c13n #28

c13n

2025 年 11 月 19 日

第 I 部

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，原地排序的特性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。

第 II 部

循环缓冲区（Circular Buffer）数据结构

王思成

Aug 27, 2025

8　从原理到实践，掌握这一高效数据结构的核心与实现细节

在数据处理领域，先进先出（FIFO）队列是一种常见需求，例如在传送带系统或音乐播放器的播放队列中，数据需要按顺序处理。传统线性缓冲区，如普通数组或列表，在处理头部出队操作时面临显著问题：每次出队都会导致后续数据的大量移动，这不仅增加时间复杂度（通常为 $O(n)$），还可能造成「假溢出」现象，即数组前部有空闲空间却无法利用，从而降低空间利用率。这些缺陷在实时或资源受限环境中尤为突出。

循环缓冲区（或称环形缓冲区）作为一种高效解决方案，通过将线性空间逻辑上首尾相连，形成一个环形结构，巧妙避免了数据移动和空间浪费。它的应用广泛，包括多线程编程中的生产者-消费者模型（用于数据交换或日志缓冲）、网络数据包的接收与发送缓冲、音频视频处理中的数据流，以及嵌入式系统中资源高效管理。本文将深入探讨其原理，并以 C 语言实现一个基本版本，帮助读者从理论到实践全面掌握。

9　核心概念与工作原理

循环缓冲区是一种使用固定大小数组但逻辑上视为环形的数据结构。其核心组件包括底层存储数组 `buffer`、写指针 `head`（指示下一个可写入位置）、读指针 `tail`（指示下一个可读取位置）以及缓冲区容量 `capacity`。需要注意的是，为了区分空和满状态，通常实际可存储元素数为 `capacity - 1`，这是一种常见策略以避免歧义。

基本操作包括写入（put 或 enqueue）和读取（get 或 dequeue）。写入时，首先检查缓冲区是否已满；如果未满，则在 `head` 位置写入数据，然后将 `head` 指针向前移动一位，使用取模运算实现循环：`head = (head + 1) % capacity`。这里的取模运算 `%` 是关键，它确保指针在到达数组末尾时自动回绕到开头。类似地，读取时检查缓冲区是否为空；如果非空，则从 `tail` 位置读取数据，并将 `tail` 指针移动：`tail = (tail + 1) % capacity`。

判断空和满是循环缓冲区设计中的关键问题。常见方案包括三种：一是始终保持一个单元为空，空的条件是 `head == tail`，满的条件是 `(head + 1) % capacity == tail`，优点是逻辑简单高效，但牺牲一个存储单元；二是使用计数器 `count`，空时 `count == 0`，满时 `count == capacity`，优点是利用所有空间，但需维护额外变量；三是使用标志位如 `full_flag`，空时 `(head == tail) && !full`，满时 `full` 为真，需在操作中维护标志。本文选择方案一进行实现，因其经典且易于理解线程安全概念。

10　代码实现（以 C 语言为例，但思想通用）

首先，我们定义循环缓冲区的数据结构。使用 `struct` 来封装相关变量，包括指向缓冲区数组的指针、头尾指针和容量。代码如下：

```
typedef struct {
    int *buffer; // 指向缓冲区数组的指针，存储整数类型数据
    size_t head; // 写指针，表示下一个写入位置
    size_t tail; // 读指针，表示下一个读取位置
    size_t capacity; // 缓冲区总容量，注意实际可存储 capacity - 1 个元素
} circular_buf_t;
```

这段代码定义了一个名为 circular_buf_t 的结构体类型。buffer 是一个动态分配的整数数组指针，用于实际存储数据；head 和 tail 是 size_t 类型变量，分别跟踪写入和读取位置；capacity 表示缓冲区的最大容量。这种设计使得缓冲区大小在初始化时固定，确保内存使用可控。

接下来，我们设计 API 函数。包括初始化、销毁、写入、读取、判断空满和获取当前大小等函数。初始化函数 circular_buf_init 负责分配内存并设置初始状态：

```
circular_buf_t* circular_buf_init(size_t size) {
    circular_buf_t *cb = malloc(sizeof(circular_buf_t));
    if (cb == NULL) return NULL;
    cb->buffer = malloc(size * sizeof(int));
    if (cb->buffer == NULL) {
        free(cb);
        return NULL;
    }
    cb->head = 0;
    cb->tail = 0;
    cb->capacity = size;
    return cb;
}
```

此函数首先分配 circular_buf_t 结构体的内存，然后分配缓冲区数组的内存。如果任何分配失败，则清理并返回 NULL。初始化时，头尾指针都设置为 0，表示缓冲区为空。容量设置为输入参数 size，但注意实际可存储元素数为 size - 1。

销毁函数 circular_buf_free 用于释放资源：

```
void circular_buf_free(circular_buf_t *cb) {
    if (cb != NULL) {
        free(cb->buffer);
        free(cb);
    }
}
```

这个函数检查指针非空后，先释放缓冲区数组内存，再释放结构体内存，避免内存泄漏。

写入函数 circular_buf_put 实现数据添加：

```
int circular_buf_put(circular_buf_t *cb, int data) {
    if (circular_buf_full(cb)) {
        return -1; // 缓冲区已满，写入失败
    }
    cb->buffer[cb->head] = data;
    cb->head = (cb->head + 1) % cb->capacity;
    return 0; // 成功
}
```

函数首先调用 circular_buf_full 检查是否已满（满则返回错误）。如果未满，将数据写入 head 位置，然后更新 head 指针：(cb→head + 1) % cb→capacity。这里的取模运算确保指针循环，例如当 head 达到 capacity 时，会回绕到 0。

读取函数 circular_buf_get 实现数据提取：

```
int circular_buf_get(circular_buf_t *cb, int *data) {
    if (circular_buf_empty(cb)) {
        return -1; // 缓冲区为空，读取失败
    }
    *data = cb->buffer[cb->tail];
    cb->tail = (cb->tail + 1) % cb->capacity;
    return 0; // 成功
}
```

类似地，先检查空状态，然后从 tail 位置读取数据到输出参数 data，并更新 tail 指针。取模运算同样用于循环处理。

辅助函数包括判断空和满：

```
int circular_buf_empty(circular_buf_t *cb) {
    return cb->head == cb->tail;
}

int circular_buf_full(circular_buf_t *cb) {
    return (cb->head + 1) % cb->capacity == cb->tail;
}
```

circular_buf_empty 直接比较头尾指针是否相等；circular_buf_full 检查 head 的下一个位置是否等于 tail，由于使用方案一，满时总会有一个单元空闲。

获取当前数据量的函数 circular_buf_size 计算已存储元素数：

```
size_t circular_buf_size(circular_buf_t *cb) {
    if (cb->head >= cb->tail) {
        return cb->head - cb->tail;
    } else {
        return cb->capacity - cb->tail + cb->head;
    }
}
```

这个函数处理头尾指针的相对位置：如果 head 大于或等于 tail，大小 simply 为 head - tail；否则，大小为 capacity - tail + head，accounting for the wrap-around。例如，如果 capacity 为 5，head 为 2，tail 为 4，则大小为 3（计算为 5 - 4 + 2 = 3）。对于扩展性，读者可以修改代码支持泛型数据，例如使用 void* 指针和元素大小参数，但这会增加复杂度，本文专注于基本整数类型以保持简洁。

11 边界情况与优化技巧

在实现循环缓冲区时，边界情况需特别注意。首先，线程安全性是一个重要问题：上述基础实现是非线程安全的，如果在多线程环境中使用，可能导致数据竞争。例如，生产者和消费者线程同时访问共享缓冲区时，需通过互斥锁或原子操作来同步。简单加锁方式是在每个操作前后加锁和解锁，但这可能影响性能；无锁编程则更复杂，涉及原子指令，本文不深入讨论。

批量操作是另一种优化方向。实现 `put_n` 和 `get_n` 函数可以一次性处理多个数据，减少函数调用开销。思路是计算连续可用空间，可能分两段进行内存拷贝。例如，在写入多个数据时，先检查从 `head` 到数组末尾的连续空间，然后处理回绕部分，但需注意边界检查以避免溢出。

动态扩容通常不是循环缓冲区的设计目标，因为其优势在于固定大小带来的确定性和效率。然而，如果需要，可以实现扩容逻辑：当缓冲区满时，分配更大数组，复制现有数据并调整指针。但这会引入复杂性，如数据复制成本和指针重定位，可能违背循环缓冲区的初衷。因此，在大多数场景下，建议预先规划足够容量。

循环缓冲区 offers 显著优点：操作时间复杂度为 $O(1)$，非常高效；内存使用预分配且可控，适合资源受限环境；尤其适用于 FIFO 队列和数据流缓冲。然而，它也有缺点：固定容量需提前规划，可能不够灵活；基础实现非线程安全，需额外处理；空满判断逻辑需小心实现以避免错误。

鼓励读者亲自实现并测试这个数据结构，在实践中加深理解。下一步可以探索并发版本或应用于具体项目，如网络编程或嵌入式系统。

12 附录/延伸阅读

完整代码可参考 GitHub 仓库（提供链接），包含可编译运行的示例。编写单元测试至关重要，测试案例应包括空满状态切换、指针回绕场景和批量操作验证。与其他数据结构如链式队列或动态数组相比，循环缓冲区在固定大小和性能关键场景中表现优异，但链式队列更灵活于动态扩容，动态数组则可能更适合随机访问。深入阅读推荐操作系统或并发编程相关书籍，以了解更多高级应用。

第 III 部

归并排序（Merge Sort）算法

杨其臻

Aug 29, 2025

排序算法是计算机科学中最基础且重要的研究领域之一，它不仅是编程入门的关键课题，更是评估算法效率与设计思想的经典案例。在众多排序算法中，归并排序凭借其稳定的 $O(n \log n)$ 时间复杂度以及典型的分治策略，成为了理论和实践中不可或缺的一部分。本文将引导您不仅实现归并排序，更深入理解其背后的分治哲学和运作机制。

13 归并排序的核心思想：分而治之

归并排序的核心是“分治”策略，这是一种将复杂问题分解为多个相同或相似的子问题，再递归解决子问题，最后合并子问题解以得到原问题解的方法。想象一下组织一场大型晚会：您不会试图一次性管理所有细节，而是将任务分解为场地、节目、餐饮等子任务，分别处理后再整合成果。归并排序正是如此运作，它通过递归将数组不断二分，直到每个子数组只含一个元素（自然有序），再通过合并操作将这些有序片段组装成更大的有序数组。

具体而言，归并排序遵循三步战略：分解、解决和合并。分解阶段将数组递归地分成两个尽可能等长的子数组；解决阶段递归排序这些子数组；合并阶段则将两个已排序的子数组合并为一个有序数组。整个算法的效率与正确性高度依赖于合并过程的实现。

14 深入剖析：合并两个有序数组的过程

合并两个有序数组是归并排序的灵魂所在。假设有两个已排序数组 `A` 和 `B`，目标是将它们合并为新的有序数组 `C`。这一过程通过三个指针高效完成：指针 `i` 遍历 `A`，指针 `j` 遍历 `B`，指针 `k` 指向 `C` 的当前写入位置。

合并从比较 `A[i]` 和 `B[j]` 开始，将较小者放入 `C[k]`，并移动相应指针。此过程循环直至任一数组被完全遍历，最后将另一数组的剩余元素直接追加至 `C` 末尾。这种策略确保了合并操作的时间复杂度为 $O(n)$，其中 n 是两个子数组的长度之和。

以下是一个 Python 的 `merge` 函数实现，它清晰展示了这一过程：

```
def merge(left, right):
    # 初始化结果列表和指针
    result = []
    i = j = 0
    # 循环比较并合并
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # 将剩余元素追加到结果中
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

这段代码首先比较左右数组的当前元素，选择较小者加入结果，并移动指针。循环结束后，任一数组的剩余部分直接被并入，保证了结果的完整性与有序性。

15 从思路到代码：完整的归并排序实现

基于分治思想与合并操作，归并排序的递归实现变得直观。主函数 merge_sort 首先处理递归终止条件 —— 当数组长度不大于 1 时直接返回（因为单元素数组自然有序）。否则，计算中点将数组分为左右两半，递归排序左右子数组，最后调用 merge 合并结果。
以下是 Python 的完整实现，代码注释与上述步骤一一对应：

```
def merge_sort(arr):
    # 递归终止条件：数组长度为 0 或 1
    if len(arr) <= 1:
        return arr
    # 找到中点，分解数组
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    # 合并排序后的子数组
    return merge(left, right)
```

递归深度为 $O(\log n)$，每层合并操作总时间为 $O(n)$，因此整体效率稳定。Java 的实现类似，但需处理类型声明和数组拷贝，此处略去以保持简洁。

16 复杂度分析：它为什么高效?

归并排序的时间复杂度分析可通过递归树直观理解。递归树高度为 $O(\log n)$，每层需处理 $O(n)$ 元素，故总时间为 $O(n \log n)$。这一效率在最好、最坏和平均情况下均保持一致，体现了算法的稳定性。
空间复杂度主要来自合并所需的临时数组，每层递归需 $O(n)$ 空间。由于递归深度为 $O(\log n)$，但同一时刻最大空间使用量为 $O(n)$，故归并排序的空间复杂度为 $O(n)$。与快速排序等原地排序算法相比，这是归并排序的主要缺点，但也换来了稳定性和可预测性。
归并排序的优点显著：时间复杂度稳定在 $O(n \log n)$，适用于大规模数据；它是一种稳定排序，即相等元素的相对顺序在排序后保持不变；此外，在处理链表结构时，归并排序可优化空间使用。缺点则包括需要 $O(n)$ 额外空间，以及递归调用带来的开销，对于小规模数据，简单排序如插入排序可能更高效。

17 实战与应用

归并排序的思想远超排序本身。例如，求解逆序对问题可通过修改合并过程高效实现，统计在合并过程中右侧元素小于左侧元素的次数。现实中，归并排序的理念广泛应用于大数据处理（如 MapReduce 中的排序阶段）和编程语言基础库（如 Java 的 Arrays.sort() 和 Python 的 sorted() 在对象排序中使用变体 TimSort）。

归并排序通过分治策略和高效合并，实现了稳定且高效的排序。理解其思想不仅助于掌握算法本身，更提升了解决复杂问题的能力。未来可探索优化方向，如对小数组使用插入排序减少递归开销，或实现迭代版本避免递归深度限制。

18 互动与思考

您能尝试用迭代方式实现归并排序吗？欢迎在评论区分享您的代码或提出疑问，共同探讨排序算法的更多奥秘。

第 IV 部

堆排序（Heap Sort）算法

马浩琨

Aug 31, 2025

排序算法在计算机科学中占据着核心地位，它们是数据处理和算法设计的基础。在众多排序算法中，如快速排序和归并排序，堆排序以其独特的优势脱颖而出。堆排序的最大亮点在于其时间复杂度在任何情况下都能保持 $O(n \log n)$ 的优异性能，没有最坏情况下的退化问题。此外，堆排序是一种原地排序算法，仅需常数 $O(1)$ 的额外空间，这使得它在内存受限的环境中非常有用。堆排序还特别适合解决 Top-K 问题，例如查找前 K 个最大或最小元素。本文将深入剖析堆排序的原理，并指导读者从头开始实现它。

19 预备知识：什么是“堆”？

堆是一种特殊的完全二叉树，它具有两种主要类型：大顶堆和小顶堆。大顶堆的性质是每个节点的值都大于或等于其子节点的值，即对于任意节点 i，有 arr[parent(i)] $\geq$ arr$[i]$，其中根节点是整个树的最大值。小顶堆则相反，每个节点的值都小于或等于其子节点的值，根节点是最小值。堆排序通常使用大顶堆来进行升序排序。堆可以用数组高效地表示一个完全二叉树，利用数组索引与树节点位置的对应关系。对于下标为 i 的节点（从 0 开始），父节点下标为 parent$(i) = \lfloor (i-1)/2 \rfloor$，左孩子下标为 left_child$(i) = 2 \times i + 1$，右孩子下标为 right_child$(i) = 2 \times i + 2$。这种表示方式使得堆的操作可以在数组上高效进行。

20 堆排序的核心思想与算法步骤

堆排序的核心思想是不断从堆顶取出最大元素，放到数组末尾，并重新调整堆结构。这个过程分为两个主要步骤：构建初始大顶堆和反复交换与调整。首先，构建初始大顶堆是将给定的无序数组调整成一个最大堆。其次，反复交换与调整 involves 将堆顶元素（最大值）与当前未排序部分的最后一个元素交换，然后减小堆的大小，并对新的堆顶元素执行堆化操作以重新使其成为有效的大顶堆。重复此过程，直到堆中只剩一个元素，此时数组已完全排序。

21 核心操作详解：Heapify（堆化）

Heapify 是堆排序中的关键操作，它确保以某个节点为根的子树满足堆性质。这个过程的前提是该节点的左右子树都已经是堆。对于大顶堆，Heapify 的过程如下：首先，从当前节点 i、左孩子 l 和右孩子 r 中找出值最大的节点，记为 largest。如果 largest 不等于 i，说明当前节点不满足堆性质，需要交换 arr[i] 和 arr[largest]。交换后，可能破坏了下一级子树的结构，因此需要递归地对 largest 指向的子树调用 Heapify。这个过程的时间复杂度为 $O(\log n)$，因为最坏情况下需要从根遍历到叶子。

22 从零开始实现堆排序

为了实现堆排序，我们需要定义几个函数：主函数 heap_sort(arr)、辅助函数 heapify(arr, n, i) 和 build_max_heap(arr)。heapify 函数负责对大小为 n 的堆，从索引 i 开始堆化。build_max_heap 函数则构建初始堆，关键点是从最后一个非叶子节点开始，自底向上地调用 heapify。最后一个非叶子节点的索引是 n//2 - 1，因为叶子节点本身可以看作是合法的堆。

以下是使用 Python 实现的完整代码示例。我们将详细解读每个部分。

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def build_max_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

def heap_sort(arr):
    n = len(arr)
    build_max_heap(arr)
    for i in range(n-1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
```

在 heapify 函数中，我们首先假设当前节点 i 是最大的，然后比较其左右孩子（如果存在）来更新 largest。如果 largest 发生变化，就交换元素并递归调用 heapify。build_max_heap 函数通过从最后一个非叶子节点开始逆向遍历，确保整个数组被构建成堆。heap_sort 函数先构建堆，然后通过循环交换堆顶元素到末尾，并调整堆，最终完成排序。

23　复杂度与特性分析

堆排序的时间复杂度分析显示，heapify 操作的时间复杂度为 $O(\log n)$，build_max_heap 函数经过精细分析可证明是 $O(n)$，而不是直观的 $O(n \log n)$。排序循环执行 n-1 次，每次调用 heapify，因此为 $O(n \log n)$。总时间复杂度为 $O(n) + O(n \log n) = O(n \log n)$。空间复杂度方面，堆排序是原地排序，如果使用迭代实现 heapify（可优化递归），则空间复杂度为 $O(1)$。稳定性方面，堆排序是不稳定的排序算法，例如在数组 [5a, 5b, 3] 中，排序后 5a 和 5b 的相对顺序可能改变。

堆排序的优点包括最坏情况下仍为 $O(n \log n)$ 的时间复杂度和原地排序的特性，但缺点是不稳定、常数项较大，在实际中通常比快速排序慢一些，且缓存局部性较差。堆排序在 Top-K 问题和优先级队列中有广泛应用。未来，读者可以探索标准库中的堆实现，如

Python 的 heapq 模块，以及堆的其他变体和应用。

24 附录：常见问题（Q&A）

为什么构建堆要从最后一个非叶子节点开始？这是因为叶子节点本身已经是合法的堆，从最后一个非叶子节点开始可以确保在调用 heapify 时，子树已经满足堆性质。如何使用小顶堆进行降序排序？只需将 heapify 中的比较逻辑反转，并调整构建和排序过程。堆排序和快速排序哪个更快？在实际中，快速排序通常更快 due to better cache performance，但堆排序在最坏情况下更可靠。堆排序为什么是不稳定的？因为交换操作可能改变相同元素的相对顺序，例如在交换堆顶和末尾元素时。

第 V 部

插入排序（Insertion Sort）算法

黄京

Sep 01, 2025

在编程和算法学习中，排序算法是基础且至关重要的部分。插入排序作为一种简单直观的算法，常常是初学者入门的第一选择。想象一下，您在整理一副扑克牌时，通常会一张一张地拿起牌，并将其插入到手中有序牌堆的适当位置。这个过程正是插入排序的核心思想。通过学习插入排序，您不仅能理解排序的基本概念，还能为后续学习更复杂的算法打下坚实基础。本文将带领您深入理解插入排序的原理，亲手实现它，并分析其性能特点。

25 算法核心思想：像整理扑克牌一样排序

插入排序的灵感来源于日常生活中的卡片整理过程。算法将待排序的数组分为两个部分：已排序区间和未排序区间。初始时，已排序区间只包含第一个元素，其余元素都属于未排序区间。然后，算法逐个从未排序区间取出元素，并将其插入到已排序区间的正确位置，通过从后向前扫描已排序区间来找到插入点。这个过程重复进行，直到未排序区间为空，数组完全有序。这种分而治之的视角使得算法易于理解和实现。

26 分步拆解：可视化排序过程

让我们通过一个具体数组示例来可视化插入排序的过程。考虑数组 [5，2，4，6，1，3]。初始状态时，已排序区间包含第一个元素 5，未排序区间包含其余元素。第一轮处理未排序区间的第一个元素 2，将其与已排序区间的 5 比较，由于 2 小于 5，我们将 5 向后移动，并将 2 插入到正确位置，得到 [2，5，4，6，1，3]。第二轮处理元素 4，它比 5 小但比 2 大，因此插入到 2 和 5 之间，数组变为 [2，4，5，6，1，3]。第三轮处理 6，它比已排序区间的所有元素都大，因此直接留在末尾，数组为 [2，4，5，6，1，3]。第四轮处理 1，这是一个关键步骤，因为它需要多次比较和移动：从后向前扫描，1 比 6、5、4、2 都小，因此这些元素依次向后移动，最后 1 插入到开头，数组变为 [1，2，4，5，6，3]。第五轮处理 3，它插入到 2 和 4 之间，最终得到有序数组 [1，2，3，4，5，6]。这个过程清晰地展示了算法如何逐步构建有序序列。

27 算法实现：手把手编码

我们将使用 Python 语言来实现插入排序，因为其语法简洁，易于理解。以下是基础版本的代码：

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

现在，让我们逐行解析这段代码的逻辑。外层循环 `for i in range(1, len(arr))` 从索

引 1 开始遍历数组，因为索引 0 的元素被视为初始已排序区间。变量 `i` 代表当前待处理元素的索引。 Inside the loop, we assign `key = arr[i]`, which is the element we are about to insert into the sorted region. Then, we set `j = i - 1` to point to the last element of the sorted region. The inner loop `while j >= 0 and key < arr[j]` is where the actual comparison and shifting happen: as long as `j` is within bounds and `key` is less than the element at `j`, we shift `arr[j]` to the right by assigning `arr[j + 1] = arr[j]`, and decrement `j` to move backwards. Once the loop exits, we have found the correct position for `key`, and we insert it with `arr[j + 1] = key`. This process ensures that each element is placed in its proper place in the sorted region.

28 算法分析：优点、缺点与适用场景

插入排序的时间复杂度分析显示，在最坏情况下，当数组完全逆序时，每个新元素都需要比较和移动所有已排序元素，导致时间复杂度为 $O(n^2)$。在最好情况下，如果数组已经有序，每个元素只需比较一次，时间复杂度为 $O(n)$，这是一个显著的优点。平均情况下，时间复杂度仍为 $O(n^2)$。空间复杂度方面，算法只使用常数级别的额外变量，如 `key` 和 `j`，因此是原地排序，空间复杂度为 $O(1)$。稳定性方面，插入排序是稳定排序，因为当遇到相等元素时，内层循环条件 `key < arr[j]` 不成立，循环停止，`key` 被插入到相等元素的后面，保持了相对顺序。优点包括实现简单、对于小规模或基本有序数据高效、空间效率高和稳定性。缺点是在大规模无序数据上效率低下。适用场景主要包括数据量较小（例如 n ⩽ 50）、数据基本有序，或作为高级排序算法（如快速排序）的子过程来处理小区间。

29 优化方向

一个常见的优化方向是使用二分查找来改进插入排序。思路是在已排序区间中使用二分查找快速定位插入位置，将查找时间从 $O(n)$ 降低到 $O(\log(n))$。然而，由于元素移动操作仍然是 $O(n)$，整体时间复杂度保持为 $O(n^2)$，但常数因子减小，在实际应用中可能带来性能提升。这可以作为进一步学习的主题。

通过本文，我们深入探讨了插入排序算法的核心思想、实现步骤和性能分析。插入排序通过构建有序序列并逐个插入元素来实现排序，其简单性和对小规模数据的效率使其成为算法学习的重要起点。理解插入排序有助于培养算法思维，并为学习更高级算法（如希尔排序或归并排序）奠定基础。

30 互动与练习

为了巩固学习，我鼓励您尝试一个挑战问题：不用临时变量 `key` 来实现插入排序，并思考这可能带来的问题，例如数据覆盖或逻辑错误。请在评论区分享您的代码或疑问，我很乐意与您交流。如果您想深入学习，推荐阅读关于其他排序算法的文章，如冒泡排序或选择排序。