# LeetCode26 从排序数组中删除重复项

给定一个**排序**数组，你需要在[**原地**](http://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95)删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在[**原地**](https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95)**修改输入数组**并在使用 O(1) 额外空间的条件下完成。

**示例 1:**

给定数组 *nums* = **[1,1,2]**,

函数应该返回新的长度 **2**, 并且原数组 *nums* 的前两个元素被修改为 **1**, **2**。

你不需要考虑数组中超出新长度后面的元素。

**示例 2:**

给定 *nums* = **[0,0,1,1,1,2,2,3,3,4]**,

函数应该返回新的长度 **5**, 并且原数组 *nums* 的前五个元素被修改为 **0**, **1**, **2**, **3**, **4**。

你不需要考虑数组中超出新长度后面的元素。

**说明:**

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**“引用”**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

// **nums** 是以“引用”方式传递的。也就是说，不对实参做任何拷贝

int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。

// 根据你的函数返回的长度, 它会打印出数组中**该长度范围内**的所有元素。

for (int i = 0; i < len; i++) {

    print(nums[i]);

}

**分析：**

分析：

我们知道对于数组来说，在尾部插入、删除元素是比较高效的，时间复杂度是 O(1)，但是如果在中间或者开头插入、删除元素，就会涉及数据的搬移，时间复杂度为 O(N)，效率较低。

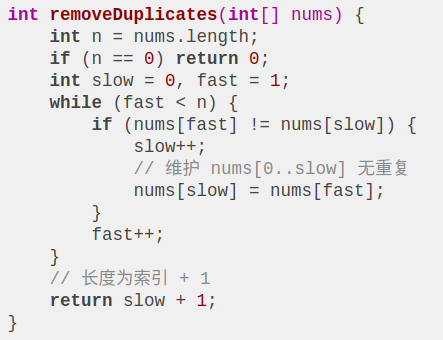
所以对于一般处理数组的算法问题，我们要尽可能只对数组尾部的元素进行操作，以避免额外的时间复杂度。

由于数组已经排序，所以重复的元素一定连在一起，找出它们并不难，但如果毎找到一个重复元素就立即删除它，就是在数组中间进行删除操作，整个时间复杂度是会达到 O(N^2)。而且题目要求我们原地修改，也就是说不能用辅助数组，空间复杂度得是 O(1)。

其实，对于数组相关的算法问题，有一个通用的技巧：要尽量避免在中间删除元素，并尽量将待删除元素换到数组尾部。这样待删除的元素都拖在数组尾部，可以被间复杂度为O(1) 的操作删除。

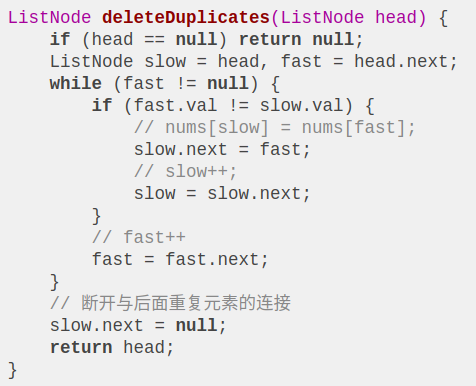
基于该思路，引申出“双指针法”，即：

使用快慢指针，慢指针（slow）走在后面，快指针（fast）走在前面探路，快指针找到一个不重复的元素就告诉slow并让slow前进一步。这样当fast指针遍历完整个数组nums后，nums[0..slow]就是不重复元素，之后的所有元素都是重复元素，返回\*\*slow+1\*\*即为题目所需。



看下算法执行的过程（见当前目录gif文件）：

再简单扩展一下，如果给你一个有序链表，如何去重呢？其实和数组是一模一样的，唯一的区别是把数组赋值操作变成操作指针而已：



对于链表去重，算法执行的过程是这样的（见当前目录gif文件）。

# LeetCode121 买卖股票的最佳时机

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

输入: [7,1,5,3,6,4]

输出: 5

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。

示例 2:

输入: [7,6,4,3,1]

输出: 0

解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

分析：

1. 如果数组长度小于等于1，则利润为0；
2. 先保存当前的最小值，然后从第二个元素开始遍历，与最小值相比，是否产生更大的利润，有则更新利润的最大值，否则，最大利润保持不变。然后更新当前的最小值，
3. 循环遍历下一个元素。
4. 返回最大利润值

# LeetCode122 买卖股票的最佳时机II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

**注意：**你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

**示例 1:**

**输入:** [7,1,5,3,6,4]

**输出:** 7

**解释:** 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。

  随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。

**示例 2:**

**输入:** [1,2,3,4,5]

**输出:** 4

**解释:** 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。

  注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。

  因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

**示例 3:**

**输入:** [7,6,4,3,1]

**输出:** 0

**解释:** 在这种情况下, 没有交易完成, 所以最大利润为 0。

**分析：**

**方法1：暴力法**

计算所有可能的组合与其对应的利润，并找出它们中的最大利润。使用分治结合递归。先计算其中部分的天数交易股票的最大利润，再计算剩余天数交易股票的最大利润。

复杂度分析

时间复杂度：O(nn)，调用递归函数nn次。

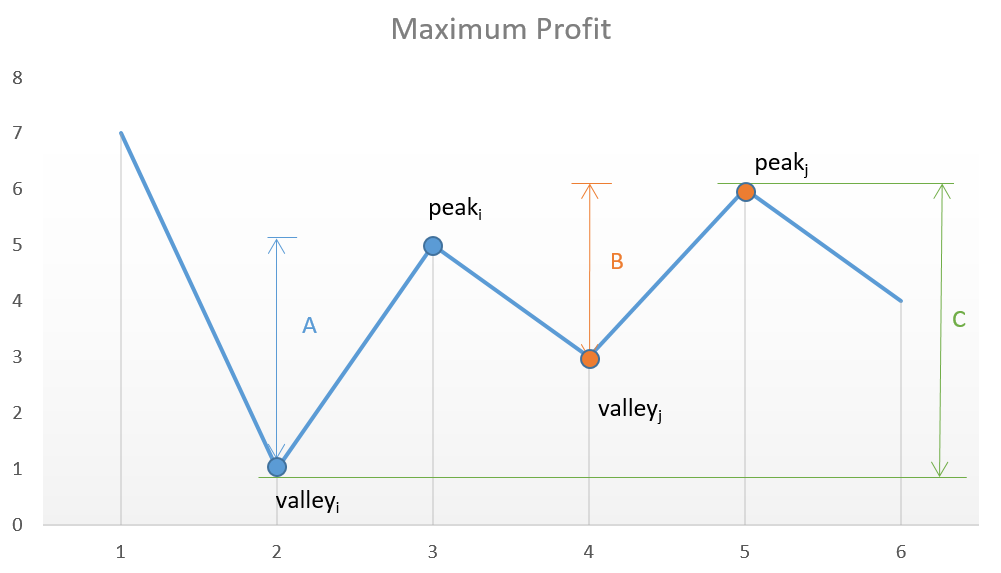
空间复杂度：O(n)，递归的深度为 n。

**方法2：峰谷法**

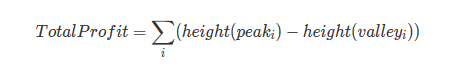
假设给定的数组为：

[7, 1, 5, 3, 6, 4]

如果我们在图表上绘制给定数组中的数字，我们将会得到：



关键点在于连续的峰和谷，总利润为：

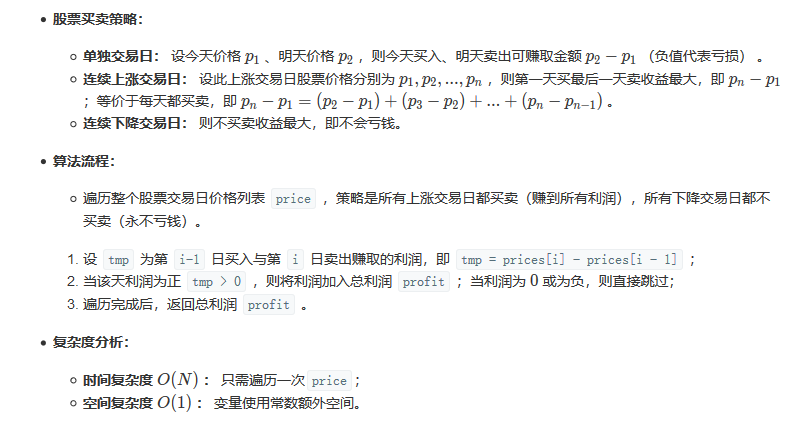


需要考虑到紧跟谷的每一个峰值以最大化利润，试图跳过其中一个峰谷都会导致总利润的降低。如，若我们跳过 *peaki* 和 *valleyj* 试图通过考虑差异较大的点以获取更多的利润，但会适得其反，使得获取的净利润总是会小于包含它们而获得的净利润，因为C<A+B。

复杂度分析：

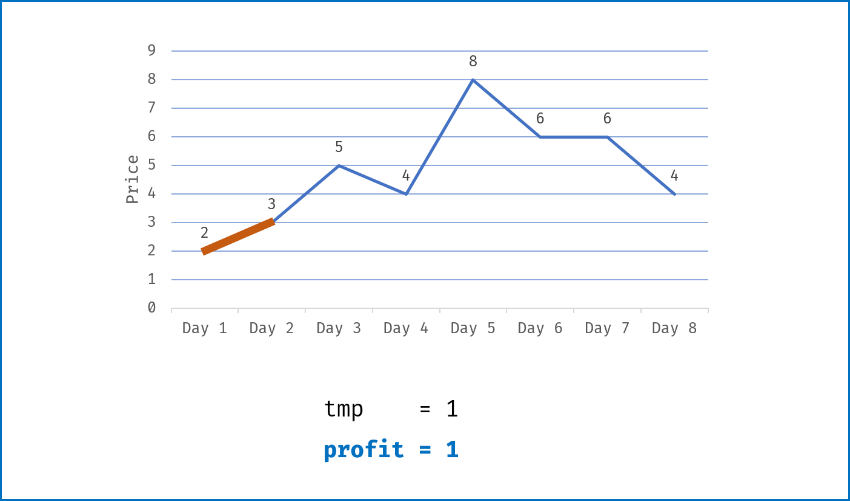
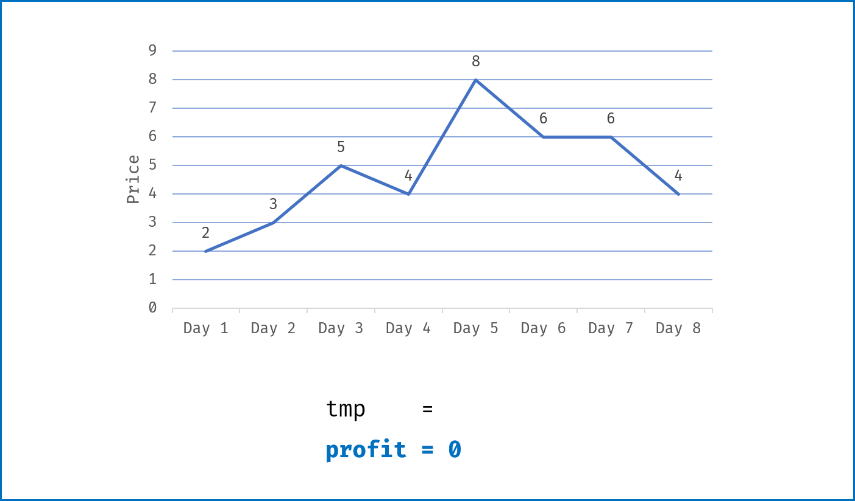
* 时间复杂度：O(n)。遍历一次。
* 空间复杂度：O(1)。需要常量的空间。

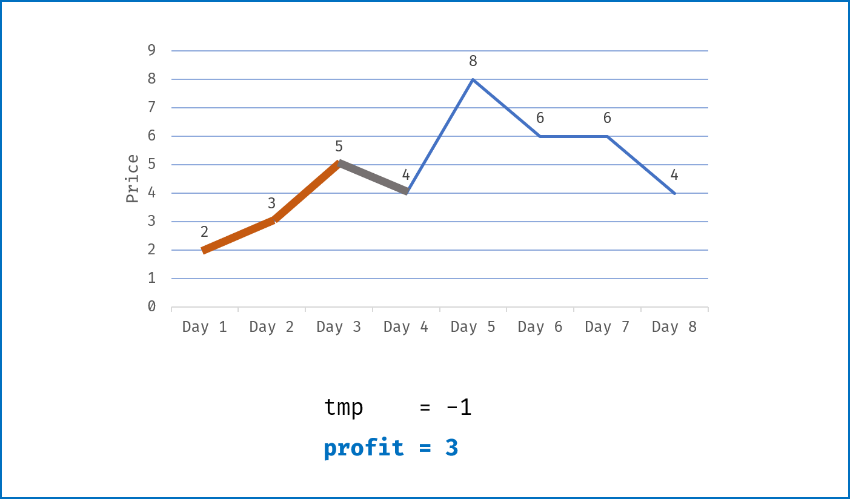
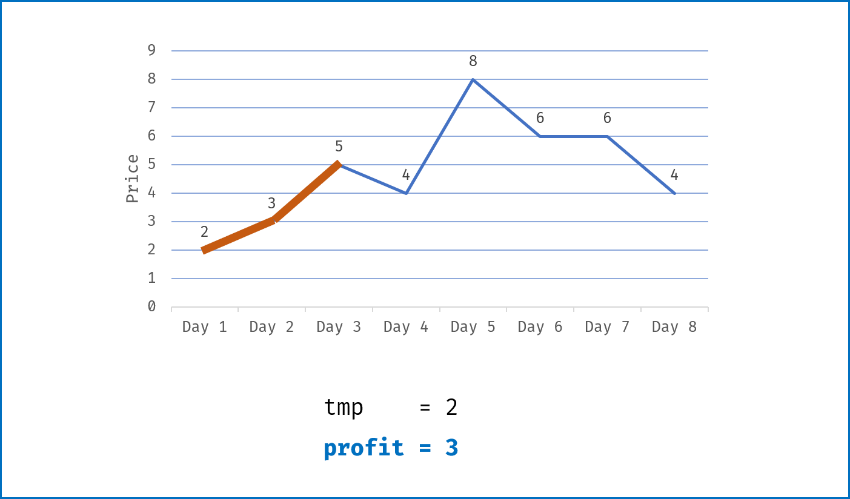
**方法3：贪心法**

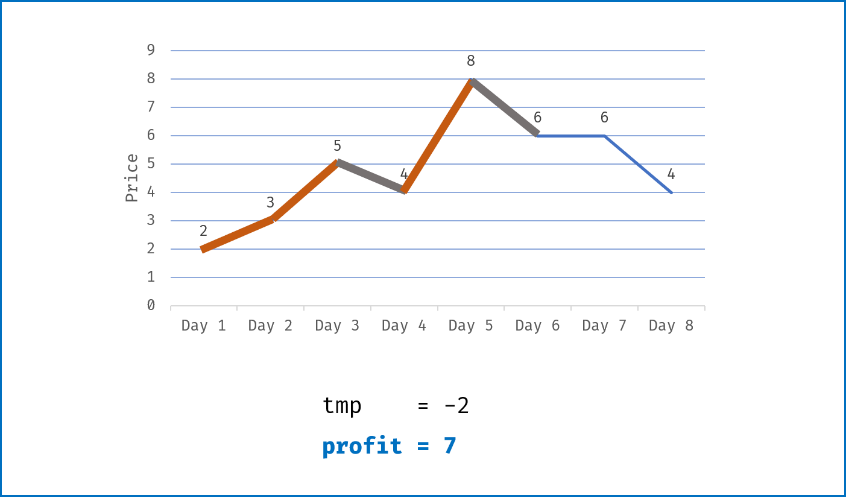
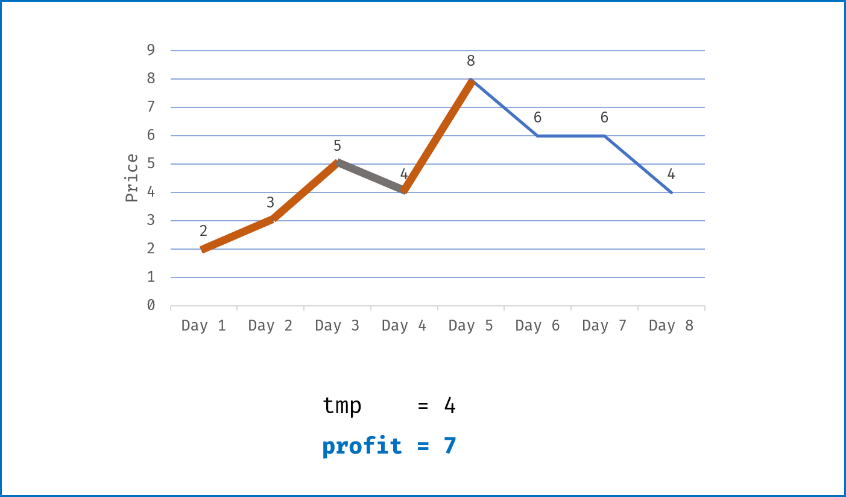


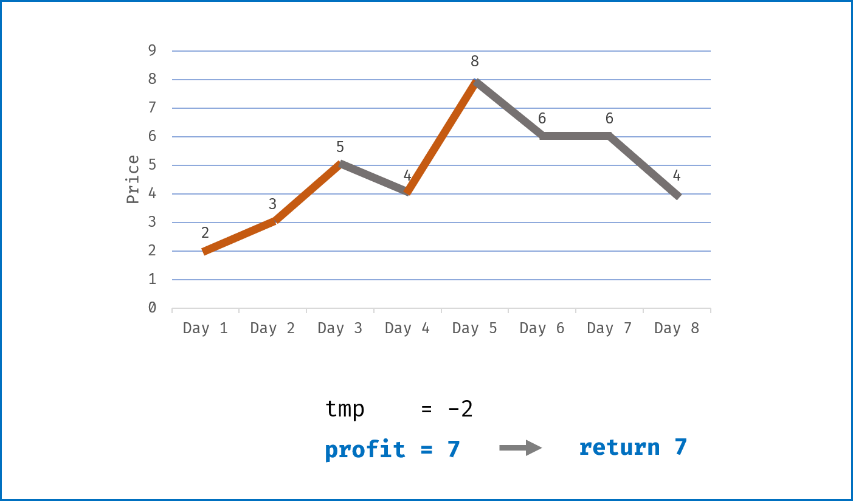
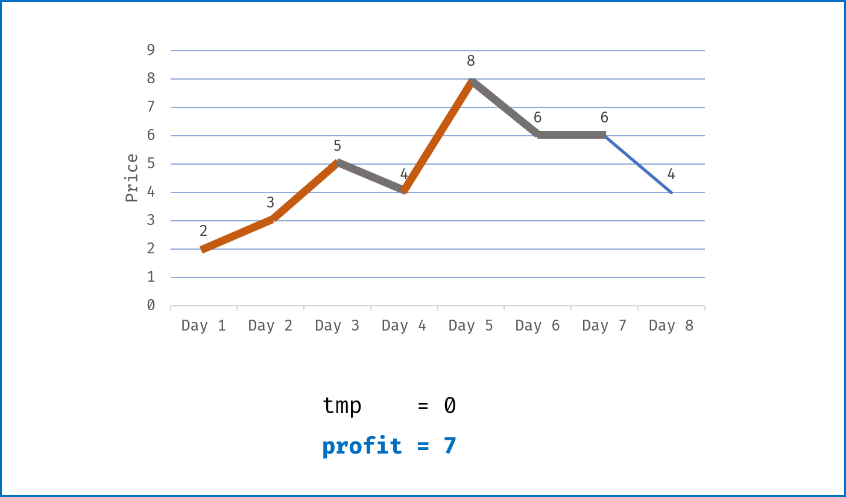
复杂度分析：

* 时间复杂度：O(n)。遍历一次。
* 空间复杂度：O(1)。需要常量的空间。









# LeetCode189 旋转数组

给定一个数组，将数组中的元素向右移动 *k*个位置，其中 *k*是非负数。

**示例 1:**

**输入:** [1,2,3,4,5,6,7] 和 *k* = 3

**输出:** [5,6,7,1,2,3,4]

**解释:**

向右旋转 1 步: [7,1,2,3,4,5,6]

向右旋转 2 步: [6,7,1,2,3,4,5]

向右旋转 3 步: [5,6,7,1,2,3,4]

**示例 2:**

**输入:** [-1,-100,3,99] 和 *k* = 2

**输出:** [3,99,-1,-100]

**解释:**

向右旋转 1 步: [99,-1,-100,3]

向右旋转 2 步: [3,99,-1,-100]

**说明:**

* 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
* 要求使用空间复杂度为 O(1) 的 **原地**算法。

**分析：**

**方法一：暴力**

最简单的方法是旋转 k 次，每次将数组旋转 1 个元素。

复杂度分析：

时间复杂度：O(n∗k) 。每个元素都被移动 1 步（O(n)） k次（O(k)） 。

空间复杂度：O(1) 。没有额外空间被使用。

**方法二：使用额外的数组**

我们可以用一个额外的数组来将每个元素放到正确的位置上，也就是原本数组里下标为 i 的我们把它放到 (i+k)%数组长度 的位置。然后把新的数组拷贝到原数组中。

复杂度分析：

时间复杂度：O(n) 。将数字放到新的数组中需要一遍遍历，另一边来把新数组的元素拷贝回原数组。

空间复杂度：O(n) 。另一个数组需要原数组长度的空间。

**方法三：使用反转**

这个方法基于这个事实：当我们旋转数组 k 次， k%n 个尾部元素会被移动到头部，剩下的元素会被向后移动。

在这个方法中，我们首先将所有元素反转。然后反转前 k 个元素，再反转后面 n−k个元素，就能得到想要的结果。

假设 n=7且 k=3 。

原始数组 : 1 2 3 4 5 6 7

反转所有数字后 : 7 6 5 4 3 2 1

反转前 k 个数字后 : 5 6 7 4 3 2 1

反转后 n-k 个数字后 : 5 6 7 1 2 3 4 --> 结果

复杂度分析

时间复杂度：O(n) 。 n 个元素被反转了总共 3 次。

空间复杂度：O(1) 。 没有使用额外的空间。

# LeetCode344 字符串反转

**分析**：双指针来解决。 i指向数组的第一个元素，j指向数组的最后一个元素。每次交换i和j的两个位置的元素，然后将i向左移动一步，j向右移动一步即可。

交换元素可使用三次异或操作交换两个值，此种操作可降低内存消耗。

如：交换a和b的值。

a ^= b;

b ^= a;

a ^= b;

# LeetCode217 存在重复元素

给定一个整数数组，判断是否存在重复元素。

如果任何值在数组中出现至少两次，函数返回 true。如果数组中每个元素都不相同，则返回 false。

**示例 1:**

**输入:** [1,2,3,1]

**输出:** true

**示例 2:**

**输入:** [1,2,3,4]

**输出:** false

**示例 3:**

**输入:** [1,1,1,3,3,4,3,2,4,2]

**输出:** true

**分析**：

**方法一：朴素线性查找（暴力解法）——超时**

对于一个有 n个整数的数组，一共有 C(n,2)=n(n-1)/2对整数。因此，我们可以对所有的 n(n+1)/2 对进行检测，看它们是否相同。

算法

为了实现这个思路，我们使用线性查找算法，这是最简单的查找算法。线性查找是一种检查特定值是否在列表中的算法，做法是依次逐个检查列表中的元素，直到找到满足的元素。

对于本问题，我们循环遍历全部 n 个数。对于第 i个整数 nums[i]，我们对前 i-1 个整数查找 nums[i] 的重复值。若找到，则返回 True; 否则继续。在程序最后，返回 False。

为了证明算法的正确性，我们定义了循环不变式。循环不变式是指在每次迭代前和后均保持不变的性质。了解循环不变式对理解循环的意义十分重要。下面就是循环不变式:

在下一次搜索之前,搜索过的整数中没有重复的整数。

循环不变式在循环之前为真，因为还没有搜索过的整数。每次循环，我们查找当前元素的任何可能重复。如果发现重复项,则函数返回 True 退出；如果没有发现，则不变式仍然成立。

因此，如果循环结束，**循环不变式**说明全部 n 个整数中不存在重复元素。

复杂度分析

时间复杂度 : O(n^2)。最坏的情况下，需要检查 n(n+1)/2​ 对整数。因此，时间复杂度为 O(n^2)。

空间复杂度 : O(1)。只使用了常数额外空间。

注意

本方法在 LeetCode 上会超时。一般而言，如果一个算法的时间复杂度为 O(n^2)，它最多能处理 n 大约为 10^4 的数据。当 n 接近 **10^5** 时就会超时。

**方法二：排序（暴力解法）——超过**

直觉

如果存在重复元素，排序后它们应该相邻。

算法

本方法使用排序算法。由于比较排序算法，如**堆排序**，可以在最坏情况下具有 O(nlogn) 的时间复杂度。因此，排序经常是很好的预处理方法。排序之后，我们可以扫描已排序的数组,以查找是否有任何连续的重复元素。

复杂度分析

时间复杂度 : O(nlogn)。

排序的复杂度是 O(nlogn)，扫描的复杂度是 O(n)O(n)O(n)。整个算法主要由排序过程决定，因此是 O(nlogn)。

空间复杂度 : O(1)。

这取决于具体的排序算法实现，通常而言，使用 堆排序 的话，是 O(1)。

注意

**此处的算法实现对原始数组进行排序，修改了原始数组。通常，除非调用方清楚输入数据将被修改，否则不应该随意修改输入数据。可以先复制 nums，然后对副本进行操作。**

**方法三：哈希表——通过**

直觉

利用支持快速搜索和插入操作的动态数据结构。

算法

从方法一中我们知道，对无序数组的查找操作的时间复杂度为 O(n)，而我们会重复调用查找操作。因此，使用搜索时间更快的数据结构将加快整个算法的速度。

有许多数据结构常用作动态集合,如二进制搜索树和哈希表。这里我们需要的操作是 search 和 insert。对于平衡二叉搜索树（Java 中的 TreeSet 或 TreeMap），search 和 insert 的时间复杂度均为 O(logn)。对于哈希表（Java 中的 HashSet 或 HashMap），search 和 insert 的平均时间复杂度为 O(1)。因此，通过使用哈希表，我们可以达到在线性时间复杂度解决问题。

复杂度分析

时间复杂度 : O(n)。

search() 和 insert() 各自使用 n 次，每个操作耗费常数时间。

空间复杂度 : O(n)。哈希表占用的空间与元素数量是线性关系。

注意

对于一些特定的 n 不太大的测试样例，本方法的运行速度可能会比方法二更慢。这是因为哈希表在维护其属性时有一些开销。要注意，程序的实际运行表现和 Big-O 符号表示可能有所不同。Big-O 只是告诉我们在 充分 大的输入下，算法的相对快慢。因此，在 n 不够大的情况下， O(n) 的算法也可以比 O(nlogn)的更慢。