深入理解并实现基本的基数排序（Radix Sort）算法

杨其臻

Nov 03, 2025

在计算机科学中，排序算法是基础且关键的主题。我们熟知的基于比较的排序算法，如快速排序和归并排序，其时间复杂度下界为 $O(n \log n)$，这意味着它们无法在比较模型下突破这一限制。然而，是否存在一种算法能够超越这一界限，在某些场景下实现更高效的排序呢？答案是肯定的 —— 基数排序作为一种非比较型整数排序算法，能够在处理特定数据时达到 $O(k \times n)$ 的时间复杂度，其中 k 表示数字的最大位数。本文旨在深入解析基数排序的原理、实现方式及其性能，帮助读者从理论到实践全面掌握这一算法。

1　理解基数排序的核心思想

基数排序的核心在于“基数”这一概念。基数指的是进制的基数，例如十进制数的基数为 10，二进制数的基数为 2。算法通过按位排序来实现整体有序，排序顺序可以从最低有效位到最高有效位（LSD 方式），或反之（MSD 方式）。这种方法的巧妙之处在于，它避免了直接比较元素大小，而是通过分配和收集操作来逐步排序。
为了直观理解基数排序，让我们以一个简单的十进制数字数组为例，如 `[170，45，75，90，802，24，2，66]`。排序过程从最低位（个位）开始，将数字分配到对应的桶中（0 到 9），然后按桶顺序收集回数组。接着处理十位数，重复分配和收集操作，最后处理百位数。每一轮排序都必须保持稳定性，即相等元素的相对顺序在排序后不变。稳定性是基数排序正确性的关键，因为如果某一轮排序不稳定，之前位数的排序信息可能会丢失，导致最终结果错误。

2　基数排序的两种实现方式

基数排序主要有两种实现方式：最低位优先（LSD）和最高位优先（MSD）。LSD 方式从数字的最低位开始排序，逐步向最高位推进。这种方式直观且易于实现，因为每一轮排序都基于前一轮的结果，通过稳定排序确保高位权重更大时能够覆盖低位的顺序。LSD 的正确性依赖于稳定排序的累积效应，最终实现整体有序。
相比之下，MSD 方式从最高位开始排序，并递归地对每个桶中的数字处理次高位。MSD 更类似于分治策略，可能在早期就将数据分割成小块，从而减少后续处理次数。然而，MSD 的实现较为复杂，需要处理递归和边界情况，通常更适用于字符串字典序排序等场景。尽管 LSD 和 MSD 都基于位的分配和收集，但它们在处理顺序和效率上各有侧重，读者可以根据具体需求选择适合的方式。

3　手把手实现 LSD 基数排序

LSD 基数排序的实现可以分为几个清晰步骤。首先，需要找到数组中的最大数字，以确定排序的轮数（即最大位数）。其次，初始化十个桶，对应数字 0 到 9。然后，从最低位开始，遍历每一位进行分配和收集操作：分配阶

段将每个元素根据当前位数字放入对应桶中，收集阶段按桶顺序将元素取回数组。重复这一过程，直到处理完最高位。

以下是一个 Python 实现示例，我们将逐步解读代码关键部分。

```
def radix_sort(arr):
    # 步骤 1: 找到最大值，确定最大位数
    max_num = max(arr)
    exp = 1 # 起始位: 个位

    while max_num // exp > 0:
        # 步骤 2: 初始化 10 个桶
        buckets = [[] for _ in range(10)]

        # 步骤 3a: 分配
        for num in arr:
            digit = (num // exp) % 10
            buckets[digit].append(num)

        # 步骤 3b: 收集
        arr_index = 0
        for bucket in buckets:
            for num in bucket:
                arr[arr_index] = num
                arr_index += 1

        # 移动到下一位
        exp *= 10

# 测试代码
if __name__ == "__main__":
    data = [170, 45, 75, 90, 802, 24, 2, 66]
    radix_sort(data)
    print("排序后的数组␣:", data)
```

在这段代码中，首先通过 max(arr) 获取数组最大值，从而确定需要处理的位数。变量 exp 初始化为 1，表示从个位开始。循环条件 max_num // exp > 0 确保在所有位数处理完毕前持续迭代。在每一轮中，初始化十个空桶用于存储数字。分配阶段使用 (num // exp) % 10 计算当前位数字，这通过整数除法和取模操作实现，例如当 exp 为 1 时，(num // 1) % 10 得到个位数；当 exp 为 10 时，得到十位数。数字被添加到对应桶中，这里使用列表作为桶，天然保证了稳定性，因为元素按添加顺序存储。收集阶段遍历所有桶，按顺序将元素放回原数组，索引 arr_index 用于跟踪数组位置。最后，exp 乘以 10 移动到下一位。测试部分演示了算法对示例数组的排序效果，输出应为有序数组。

4 深入分析与探讨

基数排序的时间复杂度为 $O(k \times n)$，其中 k 是最大数字的位数，n 是数组长度。每一轮分配和收集操作各需 $O(n)$ 时间，总共进行 k 轮。与基于比较的排序算法如快速排序的 $O(n \log n)$ 相比，当 k 小于 $\log n$ 时，基数排序可能更高效；但如果数字范围极大（k 很大），效率可能下降。空间复杂度为 $O(n + r)$，其中 r 是基数（这里为 10），因为需要额外空间存储桶和元素。

基数排序的优点包括线性时间复杂度和稳定性，适用于固定位数的整数或字符串排序。然而，它并非原地排序，需要额外内存，且仅适用于可分割为位的数据类型。优化方面，可以选择不同基数（如 256 进制）以利用位运算加速，或使用链表优化桶操作减少数据搬移。

基数排序以其非比较的独特思想，在排序算法家族中占据重要地位。通过按位分配和收集，它实现了高效排序，特别适合处理手机号、身份证号等固定位数数据。理解并实现基数排序，不仅能扩展算法知识，还能在特定场景下提升应用性能。读者可以尝试用其他语言实现，或探索 MSD 版本以加深理解。

5 互动与思考题

动手实现基数排序是巩固知识的好方法，读者可以尝试用 Java 或 C++ 重写代码，或实现 MSD 版本。思考题包括：如何修改算法以处理负数？对于长度不一的字符串排序，该如何调整？为什么在现实应用中快速排序更常见？这些问题有助于进一步探索算法的边界和优化方向。