height[j] = height[j - 1]`：`w` 变小，`h` 不变，面积一定变小
    - 移动后，j 指针对应的高度变大，即 `height[j] < height[j - 1]`：`w` 变小，`h` 不变，面积一定变小

综上所述，我们只有将高度小的指针往内移动，才会枚举到更大的面积：

代码：

```
class Solution {
    public int maxArea(int[] height) {
        int n = height.length;
        int i = 0, j = n - 1;
        int ans = 0;
        while (i < j) {
            ans = Math.max(ans, (j - i) * Math.min(height[i], height[j]));
            if (height[i] < height[j]) {
                i++;
            } else {
                j--;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：会对整个数组扫描一遍。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 **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：

```
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 **179. 最大数** ，难度为 **中等**。

Tag：「贪心」

给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整

数。

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

示例 1：

```
输入：nums = [10,2]

输出："210"
```

示例 2：

```
输入：nums = [3,30,34,5,9]

输出："9534330"
```

示例 3：

```
输入：nums = [1]

输出："1"
```

示例 4：

```
输入：nums = [10]

输出："10"
```

提示：

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 10^9

贪心解法

对于 $nums$ 中的任意两个值 a 和 b，我们无法直接从常规角度上确定其大小/先后关系。

但我们可以根据「结果」来决定 a 和 b 的排序关系：

如果拼接结果 ab 要比 ba 好，那么我们会认为 a 应该放在 b 前面。

另外，注意我们需要处理前导零（最多保留一位）。

代码：

```
class Solution {
    public String largestNumber(int[] nums) {
        int n = nums.length;
        String[] ss = new String[n];
        for (int i = 0; i < n; i++) ss[i] = "" + nums[i];
        Arrays.sort(ss, (a, b) -> {
            String sa = a + b, sb = b + a ;
            return sb.compareTo(sa);
        });

        StringBuilder sb = new StringBuilder();
        for (String s : ss) sb.append(s);
        int len = sb.length();
        int k = 0;
        while (k < len - 1 && sb.charAt(k) == '0') k++;
        return sb.substring(k);
    }
}
```

- 时间复杂度：由于是对 $String$ 进行排序，当排序对象不是 $Java$ 中的基本数据类型时，不会使用快排（考虑排序稳定性问题）。`Arrays.sort()` 的底层实现会「元素数量/元素是否大致有序」决定是使用插入排序还是归并排序。这里直接假定使用的是「插入排序」。复杂度为 $O(n^2)$
- 空间复杂度：$O(n)$

证明

上述解法，我们需要证明两个内容：

- 该贪心策略能取到全局最优解。
- 这样的「排序比较逻辑」应用在集合 $nums$ 上具有「全序关系」。

1. 该贪心策略能取到全局最优解

令我们经过这样的贪心操作得到的贪心解为 ans，真实最优解为 max。

由于真实最优解为全局最大值，而我们的贪心解至少是一个合法解（一个数），因此天然有 $ans \leqslant max$。

接下来我们只需要证明 $ans \geqslant max$，即可得 $ans = max$（贪心解即为最优解）。

我们使用「反证法」来证明 $ans \geqslant max$ 成立：

假设 $ans \geqslant max$ 不成立，即有 $ans < max$。

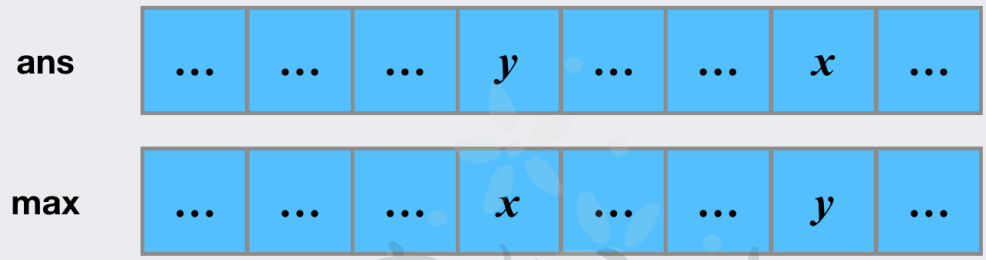
ans 和 max 都是由同样一批数字凑成的，如果有 $ans < max$。

这意味着我们可以将 ans 中的某些低位数字和高位数字互换，使得 ans 更大（调整为 max），这与我们根据「结果」进行排序的逻辑冲突。

因此 $ans < max$ 必然不成立，得证 $ans \geqslant max$ 成立，结合 $ans \leqslant max$ 可得贪心解为最优。

举个🌰，如果有 $ans < max$，那么意味着在 ans 中至少有一对数字互换可以使得 ans 变大，

那么在排序逻辑中 x 所在的整体（可能不只有 x 一个数）应该被排在 y 所在的整体（可能不只有 y 一个数）前面。



2. 全序关系

我们使用符号 $@$ 来代指我们的「排序」逻辑：

- 如果 a 必须排在 b 的前面，我们记作 $a@b$；
- 如果 a 必须排在 b 的后面，我们记作 $b@a$；
- 如果 a 既可以排在 b 的前面，也可以排在 b 的后面，我们记作 $a\#b$；

2.1 完全性

具有完全性是指从集合 $nums$ 中任意取出两个元素 a 和 b，必然满足 $a@b$、$b@a$ 和 $a\#b$ 三者之一。

这点其实不需要额外证明，因为由 a 和 b 拼接的字符串 ab 和 ba 所在「字典序大小关系中」要么完全相等，要么具有明确的字典序大小关系，导致 a 必须排在前面或者后面。

2.2 反对称性

具有反对称性是指由 $a@b$ 和 $b@a$ 能够推导出 $a\#b$。

$a@b$ 说明字符串 ab 的字典序大小数值要比字符串 ba 字典序大小数值大。

$b@a$ 说明字符串 ab 的字典序大小数值要比字符串 ba 字典序大小数值小。

这样，基于「字典序本身满足全序关系」和「数学上的 $a \geqslant b$ 和 $a \leqslant b$ 可推导出 $a = b$」。

得证 $a@b$ 和 $b@a$ 能够推导出 $a\#b$。

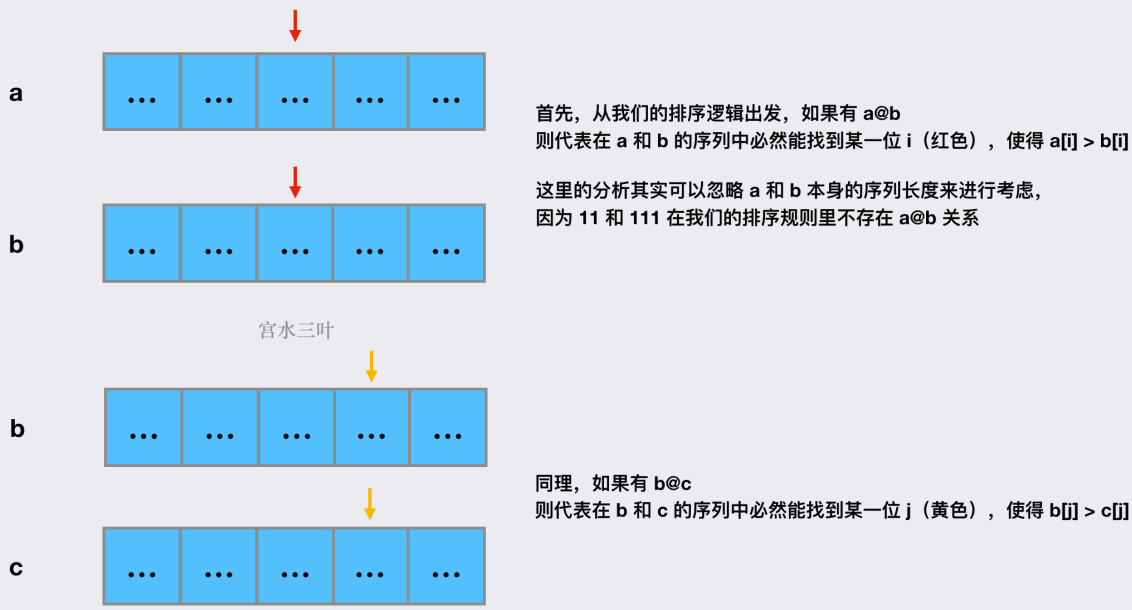
2.3 传递性

具有传递性是指由 $a@b$ 和 $b@c$ 能够推导出 $a@c$。

这里的「传递性」其实也可以使用与 官方题解 类似的手法来证明。

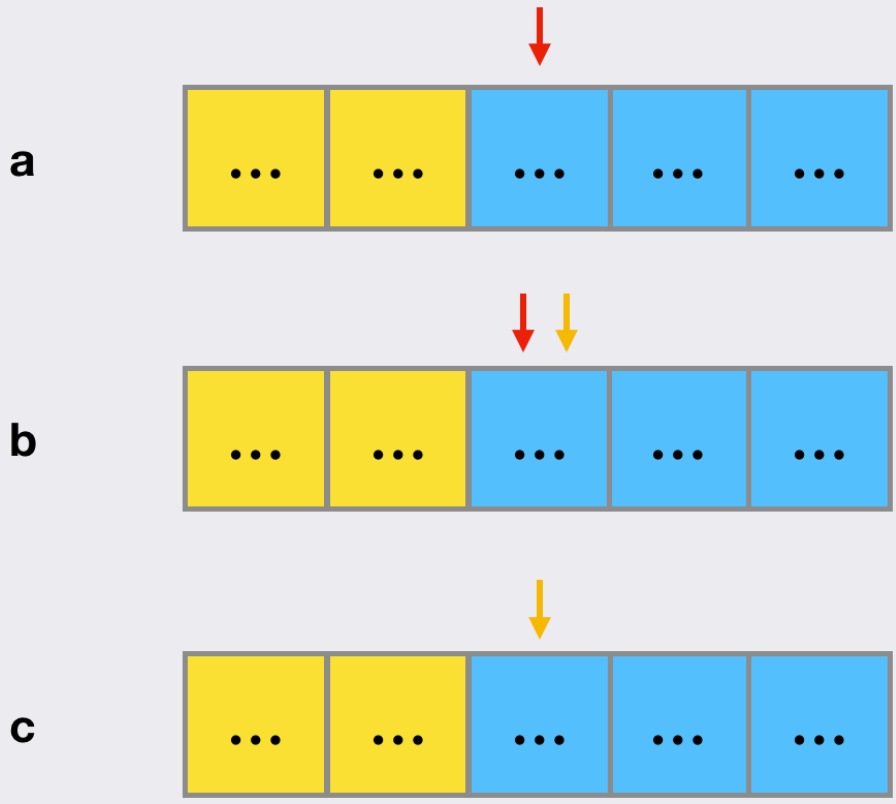
我们可以利用「两个等长的拼接字符串，字典序大小关系与数值大小关系一致」这一性质来证明，因为字符串 ac 和 ca 必然是等长的。

接下来，让我们从「自定义排序逻辑」出发，换个思路来证明 $a@c$：



然后我们只需要证明在不同的 i j 关系之间（共三种情况），$a@c$ 恒成立即可：

1. 当 $i == j$ 的时候：



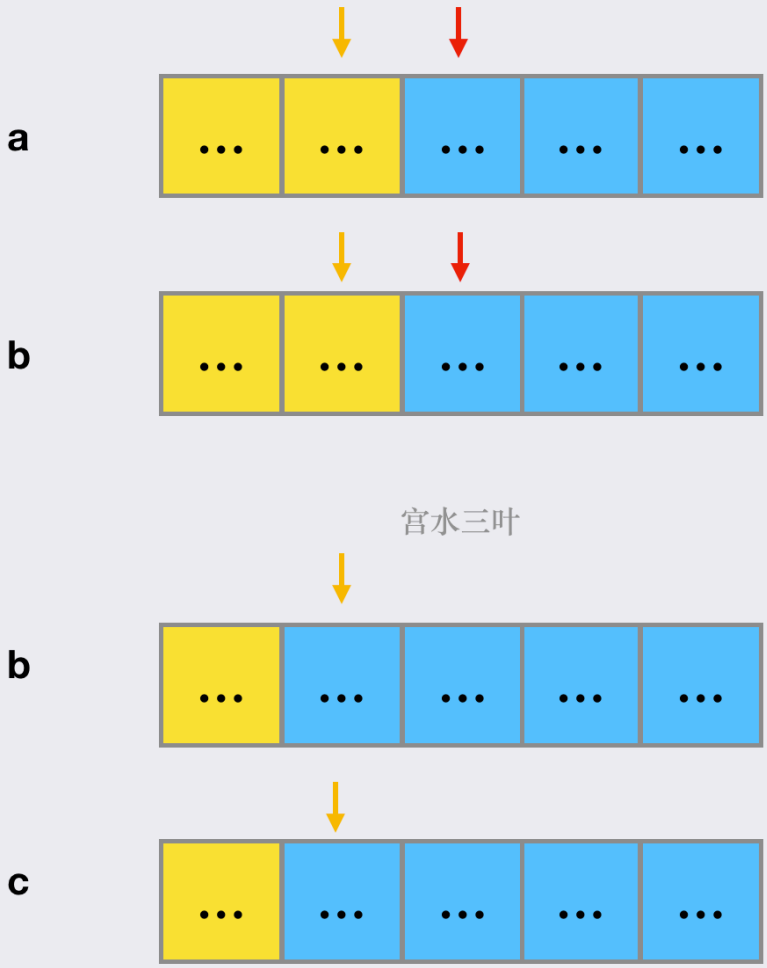
宫水三叶

情况一：

如果 i == j：此时，a b c 三者在 i/j 指针前面的部分完全相同。

通过 a[i] > b[i] = b[j] > c[j] 可得 a[i] = a[j] > c[i] = c[j]，从而得证 a@c 成立。

2. 当 $i > j$ 的时候：

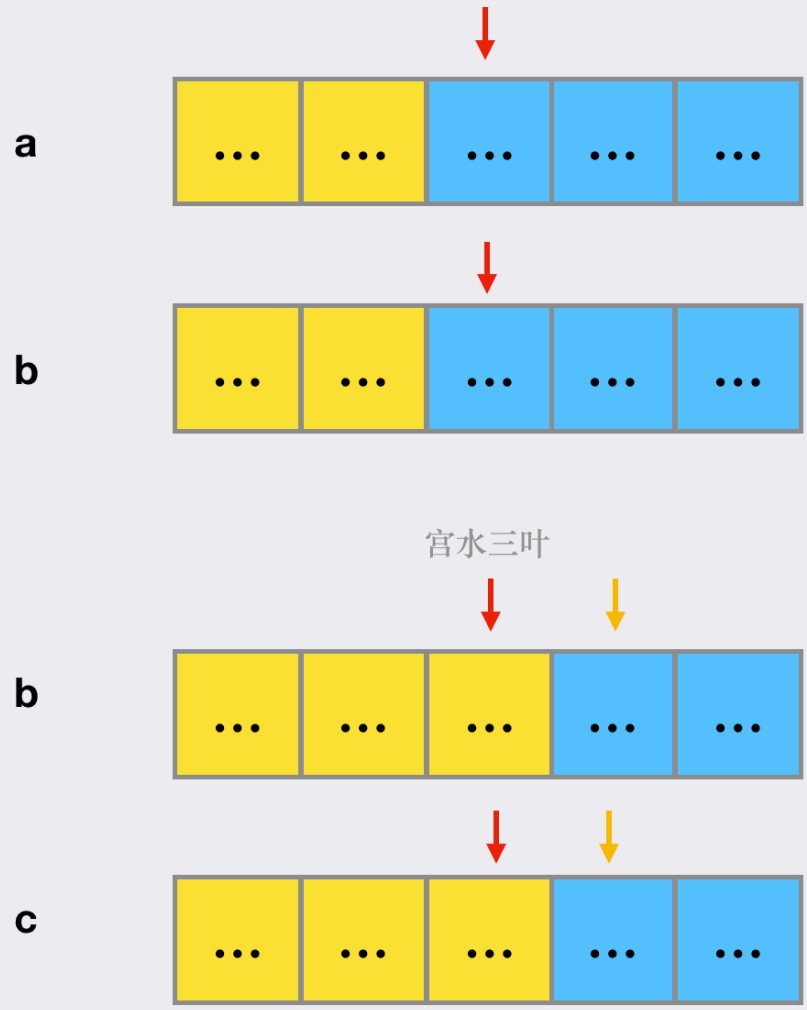


情况二：

如果 i > j ：此时，a b 在 i 指针前面的部分完全相同，而 b c 在 j 指针前面的部分完全相同。

那么通过 a[j] = b[j] 和 b[j] > c[j] 可得 a[j] > c[j]，从而得证 a@c 成立。

3. 当 $i < j$ 的时候：



情况三：

如果 i < j ：此时，a b 在 i 指针前面的部分完全相同，而 b c 在 j 指针前面的部分完全相同。

那么通过 a[i] > b[i] 和 b[i] = c[i] 可得 a[i] > c[i]，得证 a@c 成立。

综上，我们证明了无论在何种情况下，只要有 $a@b$ 和 $b@c$ 的话，那么 $a@c$ 恒成立。

我们之所以能这样证明「传递性」，本质是利用了自定义排序逻辑中「对于确定任意元素 a 和 b 之间的排序关系只依赖于 a 和 b 的第一个不同元素之间的大小关系」这一性质。

找 i 的越界问题

考虑

(1) a = 304 b = 30
(2) a = 301 b = 30

两种情况。

显然，（1）下我们会得到 $a@b$，而（2）下我们会得到 $b@a$

但是，在这种情况下 i 实际上位于 b 界外，那我们还能不能找 i 呢？$b[i]$ 是多少呢？

实际上是可以的，我们在比较 a 和 b 的时候，实际上是在比较 ab 和 ba 两个字符串，所以实际上我们是在用 $a[0], a[1], a[2]$ ... 去填补 b 本体结束后的空缺。换而言之（1）和（2）里的 b 实际上被填补为 `303` （填进来 $a[0]$）

再比如

（3）a = 3131248 b = 3131 ，比较的时候实际上是用 a 开头的 `4` 位去填补上 b 的空缺，所以 b 实际上相当于 `31313131`

题目描述

这是 LeetCode 上的 **502. IPO** ，难度为 **困难**。

Tag：「贪心」、「优先队列」、「堆」

假设 力扣（LeetCode）即将开始 IPO 。

为了以更高的价格将股票卖给风险投资公司，力扣 希望在 IPO 之前开展一些项目以增加其资本。

由于资源有限，它只能在 IPO 之前完成最多 k 个不同的项目。

帮助 力扣 设计完成最多 k 个不同项目后得到最大总资本的方式。

给你 n 个项目。对于每个项目 i ，它都有一个纯利润 profits[i] ，和启动该项目需要的最小资本 capital[i] 。

最初，你的资本为 w 。当你完成一个项目时，你将获得纯利润，且利润将被添加到你的总资本中。

总而言之，从给定项目中选择 最多 k 个不同项目的列表，以 最大化最终资本 ，并输出最终可获得的最多资本。

答案保证在 32 位有符号整数范围内。

示例 1：

```
输入：k = 2, w = 0, profits = [1,2,3], capital = [0,1,1]

输出：4

解释：
由于你的初始资本为 0，你仅可以从 0 号项目开始。
在完成后，你将获得 1 的利润，你的总资本将变为 1。
此时你可以选择开始 1 号或 2 号项目。
由于你最多可以选择两个项目，所以你需要完成 2 号项目以获得最大的资本。
因此，输出最后最大化的资本，为 0 + 1 + 3 = 4。
```

示例 2：

```
输入：k = 3, w = 0, profits = [1,2,3], capital = [0,1,2]

输出：6
```

提示：

- 1 <= k <= 10^5
- 0 <= w <= 10^9
- n == profits.length
- n == capital.length
- 1 <= n <= 10^5
- 0 <= profits[i] <= 10^4
- 0 <= capital[i] <= 10^9

贪心 + 优先队列（堆）

由于每完成一个任务都会使得总资金 `w` 增加或不变。因此对于所选的第 i 个任务而言，应该在所有「未被选择」且启动资金不超过 `w` 的所有任务里面选利润最大的。

可通过「归纳法」证明每次都在所有候选中选择利润最大的任务，可使得总资金最大。

对于第 i 次选择而言（当前所有的资金为 w），如果选择的任务利润为 cur，而实际可选的最大任务利润为 max（ $cur <= max$ ）。

将「选择 cur」调整为「选择 max」，结果不会变差：

1. 根据传递性，由 $cur <= max$ 可得 $w + cur <= w + max$，可推导出调整后的总资金不会变少；
2. 利用推论 1，由于总资金相比调整前没有变少，因此后面可选择的任务集合也不会变少。这意味着 **至少可以维持** 第 i 次选择之后的所有原有选择。

至此，我们证明了将每次的选择调整为选择最大利润的任务，结果不会变差。

当知道了「每次都应该在所有可选择的任务里选利润最大」的推论之后，再看看算法的具体流程。

由于每完成一个任务总资金都会 增大/不变，因此所能覆盖的任务集合数量也随之 增加/不变 。

因此算法核心为「**每次决策前，将启动资金不超过当前总资金的任务加入集合，再在里面取利润最大的任务**」。

「取最大」的过程可以使用优先队列（根据利润排序的大根堆），而「将启动资金不超过当前总资金的任务加入集合」的操作，可以利用总资金在整个处理过程递增，而先对所有任务进行预处理排序来实现。

具体的，我们可以按照如下流程求解：

1. 根据 `profits` 和 `capital` 预处理出总的任务集合二元组，并根据「启动资金」进行升序排序；
2. 每次决策前，将所有的启动资金不超过 w 的任务加入优先队列（根据利润排序的大根堆），然后从优先队列（根据利润排序的大根堆），将利润累加到 w；
3. 循环步骤 2，直到达到 k 个任务，或者队列为空（当前资金不足以选任何任务）。

代码：

```Java
class Solution {
    public int findMaximizedCapital(int k, int w, int[] profits, int[] capital) {
        int n = profits.length;
        List<int[]> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(new int[]{capital[i], profits[i]});
        }
        Collections.sort(list, (a,b)->a[0]-b[0]);
        PriorityQueue<Integer> q = new PriorityQueue<>((a,b)->b-a);
        int i = 0;
        while (k-- > 0) {
            while (i < n && list.get(i)[0] <= w) q.add(list.get(i++)[1]);
            if (q.isEmpty()) break;
            w += q.poll();
        }
        return w;
    }
}
```

- 时间复杂度：构造出二元组数组并排序的复杂度为 $O(n \log n)$；大根堆最多有 n 个元素，使用大根堆计算答案的复杂度为 $O(k \log n)$。整体复杂度为 $O(\max(n \log n, k \log n))$
- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **524. 通过删除字母匹配到字典里最长单词** ，难度为 **中等**。

Tag：「双指针」、「贪心」、「排序」

给你一个字符串 s 和一个字符串数组 dictionary 作为字典，找出并返回字典中最长的字符串，该字符串可以通过删除 s 中的某些字符得到。

如果答案不止一个，返回长度最长且字典序最小的字符串。如果答案不存在，则返回空字符串。

示例 1：

```
输入：s = "abpcplea", dictionary = ["ale","apple","monkey","plea"]

输出："apple"
```

示例 2：

```
输入：s = "abpcplea", dictionary = ["a","b","c"]

输出："a"
```

提示：

- 1 <= s.length <= 1000
- 1 <= dictionary.length <= 1000
- 1 <= dictionary[i].length <= 1000
- s 和 dictionary[i] 仅由小写英文字母组成

排序 + 双指针 + 贪心

根据题意，我们需要找到 `dictionary` 中为 `s` 的子序列，且「长度最长（优先级 1）」及「字典序最小（优先级 2）」的字符串。

数据范围全是 1000。

我们可以先对 `dictionary` 根据题意进行自定义排序：

1. 长度不同的字符串，按照字符串长度排倒序；
2. 长度相同的，则按照字典序排升序。

然后我们只需要对 `dictionary` 进行顺序查找，找到的第一个符合条件的字符串即是答案。

具体的，我们可以使用「贪心」思想的「双指针」实现来进行检查：

1. 使用两个指针 `i` 和 `j` 分别代表检查到 `s` 和 `dictionary[x]` 中的哪位字符；
2. 当 `s[i] != dictionary[x][j]`，我们使 `i` 指针右移，直到找到 `s` 中第一位与 `dictionary[x][j]` 对得上的位置，然后当 `i` 和 `j` 同时右移，匹配下一个字符；
3. 重复步骤 2，直到整个 `dictionary[x]` 被匹配完。

> 证明：对于某个字符 `dictionary[x][j]` 而言，选择 `s` 中 **当前** 所能选择的下标最小的位置进行匹配，对于后续所能进行选择方案，会严格覆盖不是选择下标最小的位置，因此结果不会变差。

代码：

```
class Solution {
    public String findLongestWord(String s, List<String> list) {
        Collections.sort(list, (a,b)->{
            if (a.length() != b.length()) return b.length() - a.length();
            return a.compareTo(b);
        });
        int n = s.length();
        for (String ss : list) {
            int m = ss.length();
            int i = 0, j = 0;
            while (i < n && j < m) {
                if (s.charAt(i) == ss.charAt(j)) j++;
                i++;
            }
            if (j == m) return ss;
        }
        return "";
    }
}
```

- 时间复杂度：令 `n` 为 `s` 的长度，`m` 为 `dictionary` 的长度。排序复杂度为 $O(m \log m)$；对 `dictionary` 中的每个字符串进行检查，单个字符串的检查复杂度为 $O(\min(n, dictionary[i])) \approx O(n)$。整体复杂度为 $O(m \log m + m * n)$
- 空间复杂度：$O(\log m)$

题目描述

这是 LeetCode 上的 **561. 数组拆分 I** ，难度为 **简单**。

Tag：「贪心算法」

给定长度为 2n 的整数数组 nums ，你的任务是将这些数分成 n 对, 例如 (a1, b1), (a2, b2), …, (an, bn) ，使得从 1 到 n 的 min(ai, bi) 总和最大。

返回该 最大总和 。

示例 1：

```
输入：nums = [1,4,3,2]
输出：4
解释：所有可能的分法（忽略元素顺序）为：
1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3
2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3
3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4
所以最大总和为 4
```

示例 2：

```
输入：nums = [6,2,6,5,1,2]
输出：9
解释：最优的分法为 (2, 1), (2, 5), (6, 6). min(2, 1) + min(2, 5) + min(6, 6) = 1 + 2 + 6 = 9
```

提示：

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

贪心解法

我们先对数组进行排序。

由于每两个数，我们只能选择当前小的一个进行累加。

因此我们猜想应该从第一个位置进行选择，然后隔一步选择下一个数。这样形成的序列的求和值最大。

```java
class Solution {
    public int arrayPairSum(int[] nums) {
        int n = nums.length;
        Arrays.sort(nums);
        int ans = 0;
        for (int i = 0; i < n; i += 2) ans += nums[i];
        return ans;
    }
}
```

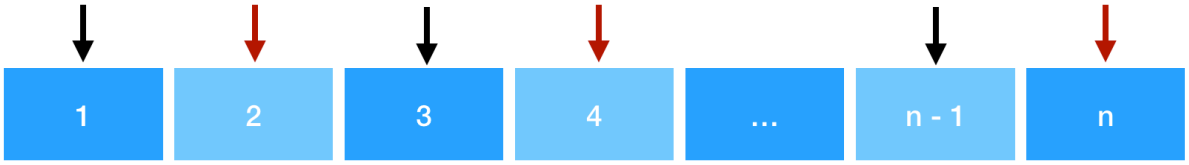
- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(\log n)$

证明

我们用反证法来证明下，为什么这样选择的序列的求和值一定是最大的：

猜想：对数组进行排序，从第一个位置进行选择，然后隔一步选择下一个数。这样形成的序列的求和值最大（下图黑标，代表当前被选择的数字）。

之所以我们能这么选择，是因为每一个被选择的数的「下一位位置」都对应着一个「大于等于」当前数的值（假设位置为 `k` ），使得当前数在 `min(a,b)` 关系中能被选择（下图红标，代表保证前面一个黑标能够被选择的辅助数）。



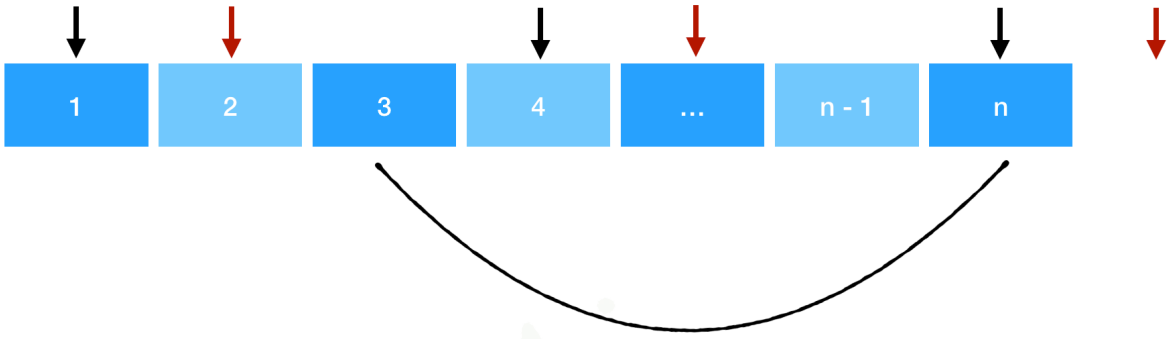
假如我们这样选择的序列求和不是最大值，那么说明至少我们有一个值选错了，应该选择更大的数才对。

那么意味着我们「某一位置」的黑标应该从当前位置指向更后的位置。

PS. 因为要满足 min(a, b) 的数才会被累加，因此每一个红标右移（变大）必然导致原本所对应的黑标发生「同样程度 或 不同程度」的右移（变大）

这会导致我们所有的红标黑标同时往后平移。

最终会导致我们最后一个黑标出现在最后一位，这时候最后一位黑标不得不与我们第 `k` 个位置的数形成一对。



我们看看这是求和序列的变化（ `k` 位置前的求和项没有发生变化，我们从 `k` 位置开始分析）：

1. 原答案 = `nums[k] + nums[k + 2] + ... + nums[n - 1]`
2. 调整后答案 = `nums[k + 1] + nums[k + 3] + ... + nums[n - 2] + min(nums[n], nums[k])`
   由于 `min(nums[n], nums[k])` 中必然是 `nums[k]` 被选择。因此：

调整后答案 = `nums[k] + nums[k + 1] + nums[k + 3] + ... + nums[n - 2]`

显然从原答案的每一项都「大于等于」调整后答案的每一项，因此**不可能在「假想序列」中通过选择别的更大的数得到更优解，假想得证。**

为什么要「证明」或「理解证明」？

证明的意义在于，**你知道为什么这样做是对的**。

带来的好处是：

1. 一道「贪心」题目能搞清楚证明，那么同类的「贪心」题目你就都会做了。**否则就会停留在“我知道这道题可以这样贪心，别的题我不确定是否也能这样做”**。
2. 在「面试」阶段，**你可以很清晰讲解你的思路**。让面试官从你的「思维方式」上喜欢上你（ emmm 当然从颜值上也可以 😃

…

更多与证明/分析相关的题解：

765. 情侣牵手：为什么交换任意一个都是对的？：两种 100% 的解法：并查集 & 贪心
1579. 保证图可完全遍历：为什么先处理公共边是对的？含贪心证明 + 数组模板 ~

1631. 最小体力消耗路径：反证法证明思路的合法性

11. 盛最多水的容器：双指针+贪心解法【含证明】

> **🍭 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 **765. 情侣牵手** ，难度为 **困难**。

Tag：「并查集」、「贪心」

N 对情侣坐在连续排列的 2N 个座位上，想要牵到对方的手。 计算最少交换座位的次数，以便每对情侣可以并肩坐在一起。 一次交换可选择任意两人，让他们站起来交换座位。

人和座位用 0 到 2N-1 的整数表示，情侣们按顺序编号，第一对是 (0, 1)，第二对是 (2, 3)，以此类推，最后一对是 (2N-2, 2N-1)。

这些情侣的初始座位 row[i] 是由最初始坐在第 i 个座位上的人决定的。

示例 1:

```
输入: row = [0, 2, 1, 3]
输出: 1
解释: 我们只需要交换row[1]和row[2]的位置即可。
```

示例 2:

```
输入: row = [3, 2, 0, 1]
输出: 0
解释: 无需交换座位，所有的情侣都已经可以手牵手了。
```

说明:

- len(row) 是偶数且数值在 [4, 60]范围内。
- 可以保证row 是序列 `0...len(row)-1` 的一个全排列。

并查集

首先，我们总是以「情侣对」为单位进行设想：

1. 当有两对情侣相互坐错了位置，ta们两对之间形成了一个环。需要进行一次交换，使得每队情侣独立（相互牵手）
2. 如果三对情侣相互坐错了位置，ta们三对之间形成了一个环，需要进行两次交换，使得每队情侣独立（相互牵手）
3. 如果四对情侣相互坐错了位置，ta们四对之间形成了一个环，需要进行三次交换，使得每队情侣独立（相互牵手）

也就是说，如果我们有 `k` 对情侣形成了错误环，需要交换 `k - 1` 次才能让情侣牵手。

于是问题转化成 `n / 2` 对情侣中，有多少个这样的环。

可以直接使用「并查集」来做。

由于 0和1配对、2和3配对 ... 因此互为情侣的两个编号除以 2 对应同一个数字，可直接作为它们的「情侣组」编号：

```Java
class Solution {
    int[] p = new int[70];
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }
    public int minSwapsCouples(int[] row) {
        int n = row.length, m = n / 2;
        for (int i = 0; i < m; i++) p[i] = i;
        for (int i = 0; i < n; i += 2) union(row[i] / 2, row[i + 1] / 2);
        int cnt = 0;
        for (int i = 0; i < m; i++) {
            if (i == find(i)) cnt++;
        }
        return m - cnt;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

贪心

还是以「情侣对」为单位进行分析：

由于题目保证有解，我们也可以从前往后（每两格作为一步）处理，对于某一个位置而言，如果下一个位置不是应该出现的情侣的话。

则对下一个位置进行交换。

同时为了方便我们找到某个值的下标，需要先对 `row` 进行预处理（可以使用哈希表或数组）。

```
class Solution {
    public int minSwapsCouples(int[] row) {
        int n = row.length;
        int ans = 0;
        int[] cache = new int[n];
        for (int i = 0; i < n; i++) cache[row[i]] = i;
        for (int i = 0; i < n - 1; i += 2) {
            int a = row[i], b = a ^ 1;
            if (row[i + 1] != b) {
                int src = i + 1, tar = cache[b];
                cache[row[tar]] = src;
                cache[row[src]] = tar;
                swap(row, src, tar);
                ans++;
            }
        }
        return ans;
    }
    void swap(int[] nums, int a, int b) {
        int c = nums[a];
        nums[a] = nums[b];
        nums[b] = c;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

证明/分析

我们这样的做法本质是什么？

其实相当于，当我处理到第 `k` 个位置的时候，前面的 `k - 1` 个位置的情侣已经牵手成功了。我接下来怎么处理，能够使得总花销最低。

分两种情况讨论：

a. 现在处理第 `k` 个位置，使其牵手成功：

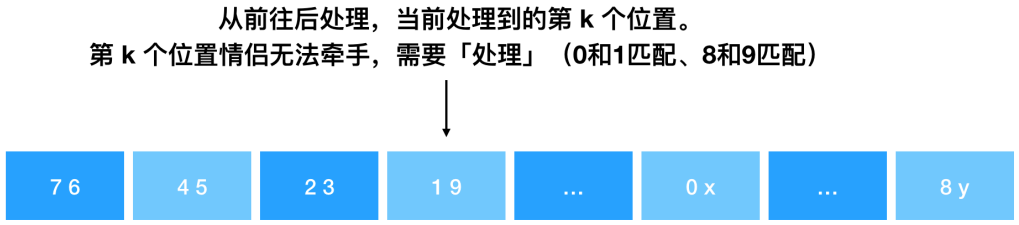
那么我要使得第 `k` 个位置的情侣也牵手成功，那么必然是保留第 `k` 个位置的情侣中其中一位，再进行修改，这样的成本是最小的（因为只需要交换一次）。

而且由于前面的情侣已经牵手成功了，因此交换的情侣必然在 `k` 位置的后面。

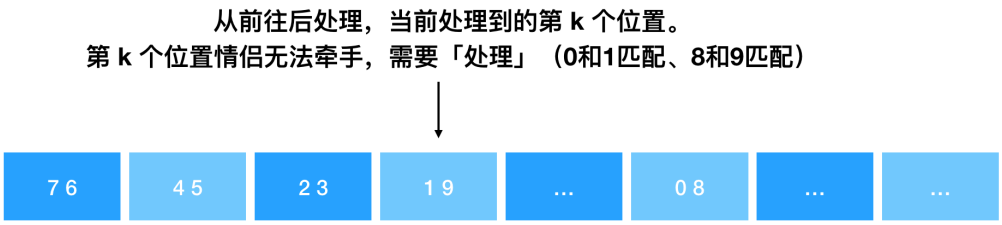
然后我们再考虑交换左边或者右边对最终结果的影响。

分两种情况来讨论：

1. 与第 `k` 个位置的匹配的两个情侣不在同一个位置上：这时候无论交换左边还是右边，后面需要调整的「情侣对数量」都是一样。假设处理第 `k` 个位置前需要调整的数量为 `n` 的话，处理完第 `k` 个位置（交换左边或是右边），需要调整的「情侣对数量」都为 `n - 1` 。



2. 与第 `k` 个位置的匹配的两个情侣在同一个位置上：这时候无论交换左边还是右边，后面需要调整的「情侣对数量」都是一样。假设处理第 `k` 个位置前需要调整的数量为 `n` 的话，处理完第 `k` 个位置（交换左边或是右边），需要调整的「情侣对数量」都为 `n - 2` 。



因此对于第 `k` 个位置而言，交换左边还是右边，并不会影响后续需要调整的「情侣对数量」。

b. 现在先不处理第 `k` 个位置，等到后面的情侣处理的时候「顺便」处理第 `k` 位置：

由于我们最终都是要所有位置的情侣牵手，而且每一个数值对应的情侣数值是唯一确定的。

因此我们这个等“后面”的位置处理，其实就是等与第 `k` 个位置互为情侣的位置处理（对应上图的就是我们是在等 【0 x】和【8 y】或者【0 8】这些位置被处理）。

由于被处理都是同一批的联通位置，因此和「a. 现在处理第 `k` 个位置」的分析结果是一样的。

不失一般性的，我们可以将这个分析推广到第一个位置，其实就已经是符合「当我处理到第 `k` 个位置的时候，前面的 `k - 1` 个位置的情侣已经牵手成功了」的定义了。

综上所述，我们只需要确保从前往后处理，并且每次处理都保留第 `k` 个位置的其中一位，无论保留的左边还是右边都能得到最优解。

> **🍭 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 **781. 森林中的兔子** ，难度为 **中等**。

Tag : 「贪心算法」

森林中，每个兔子都有颜色。其中一些兔子（可能是全部）告诉你还有多少其他的兔子和自己有相同的颜色。我们将这些回答放在 answers 数组里。

返回森林中兔子的最少数量。

示例:

```
输入: answers = [1, 1, 2]

输出: 5

解释:
两只回答了 "1" 的兔子可能有相同的颜色，设为红色。
之后回答了 "2" 的兔子不会是红色，否则他们的回答会相互矛盾。
设回答了 "2" 的兔子为蓝色。
此外，森林中还应有另外 2 只蓝色兔子的回答没有包含在数组中。
因此森林中兔子的最少数量是 5: 3 只回答的和 2 只没有回答的。

输入: answers = [10, 10, 10]
输出: 11

输入: answers = []
输出: 0
```

说明:

- answers 的长度最大为1000。
- answers[i] 是在 [0, 999] 范围内的整数。

基本分析

首先，兔子它不会说谎 (‘·ω·’)，因此我们可以得出以下结论：

- **同一颜色的兔子回答的数值必然是一样的**
- **但回答同样数值的，不一定就是同颜色兔子**

举个🌰，假如有 3 只白兔，每只白兔的回答必然都是 2（对应结论 1）；但假如有兔子回答了数值 2，可能只是三只白兔，也可能是三只白兔和三只灰兔同时进行了回答（对应结论 2）。

答案要我们求最少的兔子数量。

不妨设有某种颜色的兔子 m 只，它们回答的答案数值为 cnt，那么 m 和 cnt 满足什么关系？

显然两者关系为 $m = cnt + 1$。

但如果是在 $answers$ 数组里，回答 cnt 的数量为 t 的话呢？这时候我们需要分情况讨论：

- $t \leqslant cnt + 1$：为达到「最少的兔子数量」，我们可以假定这 t 只兔子为同一颜色，这样能够满足题意，同时不会导致「额外」兔子数量增加（颜色数量最少）。
- $t > cnt + 1$：我们知道回答 cnt 的兔子应该有 $cnt + 1$ 只。这时候说明有数量相同的不同颜色的兔子进行了回答。为达到「最少的兔子数量」，我们应当将 t 分为若干种颜色，并尽可能让某一种颜色的兔子为 $cnt + 1$ 只，这样能够满足题意，同时不会导致「额外」兔子数量增加（颜色数量最少）。

换句话说，我们应该让「同一颜色的兔子数量」尽量多，从而实现「总的兔子数量」最少。

证明

我们来证明一下，为什么这样的贪心思路是对的：

基于上述分析，我们其实是在处理 $answers$ 数组中每一个 cnt ，使得满足题意的前提下，数值 cnt 带来的影响（总的兔子数量，或者说总的颜色数量）最小。

首先 $answers$ 中的每个数都会导致我们总的兔子数量增加，**因此我们应该「让 $answers$ 的数字尽可能变少」，也就是我们需要去抵消掉 $answers$ 中的一些数字。**

对于 $answers$ 中的某个 cnt 而言（注意是某个 cnt，含义为一只兔子的回答），必然代表了有 $cnt + 1$ 只兔子，同时也代表了数值 cnt 最多在 $answers$ 数组中出现 $cnt + 1$ 次（与其颜色相同的兔子都进行了回答）。

这时候我们可以从数组中移走 $cnt + 1$ 个数值 cnt（如果有的话）。

当每次处理 cnt 的时候，我们都执行这样的抵消操作。最后得到的 $answers$ 数值数量必然最少（而固定），抵消完成后的 $answers$ 中的每个 cnt 对答案的影响也固定（增加 $cnt + 1$ ），从而实现「总的兔子数量」最少。

相反，如果不执行这样的操作的话，得到的 $answers$ 数值数量必然会更多，「总的兔子数量」也必然会更多，也必然不会比这样做法更优。

模拟解法

按照上述思路，我们可以先对 $answers$ 进行排序，然后根据遍历到某个 cnt 时，将其对答案的影响应用到 ans 中（`ans += cnt + 1`），并将后面的 cnt 个 cnt 进行忽略。

代码：

```
class Solution {
    public int numRabbits(int[] cs) {
        Arrays.sort(cs);
        int n = cs.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int cnt = cs[i];
            ans += cnt + 1;
            // 跳过「数值 cnt」后面的 cnt 个「数值 cnt」
            int k = cnt;
            while (k-- > 0 && i + 1 < n && cs[i] == cs[i + 1]) i++;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(1)$

统计分配

我们也可以先对所有出现过的数字进行统计，然后再对数值进行（颜色）分配。

代码：

```Java
class Solution {
    int N = 1009;
    int[] counts = new int[N];
    public int numRabbits(int[] cs) {
        // counts[x] = cnt 代表在 cs 中数值 x 的数量为 cnt
        for (int i : cs) counts[i]++;
        int ans = counts[0];
        for (int i = 1; i < N; i++) {
            int per = i + 1;
            int cnt = counts[i];
            int k = cnt / per;
            if (k * per < cnt) k++;
            ans += k * per;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

拓展

保持题目现有的条件不变，假定颜色相同的兔子至少有一只发声，问题改为「问兔子颜色数量可能有多少种」，又该如何求解呢？

> **🍭 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 **881. 救生艇** ，难度为 **中等**。

Tag：「贪心」、「双指针」

第 `i` 个人的体重为 `people[i]` ，每艘船可以承载的最大重量为 `limit` 。

每艘船最多可同时载两人，但条件是这些人的重量之和最多为 `limit` 。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1：

```
输入：people = [1,2], limit = 3

输出：1

解释：1 艘船载 (1, 2)
```

示例 2：

```
输入：people = [3,2,2,1], limit = 3

输出：3

解释：3 艘船分别载 (1, 2), (2) 和 (3)
```

示例 3：

```
输入：people = [3,5,3,4], limit = 5

输出：4

解释：4 艘船分别载 (3), (3), (4), (5)
```

提示：

- 1 <= people.length <= 50000
- 1 <= people[i] <= limit <= 30000

贪心

一个直观的想法是：由于一个船要么载两人要么载一人，在人数给定的情况下，为了让使用的总船数最小，要当尽可能让更多船载两人，即尽可能多的构造出数量之和不超过 $limit$ 的二元组。

先对 $people$ 进行排序，然后使用两个指针 `l` 和 `r` 分别从首尾开始进行匹配：

- 如果 $people[l] + people[r] <= limit$，说明两者可以同船，此时船的数量加一，两个指针分别往中间靠拢；
- 如果 $people[l] + people[r] > limit$，说明不能成组，由于题目确保人的重量不会超过 $limit$，此时让 $people[r]$ 独立成船，船的数量加一，`r` 指针左移。

我们猜想这样「最重匹配最轻、次重匹配次轻」的做法能使双人船的重量之和尽可能平均，从而使双人船的数量最大化。

接下来，我们使用「归纳法」证明猜想的正确性。

假设最优成船组合中二元组的数量为 $c1$，我们贪心做法的二元组数量为 $c2$。

最终答案 = 符合条件的二元组的数量 + 剩余人数数量，而在符合条件的二元组数量固定的情况下，剩余人数也固定。因此我们只需要证明 $c1 = c2$ 即可。

通常使用「归纳法」进行证明，都会先从边界入手。

当我们处理最重的人 $people[r]$（此时 r 为原始右边界 $n - 1$）时：

- 假设其与 $people[l]$（此时 l 为原始左边界 0）之和超过 $limit$，说明 $people[r]$ 与数组任一成员组合都会超过 $limit$，即无论在最优组合还是贪心组合中，$people[r]$ 都会独立成船；
- 假设 $people[r] + people[l] <= limit$，说明数组中存在至少一个成员能够与 $people[l]$ 成船：
    - 假设在最优组合中 $people[l]$ 独立成船，此时如果将贪心组合 $(people[l], people[r])$ 中的 $people[l]$ 拆分出来独立成船，贪心二元组数量 $c2$ 必然不会变大（可能还会变差），即将「贪心解」调整成「最优解」结果不会变好；
    - 假设在最优组合中，$people[l]$ 不是独立成船，又因此当前 r 处于原始右边界，因此与 $people[l]$ 成组的成员 $people[x]$ 必然满足 $people[x] <= people[r]$。
      此时我们将 $people[x]$ 和 $people[r]$ 位置进行交换（将贪心组合调整成最优组合），此时带来的影响包括：
        - 与 $people[l]$ 成组的对象从 $people[r]$ 变为 $people[x]$，但因为 $people[x] <= people[r]$，即有 $people[l] + people[x] <= people[l] + people[r] <= limit$，仍为合法二元组，消耗船的数量为 1；
        - 原本位置 x 的值从 $people[x]$ 变大为 $people[r]$，如果调

整后的值能组成二元组，那么原本更小的值也能组成二元组，结果没有变化；如果调整后不能成为组成二元组，那么结果可能会因此变差。

综上，将 $people[x]$ 和 $people[r]$ 位置进行交换（将贪心组合调整成最优组合），贪心二元组数量 $c2$ 不会变大，即将「贪心解」调整成「最优解」结果不会变好。

对于边界情况，我们证明了从「贪心解」调整为「最优解」不会使得结果更好，因此可以保留当前的贪心决策，然后将问题规模缩减为 $n-1$ 或者 $n-2$，同时数列仍然满足升序特性，即归纳分析所依赖的结构没有发生改变，可以将上述的推理分析推广到每一个决策的回合（新边界）中。

至此，我们证明了将「贪心解」调整为「最优解」结果不会变好，即贪心解是最优解之一。

代码：

```
class Solution {
    public int numRescueBoats(int[] people, int limit) {
        Arrays.sort(people);
        int n = people.length;
        int l = 0, r = n - 1;
        int ans = 0;
        while (l <= r) {
            if (people[l] + people[r] <= limit) l++;
            r--;
            ans++;
        }
        return ans;
    }
}
```

- 时间复杂度：排序复杂度为 $O(n\log n)$；双指针统计答案复杂度为 $O(n)$。整体复杂度为 $O(n\log n)$
- 空间复杂度：$O(\log n)$

题目描述

这是 LeetCode 上的 **995. K 连续位的最小翻转次数** ，难度为 **困难**。

Tag：「贪心」、「差分」

在仅包含 0 和 1 的数组 A 中，一次 K 位翻转包括选择一个长度为 K 的（连续）子数组，同时将子数组中的每个 0 更改为 1，而每个 1 更改为 0。

返回所需的 K 位翻转的最小次数，以便数组没有值为 0 的元素。如果不可能，返回 -1。

示例 1：

```
输入：A = [0,1,0], K = 1
输出：2
解释：先翻转 A[0]，然后翻转 A[2]。
```

示例 2：

```
输入：A = [1,1,0], K = 2
输出：-1
解释：无论我们怎样翻转大小为 2 的子数组，我们都不能使数组变为 [1,1,1]。
```

示例 3：

```
输入：A = [0,0,0,1,0,1,1,0], K = 3
输出：3
解释：
翻转 A[0],A[1],A[2]: A变成 [1,1,1,1,0,1,1,0]
翻转 A[4],A[5],A[6]: A变成 [1,1,1,1,1,0,0,0]
翻转 A[5],A[6],A[7]: A变成 [1,1,1,1,1,1,1,1]
```

提示：

- 1 <= A.length <= 30000
- 1 <= K <= A.length

贪心解法

目标是将数组的每一位都变为 1 ，因此对于每一位 0 都需要翻转。

我们可以从前往后处理，遇到 0 则对后面的 `k` 位进行翻转。

这样我们的算法复杂度是 $O(nk)$ 的，数据范围是 3w（数量级为 10^4），极限数据下单秒的运算量在 10^8 以上，会有超时风险。

```
class Solution {
    public int minKBitFlips(int[] nums, int k) {
        int n = nums.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (nums[i] == 0) {
                if (i + k > n) return -1;
                for (int j = i; j < i + k; j++) nums[j] ^= 1;
                ans++;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(nk)$
- 空间复杂度：$O(1)$

补充

评论有同学提出了一些有价值的疑问，我觉得挺有代表性的，因此补充到题解：

1. **为什么这样的解法是「贪心解法」，而不是「暴力解法」？**

首先「暴力解法」必然是**对所有可能出现的翻转方案进行枚举**，然后检查每一个方案得到的结果是否符合全是 1 的要求。

这样的解法，才是暴力解法，它的本质是通过「穷举」找答案。复杂度是指数级别的。

而我们的「朴素贪心解法」只是执行了众多翻转方案中的一种。

举个 🌰，对于 `nums = [0,0,1,1]` 并且 `k = 2` 的数据：

暴力解法应该是「枚举」以下三种方案：

1. 只翻转以第一个 0 开头的子数组（长度固定为 2）
2. 只翻转以第二个 0 开头的子数组（长度固定为 2）
3. 同时翻转第一个 0 开头和第二个 0 开头的子数组（长度固定为 2，只不过这时候第一个 0 被翻转了一次，第二个 0 被翻转了两次）

然后对三种方案得到的最终解进行检查，找出符合结果全是 1 的方案。这种通过「穷举」方案检查合法性的解法才是「暴力」解法。

2. **为什么我采用了与「朴素贪心」解法相似的做法，超时了？**

结果测试 C++、Python 超时，只有 Java 能过。

同样是 97 号样例数据，提交给 LeetCode 执行。Java 运行 200 ms 以内，而 C++ 运行 600 ms。

贪心 + 差分解法

由于我们总是对连续的一段进行「相同」的操作，同时只有「奇数」次数的翻转才会真正改变当前位置上的值。

自然而然，我们会想到使用数组 `arr` 来记录每一位的翻转次数。

同时我们又不希望是通过「遍历记 `arr` 的 `k` 位进行 +1」来完成统计。

因此可以使用差分数组来进行优化：当需要对某一段 `[l,r]` 进行 +1 的时候，只需要 `arr[l]++` 和 `arr[r + 1]--` 即可。

```java
class Solution {
    public int minKBitFlips(int[] nums, int k) {
        int n = nums.length;
        int ans = 0;
        int[] arr = new int[n + 1];
        for (int i = 0, cnt = 0; i < n; i++) {
            cnt += arr[i];
            if ((nums[i] + cnt) % 2 == 0) {
                if (i + k > n) return -1;
                arr[i + 1]++;
                arr[i + k]--;
                ans++;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

证明

为什么「一遇到 0 就马上进行翻转」这样的做法得到的是最优解？

这道题的贪心证明思路和 765. 情侣牵手 是一样的。

本质上是在证明**当我在处理第 `k` 个位置的 0 的时候，前面 `k - 1` 个位置不存在 0，接下来要如何进行操作，可使得总的翻转次数最小。**

如果你上次真正理解了我的证明过程的话，那么你会很容易就能证明出本题的贪心思路。

所以这次将这个证明过程留给大家思考 ~

为什么要「证明」或「理解证明」？

证明的意义在于，**你知道为什么这样做是对的**。

带来的好处是：

1. 一道「贪心」题目能搞清楚证明，那么同类的「贪心」题目你就都会做了。**否则就会停留在“我知道这道题可以这样贪心，别的题我不确定是否也能这样做”**。
2. 在「面试」阶段，**你可以很清晰讲解你的思路**。让面试官从你的「思维方式」上喜欢上你

更多与证明/分析相关的题解：

561. 数组拆分 I：反证法证明贪心算法的正确性

765. 情侣牵手：为什么交换任意一个都是对的？：两种 100% 的解法：并查集 & 贪心

1579. 保证图可完全遍历：为什么先处理公共边是对的？含贪心证明 + 数组模板 ~

1631. 最小体力消耗路径：反证法证明思路的合法性

11. 盛最多水的容器：双指针+贪心解法【含证明】

> **🍭 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 **1221. 分割平衡字符串** ，难度为 **简单**。

Tag：「贪心」、「双指针」

在一个 平衡字符串 中，‘L’ 和 ‘R’ 字符的数量是相同的。

给你一个平衡字符串 s，请你将它分割成尽可能多的平衡字符串。

注意：分割得到的每个字符串都必须是平衡字符串。

返回可以通过分割得到的平衡字符串的 最大数量 。

示例 1：

输入：s = "RLRRLLRLRL"

输出：4

解释：s 可以分割为 "RL"、"RRLL"、"RL"、"RL" ，每个子字符串中都包含相同数量的 'L' 和 'R' 。

示例 2：

输入：s = "RLLLLRRRLR"

输出：3

解释：s 可以分割为 "RL"、"LLLRRR"、"LR" ，每个子字符串中都包含相同数量的 'L' 和 'R' 。

示例 3：

输入：s = "LLLLRRRR"

输出：1

解释：s 只能保持原样 "LLLLRRRR".

示例 4：

输入：s = "RLRRRLLRLL"

输出：2

解释：s 可以分割为 "RL"、"RRRLLRLL" ，每个子字符串中都包含相同数量的 'L' 和 'R' 。

提示：

- 1 <= s.length <= 1000
- s[i] = ‘L’ 或 ‘R’
- s 是一个 平衡 字符串

基本分析

题目确保了 `s` 为一个平衡字符串，即必然能分割成若干个 `LR` 子串。

一个合法的 `LR` 子串满足 `L` 字符和 `R` 字符数量相等，常规检查一个字符串是否为合格的 `LR` 子串可以使用 $O(n)$ 的遍历方式，可以使用记录前缀信息的数据结构，而对于成对结构的元素统计，更好的方式是转换为数学判定，使用 `1` 来代指 `L` 得分，使用 `-1` 来代指 `R` 得分。

那么一个子串为合格 `LR` 子串的充要条件为 **整个 `LR` 子串的总得分为 0**。

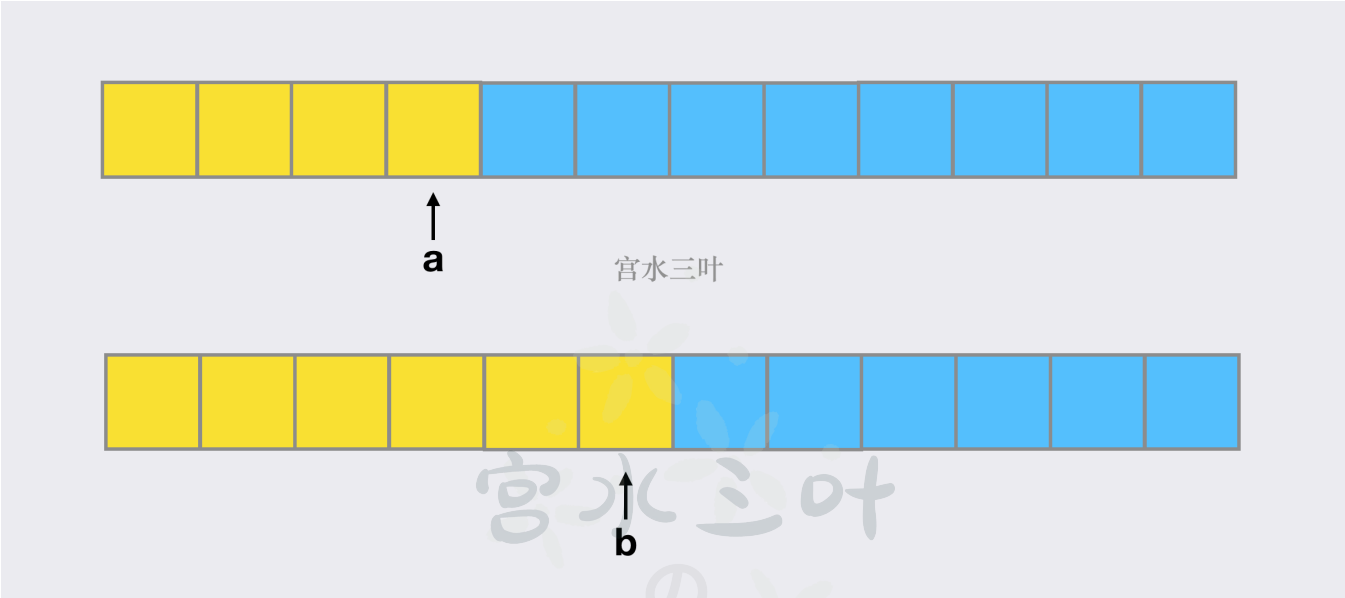
这种方式最早应该在 (题解) 301. 删除无效的括号 详细讲过，可延伸到任意的成对结构元素统计题目里去。

贪心

回到本题，题目要求分割的 `LR` 子串尽可能多，直观上应该是尽可能让每个分割串尽可能短。

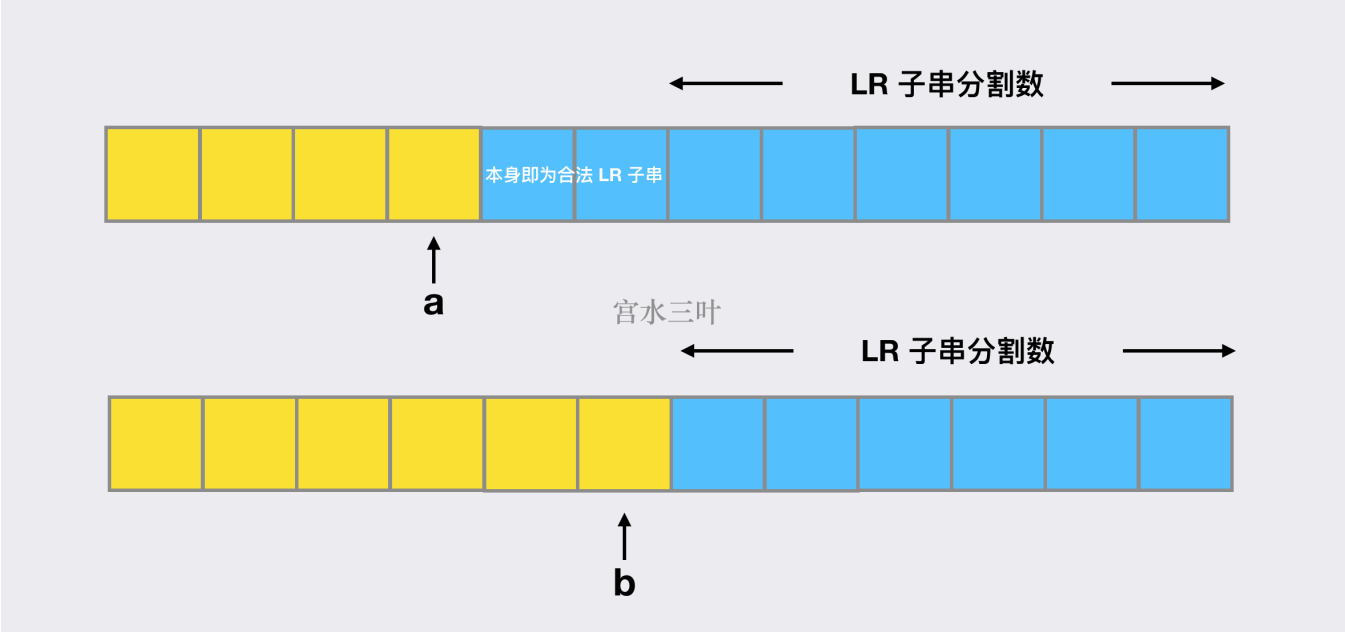
我们使用「归纳法」来证明该猜想的正确性。

首先题目数据保证给定的 `s` 本身是合法的 `LR` 子串，假设从 $[0...a]$ 可以从 `s` 中分割出 **长度最小** 的 `LR` 子串，而从 $[0...b]$ 能够分割出 **长度更大** 的 `LR` 子串（即 `a <= b`）。



我们来证明起始时（第一次分割）「将从 `b` 分割点将 `s` 断开」调整为「从 `a` 分割点将 `s` 断开」结果不会变差：

1. 从 `b` 点首次分割调整为从 `a` 点首次分割，两种分割形式分割点两端仍为合法 `LR` 子串，因此不会从“有解”变成“无解”；
2. 从 `b` 分割后的剩余部分长度小于从 `a` 分割后的剩余部分，同时由 `b` 分割后的剩余部分会被由 `a` 分割后的剩余部分所严格覆盖，因此「对 `a` 分割的剩余部分再分割所得的子串数量」**至少** 与「从 `b` 点分割的剩余部分再分割所得的子串数量」相等（不会变少）。



至此，我们证明了对于首次分割，将任意合格分割点调整为最小分割点，结果不会变得更差（当 `a < b` 时还会更好）。

同时，由于首次分割后的剩余部分仍为合格的 `LR` 子串，因此归纳分析所依赖的结构没有发生改变，可以将上述的推理分析推广到每一个决策的回合（新边界）中。

至此，我们证明了只要每一次都从最小分割点进行分割，就可以得到最优解。

代码：

```java
class Solution {
    public int balancedStringSplit(String s) {
        char[] cs = s.toCharArray();
        int n = cs.length;
        int ans = 0;
        for (int i = 0; i < n; ) {
            int j = i + 1, score = cs[i] == 'L' ? 1 : -1;
            while (j < n && score != 0) score += cs[j++] == 'L' ? 1 : -1;
            i = j;
            ans++;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：调用 `toCharArray` 会拷贝新数组进行返回（为遵循 `String` 的不可变原则），因此使用 `toCharArray` 复杂度为 $O(n)$，使用 `charAt` 复杂度为 $O(1)$

> **🍭 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 **1707. 与数组中元素的最大异或值** ，难度为 **困难**。

Tag：「Trie」、「字典树」、「二分」

给你一个由非负整数组成的数组 nums 。另有一个查询数组 queries ，其中 queries[i] = [xi, mi] 。

第 i 个查询的答案是 xi 和任何 nums 数组中不超过 mi 的元素按位异或（XOR）得到的最大值。

换句话说，答案是 max(nums[j] XOR xi) ，其中所有 j 均满足 nums[j] <= mi 。如果 nums 中的所有元素都大于 mi，最终答案就是 -1 。

返回一个整数数组 answer 作为查询的答案，其中 answer.length == queries.length 且 answer[i] 是第 i 个查询的答案。

示例 1：

```
输入：nums = [0,1,2,3,4], queries = [[3,1],[1,3],[5,6]]

输出：[3,3,7]

解释：
1) 0 和 1 是仅有的两个不超过 1 的整数。0 XOR 3 = 3 而 1 XOR 3 = 2 。二者中的更大值是 3 。
2) 1 XOR 2 = 3.
3) 5 XOR 2 = 7.
```

示例 2：

```
输入：nums = [5,2,4,6,6,3], queries = [[12,4],[8,1],[6,3]]

输出：[15,-1,5]
```

提示：

- 1 <= nums.length, queries.length <= 10^5
- queries[i].length == 2
- 0 <= nums[j], xi, mi <= 10^9

基本分析

在做本题之前，请先确保已经完成 421. 数组中两个数的最大异或值。

这种提前给定了所有询问的题目，我们可以运用离线思想（调整询问的回答顺序）进行求解。

对于本题有两种离线方式可以进行求解。

普通 Trie

第一种方法基本思路是：**不一次性地放入所有数，而是每次将需要参与筛选的数字放入 $Trie$，再进行与 421. 数组中两个数的最大异或值 类似的贪心查找逻辑。**

具体的，我们可以按照下面的逻辑进行处理：

1. 对 `nums` 进行「从小到大」进行排序，对 `queries` 的第二维进行「从小到大」排序（排序前先将询问原本的下标映射关系存下来）。
2. 按照排序顺序处理所有的 `queries[i]`：
    1. 在回答每个询问前，将小于等于 `queries[i][1]` 的数值存入 $Trie$。由于我们已经事先对 `nums` 进行排序，因此这个过程只需要维护一个在 `nums` 上有往右移动的指针即可。
    2. 然后利用贪心思路，查询每个 `queries[i][0]` 所能找到的最大值是多少，计算异或和（此过程与 421. 数组中两个数的最大异或值 一致）。
    3. 找到当前询问在原询问序列的下标，将答案存入。

代码：

```java
class Solution {
    static int N = (int)1e5 * 32;
    static int[][] trie = new int[N][2];
    static int idx = 0;
    public Solution() {
        for (int i = 0; i <= idx; i++) {
            Arrays.fill(trie[i], 0);
        }
        idx = 0;
    }
    void add(int x) {
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int u = (x >> i) & 1;
            if (trie[p][u] == 0) trie[p][u] = ++idx;
            p = trie[p][u];
        }
    }
    int getVal(int x) {
        int ans = 0;
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int a = (x >> i) & 1, b = 1 - a;
            if (trie[p][b] != 0) {
                p = trie[p][b];
                ans = ans | (b << i);
            } else {
                p = trie[p][a];
                ans = ans | (a << i);
            }
        }
        return ans ^ x;
    }
    public int[] maximizeXor(int[] nums, int[][] qs) {
        int m = nums.length, n = qs.length;

        // 使用哈希表将原本的顺序保存下来
        Map<int[], Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) map.put(qs[i], i);

        // 将 nums 与 queries[x][1] 进行「从小到大」进行排序
        Arrays.sort(nums);
        Arrays.sort(qs, (a, b)->a[1]-b[1]);

        int[] ans = new int[n];
        int loc = 0; // 记录 nums 中哪些位置之前的数已经放入 Trie
```

```
        for (int[] q : qs) {
            int x = q[0], limit = q[1];
            // 将小于等于 limit 的数存入 Trie
            while (loc < m && nums[loc] <= limit) add(nums[loc++]);
            if (loc == 0) {
                ans[map.get(q)] = -1;
            } else {
                ans[map.get(q)] = getVal(x);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：令 `nums` 的长度为 `m`，`qs` 的长度为 `n`。两者排序的复杂度为 $O(m \log m)$ 和 $O(n \log n)$；将所有数插入 $Trie$ 和从 $Trie$ 中查找的复杂度均为 $O(Len)$，Len 为 32。
  整体复杂度为 $O(m \log m + n \log n + (m + n) * Len) = O(m * \max(\log m, Len) + n * \max(\log n, Len))$。
- 空间复杂度：$O(C)$。其中 C 为常数，固定为 $1e5 * 32 * 2$。

计数 Trie & 二分

另外一个比较「增加难度」的做法是，将整个过程翻转过来：**一次性存入所有的 $Trie$ 中，然后每次将不再参与的数从 $Trie$ 中移除。相比于解法一，这就要求我们为 $Trie$ 增加一个「删除/计数」功能，并且需要实现二分来找到移除元素的上界下标是多少。**

具体的，我们可以按照下面的逻辑进行处理：

1. 对 `nums` 进行「从大到小」进行排序，对 `queries` 的第二维进行「从大到小」排序（排序前先将询问原本的下标映射关系存下来）。
2. 按照排序顺序处理所有的 `queries[i]`：
    1. 在回答每个询问前，通过「二分」找到在 `nums` 中第一个满足「小于等于 `queries[i][1]` 的下标在哪」，然后将该下标之前的数从 $Trie$ 中移除。同理，这个过程我们需要使用一个指针来记录上一次删除的下标位置，避免重复删除。
    2. 然后利用贪心思路，查询每个 `queries[i][0]` 所能找到的最大值是多

少。注意这是要判断当前节点是否有被计数，如果没有则返回 -1。

3. 找到当前询问在原询问序列的下标，将答案存入。

代码：

```java
class Solution {
    static int N = (int)1e5 * 32;
    static int[][] trie = new int[N][2];
    static int[] cnt = new int[N];
    static int idx = 0;
    public Solution() {
        for (int i = 0; i <= idx; i++) {
            Arrays.fill(trie[i], 0);
            cnt[i] = 0;
        }
        idx = 0;
    }
    // 往 Trie 存入(v = 1)/删除(v = -1) 某个数 x
    void add(int x, int v) {
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int u = (x >> i) & 1;
            if (trie[p][u] == 0) trie[p][u] = ++idx;
            p = trie[p][u];
            cnt[p] += v;
        }
    }
    int getVal(int x) {
        int ans = 0;
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int a = (x >> i) & 1, b = 1 - a;
            if (cnt[trie[p][b]] != 0) {
                p = trie[p][b];
                ans = ans | (b << i);
            } else if (cnt[trie[p][a]] != 0) {
                p = trie[p][a];
                ans = ans | (a << i);
            } else {
                return -1;
            }
        }
        return ans ^ x;
    }
    public int[] maximizeXor(int[] nums, int[][] qs) {
        int n = qs.length;

        // 使用哈希表将原本的顺序保存下来
        Map<int[], Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) map.put(qs[i], i);
```

```
        // 对两者排降序
        sort(nums);
        Arrays.sort(qs, (a, b)->b[1]-a[1]);

        // 将所有数存入 Trie
        for (int i : nums) add(i, 1);

        int[] ans = new int[n];
        int left = -1; // 在 nums 中下标「小于等于」left 的值都已经从 Trie 中移除
        for (int[] q : qs) {
            int x = q[0], limit = q[1];
            // 二分查找到待删除元素的右边界，将其右边界之前的所有值从 Trie 中移除。
            int right = getRight(nums, limit);
            for (int i = left + 1; i < right; i++) add(nums[i], -1);
            left = right - 1;
            ans[map.get(q)] = getVal(x);
        }
        return ans;
    }
    // 二分找到待删除的右边界
    int getRight(int[] nums, int limit) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int mid = l + r >> 1;
            if (nums[mid] <= limit) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return nums[r] <= limit ? r : r + 1;
    }
    // 对 nums 进行降序排序（Java 没有 Api 直接支持对基本类型 int 排倒序，其他语言可忽略）
    void sort(int[] nums) {
        Arrays.sort(nums);
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int c = nums[r];
            nums[r--] = nums[l];
            nums[l++] = c;
        }
    }
}
```

- 时间复杂度：令 `nums` 的长度为 `m`，`qs` 的长度为 `n`，常数 $Len = 32$。两者排序的复杂度为 $O(m \log m)$ 和 $O(n \log n)$；将所有数插入 $Trie$ 的复杂度为

$O(m * Len)$；每个查询都需要经过「二分」找边界，复杂度为 $O(n \log m)$；最坏情况下所有数都会从 $Trie$ 中被标记删除，复杂度为 $O(m * Len)$。
整体复杂度为 $O(m \log m + n \log n + n \log m + mLen) = O(m * \max(\log m, Len) + n * \max(\log m, \log n))$。

- 空间复杂度：$O(C)$。其中 C 为常数，固定为 $1e5 * 32 * 3$。

说明

这两种方法我都是采取「数组实现」，而且由于数据范围较大，都使用了 `static` 来优化大数组创建，具体的「优化原因」与「类实现 Trie 方式」可以在题解 421. 数组中两个数的最大异或值 查看，这里不再赘述。

> **🍭 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？

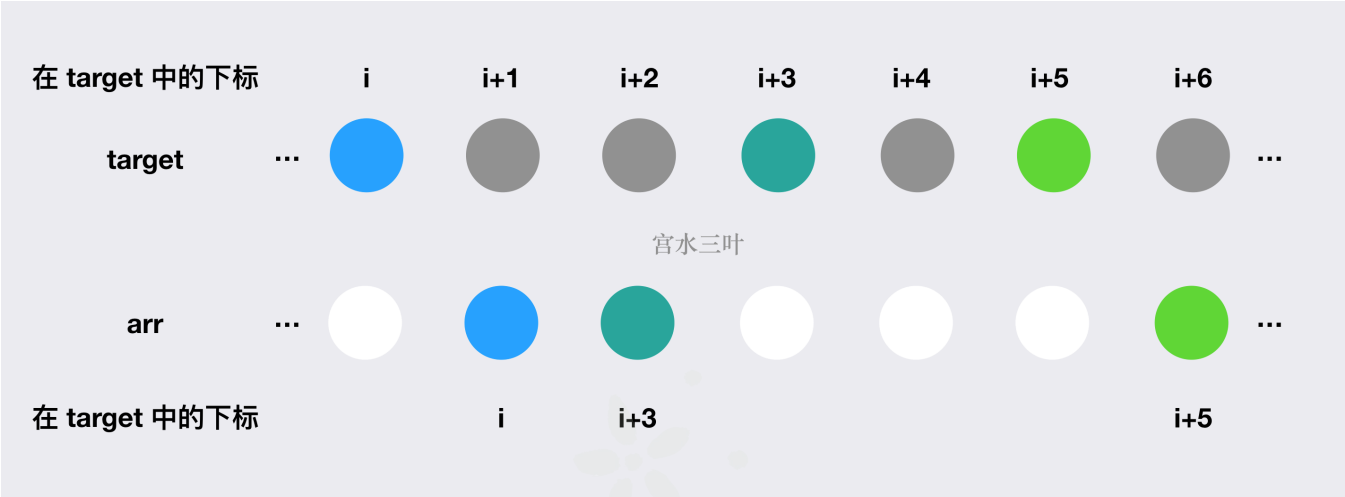
本质是利用「当其中一个数组元素各不相同时，这时候每一个“公共子序列”都对应一个不重复元素数组的下标数组“上升子序列”，反之亦然」。

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：$O(n + m)$

题目描述

这是 LeetCode 上的 **1736. 替换隐藏数字得到的最晚时间** ，难度为 **简单**。

Tag：「贪心」

给你一个字符串 `time` ，格式为 `hh:mm`（小时：分钟），其中某几位数字被隐藏（用 `?` 表示）。

有效的时间为 `00:00` 到 `23:59` 之间的所有时间，包括 `00:00` 和 `23:59` 。

替换 `time` 中隐藏的数字，返回你可以得到的最晚有效时间。

示例 1：

```
输入：time = "2?:?0"

输出："23:50"

解释：以数字 '2' 开头的最晚一小时是 23 ，以 '0' 结尾的最晚一分钟是 50 。
```

示例 2：

```
输入：time = "0?:3?"

输出："09:39"
```

示例 3：

```
输入：time = "1?:22"

输出："19:22"
```

提示：

- `time` 的格式为 `hh:mm`
- 题目数据保证你可以由输入的字符串生成有效的时间

贪心 + 模拟

规则十分简单，对每一位进行分情况讨论即可：

- 第一位：如果需要被替换，优先替换为 `2`，当然前提是第二位不能超过 `4`。否则会出现 `24:xx`、`25:xx` 等；
- 第二位：如果需要被替换，根据第一位是什么，决定替换为 `9` 还是 `3`；
- 第三位：固定为 `:`；
- 第四位：如果需要被替换，替换为 `5`；
- 第五位：如果需要被替换，替换为 `9`。

代码：

```
class Solution {
    public String maximumTime(String time) {
        StringBuilder sb = new StringBuilder();
        sb.append(time.charAt(0) == '?' ? (time.charAt(1) == '?' || time.charAt(1) < '4')
        sb.append(time.charAt(1) == '?' ? sb.charAt(0) == '2' ? '3' : '9' : time.charAt(1)
        sb.append(':');
        sb.append(time.charAt(3) == '?' ? '5' : time.charAt(3));
        sb.append(time.charAt(4) == '?' ? '9' : time.charAt(4));
        return sb.toString();
    }
}
```

- 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 **

题目描述

这是 LeetCode 上的 **1833. 雪糕的最大数量** ，难度为 **中等**。

Tag：「贪心」、「排序」

夏日炎炎，小男孩 Tony 想买一些雪糕消消暑。

商店中新到 `n` 支雪糕，用长度为 n 的数组 `costs` 表示雪糕的定价，其中 `costs[i]` 表示第

`i` 支雪糕的现金价格。

Tony 一共有 `coins` 现金可以用于消费，他想要买尽可能多的雪糕。

给你价格数组 `costs` 和现金量 `coins`，请你计算并返回 Tony 用 `coins` 现金能够买到的雪糕的 最大数量 。

注意：Tony 可以按任意顺序购买雪糕。

示例 1：

```
输入：costs = [1,3,2,4,1], coins = 7

输出：4

解释：Tony 可以买下标为 0、1、2、4 的雪糕，总价为 1 + 3 + 2 + 1 = 7
```

示例 2：

```
输入：costs = [10,6,8,7,7,8], coins = 5

输出：0

解释：Tony 没有足够的钱买任何一支雪糕。
```

```
示例 3：

输入：costs = [1,6,3,1,2,5],

输出：6

解释：Tony 可以买下所有的雪糕，总价为 1 + 6 + 3 + 1 + 2 + 5 = 18 。
```

提示：

- costs.length == n
- 1 <= n <= 10^5
- 1 <= costs[i] <= 10^5
- 1 <= coins <= 10^8

基本分析

从题面看，是一道「01 背包」问题，每个物品的成本为 $cost[i]$，价值为 1。

但「01 背包」的复杂度为 $O(N*C)$，其中 N 为物品数量（数量级为 10^5），C 为背包容量(数量级为 10^8)。显然会 TLE。

换个思路发现，每个被选择的物品对答案的贡献都是 1，优先选择价格小的物品会使得我们剩余金额尽可能的多，将来能够做的决策方案也就相应变多。

因此一个直观的做法是，对物品数组进行「从小到大」排序，然后「从前往后」开始决策购买。

证明

直观上，这样的贪心思路可以使得最终选择的物品数量最多。

接下来证明一下该思路的正确性。

假定贪心思路取得的序列为 $[a1, a2, a3, ..., an]$（长度为 n），真实最优解所取得的序列为 $[b1, b2, b3, .., bm]$（长度为 m）。

两个序列均为「单调递增序列」。

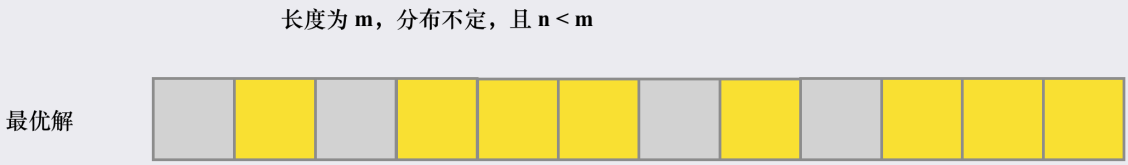
其中最优解所对应具体方案不唯一，即存在多种选择方案使得物品数量相同。

因此，**我们只需要证明两个序列长度一致即可**。

按照贪心逻辑，最终选择的方案总成本不会超过 $coins$，因此至少是一个合法的选择方案，天然有 $n \leq m$，只需要证明 $n \geq m$ 成立，即可得证 $n = m$。

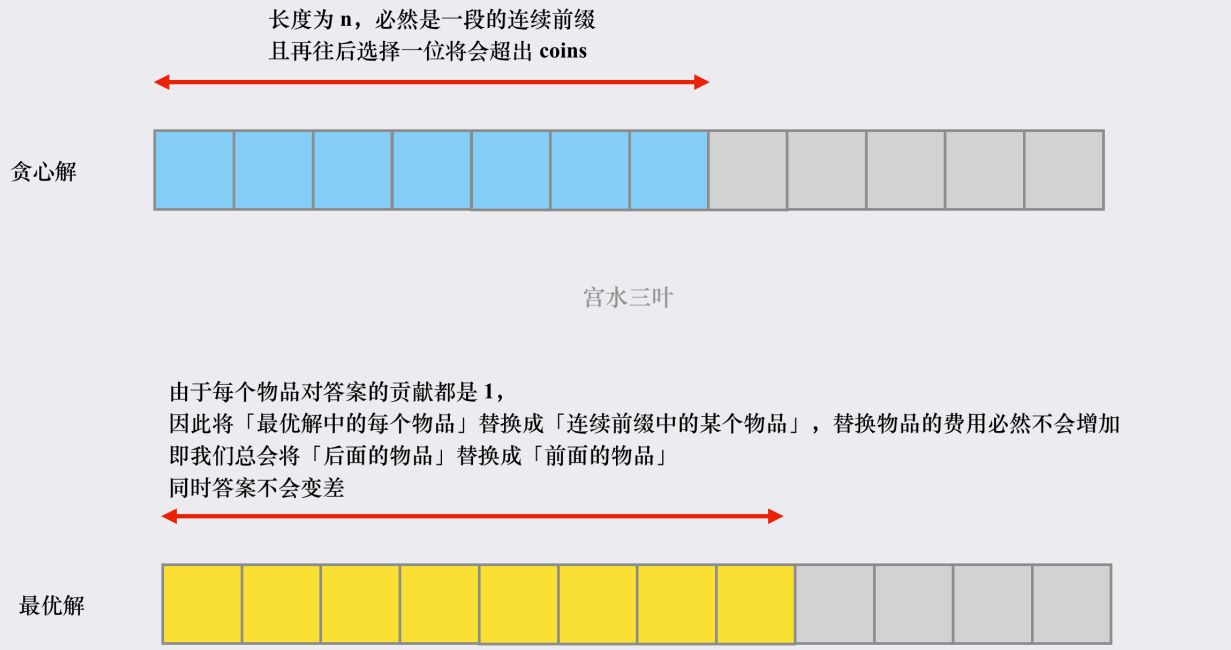
通过反证法证明 $n \geq m$ 成立，假设 $n \geq m$ 不成立，即有 $n < m$。

根据贪心决策，**我们选择的物品序列在「排序好的 $cost$ 数组」里，必然是一段连续的前缀。并且再选择下一物品将会超过总费用 $coins$；而真实最优解的选择方案在「排序好的 $cost$ 数组」里分布不定。**



这时候我们可以利用「**每个物品对答案的贡献均为 1，将最优解中的分布靠后的物品，替换为分布较前的物品，不会使得费用增加，同时答案不会变差**」。

从而将真实最优解也调整为某段连续的前缀。



综上，通过反证法得证 $n \geq m$ 成立，结合 $n \leq m$，可推出 $n = m$。

即贪心解必然能够取得与最优解一样的长度。

贪心

排序，从前往后决策，直到不能决策为止。

代码：

```java
class Solution {
    public int maxIceCream(int[] cs, int t) {
        int n = cs.length;
        Arrays.sort(cs);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (t >= cs[i]) {
                ans++;
                t -= cs[i];
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$；获取答案的复杂度为 $O(n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：排序复杂度为 $O(\log n)$。整体复杂度为 $O(\log n)$

PS. 这里假定 `Arrays.sort` 使用的是「双轴排序」的实现。

> **🍭 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 **1846. 减小和重新排列数组后的最大元素** ，难度为 **中等**。

Tag：「贪心」

给你一个正整数数组 arr 。请你对 arr 执行一些操作（也可以不进行任何操作），使得数组满足以下条件：

- arr 中 第一个 元素必须为 1 。
- 任意相邻两个元素的差的绝对值 小于等于 1 ，也就是说，对于任意的 1 <= i < arr.length （数组下标从 0 开始），都满足 abs(arr[i] - arr[i - 1]) <= 1 。abs(x) 为 x 的绝对值。

你可以执行以下 2 种操作任意次：

- 减小 arr 中任意元素的值，使其变为一个 更小的正整数 。
- 重新排列 arr 中的元素，你可以以任意顺序重新排列。

请你返回执行以上操作后，在满足前文所述的条件下，arr 中可能的 最大值 。

示例 1：

```
输入：arr = [2,2,1,2,1]

输出：2

解释：
我们可以重新排列 arr 得到 [1,2,2,2,1] ，该数组满足所有条件。
arr 中最大元素为 2 。
```

示例 2：

```
输入：arr = [100,1,1000]

输出：3

解释：
一个可行的方案如下：
1. 重新排列 arr 得到 [1,100,1000] 。
2. 将第二个元素减小为 2 。
3. 将第三个元素减小为 3 。
现在 arr = [1,2,3] ，满足所有条件。
arr 中最大元素为 3 。
```

示例 3：

```
输入：arr = [1,2,3,4,5]

输出：5

解释：数组已经满足所有条件，最大元素为 5 。
```

提示：

- 1 <= arr.length <= 10^5
  1 <= arr[i] <= 10^9

基本分析 & 证明

根据题意，数组的第一位必须是 1，且每个数只能 **减小** 或 **不变**，数值位置可以任意调整。

求解经过调整后，符合要求的数组中的最大值是多少。

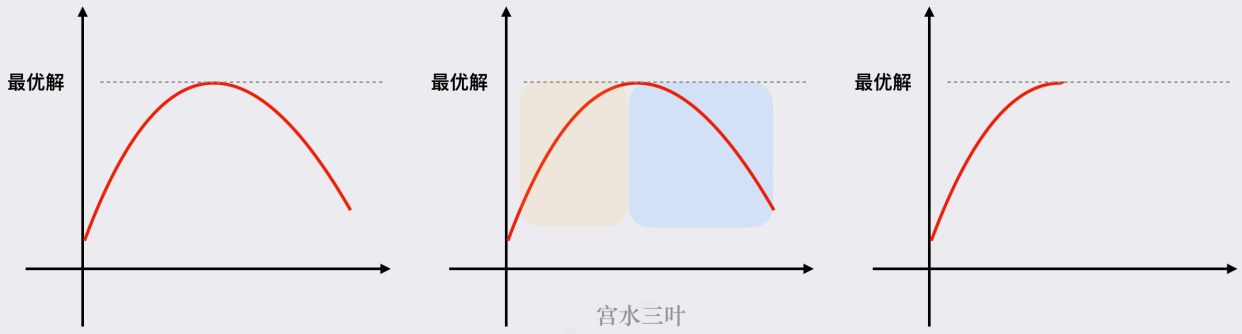
首先符合条件的数组相邻位差值绝对值不超过 1，这限定了数组的必然是如下三种分布之一：

- （非严格）单调递减
- 存在波段
- （非严格）单调递增

证明一：取得最优解对应的数组「必然是」或者「可调整为」（非严格）单调递增的形式。

我们使用反证法来证明另外两种分布不能取得最优解：

- （非严格）单调递减：题目限定了数的范围为正整数，且第一位为 1，这种情况不用讨论了，跳过；
- 存在波段：我们始终可以将波峰的右侧出现的值，纳入到波峰的左侧，从而消掉这个波峰，最终将整个分布调整为「（非严格）单调递增」的形式，结果不会变差：



将波峰右侧（蓝色）的点调整至波峰左侧（黄色）区域内

多个波段的情况也是同理，可以自己在纸上画画。

都是利用 **波峰右侧的点可以调整成波峰左侧的点，从而使分布变为（非严格）单调递增。**

至此，我们证明了最优解对应的数组必然符合（非严格）单调递增。

这启发我们可以先对原数组排个序，在此基础上进行分析。

对原数组排序得到的有序数组，不一定是符合「相邻位差值绝对值不超过 1」的，同时由于每个数值可以选择 **减小** 或 **不变**。

证明二：当必须要对当前位进行调整的时，优先选择调整为「与前一值差值为 1 的较大数」不会比调整为「与前一差值为 0 的较小数」更差。

这可以使用归纳推理，假设采取「优先调整为与前一值差值为 1 的较大数」得到的序列为 `a`，采用「优先调整与前一差值为 0 的较小数」得到的序列为 `b`。

根据「$a[0] = b[0] = 1$」、「`a` 和 `b` 长度一致」、「`a` 和 `b` 均为（非严格）单调递增」以及「`a` 和 `b` 均满足相邻位差值不超过 1」，可推导出 $sum(a) >= sum(b)$，和任意位置 $a[i] >= b[i]$，从而推导出 `a` 序列的最后一位必然大于等于 `b` 的最后一位。

即 `b` 不会比 `a` 更优。

证明三：调整大小的操作不会改变数组元素之间的相对位置关系。

在证明二的分析中，我们会对某些元素进行“减小”操作，使得整个数组最终满足「相邻位差值绝对值不超过 1」。

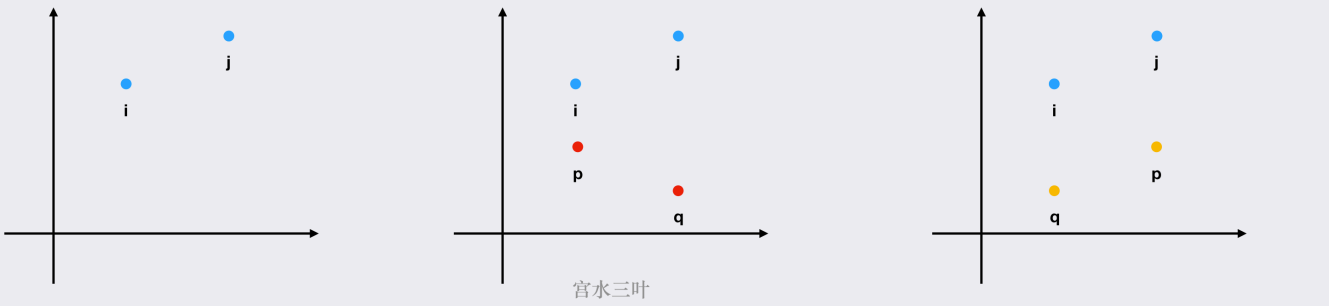
但该证明成立的还有一个很重要的前提条件，就是调整操作不会出发元素的位置重排。

那么该前提条件是否必然成立呢？答案是必然成立。

假设原排序数组中存在需要调整的点 i 和点 j，且 $nums[i] <= nums[j]$。

为了让数组满足条件，它们都进行了“减少”操作的调整，分别变为了 p 和 q，如果触发位置重排的话，必然有 $nums[p] >= nums[q]$。

此时，我们能够通过调整它们的变化关系：点 i 变为点 q、点 j 变成点 p 来确保同样满足条件，且不触发元素在有序数组中的位置重排。



贪心

排序，限定第一位值为 1，从前往后处理，根据每一位是否「必须修改（与上一位差值是否大于 1）」做决策，如果必须被修改，则修改为与前一值差值为 1 的较大数。

代码：

```
class Solution {
    public int maximumElementAfterDecrementingAndRearranging(int[] arr) {
        int n = arr.length;
        Arrays.sort(arr);
        arr[0] = 1;
        for (int i = 1; i < n; i++) {
            if (arr[i] - arr[i - 1] > 1) {
                arr[i] = arr[i - 1] + 1;
            }
        }
        return arr[n - 1];
    }
}
```

- 时间复杂度：假定 `Arrays.sort` 使用的是双轴快排实现。复杂度为 $O(n \log n)$
- 空间复杂度：假定 `Arrays.sort` 使用的是双轴快排实现。复杂度为 $O(\log n)$

> **🍭 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 **1877. 数组中最大数对和的最小值** ，难度为 **中等**。

Tag：「贪心」

一个数对 (a,b) 的 数对和 等于 a + b 。最大数对和 是一个数对数组中最大的 数对和 。

比方说，如果我们有数对 (1,5) ，(2,3) 和 (4,4)，最大数对和 为 max(1+5, 2+3, 4+4) = max(6, 5, 8) = 8 。
给你一个长度为 偶数 n 的数组 nums ，请你将 nums 中的元素分成 n / 2 个数对，使得：

- nums 中每个元素 恰好 在 一个 数对中，且
- 最大数对和 的值 最小 。

请你在最优数对划分的方案下，返回最小的 最大数对和 。

示例 1：

```
输入：nums = [3,5,2,3]

输出：7

解释：数组中的元素可以分为数对 (3,3) 和 (5,2) 。
最大数对和为 max(3+3, 5+2) = max(6, 7) = 7 。
```

示例 2：

```
输入：nums = [3,5,4,2,4,6]

输出：8

解释：数组中的元素可以分为数对 (3,5)，(4,4) 和 (6,2) 。
最大数对和为 max(3+5, 4+4, 6+2) = max(8, 8, 8) = 8 。
```

提示：

- n == nums.length
- 2 <= n <= 10^5

- n 是 偶数 。
- 1 <= nums[i] <= 10^5

基本分析 & 证明

直觉上，我们会认为「**尽量让“较小数”和“较大数”组成数对，可以有效避免出现“较大数成对”的现象**」。

我们来证明一下该猜想是否成立。

假定 $nums$ 本身有序，由于我们要将 $nums$ 拆分成 $n/2$ 个数对，根据猜想，我们得到的数对序列为：

$$(nums[0], nums[n-1]), (nums[1], nums[n-2]), ..., (nums[(n/2)-1], nums[n/2])$$

换句话说，构成答案的数对必然是较小数取自有序序列的左边，较大数取自有序序列的右边，且与数组中心对称。

假设最大数对是 $(nums[i], nums[j])$，其中 $i < j$，记两者之和为 $ans = nums[i] + nums[j]$。

反证法证明，不存在别的数对组合会比 $(nums[i], nums[j])$ 更优：

假设存在数对 $(nums[p], nums[q])$ 与 $(nums[i], nums[j])$ 进行调整使答案更优。

接下来分情况讨论：

- 调整为 $(nums[i], nums[p])$ 和 $(nums[q], nums[j])$：此时最大数对答案为 $nums[q] + nums[j]$，显然 $nums[q] + nums[j] >= nums[i] + nums[j] = ans$。我们要最小化最大数对和，因此该调整方案不会让答案更好；
- 调整为 $(nums[i], nums[q])$ 和 $(nums[p], nums[j])$：此时最大数对答案为 $\max(nums[i] + nums[q], nums[p] + nums[j]) = nums[p] + nums[j] >= nums[i] + nums[j] = ans$。我们要最小化最大数对和，因此该调整方案不会让答案更好；

上述分析可以归纳推理到每一个“非对称”的数对配对中。

至此我们得证，将原本对称的数对调整为不对称的数对，不会使得答案更优，即贪心解可取得最优解。

贪心

对原数组 $nums$ 进行排序，然后从一头一尾开始往中间组「数对」，取所有数对中的最大值即是答案。

代码：

```
class Solution {
    public int minPairSum(int[] nums) {
        Arrays.sort(nums);
        int n = nums.length;
        int ans = nums[0] + nums[n - 1];
        for (int i = 0, j = n - 1; i < j; i++, j--) {
            ans = Math.max(ans, nums[i] + nums[j]);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(\log n)$

答疑

关于「证明」部分，不少小伙伴有一些疑问，觉得挺有代表性的，特意加到题解内。

Q1.「证明」部分是不是缺少了“非对称”得最优的情况？

A1. 并没有，证明的基本思路如下：

1. 猜想对称组数对的方式，会得到最优解；
2. 证明非对称数组不会被对称数对方式更优。

然后我们证明了“非对称方式”不会比“对称方式”更优，因此“对称方式”可以取得最优解。

至于非对称和非对称之间怎么调整，会更优还是更差，我不关心，也不需要证明，因为已经证明了非对称不会比对称更优。

Q2. 证明部分的图 p、q 是在 i、j 内部，那么其他 p、q、i、j 大小关系的情况呢？

A2. 有这个疑问，说明没有重点理解「证明」中的加粗部分（原话）：

> **上述分析可以归纳推理到每一个“非对称”的数对配对中。**

也就是说，上述的分析是可以推广到每一步都成立的，包括第一步，当 i 为排序数组的第一位原始，j 为排序数组中最后一位时，任意 p 和 q 都是在 i、j 内部的。

因此，「证明」对边界情况成立，同时对任意不成“对称”关系的数对也成立（其实也就是「证明」部分中的原话）。

更大白话一点是：对于任意“非对称”的数对组合，将其调整为“对称”数对组合，结果不会变差。

"给作者手机充个电"

YOLO 的赞赏码