

11.5 快速排序

快速排序 (quick sort) 是一种基于分治策略的排序算法，运行高效，应用广泛。

快速排序的核心操作是“哨兵划分”，其目标是：选择数组中的某个元素作为“基准数”，将所有小于基准数的元素移到其左侧，而大于基准数的元素移到其右侧。具体来说，哨兵划分的流程如图 11-8 所示。

1. 选取数组最左端元素作为基准数，初始化两个指针 `i` 和 `j` 分别指向数组的两端。
2. 设置一个循环，在每轮中使用 `i` (`j`) 分别寻找第一个比基准数大 (小) 的元素，然后交换这两个元素。
3. 循环执行步骤 2.，直到 `i` 和 `j` 相遇时停止，最后将基准数交换至两个子数组的分界线。

哨兵划分完成后，原数组被划分成三部分：左子数组、基准数、右子数组，且满足“左子数组任意元素 \leq 基准数 \leq 右子数组任意元素”。因此，我们接下来只需对这两个子数组进行排序。

快速排序的分治策略

哨兵划分的实质是将一个较长数组的排序问题简化为两个较短数组的排序问题。

Python

```
quick_sort.py
```

```
def partition(self, nums: list[int], left: int, right: int) -> int:
    """哨兵划分"""
    # 以 nums[left] 为基准数
    i, j = left, right
    while i < j:
        while i < j and nums[j] >= nums[left]:
            j -= 1 # 从右向左找首个小于基准数的元素
        while i < j and nums[i] <= nums[left]:
            i += 1 # 从左向右找首个大于基准数的元素
        # 元素交换
        nums[i], nums[j] = nums[j], nums[i]
    # 将基准数交换至两子数组的分界线
    nums[i], nums[left] = nums[left], nums[i]
    return i # 返回基准数的索引
```

11.5.1 算法流程

快速排序的整体流程如图 11-9 所示。

1. 首先，对原数组执行一次“哨兵划分”，得到未排序的左子数组和右子数组。
2. 然后，对左子数组和右子数组分别递归执行“哨兵划分”。
3. 持续递归，直至子数组长度为 1 时终止，从而完成整个数组的排序。

www.hello-algo.com

图 11-9 快速排序流程

Python

quick_sort.py

```
def quick_sort(self, nums: list[int], left: int, right: int):  
    """快速排序"""  
    # 子数组长度为 1 时终止递归  
    if left >= right:  
        return  
    # 哨兵划分  
    pivot = self.partition(nums, left, right)  
    # 递归左子数组、右子数组  
    self.quick_sort(nums, left, pivot - 1)  
    self.quick_sort(nums, pivot + 1, right)
```

11.5.2 算法特性

- **时间复杂度为 $O(n \log n)$ 、自适应排序**：在平均情况下，哨兵划分的递归层数为 $\log n$ ，每层中的总循环数为 n ，总体使用 $O(n \log n)$ 时间。在最差情况下，每轮哨兵划分操作都将长度为 n 的数组划分为长度为 0 和 $n - 1$ 的两个子数组，此时递归层数达到 n ，每层中的循环数为 n ，总体使用 $O(n^2)$ 时间。
- **空间复杂度为 $O(n)$ 、原地排序**：在输入数组完全倒序的情况下，达到最差递归深度 n ，使用 $O(n)$ 栈帧空间。排序操作是在原数组上进行的，未借助额外数组。
- **非稳定排序**：在哨兵划分的最后一步，基准数可能会被交换至相等元素的右侧。

11.5.3 快速排序为什么快

从名称上就能看出，快速排序在效率方面应该具有一定的优势。尽管快速排序的平均时间复杂度与“归并排序”和“堆排序”相同，但通常快速排序的效率更高，主要有以下原因。

- **出现最差情况的概率很低**：虽然快速排序的最差时间复杂度为 $O(n^2)$ ，没有归并排序稳定，但在绝大多数情况下，快速排序能在 $O(n \log n)$ 的时间复杂度下运行。
- **缓存使用效率高**：在执行哨兵划分操作时，系统可将整个子数组加载到缓存，因此访问元素的效率较高。而像“堆排序”这类算法需要跳跃式访问元素，从而缺乏这一特性。
- **复杂度的常数系数小**：在上述三种算法中，快速排序的比较、赋值、交换等操作的总数量最少。这与“插入排序”比“冒泡排序”更快的原因类似。

11.5.4 基准数优化

快速排序在某些输入下的时间效率可能降低。举一个极端例子，假设输入数组是完全倒序的，由于我们选择最左端元素作为基准数，那么在哨兵划分完成后，基准数被交换至数组最右端，导致左子数组长度为 $n - 1$ 、右子数组长度为 0。如此递归下去，每轮哨兵划分后都有一个子数组的长度为 0，分治策略失效，快速排序退化为“冒泡排序”的近似形式。

为了尽量避免这种情况发生，**我们可以优化哨兵划分中的基准数的选取策略。**例如，我们可以随机选取一个元素作为基准数。然而，如果运气不佳，每次都选到不理想的基准数，效率仍然不尽如人意。

需要注意的是，编程语言通常生成的是“伪随机数”。如果我们针对伪随机数序列构建一个特定的测试样例，那么快速排序的效率仍然可能劣化。

为了进一步改进，我们可以在数组中选取三个候选元素（通常为数组的首、尾、中点元素），**并将这三个候选元素的中位数作为基准数。**这样一来，基准数“既不太小也不太大”的概率将大幅提升。当然，我们还可以选取更多候选元素，以进一步提高算法的稳健性。采用这种方法后，时间复杂度劣化至 $O(n^2)$ 的概率大大降低。

示例代码如下：

Python

quick_sort.py

```
def median_three(self, nums: list[int], left: int, mid: int, right: int) -> int:
    """选取三个候选元素的中位数"""
    l, m, r = nums[left], nums[mid], nums[right]
    if (l <= m <= r) or (r <= m <= l):
        return mid # m 在 l 和 r 之间
    if (m <= l <= r) or (r <= l <= m):
        return left # l 在 m 和 r 之间
    return right

def partition(self, nums: list[int], left: int, right: int) -> int:
    """哨兵划分（三数取中值）"""
    # 以 nums[left] 为基准数
    med = self.median_three(nums, left, (left + right) // 2, right)
    # 将中位数交换至数组最左端
    nums[left], nums[med] = nums[med], nums[left]
    # 以 nums[left] 为基准数
    i, j = left, right
    while i < j:
        while i < j and nums[j] >= nums[left]:
            j -= 1 # 从右向左找首个小于基准数的元素
        while i < j and nums[i] <= nums[left]:
            i += 1 # 从左向右找首个大于基准数的元素
        # 元素交换
        nums[i], nums[j] = nums[j], nums[i]
    # 将基准数交换至两子数组的分界线
    nums[i], nums[left] = nums[left], nums[i]
    return i # 返回基准数的索引
```

11.5.5 尾递归优化

在某些输入下，快速排序可能占用空间较多。以完全有序的输入数组为例，设递归中的子数组长度为 m ，每轮哨兵划分操作都将产生长度为 0 的左子数组和长度为 $m - 1$ 的右子数组，这意味着每一层递归调用减少的问题规模非常小（只减少一个元素），递归树的高度会达到 $n - 1$ ，此时需要占用 $O(n)$ 大小的栈帧空间。

为了防止栈帧空间的累积，我们可以在每轮哨兵排序完成后，比较两个子数组的长度，**仅对较短的子数组进行递归**。由于较短子数组的长度不会超过 $n/2$ ，因此这种方法能确保递归深度不超过 $\log n$ ，从而将最差空间复杂度优化至 $O(\log n)$ 。代码如下所示：

Python

quick_sort.py

```
def quick_sort(self, nums: list[int], left: int, right: int):
    """快速排序（尾递归优化）"""
    # 子数组长度为 1 时终止
    while left < right:
        # 哨兵划分操作
        pivot = self.partition(nums, left, right)
        # 对两个子数组中较短的那个执行快速排序
        if pivot - left < right - pivot:
            self.quick_sort(nums, left, pivot - 1) # 递归排序左子数组
            left = pivot + 1 # 剩余未排序区间为 [pivot + 1, right]
        else:
            self.quick_sort(nums, pivot + 1, right) # 递归排序右子数组
            right = pivot - 1 # 剩余未排序区间为 [left, pivot - 1]
```

[上一页](#)



[11.4 插入排序](#)

[下一页](#)

[11.6 归并排序](#)

