# 排序算法总结

MLE 算法指北

2025 年 2 月 5 日

## 1 排序算法分类

排序算法主要分为两类：

- **比较排序**：基于元素两两比较，时间复杂度为 $O(n \log n)$ 或更高。
  - 冒泡排序（Bubble Sort）
  - 选择排序（Selection Sort）
  - 插入排序（Insertion Sort）
  - 归并排序（Merge Sort）
  - 快速排序（Quick Sort）
  - 堆排序（Heap Sort）
- **非比较排序**：基于元素特性，如计数、基数和桶排序。
  - 计数排序（Counting Sort）
  - 基数排序（Radix Sort）
  - 桶排序（Bucket Sort）

## 2 时间复杂度分析

| 排序算法 | 平均时间复杂度 | 最坏情况 | 最优情况 | 空间复杂度 |
|---|---|---|---|---|
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ |
| 插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ |
| 快速排序 | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ | $O(\log n)$ |
| 堆排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ |
| 计数排序 | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | $O(k)$ |
| 基数排序 | $O(nk)$ | $O(nk)$ | $O(nk)$ | $O(n+k)$ |
| 桶排序 | $O(n+k)$ | $O(n^2)$ | $O(n)$ | $O(n)$ |

表 1: 排序算法时间复杂度对比

## 3 排序算法实现

### 3.1 冒泡排序（Bubble Sort）

**原理**：每次比较相邻元素并交换，使得较大元素逐步上浮。

**Python 实现**：

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

### 3.2 选择排序（Selection Sort）

**原理**：每次选择最小元素，放到未排序部分的最前面。

**Python 实现**：

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

### 3.3 插入排序（Insertion Sort）

**原理**：将未排序的元素逐个插入到已排序序列的正确位置。

**Python 实现**：

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

### 3.4 归并排序（Merge Sort）

**原理**：递归将数组拆分为两部分，分别排序后合并。

**Python 实现**：

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    while left and right:
        if left[0] < right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    return result + left + right
```

### 3.5 快速排序（Quick Sort）

**原理**：选取基准值（pivot），将数组划分为小于 pivot 和大于 pivot 的两部分，递归排序。

**Python 实现**：

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

### 3.6 堆排序 Heap sort

**堆（Heap）**是一种特殊的**二叉树**，满足以下性质：

- **完全二叉树（Complete Binary Tree）**：除了最后一层，所有层都必须填满，且最后一层的叶子节点从左到右排列。

- **堆序性（Heap Property）**：
  - **最大堆（Max-Heap）**：每个节点的值都 ** 大于等于 ** 其子节点。
  - **最小堆（Min-Heap）**：每个节点的值都 ** 小于等于 ** 其子节点。

#### 3.6.1 堆排序原理

堆排序利用 ** 最大堆（Max-Heap）** 进行排序，主要分为以下步骤：

1. ** 构建最大堆 **：将无序数组转换成最大堆，使得堆顶（即数组索引 0）存储最大元素。
2. ** 交换堆顶和最后一个元素 **，然后将剩余元素重新调整为最大堆。
3. ** 重复步骤 2**，直到所有元素排序完成。

#### 3.6.2 堆调整（Heapify）

为了维持最大堆的性质，我们需要 **heapify（堆化）** 操作：

- 设当前父节点索引为 $i$，左子节点为 $2i+1$，右子节点为 $2i+2$。
- 若子节点比父节点大，则交换，并递归调整子树。

#### 3.6.3 时间复杂度分析

- **构建最大堆**：遍历所有非叶子节点，执行堆调整，每次操作时间复杂度为 $O(\log n)$，总共 $O(n)$。
- **排序过程**：每次取出堆顶元素，并重新调整堆，总共进行 $n$ 次，每次调整 $O(\log n)$，所以时间复杂度 $O(n \log n)$。
- **总时间复杂度**：$O(n + n \log n) = O(n \log n)$。

#### 3.6.4 堆排序 Python 实现

```python
def heapify(arr, n, i):
    """维护最大堆"""
    largest = i  # 初始化最大元素
    left = 2 * i + 1  # 左子节点
    right = 2 * i + 2  # 右子节点

    # 如果左子节点大于当前最大值
    if left < n and arr[left] > arr[largest]:
        largest = left

    # 如果右子节点大于当前最大值
    if right < n and arr[right] > arr[largest]:
        largest = right

    # 如果最大值发生改变，则交换并递归堆化
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    """堆排序"""
    n = len(arr)

    # 1. 建立最大堆
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # 2. 逐个取出最大元素并调整堆
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # 交换堆顶和末尾元素
        heapify(arr, i, 0)  # 重新调整堆
```

## 3.7 基数排序（Radix Sort）

### 3.7.1 基本概念

**基数排序**（Radix Sort）是一种**非比较排序**，适用于整数和定长字符串排序。它的核心思想是**按位（位数）进行排序**，从最低位到最高位（LSD）或从最高位到最低位（MSD）。

### 3.7.2 排序原理

基数排序采用**稳定的子排序算法**（如计数排序）对数据的每个位进行排序：

1. 确定最大数的位数 $d$。
2. 从**最低位**（LSD 方法）到**最高位**，依次对所有数进行桶式排序（通常用计数排序）。
3. 经过 $d$ 次排序后，整个数组变为有序。

### 3.7.3 时间复杂度分析

- 设数组大小为 $n$，最大数的位数为 $d$，基数（桶数）为 $k$。
- 每次位排序使用 $O(n+k)$ 时间，总共有 $d$ 轮，因此时间复杂度为：

$$O(d \cdot (n+k))$$

- 若基数 $k$ 取 $O(n)$，则 $O(d \cdot n)$ 近似为 $O(n)$，接近线性排序。

### 3.7.4 Python 实现

```
def counting_sort_for_radix(arr, exp):
    """ 计数排序（基于某个位）"""
    n = len(arr)
    output = [0] * n
    count = [0] * 10  # 计数数组（0-9）

    for i in arr:
        index = (i // exp) % 10
        count[index] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    for i in reversed(arr):
        index = (i // exp) % 10
        output[count[index] - 1] = i
        count[index] -= 1

    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    """ 基数排序 """
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10
```

## 3.8 计数排序（Counting Sort）

### 3.8.1 基本概念

**计数排序**（Counting Sort）是一种**非比较排序**，适用于数据范围较小的整数排序。它通过 ** 统计数组中每个元素的出现次数 **，然后计算出每个元素在最终排序数组中的位置。

### 3.8.2 排序原理

1. 找出数组中的最大值 $k$，创建大小为 $k+1$ 的计数数组。
2. 遍历原数组，记录每个元素出现的次数。
3. 计算前缀和，确定元素在排序数组中的位置。
4. 根据前缀和，将元素放入正确的位置，并生成有序数组。

## 3.9 时间复杂度分析

- 设数据范围最大值为 $k$，数组大小为 $n$。
- 计数和前缀和计算 $O(k)$，遍历数组 $O(n)$，总时间复杂度：

$$O(n+k)$$

- 适用于 $k=O(n)$ 的情况，此时时间复杂度为 $O(n)$，非常高效。

### 3.9.1 Python 实现

```
def counting_sort(arr):
    """ 计数排序 """
    max_val = max(arr)
    count = [0] * (max_val + 1)
    output = [0] * len(arr)

    for num in arr:
        count[num] += 1

    for i in range(1, len(count)):
        count[i] += count[i - 1]

    for num in reversed(arr):
        output[count[num] - 1] = num
        count[num] -= 1

    for i in range(len(arr)):
        arr[i] = output[i]
```

## 3.10 桶排序（Bucket Sort）

**原理**：将元素放入不同的桶中，每个桶单独排序，最后合并。

**Python 实现**：

```
def bucket_sort(arr):
    max_val = max(arr)
    size = max_val // len(arr) + 1
    buckets = [[] for _ in range(len(arr))]

    for num in arr:
        index = num // size
        buckets[index].append(num)

    for bucket in buckets:
        bucket.sort()

    return [num for bucket in buckets for num in bucket]
```