快速排序（Quick Sort）算法

马浩琨

Aug 26, 2025

排序算法是计算机科学中最基础且重要的主题之一，它不仅在学术研究中占据核心地位，更是软件开发中日常使用的工具。在众多排序算法中，快速排序以其卓越的平均性能脱颖而出，成为实际应用中最广泛采用的算法之一。它的核心优势在于采用了「分而治之」的策略，并实现了原地排序，这意味着它不需要额外的存储空间，仅通过重新排列元素就能达到排序的目的。本文将系统性地解析快速排序的工作原理，逐步指导实现其代码，并深入探讨其性能特征，以帮助读者全面掌握这一经典算法。

1　算法核心思想：分而治之

快速排序的成功建立在「分而治之」这一强大的算法设计范式之上。该范式通过将复杂问题分解为多个相同类型的子问题来解决原始问题。具体来说，它包含三个主要步骤：首先是将问题划分为更小的子问题；其次是递归地解决这些子问题；最后是将子问题的解组合起来形成原问题的解。在快速排序的语境下，这一哲学得到了完美体现。算法的「分」阶段通过选择一个基准元素并将数组分割为两个部分来实现，其中左侧部分的所有元素都不大于基准，右侧部分的所有元素都不小于基准。随后的「治」阶段递归地对这两个子数组应用相同的快速排序过程。而「合」阶段则异常简单，由于排序是原地进行的，当递归完成时，整个数组已然有序，无需额外的合并操作。

2　关键步骤：分区策略详解

分区是整个快速排序算法的核心，其效率直接决定了算法的性能。该过程的目标是选择一个基准值，并重新排列数组，使得所有小于等于基准的元素位于其左侧，所有大于基准的元素位于其右侧，同时返回基准元素的最终位置索引。有多种分区策略，其中 Lomuto 分区方案因其实现简单而常被初学者采用。

在 Lomuto 方案中，通常选择数组最右侧的元素作为基准。算法维护一个指针 `i`，其初始位置为 `low - 1`，用于标记小于基准的子数组的末尾。随后，另一个指针 `j` 从 `low` 遍历至 `high - 1`。对于每个元素 `arr[j]`，如果其值小于或等于基准，则将 `i` 向右移动一位，并交换 `arr[i]` 和 `arr[j]` 的值。这一操作确保了 `i` 左侧的元素始终不大于基准。遍历完成后，基准元素仍位于数组末尾，此时将基准与 `i + 1` 位置的元素交换，使其就位于正确的位置。函数最终返回基准的索引 `i + 1`。

考虑数组 `[10, 80, 30, 90, 40, 50, 70]`，并选择最右侧的 `70` 作为基准。初始化时 `i` 为 `-1`，`j` 从 `0` 开始。当 `j` 指向 `10`，由于其小于 `70`，`i` 增至 `0` 并交换 `arr[0]` 和 `arr[0]`（无变化）。`j` 指向 `80` 时，因其大于 `70`，无操作。`j` 指向 `30` 时，`i` 增至 `1`，交换 `arr[1]`（`80`）和 `arr[2]`（`30`），数组变为 `[10, 30, 80, 90, 40, 50, 70]`。后续元素 `90`、`40`、`50` 中，只有 `40` 和 `50` 触发交换。最终，`i` 为 `4`，交换 `arr[5]`（`50`）与基准 `70`，得到分区后的数组 `[10, 30, 40, 50, 70, 90, 80]`，基准索引为 `4`。

3 从思路到代码：实现基本的快速排序

在理解了分区过程后，实现快速排序本身变得直观。算法是递归的，其基准情况是当子数组的长度为 0 或 1 时，即 low >= high，此时无需任何操作。否则，首先调用分区函数获取基准索引，然后递归地对基准左侧和右侧的子数组进行排序。

以下是使用 Python 实现的 Lomuto 分区方案和快速排序函数。分区函数 partition 接受数组 arr 及边界索引 low 和 high，返回基准的最终位置。主函数 quick_sort 则递归地应用这一过程。

```
def partition(arr, low, high):
    pivot = arr[high] # 选择最右侧元素作为基准
    i = low - 1 # 指向小于基准的子数组的末尾
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i] # 将较小元素交换到左侧
    arr[i + 1], arr[high] = arr[high], arr[i + 1] # 将基准放置到正确位置
    return i + 1
```

分区函数中，pivot 存储基准值。i 初始化为 low - 1，表示小于基准的区域尚未包含任何元素。循环变量 j 从 low 遍历至 high - 1，逐个检查元素。若当前元素 arr[j] 不大于基准，则扩展小于基准的区域（i 增加），并将该元素交换至区域末尾。循环结束后，i + 1 即为基准应处的位置，通过交换将其放置于此。

```
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high) # 获取基准索引
        quick_sort(arr, low, pi - 1) # 递归排序左半部分
        quick_sort(arr, pi + 1, high) # 递归排序右半部分
```

主排序函数首先检查子数组长度是否大于 1。若是，则调用 partition 进行分区，获取基准索引 pi。随后，递归地对基准左侧（low 至 pi - 1）和右侧（pi + 1 至 high）的子数组进行快速排序。由于是原地排序，递归完成后原始数组即已有序。

4 复杂度分析：它到底有多快?

快速排序的性能高度依赖于分区操作的质量。在理想情况下，每次分区都能将数组均分为两个大小相近的子数组，此时递归的深度为 依赖于分区操作的质量。在理想情况下,每次分区都能将数组均分为两个大小相近的子数组,此时递归的深度为\Theta(n),每一层需要进行\Theta(n)次比较,故最佳时间复杂度为\Theta(n\log n)$$，每一层需要进行 $\Theta(n)$ 次比较，故最佳时间复杂度为 $\Theta(n \log n)$。平均情况下，随机输入也能达到这一效率，这正是快速排序名称的由来。然而，在最坏情况下，每次分区极不均衡，例如当数组已经有序且始终选择边缘元素作为基准时，递归树退化为链状，深度为 $\Theta(n)$，每层仍需 $\Theta(n)$ 次操作，导致最坏时间复杂度为 $\Theta(n^2)$。尽管如此，通过简单优化（如随机选择基准），最坏情况可概率性地避免。

空间复杂度方面，由于是原地排序，不需要额外存储数据，但递归调用需要栈空间。递归深度平均为 $\Theta(\log n)$，故平均空间复杂度为 $\Theta(\log n)$。这与归并排序的 $\Theta(n)$ 额外空间相比更具优势。

5 优缺点与应用场景

快速排序的主要优点在于其优异的平均性能，时间复杂度 $\Theta(n \log n)$ 使其在处理大规模数据时效率显著。同时，原地排序的特性节省了内存空间。然而，它也存在缺点：最坏情况下的$\Theta(n^2)显著。同时，原地排序的特性节省了内存空间。然而，它也存在缺点：最坏情况下的\Theta(n^2)显著。同时，原地排序的
不过通过优化策略可 mitigate 这些问题。
在实际应用中，快速排序及其变体被广泛集成于编程语言的标准库中，如 Java 的 \texttt{Arrays.sort} 和 Python 的 \texttt{list.sort}。它特别适用于通用排序场景，其中数据随机分布且对稳定性无严格要求。
\section{优化思路简介}
为提高鲁棒性，几种优化策略常被采用。随机化快速排序通过随机选择基准元素来避免最坏情况，使得算法期望复杂度保持为 $\Theta(n \log n)$。三数取中法选择数组头、尾、中间元素的中位数作为基准，进一步减少分区不平衡的概率。对于小数组，快速排序的递归开销可能超过其效率优势，因此当子数组规模较小（如少于 10 元素）时，切换至插入排序等简单算法可提升整体性能。
快速排序凭借其分治策略和高效的平均性能，成为排序算法中的佼佼者。核心的分区操作将数组划分为较小和较大的两部分，递归应用后得到有序结果。通过代码实现，我们展示了如何将这一思想转化为实际算法。尽管存在最坏情况，但优化手段能有效应对。鼓励读者亲手实现算法，并通过测试加深理解。进一步学习可探索 Hoare 分区方案或其它高级排序算法。

6 附录：完整代码示例

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# 示例用法
```

```
example_arr = [10, 80, 30, 90, 40, 50, 70]
quick_sort(example_arr, 0, len(example_arr) - 1)
print(example_arr) # 输出排序后的数组
```

此代码实现了基本的快速排序。partition 函数完成分区操作，quick_sort 函数管理递归过程。示例数组排序后应输出 [10, 30, 40, 50, 70, 80, 90]。

7 互动与思考题

如何修改代码以实现降序排序？提示：仅需调整分区中的比较条件。挑战：实现随机选择基准的版本，以避免最坏情况。欢迎在评论区分享你的解决方案或提出疑问。