基本的基数排序（Radix Sort）算法

李睿远

Nov 11, 2025

在计算机科学中，排序算法是基础且关键的主题。常见的比较排序算法如快速排序和归并排序，其时间复杂度通常为 O(n log n)，这是基于比较操作的理论下限。然而，是否存在一种方法能够突破「比较」这一范式，实现更优的性能呢？答案是肯定的，基数排序作为一种非比较型整数排序算法，通过逐位处理数字，可以在特定条件下达到线性时间复杂度。本文将深入解析基数排序的核心原理、实现方式及其应用，帮助你全面掌握这一高效算法。

1　基数排序初探

基数排序的核心思想在于将待排序元素视为由多个「位」组成的序列，而不是直接比较整体大小。具体来说，算法从最低位或最高位开始，依次对每一位进行稳定的排序操作。这个过程可以类比为整理扑克牌：如果我们先按点数分类，再在同点数内按花色排序，就类似于最低位优先方法；反之，如果先按花色分，再按点数排序，则类似于最高位优先方法。这种分步处理的方式使得基数排序能够避免直接比较元素，从而在整数或字符串排序中展现出独特优势。

基数排序的关键特性之一是其对稳定性的依赖。稳定性指的是在排序过程中，相等元素的相对顺序保持不变。基数排序的每一轮排序都必须使用稳定的次级算法，通常选择计数排序，因为如果次级排序不稳定，整个算法的正确性将无法保证。此外，基数排序主要适用于整数或可以分解为「位」的数据类型，如字符串。对于浮点数或其他复杂类型，则需要额外处理，例如通过转换或偏移来适应算法要求。

2　深入原理

最低位优先（LSD）方法是基数排序中最常见的实现方式。它从数字的最低位（如个位）开始，依次向高位进行稳定排序。例如，给定数组 `[170, 45, 75, 90, 2, 802, 24, 66]`，LSD 会先按个位排序，结果可能为 `[170, 90, 2, 802, 24, 45, 75, 66]`；接着按十位排序，得到 `[2, 802, 24, 45, 66, 170, 75, 90]`；最后按百位排序，完成整个排序过程。LSD 实现简单直观，但它是一种离线算法，需要预先知道最大数字的位数，以确定排序轮数。排序结束后，序列自然有序，无需额外合并步骤。

最高位优先（MSD）方法则从最高位开始排序，然后递归处理每个子桶。例如，对同一数组，MSD 会先按百位分桶，将数字分配到不同范围，再对每个桶内的数字递归排序低位。这种方法更接近分治策略，可能在某些情况下提前终止，例如当某个桶内只有一个元素时。MSD 在字符串字典序排序中尤为自然，因为它可以直接处理前缀。与 LSD 相比，MSD 在实现上可能更复杂，且性能受数据分布影响较大，但它能更早地排除无关比较。

3 代码实现

在实现基数排序时，我们通常选择计数排序作为稳定的次级排序算法。计数排序通过统计每个数字的出现次数，并利用前缀和确定元素位置，从而保证稳定性。以下以 LSD 方法为例，使用 Python 语言实现基数排序。代码将分步解释，确保每个细节清晰易懂。

首先，我们需要确定最大数字的位数，以决定排序轮数。这可以通过遍历数组并计算最大值来实现。例如，如果最大数字是 802，其位数为 3，则需进行三轮排序（个位、十位、百位）。

```
def radix_sort(arr):
    # 步骤 1：寻找最大数，确定位数
    max_num = max(arr)
    exp = 1 # 从个位开始
    while max_num // exp > 0:
        # 使用计数排序对当前位进行排序
        n = len(arr)
        output = [0] * n
        count = [0] * 10 # 十进制数字范围 0-9

        # a. 计数：统计当前位上每个数字的出现次数
        for i in range(n):
            index = (arr[i] // exp) % 10
            count[index] += 1

        # b. 计算位置：将计数转换为前缀和，表示起始索引
        for i in range(1, 10):
            count[i] += count[i - 1]

        # c. 构建输出：从后向前遍历，保证稳定性
        i = n - 1
        while i >= 0:
            index = (arr[i] // exp) % 10
            output[count[index] - 1] = arr[i]
            count[index] -= 1
            i -= 1

        # d. 复制回原数组
        for i in range(n):
            arr[i] = output[i]

```

```
    exp *= 10 # 移动到下一位
```

在这段代码中，我们首先计算最大数字以确定循环次数。然后，在每一轮中，我们使用计数排序处理当前位。计数步骤统计每个数字（0-9）的出现频率；位置计算步骤将计数数组转换为前缀和，以确定每个数字在输出数组中的起始索引；构建输出步骤从原数组末尾开始遍历，确保相等元素的顺序不变；最后，将结果复制回原数组。关键点在于从后向前遍历，这维护了稳定性，因为计数排序中后出现的元素会被放置在输出数组的较后位置。每轮结束后，exp 乘以 10，以处理更高位。

实现基数排序时，常见的陷阱包括忽略稳定性或错误处理数字位。例如，如果构建输出时从前向后遍历，可能会破坏相对顺序。另外，确保 exp 正确递增，避免遗漏高位或重复处理。

4 算法分析

基数排序的时间复杂度为 O(d * (n + k))，其中 d 是最大数字的位数，n 是元素个数，k 是每位可能取值的范围（对于十进制，k=10）。当 d 为常数且 k 与 n 同阶时，时间复杂度可视为线性 O(n)。相比之下，比较排序算法如快速排序的平均时间复杂度为 O(n log n)，基数排序在特定条件下更具优势。例如，如果数字范围有限且位数较少，基数排序能显著提升性能。

空间复杂度主要来自计数排序的辅助数组，包括计数数组和输出数组，因此为 O(n + k)。这表示基数排序不是原地算法，需要额外内存空间。尽管这可能成为内存受限环境中的缺点，但其稳定性和高效性在许多应用中值得权衡。

基数排序的优缺点总结如下：优点包括在整数排序中可能达到线性时间、稳定性高；缺点则在于适用范围有限、需要额外空间，且当数字位数差异大时，性能可能不如某些自适应比较排序。因此，在选择算法时，需考虑数据特性和环境约束。

基数排序在实际应用中常用于处理固定位数的整数序列，例如身份证号、电话号码排序，或字符串字典序排列。在计算机图形学中，它也可用于某些像素处理算法。这些场景充分利用了基数排序的稳定性和高效性。

总结来说，基数排序通过「按位排序」和「依赖稳定性」的核心思想，实现了非比较排序的突破。其线性时间复杂度的优势在合适条件下显著，但需注意数据类型的限制。鼓励读者在涉及整数或字符串排序的任务中，尝试应用这一算法。

思考与拓展部分留给读者进一步探索：例如，如何对包含负数的数组进行基数排序？可以通过分离正负数组或使用偏移量处理；MSD 实现需调整代码结构为递归形式；对于非十进制数字，只需修改进制基数即可适应。这些拓展问题有助于深化对算法灵活性的理解。