深入理解并实现基本的基数排序（Radix Sort）算法

马浩琨

Oct 06, 2025

假设我们面对一个简单的问题：如何对数组 [170，45，75，90，802，24，2，66] 进行排序？许多读者可能会立刻联想到快速排序、归并排序等经典的比较排序算法。这些算法通过直接比较元素的大小来决定它们的顺序，但它们在理论上的时间复杂度下界是 $O(n \log n)$，这意味着在最坏情况下，排序 n 个元素至少需要与 $n \log n$ 成正比的时间。然而，是否存在一种方法能够突破这个下界呢？答案是肯定的，基数排序就是一种非比较排序算法，它不依赖于元素之间的直接比较，而是通过逐位处理来实现排序。在特定条件下，基数排序可以达到线性时间复杂度 $O(n \cdot k)$，其中 k 是数字的最大位数。本文的目标是深入解析基数排序的核心思想，详细说明其工作流程，并提供可实现的代码示例，同时分析其优缺点和适用场景，帮助读者全面掌握这一算法。

1　基数排序的核心思想解析

基数排序的核心思想在于逐位排序。它将待排序元素视为由多个“位”组成的序列，例如整数可以分解为个位、十位、百位等。算法从最低有效位开始，依次对每一位进行排序，直到最高有效位。这一过程的关键在于每次排序必须是稳定的，即如果两个元素在某一位上相等，排序后它们的相对顺序保持不变。稳定性是基数排序能够正确工作的基石，因为它确保了高位排序的成果不会被低位的排序破坏。例如，假设我们先按十位排序数组 [21，11，22，12]，得到 [11，21，12，22]，其中 11 在 21 之前，12 在 22 之前。如果后续按个位排序时不稳定，11 和 21 的相对顺序可能被打乱，导致最终结果错误。稳定性保证了在逐位排序过程中，先前排序的顺序得以保留。

一个形象的比喻是整理扑克牌。想象你有多张扑克牌，需要先按花色排序，再按点数排序。首先，你将所有牌按花色分成四堆，然后在不打乱每堆内部顺序的前提下，再按点数排序。这个“保持原有顺序”的过程正是稳定性的体现。基数排序也类似，它是一种多关键字排序方法，通过逐位处理来构建最终的有序序列。

2　算法流程详解：以 LSD 为例

基数排序通常有两种实现方式：最低有效位优先和最高有效位优先。这里我们以最低有效位优先为例进行详细说明。假设我们使用示例数组 [170，45，75，90，802，24，2，66]。首先，我们需要找到数组中的最大值，以确定排序的轮数。最大值是 802，它有 3 位数字，因此我们需要进行 3 轮排序，分别对应个位、十位和百位。

第一轮排序针对个位。我们创建 10 个桶，对应数字 0 到 9。将每个数字按其个位数放入对应的桶中，例如 170 的个位是 0，放入 0 号桶；45 的个位是 5，放入 5 号桶。完成分配后，我们按顺序从桶 0 到桶 9 收集数字，形成新的数组。此时，数组按个位有序，但整体可能仍未排序。

第二轮排序针对十位。我们对上一轮得到的新数组，根据十位数进行分配。需要注意的是，对于位数不足的数字，如 2，其十位视为 0。在分配过程中，稳定性至关重要。例如，在个位排序后的数组中，170 和 90 的十位

都是 7，稳定性确保了 170 依然在 90 之前。分配完成后，再次按顺序收集数字。
第三轮排序针对百位。同样地，我们根据百位数进行分配和收集。经过这三轮排序，数组最终完全有序。整个过程通过逐位处理，利用稳定性保证了排序的正确性，而无需直接比较元素大小。

3 代码实现：以 Python 为例

下面我们提供一个基数排序的 Python 实现代码，并详细解读每一步。该代码针对非负整数设计，后续我们会讨论如何处理负数。

```
def radix_sort(arr):
    if len(arr) < 2:
        return arr

    max_val = max(arr)
    exp = 1

    while max_val // exp > 0:
        buckets = [[] for _ in range(10)]

        for num in arr:
            digit = (num // exp) % 10
            buckets[digit].append(num)

        arr_index = 0
        for bucket in buckets:
            for num in bucket:
                arr[arr_index] = num
                arr_index += 1

        exp *= 10

    return arr
```

首先，函数检查数组长度是否小于 2，如果是，则直接返回，因为单个元素或空数组已经有序。接下来，找到数组中的最大值 `max_val`，以确定排序的轮数。变量 `exp` 初始化为 1，代表当前处理的位数（从个位开始）。
在循环中，只要 `max_val // exp` 大于 0，就继续排序。每一轮循环中，我们初始化 10 个空桶，对应数字 0 到 9。然后遍历数组，对每个数字计算当前位的值，使用表达式 `(num // exp) % 10`。例如，当 `exp` 为 1 时，这计算个位；当 `exp` 为 10 时，计算十位。数字被放入对应的桶中。
收集过程按桶的顺序（0 到 9）进行，将每个桶中的数字依次放回原数组。这一步保证了排序的稳定性，因为桶内元素的顺序保持不变。最后，`exp` 乘以 10，移动到下一位，循环继续直到处理完所有位。
对于负数的处理，我们可以将数组分成负数和非负数两部分。对负数部分取绝对值，进行基数排序后反转顺序；

对非负数部分直接排序；最后合并两部分。这扩展了算法的适用性，但需要额外注意边界情况。

4 算法分析

基数排序的时间复杂度为 $O(k \cdot n)$，其中 k 是最大数字的位数，n 是数组长度。算法需要进行 k 轮排序，每轮包括分配和收集两个步骤，每个步骤的时间复杂度为 $O(n)$，因此总时间为 $O(k \cdot n)$。当 k 远小于 n 时，基数排序的性能优于比较排序的 $O(n \log n)$ 下界。

空间复杂度为 $O(n + r)$，其中 r 是基数的大小（这里 $r = 10$）。算法需要额外的空间来存储 r 个桶和桶中的 n 个元素。基数排序是稳定的排序算法，这在多关键字排序中尤为重要。

5 优缺点与应用场景

基数排序的主要优点在于其速度，尤其在数据范围不大且数据量巨大时，它可以实现线性时间复杂度。此外，它的稳定性使其适用于需要保持相对顺序的场景。然而，基数排序也有明显的缺点：它需要额外的内存空间，且对数据格式有严格要求，通常只适用于整数或可以分解为“位”的结构。如果数据范围很大（k 很大），效率会显著下降。

在实际应用中，基数排序常用于对电话号码、身份证号等固定位数的数字进行排序。它还在后缀数组构造和计算机图形学中的某些算法中发挥作用。选择基数排序时，需权衡其线性时间优势与空间开销及数据限制。

6 进阶与变种

除了最低有效位优先的实现，基数排序还有最高有效位优先的变种。最高有效位优先从最高位开始排序，通常需要递归处理，可能在中间就完成排序，但实现更复杂且不稳定。另一种变种是改变基数，例如使用 2 的幂次（如 256 进制）作为基数，这可以通过位运算快速获取“位”，但会增加桶的数量，影响空间效率。这些进阶内容为算法优化提供了更多可能性。

基数排序通过逐位排序和稳定性的结合，实现了在特定条件下的线性时间复杂度。它的核心思想简单而强大，但适用场景有限。读者在理解本文内容后，可以尝试亲手实现代码，并扩展处理负数的情况，以加深对非比较排序的理解。基数排序虽然不是万能算法，但在合适的数据集上，它无疑是一把高效的排序利器。