计数排序（Counting Sort）算法

马浩琨

Aug 15, 2025

排序算法作为计算机科学的核心基础之一，通常分为两大类别：比较排序与非比较排序。传统比较排序如快速排序、归并排序等，其时间复杂度下限为 $O(n \log n)$，这一理论极限由决策树模型所证明。然而当我们面对特定数据类型时，非比较排序算法能够突破这一界限，实现线性时间复杂度。计数排序正是这样一种独特的算法，其在最优情况下时间复杂度可达 $O(n+k)$，其中 k 表示数据范围大小。这种特性使其在整数排序、数据范围有限且稳定性要求高的场景中大放异彩，例如年龄统计、考试分数排名等实际应用场景。

1　计数排序核心思想

计数排序的核心原理在于利用桶思想直接统计元素出现频次，通过频次信息重建有序序列，完全规避了元素间的比较操作。该算法有两个关键前提：待排序数据必须为整数，且数据范围必须已知或可提前确定。当处理 [4，2，0，1，3，4，1] 这类数据时，算法会创建索引 0 到 4 的计数桶。稳定性在此算法中尤为重要，它能确保相同元素的原始相对顺序得以保留，这对多关键字排序等场景至关重要。

2　算法步骤拆解

我们以数组 [4，2，0，1，3，4，1] 为例（数据范围 0~4），详细拆解计数排序的五个关键步骤。首先确定数据范围：最大值 max=4，最小值 min=0，则范围长度 k=5。接着初始化计数数组 count[0..4]，所有元素初始值为 0。第三步遍历原始数组统计频次，得到 count = [1，2，1，1，2]，表示数字 0 出现 1 次，数字 1 出现 2 次，依此类推。

第四步是计算累加频次：将计数数组转换为 count = [1，3，4，5，7]。这一步的数学意义在于 count[i] 表示值小于等于 i 的元素总数，为后续定位提供依据。最后进行反向填充：从原数组末尾向前遍历，根据 count 数组确定每个元素在输出数组中的位置。以最后一个元素 1 为例，查询 count[1] = 3 表示应放在索引 2 的位置（3-1=2），放置后 count[1] 减 1 更新为 2。反向遍历确保相同元素的顺序稳定不变。

3　代码实现

```
def counting_sort(arr):
    if len(arr) == 0:
        return arr

    # 确定数据范围（支持负数处理）
```

```
    max_val, min_val = max(arr), min(arr)
    k = max_val - min_val + 1

    # 初始化计数数组与输出数组
    count = [0] * k
    output = [0] * len(arr)

    # 统计每个元素的出现频次
    for num in arr:
        count[num - min_val] += 1 # 偏移量处理负数

    # 计算累加频次：count[i]表示≤ i 的元素总数
    for i in range(1, k):
        count[i] += count[i-1]

    # 反向填充保证稳定性
    for i in range(len(arr)-1, -1, -1):
        num = arr[i]
        # 计算元素在输出数组中的正确位置
        pos = count[num - min_val] - 1 # 转换为 0-based 索引
        output[pos] = num
        count[num - min_val] -= 1 # 更新计数器

    return output
```

在代码实现中，三个关键设计点值得关注。首先通过 `min_val` 偏移量处理负数：当元素为负值时，`num - min_val` 将其映射到非负索引区间。其次累加频次计算 `count[i] += count[i-1]` 将频次统计转换为位置信息，`count[i]` 表示所有小于等于 `i` 的元素总数。最重要的是反向填充机制：从数组末尾向前遍历，结合 `count` 数组确定位置后立即更新计数器，确保相同元素维持原始顺序。该实现时间复杂度为 $O(n+k)$，其中 n 为元素数量，k 为数据范围大小。

4 算法特性深度分析

计数排序的时间复杂度在最优、最差和平均情况下均为 $O(n+k)$，当 $k=O(n)$ 时达到线性复杂度。空间复杂度为 $O(n+k)$，包含输出数组的 $O(n)$ 和计数数组的 $O(k)$。稳定性是该算法的显著优势，通过反向填充严格保证相同元素的相对位置不变。与其他排序算法对比：快速排序虽平均 $O(n\log n)$ 但不稳定；归并排序稳定但需要 $O(n)$ 额外空间；桶排序同样线性但要求数据均匀分布。计数排序在小范围整数排序场景中具有显著性能优势。

5 优化技巧与边界处理

面对不同数据特征，计数排序有多种优化策略。范围压缩技术通过 `min_val` 偏移减少桶数量，如处理 `[-100, 100]` 范围时，使用偏移量只需 201 个桶而非 201 个。当桶数量过大（如 $k > 10^6$）时，应改用快速排序等算法避免空间浪费。特殊场景适配包括负数处理（代码已实现）和浮点数处理（缩放取整但损失精度）。边界情况需单独处理：空数组直接返回；单元素数组无需排序；全相同元素数组仍正常执行但计数数组仅单个桶有值。

6 实际应用场景

计数排序在现实中有诸多高效应用案例。成绩排名系统处理 0~100 分数据时，只需 101 个桶即可线性完成百万级数据排序。人口年龄统计中，0~120 岁范围同样适用。作为基数排序的子过程，它负责单一位的稳定排序。海量数据预处理时，可结合分治策略先用计数排序处理数据块。这些场景共同特点是数据范围有限且为整数。

7 局限性讨论

计数排序有两个主要局限：仅适用于整数排序，浮点数需近似处理会损失精度；数据范围过大时空间效率骤降，例如处理 `[1, 10^9]` 范围需要十亿级桶。不适用场景包括字符串排序（应改用桶排序或基数排序）和范围未知的大整数集合。当 k 远大于 n 时，空间浪费严重，时间复杂度退化为 $O(k)$。

8 扩展：计数排序的变种

计数排序存在多个实用变种。前缀和优化版直接使用累加计数替代二次遍历，但实现更复杂。原地计数排序通过元素交换减少空间占用，但牺牲稳定性。在数据分析领域，计数数组本身可作为频率直方图，直接展示数据分布特征，无需完整排序过程。

计数排序在特定场景下展现出无可比拟的效率优势，其核心价值在于利用空间换时间策略实现线性排序。选择原则需权衡数据范围 k 与数据量 n 的关系：当 $k = O(n)$ 时是最佳选择。作为非比较排序的经典案例，它深刻揭示了算法设计中空间与时间的辩证关系，是理解桶排序、基数排序等高级算法的重要基础。