深入理解并实现基本的基数排序（Radix Sort）算法

杨子凡

Oct 18, 2025

你是否听说过一种不基于比较的排序算法？它像整理档案一样，逐位、逐层地对数字进行归类，最终神奇地实现有序。本文将带你深入探索基数排序的独特魅力，从核心思想到代码实现，彻底掌握这一高效的非比较排序算法。

在排序算法的广阔领域中，主流比较排序算法如快速排序和归并排序的时间复杂度下限为 $O(n \log n)$，这是基于比较操作的固有限制。然而，基数排序作为一种非比较型整数排序算法，打破了这一瓶颈，其时间复杂度可以达到 $O(k \times n)$，其中 k 代表数字的最大位数。这种算法在特定场景下，例如处理固定位数的整数数据时，能够展现出超越传统排序方法的效率。本文旨在帮助读者彻底理解基数排序的工作原理，掌握最低位优先（LSD）的实现方式，并深入探讨其优势、局限性以及适用场景，从而为实际应用提供坚实的技术基础。

1　基数排序的核心思想

基数排序的核心思想可以通过一个生动的比喻来理解：想象整理一副扑克牌，我们首先按照花色（如红心、黑桃等）将牌分成几个大类，然后在每个花色内部按照点数（从 A 到 K）进行排序。类似地，基数排序通过逐位处理数字，先按最低位（如个位数）将数字分配到不同的「桶」中，然后按顺序收集，再处理更高位（如十位数、百位数），依次类推，直到最高位处理完毕，最终实现整体有序。这里涉及两个关键概念：基数和关键码。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 10（涵盖 0 到 9）；关键码则是排序所依据的每一位数字本身。基数排序主要有两种策略：最低位优先（LSD）和最高位优先（MSD）。LSD 从数字的最低位开始排序，是本文的重点；而 MSD 则从最高位开始，采用递归分治的方式，本文仅作简要提及以供拓展。

2　算法步骤详解（以 LSD 为例）

基数排序的 LSD 实现过程可以分解为几个清晰的步骤，无需依赖图示，我们通过文字描述来构建完整的理解。首先，进行准备工作：确定待排序数组中的最大值 `max_val`，并计算最大数字的位数 k，公式为 $k = \lfloor \log_{10}(\text{max\_val}) \rfloor + 1$，这决定了排序所需的轮数。接下来，从最低位（个位）开始，逐位进行排序。例如，在第一轮中，我们创建 10 个桶，分别对应数字 0 到 9，然后遍历数组，根据每个数字的个位数将其放入对应桶中。完成后，按桶的编号顺序（从 0 到 9）依次将元素收集回原数组。这时，数组在个位数上已经达到局部有序。进入第二轮，我们清空桶，基于十位数重复上述分发和收集过程。关键点在于，必须从低到高排序，因为高位相同的数字，其顺序由低位的排序结果决定，这依赖于排序的稳定性。稳定性确保前一轮的排序成果不被破坏，例如如果两个数字的十位数相同，它们的相对顺序由个位数的排序结果保持。重复这一过程，直到处理完最高位，数组便完全有序。整个算法强调稳定性的重要性，它是基数排序正确性的基石。

3 代码实现（以 Python 为例）

以下是一个完整的 Python 实现基数排序的代码示例，我们将逐段进行详细解读，帮助读者理解每一部分的功能。

```
def radix_sort(arr):
    # 特殊情况处理：如果数组为空或只有一个元素，直接返回
    if len(arr) <= 1:
        return arr

    # 寻找数组中的最大值，以确定最大位数
    max_val = max(arr)
    # 计算最大位数 k，使用对数函数并取整
    exp = 1
    while max_val // exp > 0:
        # 创建 10 个桶，每个桶是一个空列表，用于存放对应数字的元素
        buckets = [[] for _ in range(10)]

        # 分发阶段：遍历数组，根据当前位数将元素放入对应桶中
        for num in arr:
            digit = (num // exp) % 10 # 计算当前位数字
            buckets[digit].append(num) # 将数字放入对应桶

        # 收集阶段：按桶顺序（0 到 9）将元素覆盖回原数组
        arr = []
        for bucket in buckets:
            arr.extend(bucket) # 将每个桶中的元素依次添加到数组

        # 更新指数，进入下一位处理
        exp *= 10

    return arr
```

在这段代码中，我们首先处理边界情况，如果数组长度小于等于 1，则无需排序直接返回。接着，通过 max(arr) 找到最大值 max_val，并使用 exp 变量来表示当前处理的位数（初始为 1，代表个位）。主循环继续执行，只要 max_val // exp 大于 0，就意味着还有更高位需要处理。在每一轮循环中，我们创建 10 个桶，用于存放数字 0 到 9 对应的元素。分发阶段通过 (num // exp) % 10 计算每个数字在当前位的值，并将其添加到相应桶中。例如，如果 exp 为 1，则计算个位数；如果 exp 为 10，则计算十位数。收集阶段则按桶的顺序将元素重新组合到数组中，确保稳定性，因为桶内元素顺序保持不变。最后，通过 exp *= 10 更新指数，处理下一位。整个过程重复直到最高位处理完毕，返回排序后的数组。为了验证代码，我们可以使用测试用例，例如输入 [170, 45,

75，90，2，802，24，66]，并观察每轮排序后的中间结果，例如第一轮后数组在个位数上有序，第二轮后在十位数上局部有序，最终完全有序。

4 算法分析

基数排序的时间复杂度为 $O(k \times n)$，其中 k 是最大数字的位数，n 是数组长度。详细来说，每一轮分发需要遍历所有 n 个元素，收集同样需要 $O(n)$ 时间，总共进行 k 轮，因此总时间为 $O(k \times n)$。需要注意的是，当 k 远小于 n 时（例如数字范围较小），基数排序效率很高；但如果数字范围很大（如 2^{32}），k 值较大，效率可能不如 $O(n \log n)$ 的比较排序算法。空间复杂度为 $O(n + r)$，其中 r 是基数（十进制下为 10），因为需要额外的桶空间来存储 n 个元素。稳定性方面，基数排序是稳定的排序算法，这得益于每一轮的分发和收集过程都保持了元素的相对顺序，这对于多关键字排序至关重要。

5 进阶讨论与变体

除了基本的 LSD 实现，基数排序还有多种变体和扩展应用。最高位优先（MSD）基数排序从最高位开始处理，采用递归分治策略，适用于数据分布不均匀的场景，但可能需要额外处理空桶问题。对于负数处理，基本实现无法直接适用，因为负数的位表示不同。一种解决方案是将数组分割为负数和正数两部分，分别进行基数排序：对负数取绝对值排序后再反转顺序，最后与正数部分合并。另一种方法是使用偏移量将所有数转换为非负数后再排序。此外，基数排序可以扩展至其他数据类型，例如字符串或日期。对于字符串，可以将其视为字符序列，按位处理实现字典序排序；对于日期，可以分解为年、月、日等部分，逐部分排序。这些扩展体现了基数排序思想的通用性，关键在于将数据抽象为固定位的关键码序列。

回顾基数排序的核心，它通过「按位分配、稳定收集」的方式，实现了高效的非比较排序。其优势包括线性时间复杂度（在 k 较小的情况下）、稳定性以及对整数排序的高效性。然而，基数排序也有局限性：它通常仅适用于整数或具有固定位键的数据类型，需要额外的内存空间，且当 k 值较大时效率可能下降。应用场景广泛，例如电话号码排序、身份证号排序或日期排序，这些场景中关键码由多位组成且位数固定。通过本文的讲解，读者应能掌握基数排序的基本原理和实现，为实际开发提供参考。

6 互动与思考

鼓励读者动手实践，尝试用其他编程语言实现基数排序，或扩展代码以处理负数情况。思考题包括：如果使用不稳定的子排序方法，会发生什么？例如，假设在分发过程中不保持桶内顺序，可能导致前一轮排序结果被破坏，从而无法保证最终有序。另一个思考题是如何修改算法对字符串数组进行字典序排序？这可以通过将字符串视为字符序列，按字符的 ASCII 值逐位处理来实现。通过这些互动，读者可以加深对基数排序的理解，并探索其更多可能性。