深入理解并实现基本的堆排序（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。

1　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 $\text{arr}[\text{parent}(i)] \geq \text{arr}[i]$，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 $\text{parent}(i) = \lfloor (i-1)/2 \rfloor$，左孩子下标为 $\text{left\_child}(i) = 2 \times i + 1$，右孩子下标为 $\text{right\_child}(i) = 2 \times i + 2$。这种表示方式使得堆的操作可以在数组上高效进行。

2　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。

3　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。

4 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。

5 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。

6 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。