排序算法：经典排序算法深度剖析

前言

排序是很多算法的基础，很多优秀的算法都要求数据有序。另外排序也有着很多实际的使用场景。比如：

- 按照成绩高低给大家排名次。
- 比如电脑右键的 `sort by xx`
- 电脑的任务管理器可以根据 CPU，内存的的使用情况进行各种排序
- 等等

因此排序对我们很重要，以至于大学的教材讲算法常常以排序作为开始，面试的时候也经常考察大家对于排序算法的掌握程度，这种情况在互联网早期尤为普遍。有的是直接让你实现某一种排序算法，有的是让你分析不同排序算法的复杂度和稳定性。

如今我们对具体的排序算法比较陌生的很大原因是排序算法已经很稳定了。虽然目前直接考察排序算法的比例在不断下降，但是各种排序算法的思想是很重要的，这些算法都是经过了历代聪明绝顶的人才不断优化产生的，非常值得我们拿来学习。

比如通过学习快速排序和归并排序让我深刻理解了**分治**，**递归**，**前中后序遍历**，通过学习冒泡排序和插入排序以及选择排序让我深刻理解了分类的思想，通过学习计数排序让我明白了数据离散且集中的情况可以有更快的排序方法等等。

排序的算法有非常多，比较有名的排序有十种，分别是：

- 冒泡排序
- 插入排序
- 选择排序
- 快速排序
- 希尔排序
- 归并排序
- 堆排序
- 计数排序
- 基数排序

- 桶排序

这十种排序算法我们称之为十大排序。

由于篇幅的原因，这里不会对十种排序分别讲解， 也没有必要。这里只介绍五种最经典的，面试中经常考察的排序算法，它们分别是 冒泡排序 ， 插入排序 ， 快速排序 ， 归并排序 和 计数排序 。

根据排序的逻辑是否是**比较两个元素大小**，我们将排序算法分为：基于比较的排序（比如插入排序，归并排序，快速排序等等）和非基于比较的排序（比如桶排序，基数排序等等）。基于比较的排序总会在**相对位置不正确**的时候进行交换，因此交互两个相对位置不正确的元素是基于比较算法的共同点。

本文要讲的这五种排序除了计数排序，其他都是基于比较的排序。

基于比较的排序时间复杂度都不会低于 nlonN，计数排序由于不是基于比较的，因此其时间复杂度理论上可以更好，计数排序本质是空间换时间的做法，而由于计数排序对数据要求比较高， 因此适用范围也比较受限。

虽然堆排序的使用场景也比较多，但是由于堆排序如果直接调用封装好的堆数据结构，那么就无比简单，以至于不需要讲解。而如果自己实现堆，则可能导致篇幅过长，对堆感兴趣的可以看下我之前写过的堆专题，这里不再赘述。

- **堆专题 - 上**
- **堆专题 - 下**

一点前提

为了方便描述，假设我们需要对数组 nums 进行**升序**排序，数组中的值**都是数字**，数组大小为 **n**，下面介绍的几种算法也是基于这个前提， 不再赘述。

另外，没有什么特殊原因，我们都应该按照传统的**从左到右遍历**方式来完成算法。因此下文如不作特殊说明我们都采取从左到右遍历数组。

值得注意的是：

- 虽然我这里只对数组进行排序，实际上这些算法都可以扩展到其他数据结构，比如链表。
- 虽然我这里限定了数组中的元素是数字， 实际上算法同样可以扩展到非数字， 只要值是**可比较**（comparable）的即可。比如 'a' 和 'b'，我认为 'a' 比 'b' 小，这就是可比较的。

另外一个比较重要的点是排序的稳定性。稳定性指的是经过排序之后，相同大小的值是否仍然能保持原有的相对位置不变。如果可以，那么算法就是稳定的，否则算法是不稳定的。而一般，如果我们排序的是数字等基本类型，稳定性是没有意义的。因此考虑稳定性的常见更多的应该是复杂对象的排序。

冒泡排序

介绍

冒泡排序是很多人接触的第一个排序算法，这个算法很经典，我们先来学习它。

由于这是我们讲的第一个排序算法，因此我尽可能详细地讲解，后面的讲解则会适当精简。

算法简述：

冒泡排序需要从序列的一端开始往另一端**冒泡**。接着依次比较相邻的两个数的大小，如果两个数字的相对位置大小不正确，则交换两个元素，否则不做任何操作，继续移动即可。

> 你可以从左往右冒泡，也可以从右往左冒泡，这里我们以从左到右冒泡为例。

具体来说， 就是从数组索引 0 开始，两两比较相邻元素， 如果需要交换则交换。接下来从数组索引 1 开始，两两比较相邻元素， 如果需要交换则交换 。。。直到到达数组索引为 n - 1 的元素为止。这样的一次**从数组一端到另一端的过程**我们称之为一轮。

经过这样一轮的操作，数组索引为 n - 1 的元素已经就位了。因此下一轮从数组索引 0 到数组索引 n - 2 即可。

接下来，我们通过一个具体的例子来帮我消化一下这句话。

图解

以 数组 nums: [2,3,8, 1, 5] 为例。

遍历到数组中的第一项，这个时候我们比较相邻的数组的大小。上面说了 **如果相对位置不正确，则交换两个元素**。而由于我们是要升序，因此如果**后面的元素比前面的元素小就是相对位置不正确**。

我们可以从数组第一个开始，也可从数组第二项开始， 影响的只是边界条件而已。

如果从数组第一项开始，代码是：

```
for (i = 0; i < nums.length - 1; i++) {
  if (nums[i] > nums[i + 1]) {
    swap(nums, i, i + 1);
  }
}
```

如果从数据第二项开始，代码是：

```
for (i = 1; i < nums.length; i++) {
  if (nums[i - 1] > nums[i]) {
    swap(nums, i - 1, i);
  }
}
```

两种代码都是一样的， 本质都是进行了一次 N - 1 的遍历操作，这里我们不妨从数组第一项开始。

> 下文不再对这种边界问题进行特殊说明。

swap 的功能就是交换数组的两个元素，代码:

```
function swap(nums, i, j) {
  [nums[i], nums[j]] = [nums[j], nums[i]];
}
```

如果你愿意的话， 你可以将上面的代码复制粘贴 N - 1 次 。

```
for (i = 1; i < nums.length; i++) {
  if (nums[i - 1] > nums[i]) {
    swap(nums, i - 1, i);
  }
}
// ...
// 复制粘贴 N - 1 次
```

恭喜你，你已经完成了冒泡排序的算法。不过实际工程没有人会这么写代码， 因为有一种东西叫做循环。

```
for (round = 0; round < nums.length; round++) {
  for (i = 1; i < nums.length; i++) {
    if (nums[i - 1] > nums[i]) {
      swap(nums, i - 1, i);
    }
  }
}
```

更多的时候， 我们会把 round 命名为 i ， 这里的 i 就变成了 j。 这是一种约定俗成的命名，我强烈建议你这么做。

```
for (i = 0; i < nums.length; i++) {
  for (j = 1; j < nums.length; j++) {
    if (nums[j - 1] > nums[j]) {
      swap(nums, j - 1, j);
    }
  }
}
```

这个时候你已经完成了一个功能正常的冒泡算法。聪明的算法工作者不满足于此，他们对这个算法进行了优化，使得比较次数直接减半，下面来看下是如何做到的。

前面我提到了 经过这样一轮的操作，数组索引为 n - 1 的元素已经就位了 ， 答案就隐藏在这句话中。

由于每次操作我们都会使得一个元素就位，因此没必要每次都遍历到 n - 1，而是 n - round， 其中 round 为轮数，取值为 1,2, ... ,n - 1。

> 形象地来看， 就好像是在吐泡泡，前端事件系统的冒泡机制也是类似。

代码：

```
for (i = 0; i < nums.length; i++) {
  for (j = 1; j < nums.length - i; j++) {
    if (nums[j - 1] > nums[j]) {
      swap(nums, j - 1, j);
    }
  }
}
```

这样每次的比较次数都会减 1。由于需要执行 n - 1 轮， 每一轮需要比较的次数为等差数列，由等差数列求和公式可知冒泡排序的比较次数大概为 $\mathbf{n^2/2}$，比之前的 $\mathbf{n^2}$ 少了一半，不过时间复杂度扔为 $\mathbf{n^2}$ 。

由于每一轮都会使得一个元素就位， 因此实际上排序一个长度为 n 的数组， 只需要 n - 1 轮就行了，因为 n - 1 个就位了，最后一个肯定也就位了，因此代码还可以进一步优化：

```
for (i = 0; i < nums.length - 1; i++) {
  for (j = 1; j < nums.length - i; j++) {
    if (nums[j - 1] > nums[j]) {
      swap(nums, j - 1, j);
    }
  }
}
```

复杂度分析

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(1)$

你以为这就结束了吗？ 不！，我们还要继续优化。

冒泡排序不管数组是有序还是接近有序还是完全无序等，遍历的次数都是一样的。也就是说你去排序一个 [1,2,3,4,5] ，也依然傻傻地遍历 N - 1 轮。而在这个例子中， 你在第一轮就可以得知**数组是否有序**的特征，进而直接退出循环， 这样就节省了 N - 2 轮的时间。

如果数组是接近有序的， 比如 [1,2,3,5,4]。第一轮，我们将数组变成了 [1,2,3,4,5]，下一轮我们就能够知道数组已经有序了，进而直接退出循环， 这样就节省了 N - 3 轮的时间。

总的来说， **对完全有序或者接近有序的数组，这种优化效果很明显。**

最终代码：

```
for (i = 0; i < nums.length - 1; i++) {
  sorted = true;
  for (j = 1; j < nums.length - i; j++) {
    if (nums[j - 1] > nums[j]) {
      swap(nums, j - 1, j);
      sorted = false;
    }
    if (sorted) break;
  }
}
```

复杂度分析

- 时间复杂度：这种算法优化的仅仅是特殊情况，也就是说最好的情况下时间复杂度为 $O(n)$。而我们的复杂度分析只考虑最差情况，因此时间复杂度依然是 $O(n^2)$
- 空间复杂度：$O(1)$

总结

冒泡排序就是从序列的一端开始往另一端**冒泡**。你可以从左往右冒泡，也可以从右往左冒泡，接着依次比较相邻的两个数的大小，如果两个数字的相对位置大小不正确，则交换两个元素，否则不做任何操作，继续移动即可。对数组执行 N - 1 轮这样的操作就可以完成了排序过程。

冒泡算法虽然简单，但是也有很多可以优化的地方，比如轮数可以优化到 N - 1 轮，内层比较可以每次都减少一次，以及数组有序的时候可以提前退出。

插入排序

插入排序的思路也非常简单。和前面的冒泡排序类似，它的思路也是分别就位数字的每一位，只不过具体的就位方式不太一样罢了。

首先我们需要固定数组索引为 0 的数，每次固定只和已经就位的数进行比较，将当前数字**插入到已经就位的数字中**，当所有数字都就位整个数组就有序了，这就是插入排序的核心思想。

由于只有一个数，一个数一定是有序的，因为从 0 开始是没有必要的，我们从索引 1 开始进行比较。

当遍历到索引为 1 时，我们尝试插入到前面已经就位的一个数中。由于已经就位的数字是递增的，因此我们可以从后往前依次和已经就位的进行比较，当发现相对位置关系不正确，我们则将其进行交换，直到相对位置关系正确。由于我们是升序排列，因此当前项比前一项小就是相对位置关系不正确。

具体地：

- 我们需要固定一层循环来就位每一个数，从 1 开始 到 n - 1 即可。（前面已经讲过了为什么从 1 开始，而不是 0）

- 对于每一层循环，我们和前面已经就位的数进行比较。如果相对位置不正确则进行交换，否则结束内层循环。

> 值得注意的是“如果相对位置不正确则进行交换” 的交换是不断进行的，也就是说如果 xxxab 交换后变成 xxxxba 之后，还需要将 b 和前面的数进行同样的比较，观察其是否相对位置正确，并重复这个过程直到相对位置正确为止。

我们来看下代码如何书写：

```
function sortArray(nums) {
  const n = nums.length;
  for (let i = 1; i < n; i++) {
    t = nums[i];
    j = i - 1;
    while (j > -1 && nums[j] > t) {
      nums[j + 1] = nums[j];
      j -= 1;
    }
    nums[j + 1] = t;
  }
  return nums;
}
```

插入排序和冒泡有点像。只不过冒泡的时候，假设我们是从左到右遍历，那么实际上每一轮会令右边的一位就位。

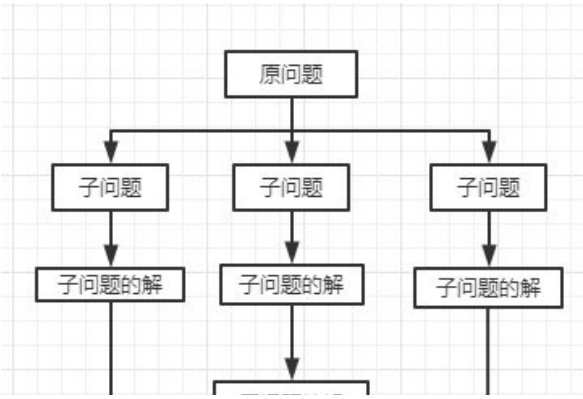
而插入排序，假设我们也是从左到右遍历，实际上每一轮我们会令左边的一位就位。

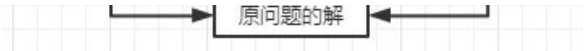
这有什么影响呢？这其实有很大的不同。这是因为插入排序每次都只需要往前进行比较，一旦**当前数字和前一个数字相对位置关系正确，则无需再往前进行比较，因此前面的已经有序了**。

而如果是冒泡，遍历的方向和数组就位的数字在数组的两端，因此不能直接退出**内层循环**。这种对比的记忆，容易加深大家对各种算法的理解。

归并排序

归并排序是一个典型的分治思想算法。核心思想就是将一个数组不断分为**两半**，直到无法分割（即一个元素），然后**两两**合并，两个小数组合并成大数组，直到最终合并完成。





如果你学过树的前中后序遍历的话，实际上**归并排序就是一种后序遍历，而后面我们要讲的快速排序则是一种先序遍历。**

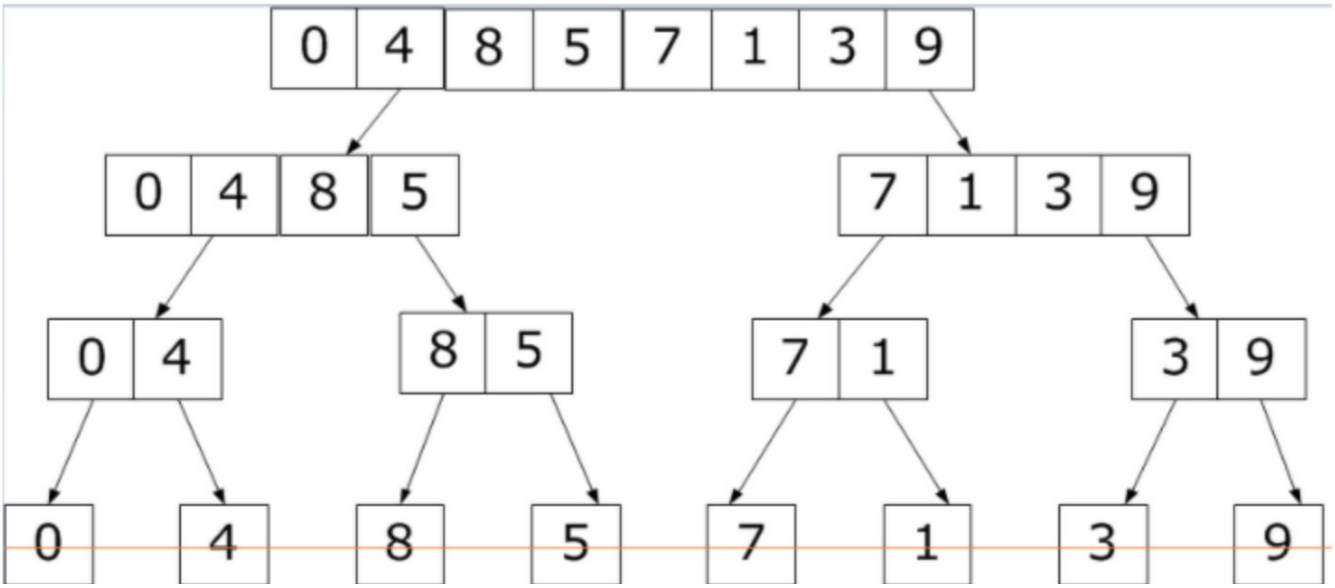
接下来，我们来简单概括一个归并排序的核心思路：

- 将原问题分解至达到求解边界的结构相同且互相独立的子问题
- 对所有的子问题进行求解
- 将所有子问题的解进行合并，从而得到原问题

比如数组 [0,4,8,5,7,1,3,9]。我们可以先将它分为 [0,4,8,5] 和 [7,1,3,9]。

> 你当然可以分为别的形式。 比如 [0,4,7,1] 和 [8,5,3,9]。只不过对算法没有好处，只是徒增了代码书写难度，因此建议直接取中点分割数组为两部分即可。

对于这两个数组， 执行同样的操作， 直到无法分割（一个数字）。



接着，我们进行合并。两个有序数组合并成一个有序数组也是力扣的简单题目。 具体来说，我们只需要使用两个变量分别记录两个数组的读取位置，然后通过比较两个指针对应的数字的相对关系，读取相应数字并更新对应指针即可。

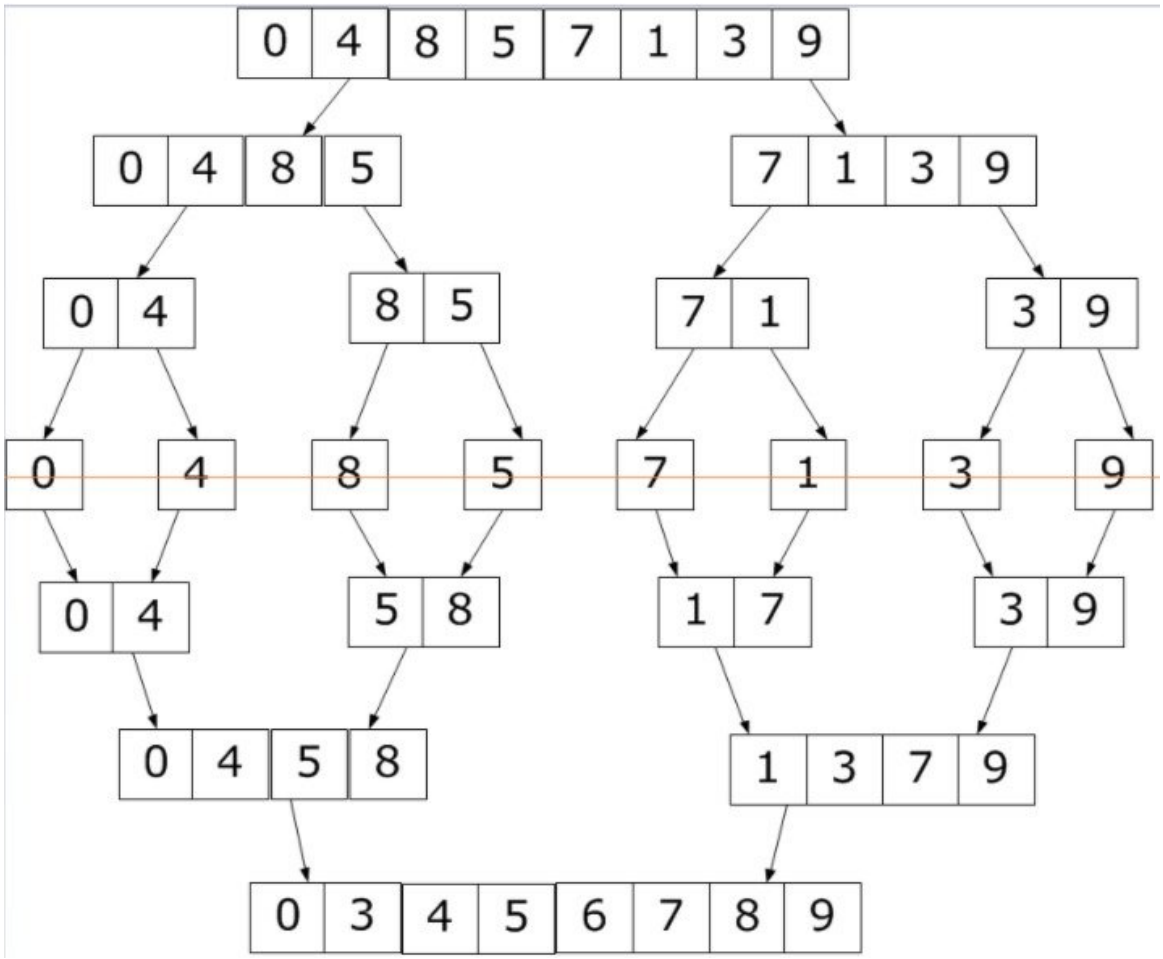
代码：

```
// 将nums1 和 nums2 合并
function mergeTwo(nums1, nums2) {
  let ret = [];
  let i = (j = 0);
  while (i < nums1.length || j < nums2.length) {
    if (i === nums1.length) {
      ret.push(nums2[j]);
```

```
      j++;
      continue;
    }

    if (j === nums2.length) {
      ret.push(nums1[i]);
      i++;
      continue;
    }
    const a = nums1[i];
    const b = nums2[j];
    if (a > b) {
      ret.push(nums2[j]);
      j++;
    } else {
      ret.push(nums1[i]);
      i++;
    }
  }
  return ret;
}
```

经过若干次合并， 我们最终就得到了原数组的有序形式。



因此我们的算法就是：

- 将数组不断进行二分，直到数组长度为 1
- 利用合并两个有序数组的思路，将子数组两两进行合并

由于算法一共有 logn 层， 每层的操作数都是 n（合并有序数组的操作），因此总的时间复杂度就是 nlogn。

如果你使用递归来完成，那么递归栈的最大长度就是上图的高度，大概是 logn，再加上合并有序数组开辟的 n 的临时数组，总的空间复杂度为 $\mathrm{n}+\mathrm{logn}$，忽略低此项后为 $\mathrm{O(n)}$

总结：归并排序核心是分治，先分到不能再分的情况（单个值），然后两两合并求解。代码上，我们只需要实现两个函数即可。

```
# 第一个函数就是主函数， mergeSort，用户只需要调用 mergeSort(0, 数组长度-1) 就可以得到一个有序数组。
def mergeSort(l, r):
    if l >= r: return [nums[l]]
    mid = (l+r)//2
    return mergeTwo(mergeSort(l, mid), mergeSort(mid+1, r))
# 第二个函数就是合并两个有序数组。力扣难度为简单。
def mergeTwo(left, right):
  pass
```

推荐题目:

- **493. 翻转对**

快速排序

快速排序是每次从数组中选出一个基准值（pivot），其他数依次和基准值做比较，比基准值大的放右边，比基准值小的放左边。

> 你也可以选择将比基准值大的放左边，比基准值小的放右边，不过尽量不要这么做。因为不符合直觉，并且算法效率不会有任何改变。

然后对左边和右边的两组数分别选出一个基准值，进行同样的操作，直到无法继续（只剩一个元素）。

每一次这样的操作，目标都是将 pivot 挪到数组某个位置 i，且以第 i 位为分界点，左边的数都小于等于 pivot，右边的数都大于等于 pivot。经过这样的操作，pivot 就已经就位了。

快速排序的每一轮处理其实就是将这一轮的基准数归位，当所有的数都归位了，那么排序就结束了。

以数组 [0,4,8,5,7,1,3,9] 来说。我们需要先选取一个基准，不妨选 5.

- 1. 先从右往左找一个小于 5 的数，再从左往右找一个大于 5 的数，然后交换他们。

> 这里可以用两个变量 i 和 j，分别指向序列最左边和最右边。

- 2. 此时 j 指向 3， i 指向 8，交换它们，此时的数组为 [0,4,3,5,7,1,8,9]。

- 3. 继续从右往左找一个小于 5 的数，再从左往右找一个大于 5 的数，然后交换他们。

- 4. 此时 j 指向 1， i 指向 5，交换它们，此时的数组为 [0,4,3,1,7,5,8,9]。

- 5. 继续从右往左找一个小于 5 的数，再从左往右找一个大于 5 的数，然后交换他们。

- 6. 此时 j 指向 5， i 指向 7，交换它们，此时的数组为 [0,4,3,1,5,7,8,9]。

- 7. i 和 j 交汇了，算法结束。

经过一轮这样的处理，数组变成了：[0,4,3,1,5,7,8,9] ， 其中基准元素 5 已经就位了，即最终排序好的数组 5 一定就在这个位置。经过 N 轮这样的处理， 所有的元素都会就位，整个数组自然就变得有序了。当然经过一轮处理，远远不止就绪基准元素这么简单， 不然和选择以及冒泡排序等就差不多了。

实际上， 除了就位基准元素，还有一个更重要的功能。我们注意到基准元素左侧都是小于基准元素的，而基准元素右侧都是大于基准元素的。因此我们**不仅就为了基准元素，我们还大致锁定了其他元素的位置。** 即[0,4,3,1] 这几个数字都在数组的前四项，[7,8,9] 都在数组的后三项。

这有什么用呢？这其实很有用！大家注意到上面算法描述部分的交换元素了么？实际上，之后的交换元素只会发生在 [0,4,3,1] 内部和[7,8,9] 内部， **这两个部分之间是不可能发生交换的**。这就是快排之所以快的原因。

相比冒泡排序的跳跃式交换两个元素。快速排序每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换两个不相邻的数的时候就不会像冒泡排序一样每次只能借助在相邻的数之间进行交换达到交换不相邻元素的目的，因此总的比较和交换次数就会变少。当然在最坏的情况下（基准元素某一侧没有元素），还是相邻的两个数进行了交换。

> 另外跳跃性的交换对 CPU 来说也不友好，使得 CPU 的某些优化算法失效。

快速排序和归并排序的**分**的过程很像，只不过归并是能保证每次都分为大小差不超过 1 的两个部分，而快速排序做不到。因此理论上快速排序，会发生一边倒的情况。即所有的元素都比 pivot 大或者所有的元素都比 pivot 小。这种极端情况，算法的

时间复杂度会退化到 $\mathbf{N^2}$。一种优化方式是随机选择 pivot，这样可以使得算法尽可能稳定，平均情况可以达到$\mathbf{nlogn}$ 的时间复杂度。

> 因此有些算法会预先对数组的无序性进行衡量。只有无序性超过某个阈值，才会使用快速排序。

至于 $\mathbf{nlogn}$ 是如何计算出来的也不复杂，大致和归并算法基本一样。 具体来说就是：由于算法一共有 logn 层， 每次的操作数都是 n，因此总的时间复杂度就是 $\mathbf{nlogn}$。

快排参考代码：

```
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        temp = [0] * len(nums)

        def partition(l, r):
            pivot = nums[l]

            while l < r:
                while l < r and nums[r] >= pivot:
                    r -= 1
                # nums[r] 是右边数第一个小于 pivot 的，将其交换到 l 位置
                nums[l] = nums[r]
                while l < r and nums[l] <= pivot:
                    l += 1
                # nums[l] 是左边数第一个大于 pivot 的，将其交换到 r 位置
                nums[r] = nums[l]
            # 最后将轴元素 nums[l] 还原
            nums[l] = pivot
            return l

        def quickSort(l, r):
            if l >= r:
                return
            pivot = partition(l, r)
            quickSort(l, pivot - 1)
            quickSort(pivot + 1, r)

        quickSort(0, len(nums) - 1)
        return nums
```

快速排序很有用。一个经典的变种是快速选择。比如我们需要求一个无序数组中的第 k 大的数。那么我们就可以在根据轴元素（pivot）进行分区的时候舍弃一半，而无需像快速排序那样分别对两侧的数组进行递归处理。 推荐题目: **215. 数组中的第 K 个最大元素** 使用快选平均可以达到 $\mathbf{O(n)}$ 的时间复杂度，而最坏和快排一样会退化到 $\mathbf{O(n^2)}$。

快选参考代码：

```python
class Solution:
    def solve(self, nums, k):
        def partition(l, r):
            pivot = nums[l]
            while l < r:
                while l < r and nums[r] >= pivot:
                    r -= 1
                nums[l] = nums[r]
                while l < r and nums[l] <= pivot:
                    l += 1
                nums[r] = nums[l]
            nums[l] = pivot
            return l

        l, r = 0, len(nums) - 1

        while l < r:
            m = partition(l, r)
            if m ==  k: return nums[m]
            elif m > k: r = m - 1
            else: l = m + 1
        return nums[l]
```

计数排序

桶排序，基数排序以及计数排序理论上复杂度都可以达到线性，这是这些算法相比上面提到的最大的优点。缺点就是空间复杂度可能比较高，对数据要求比较严格。

我们来看下计数排序的具体算法，你就明白为什么它最好可以达到线性时间，以及为什么有些情况空间复杂度会很高了。

仍然以数组 [0,4,8,5,7,1,3,9] 为例。

- 最大值是 9， 最小值是 0， 因此需要建立一个大小为 9 - 0 + 1 = 10 的数组，其中数组的索引表示的数字减去最小值的差（因此一定是非负数），值表示该数字出现的次数。

> 实际算法，很多时候不需要求最大值和最小值再计算数组大小，而是直接根据题目的数据范围确定数组大小。 比如题目说了数组每一项都在 [100, 10^5] 之间。那么就建立一个大小为 10 ^ 5 的数组即可

- 此时数组是 [0,0,0,0,0,0,0,0,0,0]。
- 依次将所有数组放到计数数组上，具体来说就是先找到他们的位置， 然后更新计数器，使其加 1。

[1,0,0,0,0,0,0,0,0,0] 先放 0

[1,0,0,0,1,0,0,0,0,0] 再放 4

[1,0,0,0,1,0,0,0,1,0] 再放 8

。。。

最后计数数组会变成：[1,1,0,1,1,1,0,1,1,1]。

接着我们遍历一次计数数组即可得到一个有序的数组，即完成了对数组的排序工作。

```
nums = [0, 4, 8, 5, 7, 1, 3, 9];
counts = [1, 1, 0, 1, 1, 1, 0, 1, 1, 1];
lower = 0;
cur = 0;
for (let i = 0; i < counts.length; i++) {
  let count = counts[i];
  // 数组可能有重复元素， 因此需要 while 循环一下
  while (count !== 0) {
    count -= 1;
    nums[cur++] = i + lower;
  }
}
console.log(nums); // [0, 1, 3, 4, 5, 7, 8, 9]
```

不难看出计数排序的时间和空间复杂度实际取决于数字的分布情况，而不是数组的大小。

计数排序的空间复杂度是 $O(\text{upper} - \text{lower})$，其中 upper 为 数组的最大值， lower 为数组的最小值。 可以看出，如果数组中的值比较集中，那么使用计数排序则非常高效，相反则会浪费大量空间。

值得注意的是，就稳定性而言上面的算法是不稳定的。

不过我们可以继续优化，使得他变得稳定，代价就是 $O(N)$的额外空间。

具体来说，我们可以：

- 使用一个额外数组， 这个数字是原数组的拷贝。
- 对所有的计数累加求**前缀和**。还是以上面的例子来说， 此时的前缀和 为 [1,2,2,3,4,5,5,6,7,8]
- 反向填充目标数组，将每个元素 a 放在新数组的第 counts[a]项，每放一个元素就更新 counts, 即将 counts[i]减去 1。

代码：

```
counts = [1, 2, 0, 1, 1, 1, 0, 1, 1, 3];
nums = [0, 4, 8, 5, 7, 1, 3, 9];
copy = [];
lower = 0;

for (let i = 1; i < counts.length; i++) {
  counts[i] += counts[i - 1];
```

```
}

for (let i = nums.length - 1; i >= 0; i--) {
  const a = nums[i];
  copy[counts[a - lower] - 1] = a;
  counts[a - lower] -= 1;
}
console.log(copy); //  [0, 1, , 3, 4, 5, 7, 8, , , 9]
```

由于 nums 是按照顺序遍历的， 因此最终排序的数组也会按照原来 nums 的顺序出现， 因此这种算法是稳定的。相应地空间复杂会增加 $O(N)$，因此总的空间复杂度会变成 $O(N + upper - lower)$。

虽然直接考计数排序的题很少见，但是计数排序的应用却很广泛， 笔者在很多题目都用到了它。只不过有的时候不需要你排序而已，另外状态压缩也会使用二进制位存储状态，区别于计数排序需要记录次数， 状态压缩往往关心的是否存在。

举一个例子，请听题。

题目描述：

```
Write a function that takes in an array of integers and returns an array of length 2 representing the largest range of i

The first number in the output array should be the first number in the range, while the second number should be the last

A range of numbers is defined as a set of numbers that come right after each other in the set of real integers. For inst

You can assume that there will only be one largest range.

Sample Input:

array = [1, 11, 3, 0, 15, 5, 2, 4, 10, 7, 12, 6]

Sample Output:

[0, 7]
```

题目是英文的， 为了不曲解题目意思， 我直接贴的原文。 题目大概意思是给你一个数组 array，要求你找出这个数组的最长连续数组。 注意这个数组的相对顺序不一定和 array 的顺序保持一致。 返回的时候也没必要把最长连续数组这个返回， 而是仅返回两个端点即可。 比如 [1, 11, 3, 0, 15, 5, 2, 4, 10, 7, 12, 6]，最长连续数组就是 [0,1,2,3,4,5,6,7]，尽管在原数组中他们不是这个顺序。由于值返回端点即可， 因此你需要返回 [0, 7]。

这道题并不需要对每一个数计数， 因为我们仅关心某一个数**是否存在**， 因此直接使用一个长度为 upper - lower 的数组记录每一个数的出现情况即可。 唯一需要注意的是负数的情况。

- 将数据离散化到 [0, upper - lower + 1]的数组上，索引为值，值为 1（二值化）。其中 upper 为 数组的最大值， lower 为数组的最小值。
- 滑动窗口思路计算最大的连续 1 即可。

代码：

```
def largestRange(array):
 upper, lower = max(array), min(array)
    temp = [0] * (upper - lower + 1)
 l = cnt = ans = 0
 for i in range(len(array)):
  temp[array[i] - lower] = 1
 for i in range(len(temp)):
  cnt += temp[i]
  if temp[i] == 0:
   cnt = 0
  if cnt > ans:
   ans = cnt
   l = i - cnt + 1
 return [l + lower, l + ans + lower - 1]
```

复杂度分析

- 时间复杂度：$O(\text{upper} - \text{lower})$，其中 upper 为 数组的最大值， lower 为数组的最小值。
- 空间复杂度：$O(\text{upper} - \text{lower})$，其中 upper 为 数组的最大值， lower 为数组的最小值。

类似的题目还有很多， 读者可以在平时的练习过程中多多感受一下。

排序技巧

就位思想

很多题目都有就位思想。这些题目的一个共同限制是数组中的元素不重复。

比如给你一个值为 1 - n 数组 nums，里面缺一个数字，让你找到这个数字。 你就可以将让 1 就位到 nums[0]， 2 就位到 nums[1]， 3 就位到 nums[2]， ...， n 就位到 nums[n - 1]。 力扣原题 **41. 缺失的第一个正数** 就是这样的题目。

另外一种是给你 nums[i] 应该在 nums 中的位置（第几大），让你根据这个规则排序。力扣有好几道题目，我一时想不起来。找了一个 Binary Search 上的题目 **Sort by Permutation** 这道题目你就可以将 p 数组就位成 p = [0,1,2,3,4,5,6,7,8,9,...] 的数组达到目标，思想和上面的题目一样。

那么如何就位？核心思想很简单，伪代码表示：

```
# 这里要就位 nums[i]，使得 nums[i] 等于 nums[i] -1
# 如何就位？不断交换 nums[i] 和 nums[nums[i] - 1]
# 因为实际上一次交换，是让 nums[i] 就位到 nums[nums[i] - 1]，此时交换回来的 nums[nums[i] - 1] 不一定是 nums[i]
while nums[nums[i] - 1] != nums[i]:
  nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]
```

两个问题：

1. 有可能无限循环么？

不可能。

交换回来的数有两种可能：等于 nums[i] 和 不等于 nums[i]。

如果等于，交换停止，不会死循环。如果不等于，由于每个数都仅出现一次，因为再次交换一定不会交换回去。如果发生死循环，那么意味着交换链路存在环，这和数组中不存在重复数矛盾。

2. 时间复杂度是多少?

O(n)，其中 n 为数组长度。 这是因为一旦被交换到正确位置，那么就不会发生交换，总的交换次数和数组线性相关。

多维数组排序

题目中输入是二维数组，不妨称其中第一个维度为 x 坐标， 第二个维度为 y 坐标。

比如题目输入是 nums = [[1,2,3,4], [5,6,7,8]]

这个时候我们可以考虑排序按一个维度排序（通常是 x 坐标排序），然后处理另外一个坐标（通常是 y 坐标）。

比如按照 x 升序排序 num.sort() 或者按照 x 降序排序 nums.sort(key=lambda x:-x[0])。这种技巧用地很多，值得大家掌握。

推荐题目：**1356 · 最大点的集合**

题目让我们求坐标轴中横纵坐标都严格大于当前点的点的个数。

我们可以按照横坐标进行排序。假设处理到点 (x, y)。那么 x 的右侧**横坐标**都是大于它的（题目限定了横纵坐标都不相同）。这样我们仅需要考虑**是否存在**右侧的点纵坐标也大于 y 的即可，如果不存在（不存在其实指的当前的 y 是右侧纵坐标最大的）我们将其将入到答案。 这个问题可以转化为一维数组中是否存在右侧大于当前数的数。解决这个问题也很简单，我们依次遍历并维护最大值即可。

参考代码：

```
class Solution:
    def MaximumPointsSet(self, points):
        n = len(points)
        points.sort()
        max_y = float('-inf')
        ans = []

        for i in range(n-1,-1,-1):
            if points[i][1] > max_y: ans.append(i)
            max_y = max(points[i][1], max_y)
        return [points[i] for i in ans][::-1]
```

再推荐一道周赛的题目供大家练习：**1996. 游戏中弱角色的数量**，基本思路是一样的，力扣中相同思路的题目不在少数，大家可以多留意一下。

总结

所有排序代码都可以在刷题插件 leetcode-cheat 代码模板中的手撕算法中获取，插件可以在公众号力扣加加中回复插件获取！

冒泡排序最差的情况以及平均情况都可以达到 $O(n^2)$，而在最好的情况是数组已经有序， 我们可以提前退出， 此时时间复杂度可以达到 $O(n)$。而空间复杂度为 $O(1)$。又由于相同的数，并不会发生交换， 因此冒泡排序是一种稳定排序。

归并排序最好的情况，最差的情况以及平均情况都可以达到 $O(nlogn))$。由于借助了中间数组，因此空间复杂度为 $O(n)$，此时递归产生的 $O(logn)$ 调用栈并不是算法的瓶颈。又由于相同的数合并的过程不会改变相对位置，因此归并排序也是一种稳定的排序算法。

快速排序最好的情况以及平均情况都可以达到 $O(nlogn)$，而在最差的情况是 pivot 分布极度不均匀， 此时时间复杂度可以达到 $O(n^2)$。而空间复杂度为递归栈的深度，因此为 $O(logn)$。又由于相同的数，可能会发生交换， 因此快速排序是一种不稳定排序。 比如数组 [3, 4, 5, 5, 4, 3]，同时选择数组第一项为基准元素。v8 为了 排序稳定，在 es10 之后元素个数大于 10 都采用 timsort，在这之前使用的不稳定的快速排序算法。快速排序也是本章唯一的一种非稳定排序算法。

计数排序不管什么情况时间和空间复杂度都是 $O(upper - lower + N)$。又由于我们使用了额外的空数组，使得相同的数可以保持原有的顺序， 因此计数排序是一种稳定排序。

送给大家一张图， 方便大家记忆。

	时间复杂度（最坏）	时间复杂度（平均）	空间复杂度（最坏）	稳定性
冒泡排序	O(n^2)	Θ(n^2)	O(1)	稳定
插入排序	O(n^2)	Θ(n^2)	O(1)	稳定
快速排序	O(n^2)	Θ(nlogn)	O(logn)	不稳定
归并排序	O(nlogn)	Θ(nlogn)	O(n)	稳定
计数排序	O(upper - lower + n)	Θ(upper - lower + n)	O(upper - lower + n)	稳定

其中有几点需要大家注意：

- 符号 O(xxx) 表示最坏复杂度，而 Θ(xxx) 表示的平均复杂度。
- 快速排序是本章唯一一个不稳定的排序，因此在对复杂类型排序的时候需要考虑是否对稳定性有要求。
- 快速排序和归并排序的空间复杂度实际上可以进一步优化，感兴趣的可以查阅相关紫资料。

一般而言，现在大多数不会考察直接手写一种经典排序算法，而是偏应用的，比如：

1. 算法某一步需要排序。很多贪心和二分题会先排序。

2. 利用排序算法中的核心思想。比如利用归并排序求逆序数，推荐题目**493. 翻转对**，这显然不是让你直接排序。再比如用桶排序思想求数组相邻最大间距，推荐题目**64. 最大间距**。再比如 **First Missing Positive**，我们可以利用排序中**交换后就位**的思想，一次扫描 $O(n)$ 让 nums[i] - 1 就位到索引为 i 的位置，再一次 $O(n)$ 的扫描找出第一个没就位的正数即可，也就是找到第一个 i 不等于 nums[i] - 1 的数（力扣也有同样的题目 **41. 缺失的第一个正数**，大家可以做做）。

3. 大小关系很重要的时候考虑排序，比如在一个数组中，我想知道所有比当前数字 nums[i] 大的数之和是多少，或者所有比当前数字 nums[i] 大的数一共有几个等等，就可以考虑排序，这样就相当于一次预处理，本身需要遍历才知道的，排序后可以在常数的时间完成。推荐题目 第 280 场周赛 T3 **6006. 拿出最少数目的魔法豆**，由于我需要知道比当前袋子魔法豆多的有几个，少的有几个，因此排序后再处理会将时间复杂度从 $O(n^2)$ 优化到 $O(nlogn)$（考虑了排序的时间复杂度）。再比如 **H 指数** 这道题我们也是想知道不比 citations[i] 小的有几个，那么排序后直接根据当前索引就能确定。由于需要求最大的，那么升序排序倒序枚举（或者降序排序正序枚举）可以剪枝（关于剪枝，会在专题篇进行进一步学习）。

4. 有时候我想求最 xxx（小或者大） 的 yyy（某个指标）。这个时候可以考虑进行一次排序，比如 **720. 词典中最长的单词** 求 最长的字典序最小yyy ，那么就可以考虑按照长度和字典序进行排序，而由于长度越长越优先的，长度相同字典序最小的优先。因此我们可以按照长度升序字典序降序排序。

> 类似这种对一个指标升序另外一个指标降序的例子很多。尤其是一些类 LIS 问题。

如果真的考察你手写排序的话，建议学习一种比较经典且高效的排序算法，比如快速排序，归并排序。另外建议大家数组排序和链表排序都练习一下。因为有的公司比较刁钻，让你手写链表排序。

这里列举链表的归并排序代码：

```
# 1. 归并排序（推荐！其他排序方法都不推荐在竞赛中使用）
class A:
    def sortList(self, head: LLNode):
        def mergeSort(head: LLNode):
            if not head or not head.next:
                return head
            dummyHead = LLNode(-1)
            dummyHead.next = head
            slow, fast = dummyHead, head
            # 找到链表中点，稍微以中点为断点，将其断开
            while fast and fast.next:
                slow = slow.next
                fast = fast.next.next
            nxt = slow.next
            # 断开链表，分别排序
            slow.next = None
            return merge(mergeSort(head), mergeSort(nxt))
        # 合并两个有序链表(难度简单，使用双指针即可)
        def merge(head1: LLNode, head2: LLNode):
            dummyHead = LLNode(-1)
            temp, l1, l2 = dummyHead, head1, head2
            while l1 and l2:
                if l1.val <= l2.val:
```

```
                temp.next = l1
                l1 = l1.next
            else:
                temp.next = l2
                l2 = l2.next
            temp = temp.next
        if l1:
            temp.next = l1
        elif l2:
            temp.next = l2
        return dummyHead.next

    return mergeSort(head)
```

以上代码摘自我的刷题插件。关于数组和链表排序的其他各种方法参考我的插件中的模板中的**手撕算法**。

上一页 下一页