# 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。

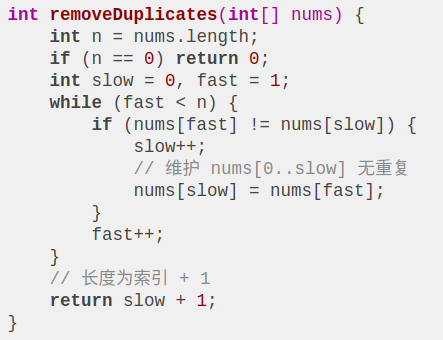
所以对于一般处理数组的算法问题，我们要尽可能只对数组尾部的元素进行操作，以避免额外的时间复杂度。

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。

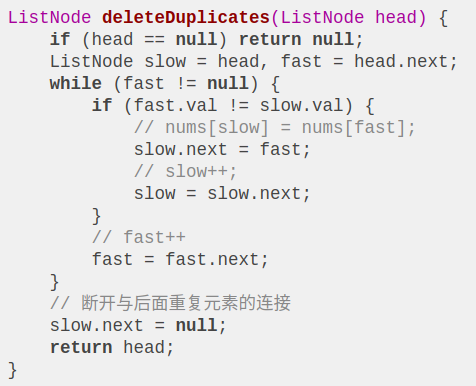
基于该思路，引申出“双指针法”，即：

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。

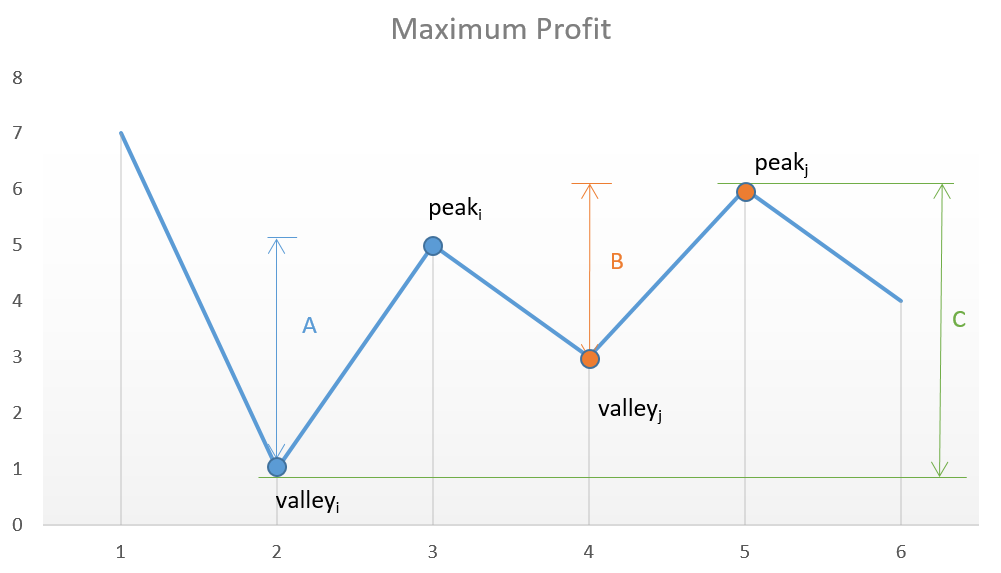
空间复杂度：O(n)，递归的深度为 n。

**方法2：峰谷法**

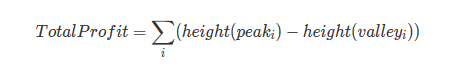
假设给定的数组为：

[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：

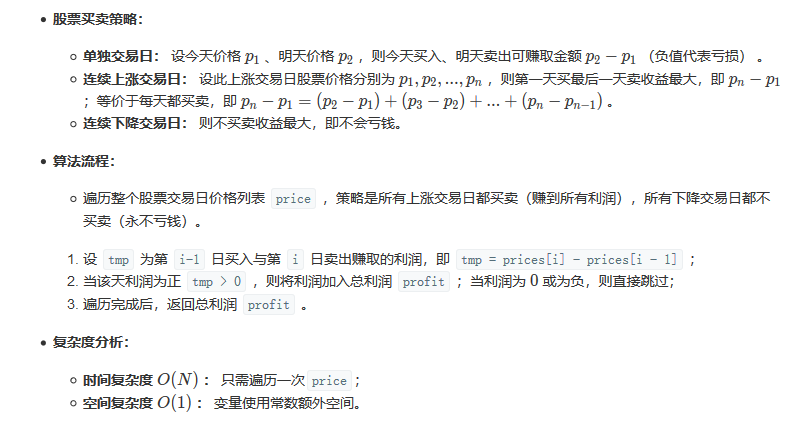


需要考虑到紧跟谷的每一个峰值以最大化利润，试图跳过其中一个峰谷都会导致总利润的降低。如，若我们跳过 *peaki* 和 *valleyj* 试图通过考虑差异较大的点以获取更多的利润，但会适得其反，使得获取的净利润总是会小于包含它们而获得的净利润，因为C<A+B。

复杂度分析：

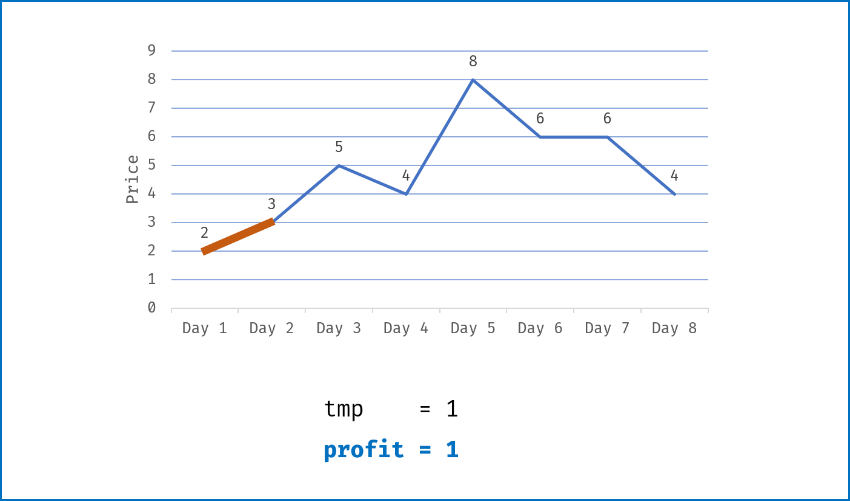
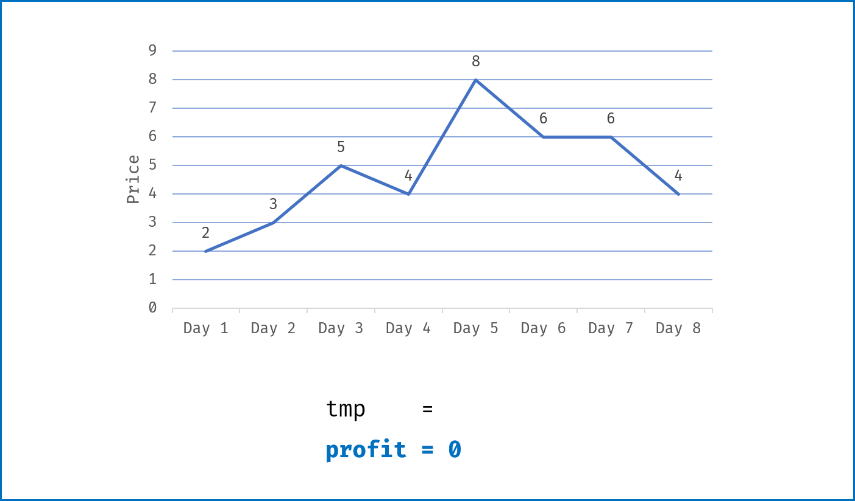
* 时间复杂度：O(n)。遍历一次。
* 空间复杂度：O(1)。需要常量的空间。

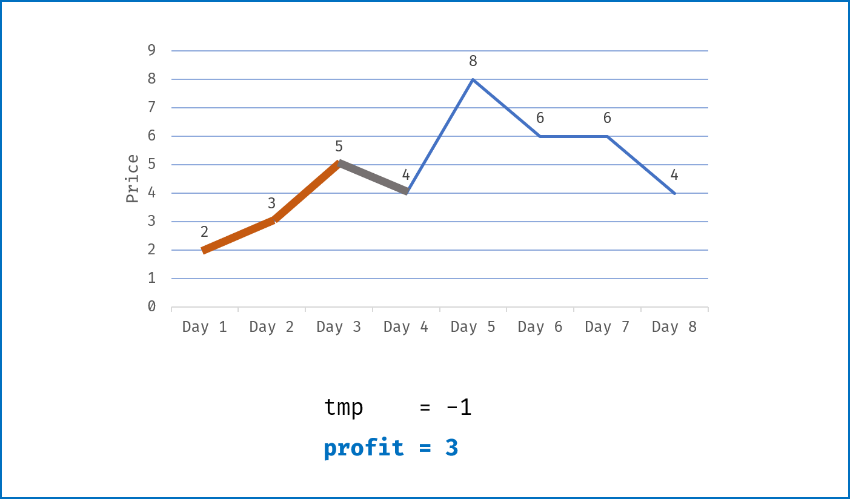
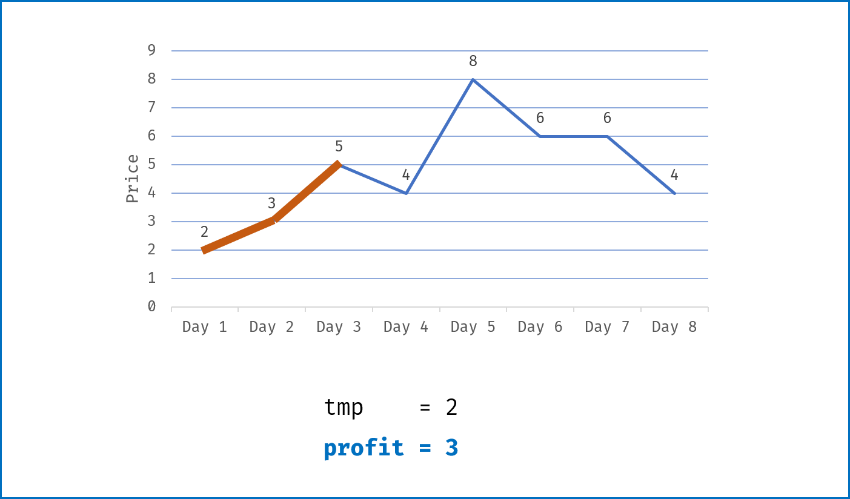
**方法3：贪心法**

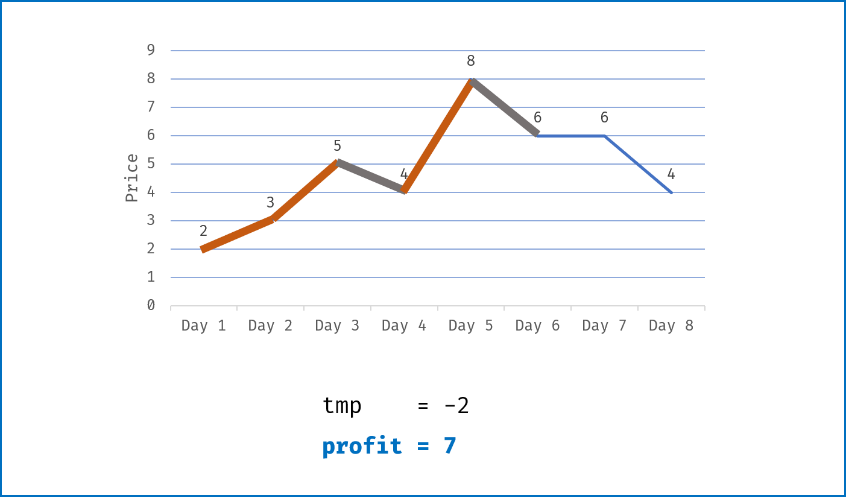
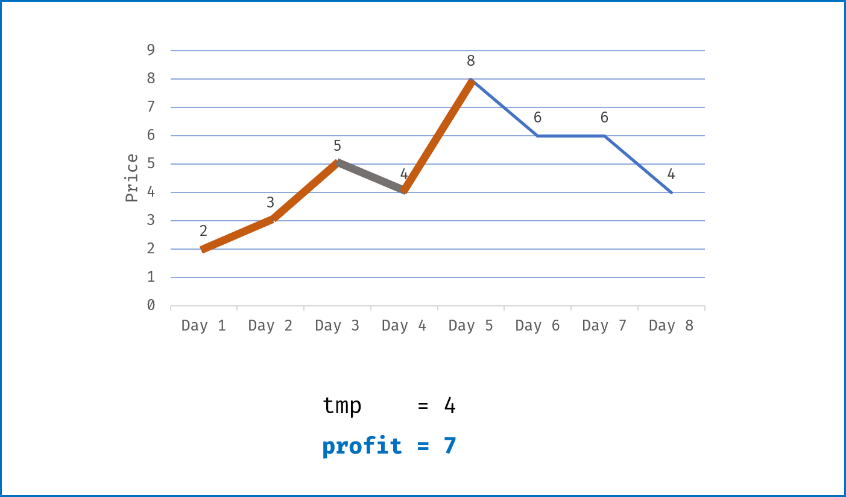


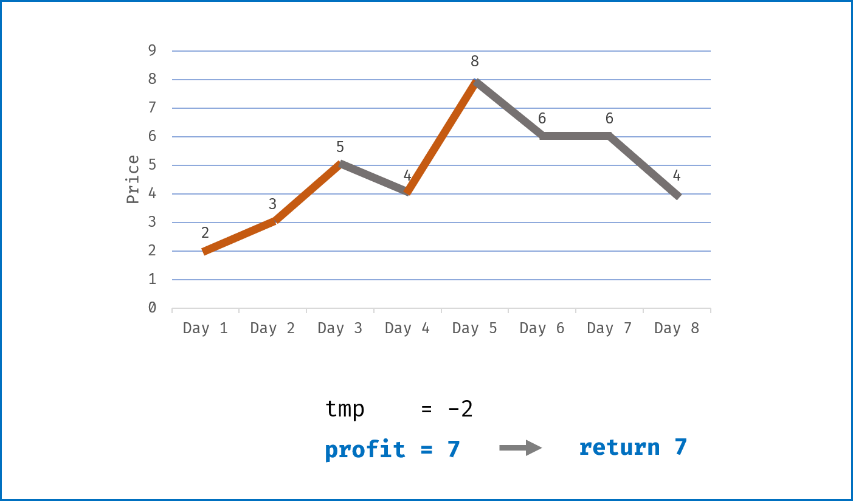
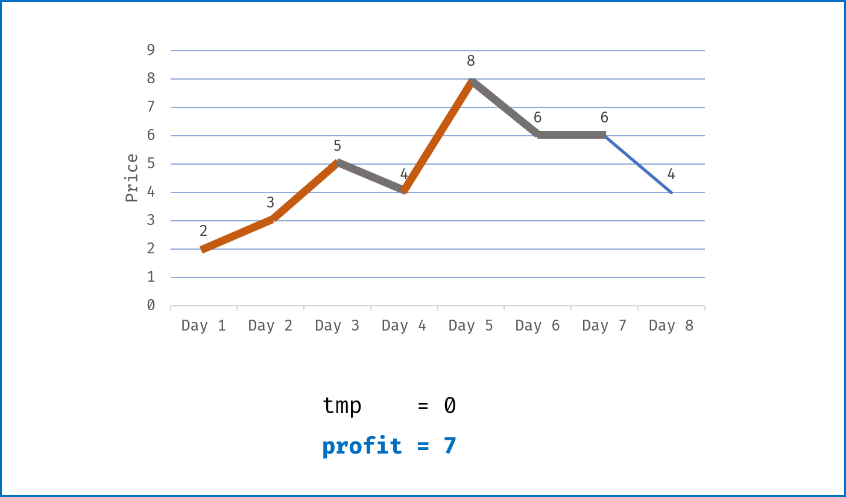
复杂度分析：

* 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。

# LeetCode1 两数之和

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9

所以返回 [0, 1]

**分析：**

**方法一：暴力法**

暴力法很简单，遍历每个元素 xxx，并查找是否存在一个值与 target−x相等的目标元素。

复杂度分析：

时间复杂度： O(n^2)

对于每个元素，我们试图通过遍历数组的其余部分来寻找它所对应的目标元素，这将耗费 O(n)的时间。因此时间复杂度为O(n^2)。

空间复杂度：O(1)

**方法二：两遍Hash表**

为了对运行时间复杂度进行优化，我们需要一种更有效的方法来检查数组中是否存在目标元素。如果存在，我们需要找出它的索引。保持数组中的每个元素与其索引相互对应的最好方法是什么？哈希表。

通过以空间换取速度的方式，我们可以将查找时间从 O(n) 降低到 O(1)。哈希表正是为此目的而构建的，它支持以 近似 恒定的时间进行快速查找。我用“近似”来描述，是因为一旦出现冲突，查找用时可能会退化到 O(n)。但只要你仔细地挑选哈希函数，在哈希表中进行查找的用时应当被摊销为 O(1)。

一个简单的实现使用了两次迭代。在第一次迭代中，我们将每个元素的值和它的索引添加到表中。然后，在第二次迭代中，我们将检查每个元素所对应的目标元素（target−nums[i]）是否存在于表中。注意，该目标元素不能是 nums[i] 本身！

复杂度分析：

时间复杂度：O(n)，

我们把包含有 n 个元素的列表遍历两次。由于哈希表将查找时间缩短到 O(1) ，所以时间复杂度为 O(n)。

空间复杂度：O(n)，

所需的额外空间取决于哈希表中存储的元素数量，该表中存储了 n 个元素。

**方法三：一遍Hash表**

事实证明，我们可以一次完成。在进行迭代并将元素插入到表中的同时，我们还会回过头来检查表中是否已经存在当前元素所对应的目标元素。如果它存在，那我们已经找到了对应解，并立即将其返回

复杂度分析：

时间复杂度：O(n)，

我们只遍历了包含有 n 个元素的列表一次。在表中进行的每次查找只花费 O(1) 的时间。

空间复杂度：O(n)，

所需的额外空间取决于哈希表中存储的元素数量，该表最多需要存储 n 个元素。

# LeetCode136 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

输入: [2,2,1]

输出: 1

示例 2:

输入: [4,1,2,1,2]

输出: 4

**分析**

**方法一：**异或操作。（相同的数字经过异或运算后结果为0；任何数字与0进行异或运算都是该数字本身）；O(n)，O(1)

**方法二：**利用HashSet去重 ；HashSet的add方法是通过HashMap来实现的，HashSet中的元素都是HashMap中的key，HashMap的put方法调用的 resize() 、 putTreeVal() 等方法本身也是O(n2)的时间复杂度。因此总的时间复杂度为O(n2)，空间复杂度为O(n)

**方法三：**先排序后比较；O(n2)，O(1)

# LeetCode349 两个数组的交集

给定两个数组，编写一个函数来计算它们的交集。

示例 1:

输入: nums1 = [1,2,2,1], nums2 = [2,2]

输出: [2]

示例 2:

输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

输出: [9,4]

说明:

输出结果中的每个元素一定是唯一的。

我们可以不考虑输出结果的顺序。

**分析：**

**方法一：暴力解法。遍历两个数组，将相同的元素存入Set去除重复。时间复杂度O(m\*n)，空间复杂度O(n)**

**方法二：两个Set法。先将两个数组存入Set中，然后迭代较小的集合是否存在较大集合中。集合中查找的时间复杂度为O(1)。则平均情况下时间复杂度为O(m+n)（O(m)时间用于转换数组1到集合中，O(n)时间用于转换数组2到集合中），空间复杂度为O(m+n)，最坏的情况是数组中的所有元素都不同。**

**方法三：利用Java自带的retainAll()函数。时间复杂度在一般情况下是O(m+n)，最坏情况下式O(m\*n)。空间复杂度为O(m+n)，当数组中元素全不同时。**

**boolean retainAll(Collection<?> c)**

**list1.retainAll(list2);**

**保留同时在list1中同时出现在list1和list2中的元素（交集）。若list1与list2完全一样，返回false；若list1包含于list2，返回false；其余情况，返回true。**

# LeetCode350 两个数组的交集II

方法一：使用哈希表。使用哈希表存储较大数组中出现的数与数出现的频次。之后遍历较小的数组，若当前索引的存在于哈希表的键中，则将其存入数组中，并将哈希表中的频次减一，若频次为0则移除它。时间复杂度O(m+n)，空间复杂度O(m+n)。

方法二：排序后使用双指针。时间复杂度O(nlogn)（排序算法），额外in空间复杂度O(1)。

首先对两个数组进行排序，然后就是双指针，p1指向nums1，p2指向nums2：

1. 如果nums1[p1] == nums2[p2]，说明俩数组中都有这个数，是其交集，所以将它丢入list中。
2. 如果不等，则移动小的那个指针。

# LeetCode66 加一

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1:

输入: [1,2,3]

输出: [1,2,4]

解释: 输入数组表示数字 123。

示例 2:

输入: [4,3,2,1]

输出: [4,3,2,2]

解释: 输入数组表示数字 4321。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/plus-one

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**分析：**

一位十进制数加1只会有两种情况：

* + 除 9之外的数字，直接加1即可
  + 数字 9，加1后进位1（仅会出现进位1）。
* 若某一数字，做加1运算后不出现进位，则运算结束，可直接返回结果；
* 若某一数字，做加1运算后出现进位，则当前位 **置0**，且进位1，循环判断直至没有进位就退出运算，返回结果；
* 若出现特殊数字如9，99，999等，循环结束后仍需进位1（此时数字长度增加一位），最高位为1，其余位均为0。再返回结果。

**代码：**

class Solution {

public int[] plusOne(int[] digits) {

for (int i = digits.length - 1; i >= 0; i--) {

digits[i]++;

// 根据模10后是否为0判断是否出现进位。

// 若出现进位，则利用循环下一位加1；未出现进位，结束运算直接返回结果

digits[i] = digits[i] % 10;

// 若不出现进位，运算结束，直接返回结果

if (digits[i] != 0) return digits;

}

// 处理特殊数字（9，99，999等），数字长度加1，最高位 **置1**，其余位均为0。

digits = new int[digits.length + 1];

digits[0] = 1;

return digits;

}

}

# LeetCode283 移动零

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

输入: [0,1,0,3,12]

输出: [1,3,12,0,0]

说明:

必须在原数组上操作，不能拷贝额外的数组。

尽量减少操作次数。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/move-zeroes

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**分析：**

**方法一：使用额外的数组，然后进行拷贝。时间复杂度O(n)，空间复杂度O(n)**

**方法二：使用冒泡，将等于0的元素上浮。时间复杂度O(n2)，空间复杂度O(1)**

**方法三：使用双指针。使用快慢指针，使用慢指针指向第一个为0的元素，使用快指针指向第一个不为0的元素。之后，交换快慢指针所指向的元素，慢指针指向第二个为0的元素，快指针指向第二个不为0的元素，重复上述操作直到快指针无法指向不为0的元素（所有0已移动到数组末尾）**

class Solution {

public void moveZeroes(int[] nums) {

int len = nums.length;

int slow = 0;

int fast = 1;

while(fast < len){

if (nums[slow] != 0){

slow++;

fast++;

}

else{

if (nums[fast] == 0)

fast++;

else{

nums[slow] = nums[fast];

nums[fast] = 0;

slow++;

fast++;

}

}

}

}

}

# LeetCode36有效的数独

判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

数字 1-9 在每一行只能出现一次。

数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。



上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用 '.' 表示。

示例 1:

输入:

[

["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

输出: true

示例 2:

输入:

[

["8","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

输出: false

解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。

但由于位于左上角的 3x3 宫内有两个 8 存在, 因此这个数独是无效的。

说明:

一个有效的数独（部分已被填充）不一定是可解的。

只需要根据以上规则，验证已经填入的数字是否有效即可。

给定数独序列只包含数字 1-9 和字符 '.' 。

给定数独永远是 9x9 形式的。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/valid-sudoku

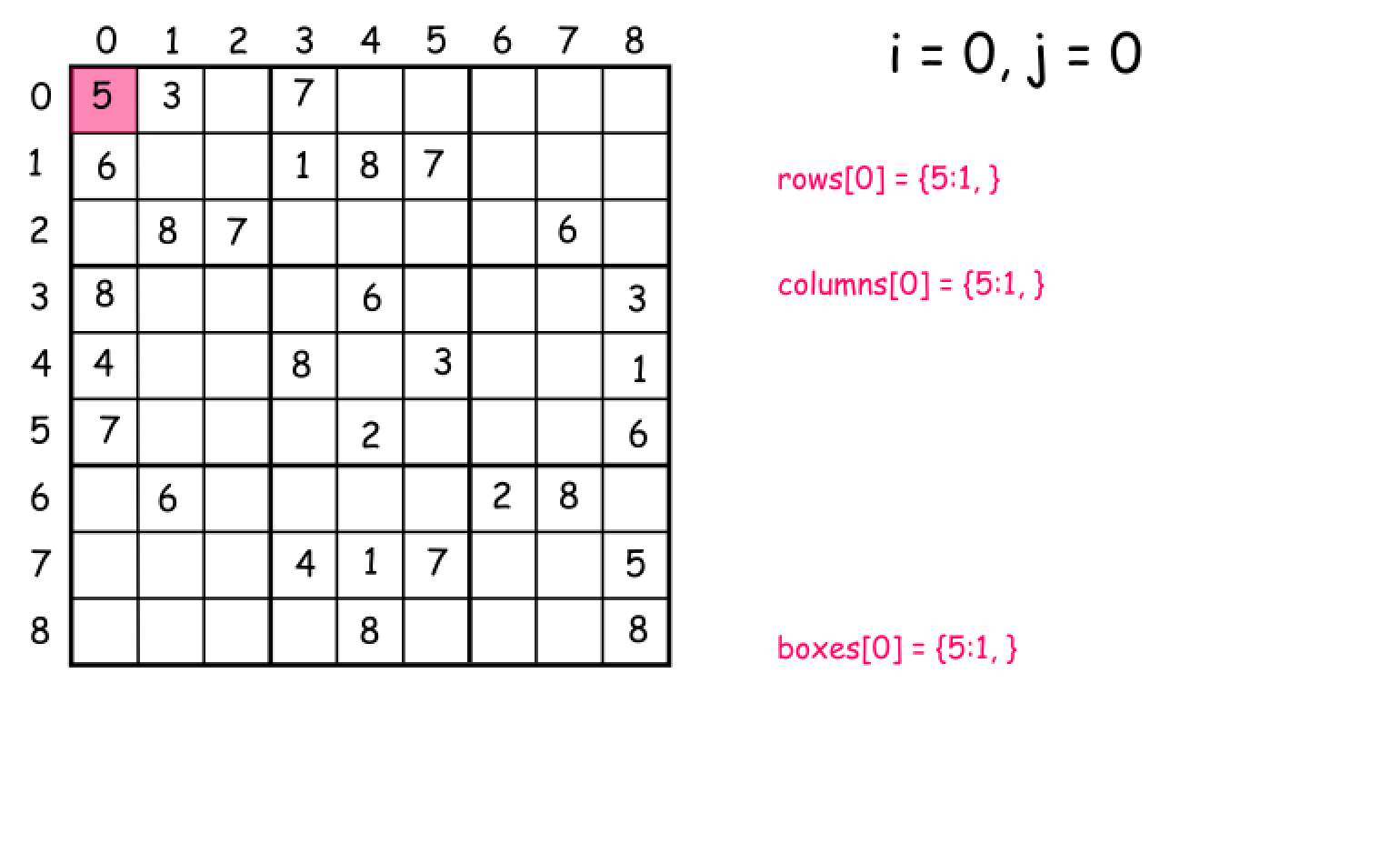
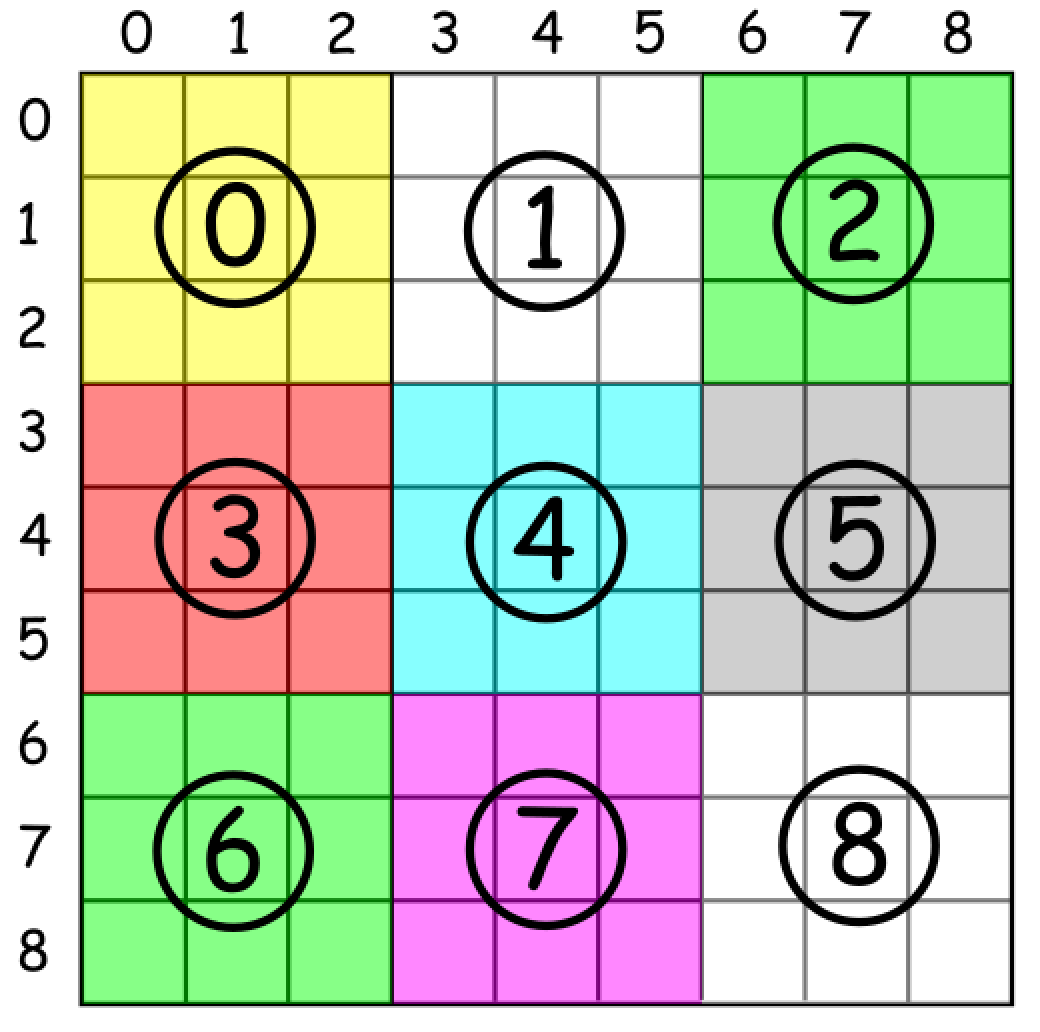
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**分析：使用键值对记录数字出现的次数，键为1~9，值为出现的次数。**

方法1：三次遍历，分别去判断行、列、3\*3子数独中没有重复的数字。

方法2：一次遍历，同时判断行、列、3\*3子数独中没有重复的数字。

给子数独编号，box\_index = (row / 3) \* 3 + columns / 3，



# LeetCode48 旋转图像

给定一个 n × n 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

说明：

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

示例 1:

给定 matrix =

[

[1,2,3],

[4,5,6],

[7,8,9]

],

原地旋转输入矩阵，使其变为:

[

[7,4,1],

[8,5,2],

[9,6,3]

]

示例 2:

给定 matrix =

[

[ 5, 1, 9,11],

[ 2, 4, 8,10],

[13, 3, 6, 7],

[15,14,12,16]

],

原地旋转输入矩阵，使其变为:

[

[15,13, 2, 5],

[14, 3, 4, 1],

[12, 6, 8, 9],

[16, 7,10,11]

]

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/rotate-image

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**分析：**

使用辅助矩阵的方案。时间复杂度为O(n2),空间复杂度为O(n2).

newMatrix[i][j] = matrix[n - 1 - j][i];

不使用辅助矩阵的方案：

方法一：转置加翻转。先转置（b[j][i] = a[i][j]），再水平翻转每一行再水平翻转（b[i][j] = a[i][n - 1 -j]）。时间复杂度为O(n2)，空间复杂度为O(1)。

方法二：自外向内顺时针循环。自外向内一共有不超过 n/2 层（单个中心元素不算一层）矩形框。对于第 times 层矩形框，其框边长 len=nums−(times∗2)，将其顺时针分为 4 份 len−1 的边，对四条边进行元素的循环交换即可。

1 2 3 4 5 6

7 8 9 10 11 12

13 14 15 16 17 18

19 20 21 22 23 24

25 26 27 28 29 30

31 32 33 34 35 36

总共有3层，第一层有4个边长为6的边（如1-6）；第二层有4个边长为4的边（如8-11）；第三次有4个边长为2的边（如15-16）。

public void rotate(int[][] matrix) {

        if (matrix.length == 0 || matrix.length != matrix[0].length)

            return;

        int n = matrix.length;

        int times = 0;

        while (times <= n/2){

            int len = n - times \* 2;

            for (int i = 0; i < len - 1; i++){

                int tmp = matrix[times][times + i];

                matrix[times][times + i] = matrix[times + len - 1 - i][times];

                matrix[times + len - 1 - i][times] = matrix[times + len - 1][times + len - 1 -i];

                matrix[times + len - 1][times + len - 1 - i] = matrix[times + i][times + len - 1];

                matrix[times + i][times + len - 1] = tmp;

            }

            times++;

        }

    }

# LeetCode206 反转链表

反转一个单链表。

示例:

输入: 1->2->3->4->5->NULL

输出: 5->4->3->2->1->NULL

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题？

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/reverse-linked-list

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**分析：**

Node定义如下：

public static class Node {

public int value;

public Node next;

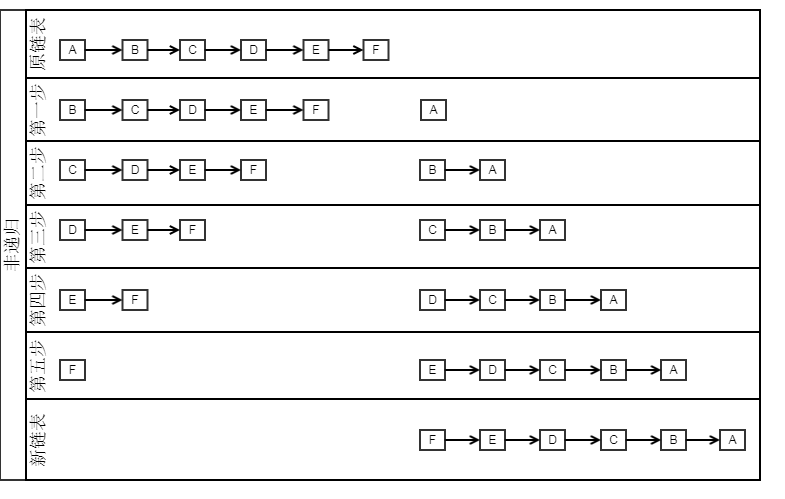
public Node(int data) {

this.value = data;

}

}

方法一：遍历法（链表遍历的过程中将指针顺序置换）。p指向前一个结点，q指向当前结点，temp临时保存q的下一个结点。时间复杂度O(N)，空间复杂度O(1)



public static Node reverseList(Node node) {

Node pre = null;

Node next = null;

while (node != null) {

next = node.next;

node.next = pre;

pre = node;

node = next;

}

return pre;

}

假设链表为1->2->3->4,则程序执行过程如下：

 准备两个空结点 pre用来保存先前结点、next用来做临时变量

 在头结点node遍历的时候此时为1结点

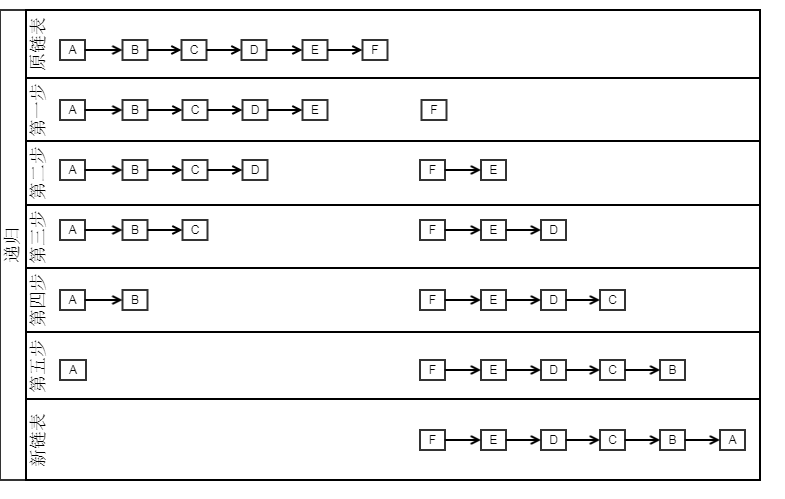
* next = 1结点.next(2结点)
* 1结点.next=pre(null)
* pre = 1结点
* node = 2结点

 进行下一次循环node=2结点

* next = 2结点.next(3结点)
* 2结点.next=pre(1结点)=>即完成2->1
* pre = 2结点
* node = 3结点

 进行循环...

方法二：递归实现。时间复杂度O(N)，空间复杂度O(N)



public Node reverse(Node head) {

if (head == null || head.next == null)

return head;

Node temp = head.next;

Node newHead = reverse(head.next);

temp.next = head;

head.next = null;

return newHead;

}

递归实质上就是系统帮你压栈的过程，系统在压栈的时候会保留现场。

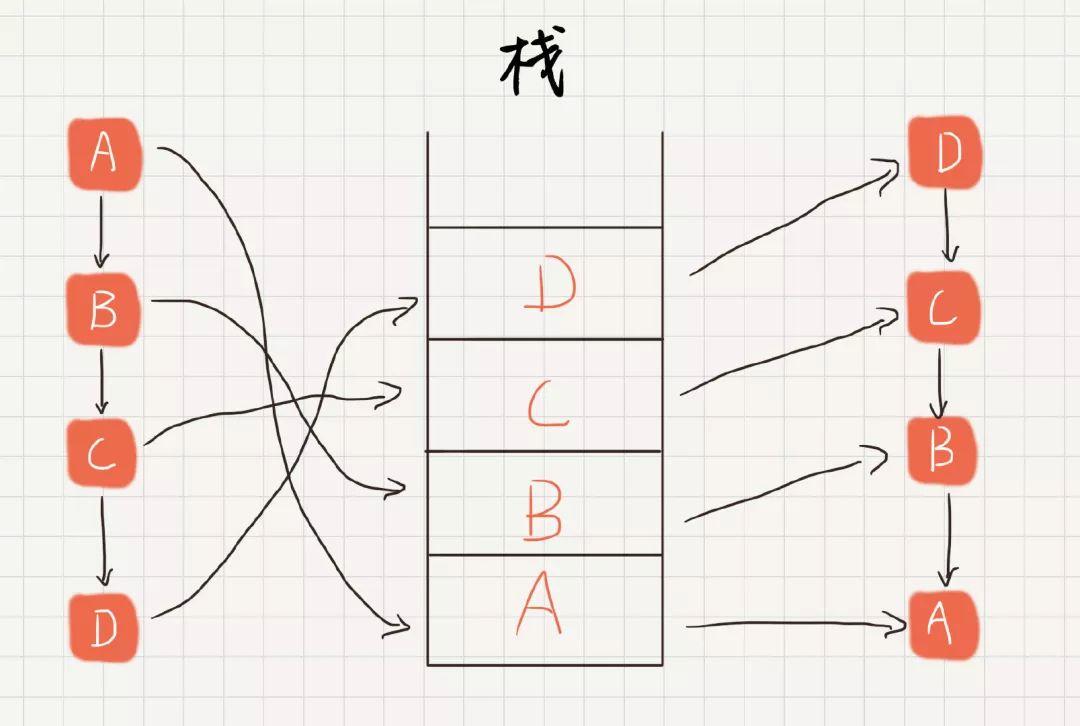
链表1->2->3->4的递归过程如下：

* 程序到达Node newHead = reverse(head.next);时进入递归
* 我们假设此时递归到了3结点，此时head=3结点，temp=3结点.next(实际上是4结点)
* 执行Node newHead = reverse(head.next);传入的head.next是4结点，返回的newHead是4结点。
* 接下来就是弹栈过程了
  + 程序继续执行 temp.next = head就相当于4->3
  + head.next = null 即把3结点指向4结点的指针断掉。
  + 返回新链表的头结点newHead

注意：当retuen后，系统会恢复2结点压栈时的现场，此时的head=2结点；temp=2结点.next(3结点)，再进行上述的操作。最后完成整个链表的翻转。

（尝试根据图结算法中的方法，绘制递归过程。）

方法三：栈实现。时间复杂度O(N)，空间复杂度O(N)



# LeetCode53 最大子序和

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

输入: [-2,1,-3,4,-1,2,1,-5,4],

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

进阶:

如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/maximum-subarray

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**分析：**

**方法一：暴力解法。n个数的子集之和有n(n+1)/2种情况，若有n个元素，则有n+(n-1)+ … + 1种，可使用两层for循环进行遍历。内存for循环求得localMax，外层循环比较localMax求得globalMax。时间复杂度O(n)，空间复杂度O(1)。**

**方法二：动态规划。时间复杂度O(N)，空间复杂度O(N)**

定义状态：

\* dp[i] ： 表示以 nums[i] 结尾的连续子数组的最大和

\* <p>

\* 状态转移方程：

\* dp[i] = max{num[i],dp[i-1] + num[i]}

**方法三：动态规划。**

**动态规划的是首先对数组进行遍历，当前最大连续子序列和为 sum，结果为 ans**

**如果 sum > 0，则说明 sum 对结果有增益效果，则 sum 保留并加上当前遍历数字**

**如果 sum <= 0，则说明 sum 对结果无增益效果，需要舍弃，则 sum 直接更新为当前遍历数字**

**每次比较 sum 和 ans的大小，将最大值置为ans，遍历结束返回结果**

**时间复杂度：O(n)，空间复杂度O(1)**

class Solution {

public int maxSubArray(int[] nums) {

int len = nums.length;

if(len < 1) return 0;

int sum = 0;

int res = nums[0];

for(int n : nums){

if(sum > 0) sum = sum + n;

else sum = n;

res = Math.max(res, sum);

}

return res;

}

}

**方法四：动态规划。和方法三思想一致，但实现更易理解。时间复杂度：O(n)，空间复杂度O(1)**

class Solution {

public int maxSubArray(int[] nums) {

int len = nums.length;

if(len < 1) return 0;

int segmentSum = nums[0];

int res = nums[0];

for (int i = 1; i < len; i++) {

segmentSum = Math.max(nums[i], segmentSum + nums[i]);

res = Math.max(res, segmentSum);

}

return res;

}

}

# LeetCode

# LeetCode

# LeetCode

# LeetCode