基本的基数排序（Radix Sort）算法

黄梓淳

Sep 15, 2025

排序算法是计算机科学中的基础主题，传统上我们依赖于比较操作，如快速排序或归并排序，它们通过元素间的比较来确定顺序。但有一个问题：排序一定要通过两两比较吗？答案是否定的。基数排序（Radix Sort）提供了一种全新的视角，它是一种非比较型整数排序算法，基于键值的各位数字或字符进行逐位处理。这种方法的核心思想是稳定地按位排序，从而避免了许多比较操作带来的开销。

基数排序的核心价值在于其线性时间复杂度，通常表示为 $O(n \times k)$，其中 n 是元素个数，k 是最大位数。这使得它在处理大规模固定长度数据时表现出色，例如排序手机号码、学号或 IP 地址。在本文中，我将带领您深入理解基数排序的原理，亲手实现它，并分析其性能与适用边界。通过这篇文章，您将不仅学会如何编码，还能 grasp 其背后的思想。

1　算法核心原理剖析

基数排序的基本思想是逐位处理数字，从最低位（LSD，Least Significant Digit）或最高位（MSD，Most Significant Digit）开始排序。一个经典的类比是扑克牌排序：假设您有一副牌，您可能先按花色排序，再按数字排序，但为了保持顺序，需要确保排序是稳定的。稳定性意味着相同键值的元素在排序后保持原有相对顺序，这对基数排序至关重要，因为高位排序时不能打乱低位已排好的顺序。

在 LSD 方法中，我们从最低位（如个位）开始，逐步向高位推进。例如，考虑数组 `[170，45，75，90，802，24，2，66]`。首先，我们找到最大数字的位数（这里是 3，因为 802 有三位）。然后，进行三轮排序：第一轮按个位排序，第二轮按十位，第三轮按百位。每一轮使用一个稳定的排序算法（通常是计数排序）来处理当前位。可视化过程如下：初始数组为 `[170，45，75，90，802，24，2，66]`；按个位排序后，顺序变为 `[170，90，802，2，24，45，75，66]`；按十位排序后，变为 `[802，2，24，45，66，170，75，90]`；最后按百位排序，得到最终有序数组 `[2，24，45，66，75，90，170，802]`。这个过程展示了如何通过逐位稳定排序达到整体有序。

2　实现细节与代码解析

在实现基数排序时，我们选择计数排序作为辅助算法，因为它具有稳定性和线性时间复杂度，完美契合基数排序的需求。计数排序的核心是统计每个数字出现的次数，并通过累积计数来确定元素位置。下面，我将使用 Python 语言逐步实现 LSD 基数排序，并对代码进行详细解读。

首先，我们需要两个辅助函数：一个用于获取数组中的最大数字的位数，另一个用于获取数字在特定位上的数字。函数 `getMaxDigits` 遍历数组，找到最大数字并计算其位数。这通过将数字转换为字符串并取长度来实现，或者通过数学运算如连续除以 10。函数 `getDigit` 则提取数字在指定位上的数字，例如对于数字 123 和位索引

1（从右向左，0-based），它返回十位数字 2。

核心函数 radixSort 的实现步骤如下：计算最大位数 k，然后循环 k 次（从最低位到最高位）。在每一轮循环中，初始化一个大小为 10 的计数数组（因为十进制数字范围是 0-9）。接着，进行计数阶段：遍历数组，统计当前位上每个数字出现的次数。然后，将计数数组转换为累积计数数组，这有助于确定每个数字的最终位置。最后，重构数组：从后向前遍历原数组，根据当前位数字和计数数组，将元素放入临时输出数组的正确位置。从后向前遍历是为了保持稳定性，确保相同数字的元素顺序不变。完成后，将输出数组复制回原数组。

以下是 Python 代码实现：

```
def getMaxDigits(arr):
    # 找到数组中的最大数字
    max_val = max(arr)
    # 计算最大数字的位数：通过转换为字符串取长度
    return len(str(max_val))

def getDigit(num, digit):
    # 获取数字在指定位上的数字，digit 从 0 开始（0 表示个位）
    # 例如，getDigit(123, 1) 返回 2（十位）
    return (num // (10 ** digit)) % 10

def radixSort(arr):
    # 获取最大位数
    k = getMaxDigits(arr)
    # 临时输出数组
    output = [0] * len(arr)
    # 进行 k 轮排序
    for digit in range(k):
        # 初始化计数数组，大小为 10（0-9）
        count = [0] * 10
        # 计数阶段：统计当前位上每个数字的出现次数
        for num in arr:
            d = getDigit(num, digit)
            count[d] += 1
        # 累加计数：将计数数组转换为累积计数
        for i in range(1, 10):
            count[i] += count[i-1]
        # 重构数组：从后向前遍历原数组，以保持稳定性
        for i in range(len(arr)-1, -1, -1):
            num = arr[i]
            d = getDigit(num, digit)
            output[count[d] - 1] = num
```

```
            count[d] -= 1
        # 将输出数组复制回原数组
        arr = output[:]
    return arr
```

代码解读：在 getMaxDigits 函数中，我们使用 max 函数找到最大值，然后通过 len(str(max_val)) 计算位数，这是一种简单直接的方法。在 getDigit 函数中，我们使用整数除法和模运算来提取特定位上的数字，例如 (num // (10 ** digit)) % 10 计算数字在 digit 位上的值。在 radixSort 函数中，外层循环运行 k 次，对应每位排序。计数数组 count 初始化为全零，然后遍历数组统计数字出现次数。累加计数步骤将 count 数组转换为每个数字的结束索引加一。重构数组时，从后向前遍历原数组，确保稳定性：将元素放入输出数组的指定位置，并递减计数。最后，复制输出数组回原数组，完成排序。这个实现的时间复杂度为 $O(n \times k)$，空间复杂度为 $O(n + 10)$，由于 10 是常数，通常简化为 $O(n)$。

3 进阶讨论与变体

基数排序的复杂度分析显示，时间复杂度为 $O(n \times k)$，其中 n 是元素个数，k 是最大位数。由于 k 通常相对较小（例如，对于 32 位整数，k 最大为 10），这可以被视为线性时间复杂度，优于许多比较排序算法如快速排序的 $O(n \log n)$。空间复杂度为 $O(n + b)$，b 是基数大小（这里 b=10），主要开销来自输出数组和计数数组。
LSD 和 MSD 是基数排序的两种变体。LSD 从最低位开始排序，实现简单，适用于位数较少的数，但必须完成所有位的排序。MSD 从最高位开始，类似递归的桶排序，可能不需要比较所有位（如果高位已能区分大小），但实现更复杂，需要处理递归开销和空桶，适用于字符串排序或字典序场景。
处理负数时，基数排序需要额外步骤。常见方法是将数组分割成负数和正数两部分。对负数部分，取绝对值后使用基数排序，然后反转顺序（因为负数取绝对值后排序顺序相反）。对正数部分直接排序，最后合并两部分。另一种方法是偏移法：将所有数加上一个最小值偏移量，使其变为非负数，排序后再减回去。例如，如果数组中有负数，先找到最小值 min_val，然后将每个元素加上 abs(min_val)，排序后再减去 abs(min_val)。
基数排序的优点包括线性时间复杂度，高效处理大规模固定长度数据，如整数或字符串。缺点是非原地排序，需要额外空间；对数据类型有限制（只适用于可分解为位的类型）；当 k 很大时（数字非常长），效率可能下降。
典型应用场景包括数据库中对整数键的排序、计算机图形学中的算法（如深度排序）、以及后缀数组的构造。在这些领域，基数排序的线性性能优势明显。
结束语：基数排序是一种独特而高效的算法，它突破了传统比较排序的局限。通过理解其原理和实现，您可以更好地应用它到实际问题中。我鼓励您动手实现一遍代码，以加深印象。

4 互动与延伸

思考题：如何修改代码来排序字符串数组？例如，对单词列表按字典序排序，这可以通过将每个字符视为一位，使用类似方法处理。如果数字的进制不是十进制，比如二进制或十六进制，算法需要调整基数大小（b），例如二进制时 b=2，计数数组大小相应改变。
相关资源：您可以访问可视化排序网站如 VisuAlgo，观看基数排序的动态演示。对于更深入的讨论，推荐阅读关于 MSD 基数排序的学术文章，以探索其递归实现和优化。