基本的基数排序（Radix Sort）算法

叶家炜

Oct 24, 2025

在计算机科学中，排序算法是基础且重要的主题。大多数常见的排序算法，如快速排序或归并排序，都基于元素之间的比较操作。这些比较排序算法的时间复杂度下限为 $O(n \log n)$，其中 n 是元素数量。然而，在某些场景下，例如对大量整数进行排序时，比较排序可能不是最高效的选择。基数排序作为一种非比较型整数排序算法，在特定条件下能够实现线性时间复杂度 $O(n \times k)$，其中 k 是数字的最大位数。本文将深入解析基数排序的核心思想、实现细节及其应用场景。

想象一下，您需要快速整理一叠按出生年月日记录的卡片。如果使用基于比较的方法，您可能需要反复对比不同卡片的日期，这个过程在数据量较大时会变得低效。基数排序通过一种独特的方式解决了这个问题：它不直接比较元素的大小，而是依据数字的每一位进行多轮排序。这种方法在处理整数数据时，尤其当数字的位数相对较少时，能够显著提升效率。基数排序的核心优势在于其非比较特性，这使得它能够突破比较排序的理论下限，在合适条件下实现近乎线性的性能。

1　第一部分：基数排序的核心思想

基数排序的核心思想是逐位排序。这类似于整理扑克牌时，先按花色分组，再在每个组内按点数排序。对于数字，基数排序会从最低位或最高位开始，依次对每一位进行排序。关键概念包括基数「Radix」和键「Key」。基数指的是每一位数字的取值范围，例如对于十进制数，基数为十，对应零到九。键则是排序所依据的特征，即数字的每一位。基数排序主要有两种方法：最低位优先「LSD」和最高位优先「MSD」。LSD 方法从数字的最低位（如个位）开始排序，逐步向高位推进，这种方法实现简单且广泛应用。MSD 方法则从最高位开始，采用分治策略递归处理，虽然可能在某些情况下提前结束，但实现较为复杂。本文将重点介绍 LSD 基数排序，因为它更易于理解和实现。

2　第二部分：LSD 基数排序的逐步拆解

为了清晰展示 LSD 基数排序的过程，我们以一个具体数组为例：[170, 45, 75, 90, 2, 802, 2, 66]。排序过程从最低位开始，逐位进行多轮排序。每一轮排序都必须保持稳定性，即相同键值的元素在排序后保持原有相对顺序。首先，我们找出数组中最大数字的位数，这里是三位（802）。然后，我们从个位开始排序：创建十个桶，对应数字零到九，将每个数字按其个位数放入对应桶中，例如 170 的个位是零，放入零号桶；45 的个位是五，放入五号桶。之后，按桶号顺序收集所有元素，得到新数组。接下来，对十位进行同样操作：基于上一轮结果，将数字按十位数放入桶中并收集。最后，对百位进行排序。经过这三轮后，数组变为有序状态。整个过程强调稳定性，例如在个位排序中，两个数字二（2 和 2）保持原顺序，确保最终结果的正确性。算法步骤可总结为：首先确定最大位数，然后从最低位到最高位循环，每轮执行分配和收集操作。分配阶段将元素按当前位数字放入对

应桶，收集阶段按桶顺序取回元素。

3 第三部分：手把手实现基数排序（代码篇）

我们将使用 Python 语言实现基数排序，因为它语法清晰，易于理解。实现过程包括两个辅助函数和一个核心函数。首先，定义辅助函数 get_digit(num, place)，用于获取数字在指定位置上的数字。例如，对于数字 170，get_digit(170, 0) 返回个位数字零，get_digit(170, 1) 返回十位数字七。该函数的实现基于数学公式：(abs(num) // (10 ** place)) % 10。这里，abs(num) 取绝对值以确保处理正数，// 表示整数除法，** 表示幂运算，% 取模得到当前位数字。另一个辅助函数 get_max_digit_count(nums) 用于计算数组中最大数字的位数。它先找到数组中的最大值，然后通过循环除以十来计数位数，例如对于 802，循环三次后得到三。

核心函数 radix_sort(nums) 负责执行排序。首先处理边界情况，如果数组为空或只有一个元素，直接返回。然后，调用 get_max_digit_count 获取最大位数。接下来，循环最大位数次，每次循环对应一位数字。在循环内，初始化十个空桶，用于存放数字。分配阶段遍历数组，使用 get_digit 获取当前位数字，将元素放入对应桶中。收集阶段清空原数组，然后按桶号顺序将桶中元素放回数组。最后，返回排序后的数组。代码测试时，使用示例数组 [170, 45, 75, 90, 2, 802, 2, 66]，打印每轮结果以验证正确性。例如，第一轮按个位排序后，数组可能变为 [170, 90, 2, 802, 2, 45, 75, 66]，其中个位数字有序。

```
def get_digit(num, place):
    return (abs(num) // (10 ** place)) % 10

def get_max_digit_count(nums):
    if not nums:
        return 0
    max_num = max(nums)
    count = 0
    while max_num != 0:
        count += 1
        max_num //= 10
    return count

def radix_sort(nums):
    if len(nums) <= 1:
        return nums
    max_digit_count = get_max_digit_count(nums)
    for k in range(max_digit_count):
        buckets = [[] for _ in range(10)]
        for num in nums:
            digit = get_digit(num, k)
            buckets[digit].append(num)
```

```
        nums = []
        for bucket in buckets:
            nums.extend(bucket)
    return nums
```

在代码解读中，get_digit 函数通过数学运算精确提取指定位的数字，确保处理各种整数。get_max_digit_count 函数使用循环计算位数，避免了对数运算可能带来的精度问题。核心函数 radix_sort 通过嵌套循环实现多轮排序，外层循环控制位数，内层循环处理分配和收集。分配阶段利用列表推导创建桶，收集阶段使用 extend 方法高效合并元素。整个实现强调代码的可读性和效率，同时确保稳定性。

4 第四部分：深入分析与探讨

基数排序的时间复杂度为 $O(n \times k)$，其中 n 是元素数量，k 是最大数字的位数。这是因为外层循环执行 k 次，内层的分配和收集操作各为 $O(n)$。空间复杂度为 $O(n + r)$，其中 r 是基数（十进制下为十），因为需要额外存储 r 个桶和桶内的 n 个元素。基数排序是稳定的排序算法，这意味着在排序过程中，相同键值的元素保持原有顺序，这对于多关键字排序至关重要。例如，如果先按十位排序，再按个位排序，稳定性确保十位相同的元素在个位排序后仍保持正确顺序。

基数排序的优点包括：当 k 远小于 n 时，效率高于 $O(n \log n)$ 的比较排序；并且它是稳定的。缺点在于：它不是原地排序，需要额外空间；通常只适用于整数或可表示为整数的类型；如果数字范围很大（k 很大）而 n 较小，效率可能不如比较排序。与其他算法相比，基数排序在整数排序中具有独特优势。例如，与快速排序相比，基数排序在特定条件下性能更优，但快排是原地排序且通用性强。与计数排序相比，基数排序通过多轮排序解决了计数排序在数字范围大时空间消耗高的问题。

处理负数时，上述实现需要扩展。我们可以将数组分为正数和负数两部分。对负数部分取绝对值，进行基数排序后反转结果；对正数部分直接排序；最后合并两部分。这确保了负数能正确排序，同时保持算法稳定性。

基数排序通过逐位排序的方式，实现了对整数的高效排序。本文详细介绍了 LSD 基数排序的核心思想、实现步骤和复杂度分析，强调了稳定性和线性时间复杂度的前提条件。读者在理解后，可以尝试实现 MSD 基数排序或探索其他基数（如二进制）以优化性能。基数排序在现实中有广泛应用，例如早期卡片排序机和现代大数据处理。鼓励读者动手实现代码，并思考如何扩展至更复杂的数据类型。通过深入理解基数排序，我们不仅掌握了一种高效算法，更提升了解决实际问题的能力。