# 基本的桶排序（Bucket Sort）算法

黄梓淳

Sep 03, 2025

排序算法在计算机科学中占据核心地位，它是许多应用程序的基础。常见的比较排序算法如快速排序和归并排序，其时间复杂度下限为 $O(n \log n)$，这意味着在最坏情况下，排序 n 个元素至少需要这么多时间。然而，桶排序（Bucket Sort）算法在某些特定场景下可以突破这个限制，达到线性时间复杂度 $O(n)$，这使其在处理大规模数据时极具吸引力。桶排序的价值在于其对数据分布的强依赖性，它巧妙地运用了“用空间换时间”和“分治”思想。本文将带领读者从零开始，全面理解桶排序的原理、实现、性能及其应用场景，旨在帮助您掌握这一高效算法。

## 1　算法核心思想：化整为零，分而治之

桶排序的核心思想可以用一个简单的比喻来理解：想象您需要整理一堆大小不一的书籍。传统比较排序方法就像一本一本地比较书籍的厚薄，然后进行排列；而桶排序则先准备几个书架（称为桶），每个书架标记特定的范围（如「A-C」、「D-F」等，这对应于映射规则），然后将书籍按书名首字母放入对应书架（分桶），再对每个书架内的少量书籍进行排序（子排序），最后按书架顺序合并所有书籍（合并）。这种方法通过化整为零、分而治之的策略，大幅提高了排序效率。

正式定义下，桶（Bucket）是一个容器，通常是数组或链表，用于存放处于特定区间的数据。映射函数（Mapping Function）是关键组件，它决定每个元素应该放入哪个桶，其设计直接影响算法效率。桶排序的基本步骤包括：设置空桶、分散入桶（Scatter）、各桶排序和依次收集（Gather）。这些步骤共同确保了算法在理想情况下的高效性。

## 2　一步一步看桶排序

让我们通过一个具体示例来逐步理解桶排序的过程。假设我们有一个数组 [29，25，3，49，9，37，21，43]，数据范围在 [0，50) 之间。首先，我们创建 5 个桶，每个桶负责一个区间，例如 [0,10)、[10,20)、[20,30)、[30,40) 和 [40,50)。这一步对应于设置空桶。

接下来，我们遍历数组，将每个元素通过映射函数放入对应的桶中。例如，数字 29 落入区间 [20,30)，因此放入第二个桶；数字 25 也落入同一桶；数字 3 落入 [0,10) 桶；以此类推。完成分散入桶后，每个桶可能包含多个元素，例如 [20,30) 桶有 [29，25]。

然后，我们对每个非空桶内的元素进行排序。这里，我们可以使用简单的排序算法如插入排序。例如，对 [20,30) 桶排序后，得到 [25，29]；对 [0,10) 桶排序后，保持 [3，9] 有序。其他桶类似处理。

最后，我们按桶的顺序（从 [0,10) 到 [40,50)）依次取出所有元素，合并成一个有序数组。结果是 [3，9，21，25，29，37，43，49]。这个过程展示了桶排序如何通过分阶段处理来高效完成排序。

## 3 关键细节与实现

实现桶排序时，关键细节包括选择桶的数量（k）和区间范围。通常，桶的数量 k 可以设置为数组长度 n，或者根据数据分布调整。区间范围应均匀覆盖所有数据，计算公式为 `range = (max - min) / k`，其中 max 和 min 是数组的最大值和最小值。映射函数的设计至关重要，核心公式为 `index = (num - min) * k / (max - min + 1)`，这将数值 num 映射到 `[0, k-1]` 的索引范围内，确保正确落入桶。注意处理边界情况，例如当 num 等于 max 时，索引可能超出范围，需要通过取整或调整来避免。

下面是一个 Python 实现桶排序的代码示例。我们将逐步解读代码，以帮助理解每个部分对应算法的哪个环节。

```
def bucket_sort(arr):
    # 计算数组的最大值和最小值
    min_val = min(arr)
    max_val = max(arr)
    n = len(arr)
    # 设置桶的数量，这里使用数组长度 n
    k = n
    # 初始化桶：创建一个列表的列表，每个子列表代表一个桶
    buckets = [[] for _ in range(k)]

    # 分散入桶：遍历数组，将每个元素放入对应桶
    for num in arr:
        # 计算索引：使用映射函数，注意处理除零和边界
        if max_val == min_val:
            index = 0 # 所有元素相同，放入第一个桶
        else:
            # 公式 : index = (num - min_val) * k / (max_val - min_val + 1)
            index = int((num - min_val) * k / (max_val - min_val + 1))
        buckets[index].append(num)

    # 各桶排序：对每个桶内的元素进行排序（这里使用内置排序函数）
    for i in range(k):
        buckets[i] = sorted(buckets[i])

    # 依次收集：合并所有桶中的元素
    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(bucket)

    return sorted_arr
```

```
31
# 示例使用
33 arr = [29, 25, 3, 49, 9, 37, 21, 43]
sorted_arr = bucket_sort(arr)
35 print("排序后的数组␣:", sorted_arr)
```

代码解读：首先，我们计算数组的最小值和最小值，以确定数据范围。然后，初始化 k 个空桶，这里 k 设置为数组长度 n，这是一种常见选择，以确保桶的数量足够。在分散入桶阶段，我们使用映射函数计算每个元素应放入的桶索引；公式中的 +1 是为了避免除零错误并处理边界。之后，对每个桶调用内置排序函数（如 sorted），这简化了实现，但强调了核心逻辑；在实际应用中，如果桶内元素少，可以使用插入排序以提高效率。最后，合并所有桶得到有序数组。这个实现展示了桶排序的完整流程，但请注意，它假设数据分布相对均匀，否则性能可能下降。

## 4 算法分析

桶排序的时间复杂度分析显示其高效性。在最佳情况下，当数据均匀分布时，时间复杂度为 $O(n+k)$，其中 n 是元素数量，k 是桶数量。分散和收集步骤各为 $O(n)$，而子排序步骤由于每个桶元素数量平均为 $n/k$，总时间为 $O(k \cdot (n/k) \log(n/k))$，简化后约为 $O(n \log(n/k))$。当 k 接近 n 时，这接近 $O(n)$，实现线性时间。平均情况通常接近最佳情况。然而，在最坏情况下，如果所有数据集中在一个桶内，算法退化为单个桶的排序，时间复杂度可能达到 $O(n^2)$，例如使用插入排序时。

空间复杂度为 $O(n+k)$，因为需要额外空间存储 k 个桶，且所有桶总共容纳 n 个元素。这体现了“以空间换时间”的策略。桶排序是稳定的排序算法，稳定性取决于入桶时保持顺序（如使用尾部插入）和桶内排序使用稳定算法（如插入排序）。在我们的实现中，由于使用列表的 append 和 sorted（Python 的 sorted 是稳定的），稳定性得以保证。

## 5 优缺点与适用场景

桶排序的优点包括：在数据分布均匀且桶数量设置合理时，效率极高，可达线性时间；它是稳定的排序算法；思想简单，易于理解和实现。这些特点使其在特定场景下非常有用。然而，桶排序也有明显缺点：严重依赖于数据分布，如果数据集中在一个桶内，效率会急剧下降；需要额外的内存空间，不适合内存受限的环境；并且不适合处理离散性很强或范围未知的数据。

典型适用场景包括数据均匀分布在一个已知区间内，例如排序浮点数或在外部排序中处理大规模数据。桶排序还常作为其他算法的子过程，如基数排序。在实际应用中，应根据数据特性谨慎选择桶排序，以发挥其最大优势。回顾桶排序的核心思想，它通过「分桶-排序-合并」的策略，实现了高效排序。其成功关键在于数据分布依赖性和空间换时间的特性。鼓励读者动手实现桶排序，并尝试不同数据分布来观察性能变化，从而深化理解。未来，我们可能会探讨基数排序等 related 算法，以扩展排序知识。

## 6 互动与思考题

如果您数据分布极度不均匀，有哪些方法可以优化桶排序？例如，可以动态调整桶的数量或使用自适应映射函数。桶排序和基数排序有什么联系和区别？两者都基于分桶思想，但基数排序按 digit 分桶，而桶排序按值范围

分桶。您能设想一个桶排序在现实生活中的具体应用例子吗？比如排序学生成绩基于分数段。欢迎在评论区分享您的想法！