

# 基本的基数排序 (Radix Sort) 算法

杨子凡

Nov 13, 2025

在数据处理领域，排序是一项基础而关键的操作。当我们面对海量数据时，例如为成千上万的手机号码排序，传统的比较排序算法如快速排序或归并排序虽然高效，但它们的时间复杂度存在一个理论下限  $O(n \log n)$ 。这自然引出一个问题：是否存在一种不通过比较就能实现排序的算法？答案是肯定的，基数排序正是这样一种非比较型的整数排序算法。它的核心思想是将整数按位数「切割」，逐位进行排序，在特定条件下时间复杂度甚至可以达到  $O(n)$ 。本文旨在带领读者彻底理解基数排序的原理，掌握其手动实现方法，并清晰认识其优缺点与适用场景。

## 1 核心思想与原理

基数排序的命名来源于「基数」这一概念。基数指的是进制的基数，例如在十进制中，基数为 10，这意味着我们需要 10 个「桶」来分别存放数字 0 到 9。关键码是排序所依据的属性，对于整数排序而言，关键码就是数字本身。稳定性是排序算法的一个重要特性，它表示如果两个元素在排序前相等，那么排序后它们的相对顺序保持不变。在基数排序中，稳定性至关重要，因为后序位的排序不能打乱前序位已经排好的顺序。

基数排序有两种主要实现方式：最低位优先 (LSD) 和最高位优先 (MSD)。LSD 从最低位（如个位）开始排序，依次向最高位进行，实现简单直观，是本文主要讲解的方式。MSD 则从最高位开始，然后递归地对每个桶内的数据进行下一位排序，更像一种分治策略，但实现稍显复杂。通过对比，LSD 因其易于理解和实现，常被用作入门教学的首选。

## 2 逐步拆解 LSD 基数排序过程

让我们通过一个具体示例来逐步拆解 LSD 基数排序的过程。假设我们有数组 [170, 45, 75, 90, 2, 802, 2, 66]。首先，我们需要找到数组中的最大数字，以确定最大位数。这里最大数字是 802，有 3 位，因此我们需要进行 3 轮排序。

第一轮从最低位（个位）开始。我们创建 10 个桶，对应数字 0 到 9。然后遍历数组，根据每个数字的个位数字将其放入对应的桶中。例如，170 的个位是 0，放入桶 0；45 的个位是 5，放入桶 5；以此类推。分配完成后，我们按桶号 0 到 9 的顺序依次收集元素放回数组。此时，数组按个位数字排序，结果为 [170, 90, 2, 802, 2, 45, 75, 66]。

第二轮处理十位数字。同样，创建 10 个桶，根据十位数字分配元素。例如，170 的十位是 7，放入桶 7；90 的十位是 9，放入桶 9。由于排序是稳定的，个位相同的数字在十位排序时会保持相对顺序。收集后，数组按十位和个位排序，结果为 [2, 802, 2, 45, 66, 170, 75, 90]。

第三轮处理百位数字，过程相同。完成后，整个数组有序，最终结果为 [2, 2, 45, 66, 75, 90, 170, 802]。这个示例清晰地展示了 LSD 基数排序的逐位排序过程，强调了稳定性在保持顺序中的作用。

### 3 动手实现基数排序

以下是用 Python 实现 LSD 基数排序的代码。我们将逐步解释关键部分，确保读者能够理解每一行代码的作用。

```
1 def radix_sort(arr):
2     # 步骤一：寻找最大值与最大位数
3     max_num = max(arr)
4     max_digit = len(str(max_num)) # 通过转换为字符串获取位数
5
6     # 步骤二：核心排序循环
7     for digit in range(max_digit):
8         # 创建 10 个桶
9         buckets = [[] for _ in range(10)]
10
11         # 分配过程：根据当前位数字将元素放入对应桶
12         for num in arr:
13             current_digit = (num // (10 ** digit)) % 10
14             buckets[current_digit].append(num)
15
16         # 收集过程：按顺序将桶中元素放回数组
17         arr = []
18         for bucket in buckets:
19             arr.extend(bucket)
20
21     return arr
```

现在，让我们详细解读这段代码。在步骤一中，我们使用 `max(arr)` 找到数组中的最大值 `max_num`，然后通过 `len(str(max_num))` 计算其位数 `max_digit`。这里，将数字转换为字符串后取长度是一种简单直观的方法，用于确定排序的轮数。

在核心循环中，`digit` 从 0 到 `max_digit-1` 迭代，表示当前处理的位数（0 表示个位，1 表示十位，以此类推）。对于每一轮，我们使用列表推导式创建 10 个空桶 `buckets`。

分配过程中，对于每个数字 `num`，我们计算当前位的数字。表达式 `(num // (10 ** digit)) % 10` 是关键：当 `digit=0`（个位）时，`10 ** 0` 等于 1，`num // 1` 仍是 `num`，然后 `% 10` 得到个位数字；当 `digit=1`（十位）时，`10 ** 1` 等于 10，`num // 10` 去掉个位，然后 `% 10` 得到十位数字。这样，我们就能准确提取指定位的数字值。

收集过程时，我们初始化一个新数组 `arr`，然后按桶号 0 到 9 的顺序，使用 `extend` 方法将每个桶中的元素依次添加回数组。由于 `extend` 保持元素顺序，且桶是按数字顺序创建的，这确保了排序的稳定性。最终，函数返回排序后的数组。

## 4 深入分析与探讨

基数排序的性能分析是理解其优势的关键。设待排序元素个数为  $n$ , 最大位数为  $k$ , 基数为  $r$  (在十进制中  $r = 10$ )。每一轮分配需要遍历所有元素, 时间复杂度为  $O(n)$ , 收集同样需要  $O(n)$  时间。由于有  $k$  轮, 总时间复杂度为  $O(k \times n)$ 。当  $k$  远小于  $n$  时, 性能接近线性, 表现出高效性。

空间复杂度方面, 我们需要额外的  $O(n + r)$  空间来存储桶和元素。这是一种典型的以空间换时间策略, 在内存充足的情况下值得采用。

基数排序的优点包括时间复杂度可能达到  $O(n)$ , 且是稳定排序。缺点是非原地排序, 需要额外空间, 且适用范围有限, 通常只适用于整数或可表示为整数的类型。如果最大位数  $k$  很大, 效率会显著下降。

与其他排序算法相比, 基数排序在特定场景下优势明显。例如, 与快速排序相比, 基数排序稳定且对数据分布不敏感; 与计数排序相比, 基数排序通过分治按位处理, 避免了数据范围大时计数排序的空间消耗问题。

## 5 扩展与变种

标准 LSD 基数排序无法直接处理负数。一个常见的解决方案是将数组拆分为正数和负数两部分。对负数部分取绝对值进行排序, 然后反转顺序 (因为负数绝对值越大, 实际值越小), 再与正数部分合并。这样可以扩展算法的适用性。

MSD 基数排序从最高位开始, 递归排序, 适用于某些场景, 如字符串排序, 但实现更复杂。对于字符串, 可以按字符的 ASCII 码逐位排序, 原理类似整数排序。

基数排序也可以应用于其他数据类型, 如日期, 只要能将它们转换为整数序列。例如, 日期可以表示为年月日的数字组合, 然后按位排序。

本文详细介绍了基数排序的核心思想、LSD 实现步骤、性能分析和扩展应用。基数排序作为一种非比较排序算法, 在特定条件下展现出高效性, 尤其适用于整数排序场景。通过手动实现代码, 读者可以更深入地理解其工作原理。鼓励读者在实践中探索基数排序的更多应用, 例如在大数据处理或数据库索引中。

## 6 附录与思考题

在附录中, 我们提出几个思考题供读者进一步探索。首先, 考虑如何使用队列数据结构来优化桶的实现, 使得收集过程更自然。其次, 尝试修改代码以处理包含负数的数组。最后, 如果待排序数字的范围已知且较小, 可以探索用计数排序替代每轮的桶排序, 以优化性能。这些练习有助于加深对基数排序的理解和应用。