基于代码实现的排序算法深入讲解

本文将围绕提供的 Java 排序算法代码，按「基础排序算法」「进阶排序算法」「快速排序算法」三大类，深入剖析每种算法的核心思想、代码逻辑细节、关键设计亮点、时间复杂度、空间复杂度及适用场景。所有讲解均紧密结合代码实现，让理论与实践深度融合。

一、基础排序算法

基础排序算法通常是入门级实现，核心逻辑简单直观，时间复杂度多为 $O(n^2)$，适用于小规模数据排序。本节包含冒泡排序、选择排序、插入排序及希尔排序（插入排序的优化版）。

1. 冒泡排序（BubbleSort）

1.1 核心思想

通过相邻元素的两两比较与交换，将最大（或最小）的元素逐步“冒泡”到数组的末尾。每一轮遍历都会确定一个未排序区间的最值元素，遍历 n-1 轮后完成排序。代码中加入了 `sorted`标志位优化，避免已排序数组的无效遍历。

1.2 代码逻辑拆解

```
public void bubbleSort(int[] nums) {
    int n = nums.length;

    for (int i = 0; i < n - 1; i++) {
        boolean sorted = true; // 优化标志：默认当前区间已排序

        for (int j = 0; j < n - i - 1; j++) {
            if (nums[j] > nums[j + 1]) {
                swap(nums, j, j + 1); // 相邻元素逆序则交换
                sorted = false; // 发生交换，说明区间未排序
            }
        }

        if (sorted) break; // 无交换则直接退出，无需后续遍历
    }
}

private void swap(int[] a, int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

- 外层循环（i 从 0 到 n-2）：控制排序轮次，共需 n-1 轮（最后一个元素无需比较）。每轮结束后，数组末尾会新增一个已排序的最值元素。

- 内层循环（j 从 0 到 n-i-2）：遍历未排序区间（0 到 n-i-1），相邻元素比较。若逆序则调用 `swap` 方法交换，同时将 `sorted` 设为 false。
- 优化逻辑：若某一轮未发生任何交换（sorted 保持 true），说明数组已完全有序，直接跳出循环，减少无效遍历。

1.3 复杂度与适用场景

- 时间复杂度：最好情况 O(n)（数组已有序，触发 sorted 优化，仅遍历 1 轮）；最坏情况 $O(n^2)$（数组逆序，每轮都需完整交换）；平均 $O(n^2)$。
- 空间复杂度：O(1)，仅使用常数级临时变量，属于原地排序。
- 适用场景：小规模数据（n ≤ 1000）、接近有序的数组；不适用于大规模数据（$O(n^2)$ 复杂度效率过低）。

2. 选择排序（SelectionSort）

2.1 核心思想

每一轮从未排序区间中找到最小元素的索引，然后将该元素与未排序区间的第一个元素交换。通过"选择最值+交换"的方式，逐步扩大已排序区间，最终完成排序。相比冒泡排序，选择排序减少了交换次数（每轮仅交换 1 次）。

2.2 代码逻辑拆解

```
public void selectionSort(int[] nums) {
    for(int i  = 0; i < nums.length; i++){
        int leftMin = findMin(nums, i); // 找到未排序区间[i, n-1]的最小值索引
        swap(nums, i, leftMin); // 交换最小值与未排序区间第一个元素
    }
}

private int findMin(int[] nums, int L){
    int minIndex = L; // 初始化最小值索引为未排序区间起点
    for(int i = L; i < nums.length; i++){
        if(nums[i] < nums[minIndex]){
            minIndex = i; // 更新最小值索引
        }
    }
    return minIndex;
}

private void swap(int[] a, int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

- 外层循环（i 从 0 到 n-1）：i 是未排序区间的起点，每轮循环后，[0, i] 成为已排序区间。
- findMin 方法：遍历未排序区间 [L, n-1]，通过逐一比较找到最小值的索引，核心是“遍历查找”而非“交换移动”。
- swap 方法：将找到的最小值与未排序区间的第一个元素（nums[i]）交换，确保已排序区间的有序性。

2.3 复杂度与适用场景

- 时间复杂度：最好、最坏、平均均为 $O(n^2)$。无论数组是否有序，都需遍历 n 轮，每轮遍历未排序区间，查找最小值的操作固定为 O(n)。
- 空间复杂度：O(1)，原地排序，仅使用常数级临时变量。
- 适用场景：小规模数据、对交换操作代价敏感的场景（相比冒泡排序，交换次数从 $O(n^2)$ 减少到 O(n)）；不适用于大规模或接近有序的数据。

3. 插入排序（InsertionSort）

3.1 核心思想

将数组分为“已排序区间”和“未排序区间”，每次取未排序区间的第一个元素（关键字 key），插入到已排序区间的合适位置，使已排序区间始终有序。类比日常生活中整理扑克牌的过程，逐步将每张牌插入到正确位置。

3.2 代码逻辑拆解

```
public void insertionSort(int[] nums) {
    for (int i = 1; i < nums.length; i++) { // 未排序区间起点为1（0号元素为初始已排序区间）
        int key = nums[i]; // 取未排序区间第一个元素作为关键字
        int j = i - 1; // j 指向已排序区间末尾

        // 遍历已排序区间，找到key的插入位置
        while (j >= 0 && nums[j] > key) {
            nums[j + 1] = nums[j]; // 元素后移，为key腾出位置
            j--;
        }

        nums[j + 1] = key; // 将key插入到正确位置
    }
}
```

- 外层循环（i 从 1 到 n-1）：初始时已排序区间为 [0, 0]，未排序区间为 [1, n-1]。i 依次遍历未排序区间的每个元素。
- 关键字 key：保存当前要插入的未排序元素，避免后续元素后移时被覆盖。
- 内层 while 循环：从已排序区间末尾（j = i-1）向前遍历，若 nums[j] > key，说明 nums[j] 应在 key 之后，将 nums[j] 后移一位（nums[j+1] = nums[j]）；直到找到 nums[j] ≤ key 或 j < 0，此时 j+1 即为 key 的插入位置。

- 插入操作：将 key 赋值到 nums[j+1]，完成一次插入，已排序区间扩大一位。

3.3 复杂度与适用场景

- 时间复杂度：最好情况 O(n)（数组已有序，无需元素后移，仅遍历 n-1 次）；最坏情况 $O(n^2)$（数组逆序，每次插入都需遍历整个已排序区间并后移所有元素）；平均 $O(n^2)$。
- 空间复杂度：O(1)，原地排序，仅使用 key 和 j 两个临时变量。
- 适用场景：小规模数据、接近有序的数据（此时效率接近 O(n)）、数据量动态增长的场景（可快速插入新元素）；是日常开发中小规模数据排序的优选方案之一。

4. 希尔排序（ShellSort）

4.1 核心思想

希尔排序是插入排序的优化版，核心是“分组插入排序”。通过设定一个递减的“步长 gap”，将数组按 gap 分为多个子数组，对每个子数组执行插入排序；逐步缩小 gap 直至为 1，此时整个数组变为一个子数组，执行最后一次插入排序，完成最终排序。其目的是通过前期的粗排序，让数组整体接近有序，从而降低最后一次插入排序的复杂度。

4.2 代码逻辑拆解

```
public static void shellSort(int[] arr) {
    if (arr == null || arr.length <= 1) {
        return; // 边界处理：空数组或单元素数组无需排序
    }
    int n = arr.length;
    for (int gap = n / 2; gap > 0; gap /= 2) { // 步长gap初始为n/2，每次减半
        // 对每个子数组执行插入排序
        for (int i = gap; i < n; i++) {
            int current = arr[i]; // 保存当前要插入的元素
            int j = i - gap; // j指向当前子数组的已排序区间末尾

            // 查找插入位置，元素后移（步长为gap）
            while (j >= 0 && arr[j] > current) {
                arr[j + gap] = arr[j];
                j -= gap;
            }
            arr[j + gap] = current; // 插入当前元素
        }
    }
}
```

- 边界处理：首先判断数组是否为空或仅有一个元素，直接返回，避免无效计算。
- 步长 gap 循环：初始 gap 为数组长度的一半（n/2），之后每次减半（gap /= 2），直至 gap > 0。gap 的递减策略决定了排序效率，常见的还有 Hibbard 步长（2^k - 1）等，但代码中采用最简单的“减半步长”，实现简洁。

- 分组插入排序：对于每个 gap，i 从 gap 开始遍历（每个子数组的未排序区间起点），current 保存当前要插入的元素；j 指向当前子数组已排序区间的末尾（i - gap），通过 while 循环向前遍历（步长为 gap），找到 current 的插入位置，完成元素后移和插入。
- 最终排序：当 gap = 1 时，数组被分为一个子数组，此时执行的就是标准插入排序，但由于前期粗排序已让数组接近有序，此时插入排序的效率接近 O(n)。

4.3 复杂度与适用场景

- 时间复杂度：取决于步长策略，代码中"减半步长"的时间复杂度为 $O(n^2)$，但实际效率远高于标准插入排序；若采用 Hibbard 步长，时间复杂度可优化到 O(n^(3/2))。
- 空间复杂度：O(1)，原地排序，仅使用常数级临时变量。
- 适用场景：中大规模数据排序（效率优于基础 $O(n^2)$ 排序）、对空间占用敏感的场景；是从基础排序过渡到进阶排序的重要算法，实际开发中应用广泛。

二、进阶排序算法

进阶排序算法采用"分治"等更优策略，时间复杂度优化到 O(n log n)，是大规模数据排序的核心方案。本节包含归并排序、快速排序和堆排序。

1. 归并排序（MergeSort）

1.1 核心思想

基于"分治思想"，将数组递归拆分为两个子数组，直到每个子数组仅含一个元素（此时子数组天然有序）；然后逐步将两个有序子数组合并为一个有序数组，最终合并为完整的有序数组。核心是"先分后合"，拆分过程不涉及元素比较，合并过程负责有序整合。

1.2 代码逻辑拆解

```
public static void mergeSort(int[] arr) {
    if (arr == null || arr.length <= 1) return; // 边界处理：空数组或单元素数组无需排序
    int[] temp = new int[arr.length]; // 临时数组，用于合并过程，避免重复创建
    split(arr, 0, arr.length - 1, temp); // 递归拆分并合并
}

private static void split(int[] arr, int left, int right, int[] temp) {
    if (left >= right) {
        return; // 递归终止条件：子数组长度为1（left == right）或无效区间
    }
    int mid = left + (right - left) / 2; // 计算中间索引（避免(left+right)溢出）

    split(arr, left, mid, temp); // 递归拆分左子数组[left, mid]
    split(arr, mid + 1, right, temp); // 递归拆分右子数组[mid+1, right]

    merge(arr, left, mid, right, temp); // 合并左右两个有序子数组
```

```
    }

    private static void merge(int[] arr, int left, int mid, int right, int[] temp) {
        int i = left; // 左子数组指针（起点left）
        int j = mid + 1; // 右子数组指针（起点mid+1）
        int k = left; // 临时数组temp的指针（起点left）

        // 合并两个有序子数组到temp
        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++]; // 左子数组元素更小，放入temp
            } else {
                temp[k++] = arr[j++]; // 右子数组元素更小，放入temp
            }
        }

        // 处理左子数组剩余元素
        while (i <= mid) {
            temp[k++] = arr[i++];
        }

        // 处理右子数组剩余元素
        while (j <= right) {
            temp[k++] = arr[j++];
        }

        // 将temp中合并后的有序元素拷贝回原数组arr
        if (right + 1 - left >= 0) System.arraycopy(temp, left, arr, left, right + 1 -
    left);
    }
```

- 入口方法 mergeSort：负责边界处理和临时数组创建。临时数组 temp 仅创建一次，避免在递归合并过程中重复创建，减少空间开销。
- 拆分方法 split：
  - 递归终止条件：当 left ≥ right 时，子数组长度为 1 或无效，直接返回（单个元素天然有序）。
  - 中间索引计算：采用 `mid = left + (right - left) / 2`，而非 `(left + right) / 2`，目的是避免 left 和 right 较大时相加溢出。
  - 递归拆分：将数组拆分为左子数组 [left, mid] 和右子数组 [mid+1, right]，直到所有子数组长度为 1。
- 合并方法 merge（核心步骤）：
  - 指针初始化：i 指向左子数组起点，j 指向右子数组起点，k 指向 temp 数组的写入起点。
  - 有序合并：遍历左右两个有序子数组，将较小的元素依次写入 temp 数组，确保 temp 中元素有序。
  - 剩余元素处理：当一个子数组遍历完毕后，将另一个子数组的剩余元素直接写入 temp（剩余元素已有序）。

- 拷贝回原数组：通过 `System.arraycopy` 将 temp 中 [left, right] 区间的有序元素拷贝回原数组 arr，完成本次合并。

1.3 复杂度与适用场景

- 时间复杂度：最好、最坏、平均均为 O(n log n)。拆分过程是 log n 层递归，每层合并过程的时间复杂度为 O(n)，整体为 O(n log n)。
- 空间复杂度：O(n)，主要消耗在临时数组 temp 上（长度为 n）；递归调用栈的空间复杂度为 O(log n)，可忽略，因此整体空间复杂度为 O(n)。
- 适用场景：大规模数据排序（O(n log n) 效率稳定）、要求排序稳定的场景（合并过程中相等元素的相对位置不变，属于稳定排序）；不适用于对空间占用敏感的场景。

2. 快速排序（QuickSort）

2.1 核心思想

同样基于“分治思想”，核心是“选基准、分区间”。从数组中选择一个元素作为“基准（pivot）”，通过一次分区操作，将数组分为两部分：左区间元素均小于基准，右区间元素均大于基准；然后递归对左右区间执行同样的操作，直至所有区间长度为 1，完成排序。快速排序的效率关键在于基准的选择和分区操作的实现。

2.2 代码逻辑拆解

```
public void quickSort(int[] arr){
    sortHelper(arr, 0, arr.length-1); // 调用辅助方法，指定排序区间[0, n-1]
}

private void sortHelper(int[] arr, int L, int R){
    if(L >= R) return; // 递归终止条件：区间长度为1或无效
    int pivot = partition(arr, L, R); // 分区操作，返回基准元素的最终位置

    sortHelper(arr, L, pivot); // 递归排序左区间[L, pivot]
    sortHelper(arr, pivot + 1, R); // 递归排序右区间[pivot+1, R]
}

private int partition(int[] arr, int L, int R){
    int pivot = arr[L]; // 选择区间左端点作为基准（简单实现，可优化）

    int i = L - 1,  j = R + 1; // i初始在L左侧，j初始在R右侧

    while(i < j){
        do { i ++; } while(arr[i] < pivot); // i向右移动，直到找到≥pivot的元素
        do { j --; } while(arr[j] > pivot); // j向左移动，直到找到≤pivot的元素

        if(i >= j) break; // i≥j时，分区完成，退出循环
        swap(arr, i, j); // 交换i和j位置的元素，确保左小右大
    }

    return j; // 返回基准元素的最终位置j
```

```
}

private void swap(int[] arr, int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

- 入口方法 quickSort：简洁的入口，直接调用辅助方法 sortHelper，指定初始排序区间 [0, n-1]。
- 辅助方法 sortHelper：负责递归控制。若区间 L ≥ R，说明区间已有序，直接返回；否则执行分区操作，得到基准元素的最终位置 pivot，然后递归排序左右两个子区间。
- 分区方法 partition（核心步骤，采用"双指针法"）：
    - 基准选择：代码中选择区间左端点 arr[L] 作为基准，实现简单，但在数组已有序时会导致分区失衡（左区间为空，右区间为 n-1 个元素），可优化为"三数取中"（左、中、右三个元素的中位数作为基准）。
    - 双指针初始化：i 初始为 L-1（左指针起点在区间左侧），j 初始为 R+1（右指针起点在区间右侧），目的是让指针先移动再判断，避免初始位置的基准元素被误判。
    - 指针移动与交换：i 向右移动，直到找到 ≥ pivot 的元素；j 向左移动，直到找到 ≤ pivot 的元素。若 i < j，交换两个元素，确保左区间元素 ≤ pivot，右区间元素 ≥ pivot；若 i ≥ j，分区完成，返回 j 作为基准元素的最终位置。

2.3 复杂度与适用场景

- 时间复杂度：最好、平均为 O(n log n)（分区均衡时，递归层数为 log n，每层分区时间为 O(n)）；最坏情况 $O(n^2)$（数组已有序或逆序，分区失衡，递归层数为 n）。通过优化基准选择（如三数取中），可避免最坏情况。
- 空间复杂度：O(log n)，主要消耗在递归调用栈上（分区均衡时，递归层数为 log n）；最坏情况 O(n)（分区失衡时）。属于原地排序（未使用额外数组存储元素）。
- 适用场景：大规模数据排序（实际效率通常优于归并排序，因为原地排序减少了数据拷贝开销）、对排序稳定性要求不高的场景（分区交换过程可能改变相等元素的相对位置，属于不稳定排序）；是实际开发中应用最广泛的排序算法之一。

3. 堆排序（HeapSort）

3.1 核心思想

基于"堆"这种数据结构（完全二叉树），利用堆的性质（大顶堆：根节点是最大值；小顶堆：根节点是最小值）实现排序。核心步骤分为两步：① 将无序数组构建为大顶堆；② 逐步提取堆顶的最大值，与堆尾元素交换，然后调整剩余元素为新的大顶堆，重复此过程直至堆为空，最终得到有序数组。

3.2 代码逻辑拆解

```java
public static void heapSort(int[] arr) {
    if (arr == null || arr.length <= 1) return; // 边界处理
    int n = arr.length;

    // 第一步：构建大顶堆（从最后一个非叶子节点开始下沉调整）
    for (int i = n / 2 - 1; i >= 0; i--) {
        siftDown(arr, i, n);
    }

    // 第二步：提取堆顶元素，调整堆
    for (int end = n - 1; end > 0; end--) {
        swap(arr, 0, end); // 交换堆顶（最大值）与堆尾元素
        siftDown(arr, 0, end); // 调整剩余[0, end-1]元素为大顶堆
    }
}

private static void siftDown(int[] arr, int index, int size) {
    int largest = index; // 初始化最大值索引为当前节点
    while (true) {
        int left = 2 * index + 1; // 当前节点的左子节点索引（完全二叉树性质）
        int right = left + 1; // 当前节点的右子节点索引

        // 比较左子节点与当前最大值
        if (left < size && arr[left] > arr[largest]) {
            largest = left;
        }
        // 比较右子节点与当前最大值
        if (right < size && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest == index) break; // 若当前节点已是最大值，调整结束

        swap(arr, index, largest); // 交换当前节点与最大值节点
        index = largest; // 更新当前节点索引，继续下沉调整
    }
}

private static void swap(int[] arr, int i, int j) {
    int t = arr[i];
    arr[i] = arr[j];
    arr[j] = t;
}
```

- 入口方法 heapSort：
    - 边界处理：空数组或单元素数组直接返回。
    - 构建大顶堆：循环从最后一个非叶子节点（索引为 n/2 - 1，完全二叉树中，叶子节点从 n/2 开始）开始，调用 siftDown 方法进行下沉调整。为什么从最后一个非叶子节点开始？因为叶子节点本身已是合法的堆结构，无需调整，从非叶子节点倒序调整可确保整个数组构建为大顶堆。

    - 提取堆顶与调整：end 从 n-1 开始（堆尾元素），每次将堆顶（索引 0，最大值）与堆尾元素交换，此时堆尾元素成为已排序的最大值；然后调用 siftDown 方法，将剩余 [0, end-1] 区间的元素调整为大顶堆，重复此过程直至 end = 0，数组完全有序。

- 下沉调整方法 siftDown（核心步骤）：
    - 初始化最大值索引 largest 为当前节点 index。
    - 计算左右子节点索引：根据完全二叉树性质，左子节点 index = 2*index + 1*，*右子节点 index = 2*index + 2。
    - 寻找最大值：比较当前节点与左右子节点，更新 largest 为最大值的索引（需确保子节点索引 < size，即子节点存在）。
    - 调整与循环：若 largest != index，说明当前节点不是最大值，交换当前节点与 largest 节点，然后将 index 更新为 largest，继续循环下沉；若 largest == index，说明当前节点已是最大值，调整结束。

3.3 复杂度与适用场景

- 时间复杂度：最好、最坏、平均均为 O(n log n)。构建大顶堆的时间复杂度为 O(n)；提取堆顶并调整的过程共 n-1 次，每次调整的时间复杂度为 O(log n)，整体为 O(n log n)。
- 空间复杂度：O(1)，原地排序，仅使用常数级临时变量。
- 适用场景：大规模数据排序、对空间占用敏感的场景、需要快速获取最大值的场景（如Top K问题）；不适用于对排序稳定性要求高的场景（交换堆顶与堆尾时可能改变相等元素的相对位置，属于不稳定排序）。

三、快速排序算法（线性时间排序）

此类算法不基于元素比较，而是利用“计数”“基数”等特性实现排序，时间复杂度可达到 O(n)，属于线性时间排序。本节包含计数排序和基数排序，适用于特定类型的数据。

1. 计数排序（CountingSort）

1.1 核心思想

计数排序的核心是“统计元素出现次数”，通过一个计数数组记录每个元素的出现次数，然后根据计数数组重构有序数组。其前提是待排序元素为非负整数且取值范围已知（存在最大值 max）。代码中实现了两种版本：不稳定版和稳定版。

1.2 代码逻辑拆解

1.2.1 不稳定版计数排序（countingSort_unStable）

```
public void countingSort_unStable(int[] nums){
    if (nums == null || nums.length <= 1) return; // 边界处理
```

```
    int max = nums[0];
    for (int num : nums) max = Math.max(max, num); // 找到数组中的最大值max

    int[] count = new int[max + 1]; // 计数数组，长度为max+1（覆盖0到max的所有元素）

    // 第一步：统计每个元素的出现次数
    for (int num : nums) count[num]++;

    // 第二步：根据计数数组重构有序数组
    int pos = 0;
    for (int i = 0; i <= max; i++){
        while (count[i] > 0){
            nums[pos++] = i; // 按元素大小顺序写入原数组
            count[i]--;
        }
    }
}
```

- 边界处理：空数组或单元素数组直接返回。
- 找最大值 max：用于确定计数数组的长度（max + 1），确保计数数组能覆盖所有待排序元素。
- 统计出现次数：遍历待排序数组，count[num] 记录元素 num 的出现次数。
- 重构有序数组：遍历计数数组（从 0 到 max），对于每个元素 i，若 count[i] > 0，将 i 重复写入原数组 count[i] 次，pos 指针记录写入位置。此版本不稳定的原因：相等元素的相对位置可能改变（例如原数组中 [2, 1, 2]，排序后两个 2 的相对位置与原数组相反）。

1.2.2 稳定版计数排序（countingSort_stable）

```
public void countingSort_stable(int[] nums) {
    if (nums == null || nums.length <= 1) return; // 边界处理

    int max = nums[0];
    for (int num : nums) max = Math.max(max, num); // 找到最大值max

    int[] count = new int[max + 1];
    for (int num : nums) {
        count[num]++; // 第一步：统计出现次数
    }

    // 第二步：计算count数组的前缀和（确定每个元素的最终位置范围）
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }

    int[] output = new int[nums.length]; // 临时数组，用于保存有序元素
    // 第三步：倒序遍历原数组，将元素放入output的正确位置（保证稳定性）
    for (int i = nums.length - 1; i >= 0; i--) {
```

```
            int value = nums[i];
            int pos = count[value] - 1; // 计算元素的最终位置
            output[pos] = value;
            count[value]--; // 更新count，确保下一个相同元素的位置正确
        }

        // 第四步：将有序元素拷贝回原数组
        System.arraycopy(output, 0, nums, 0, nums.length);
    }
```

- 前两步与不稳定版一致：统计元素出现次数。
- 计算前缀和：count[i] = count[i] + count[i-1]，此时 count[i] 表示“小于等于 i 的元素个数”，即元素 i 的最终位置最大索引为 count[i] - 1。这是稳定排序的关键一步，通过前缀和确定元素的位置范围。
- 倒序遍历与写入：倒序遍历原数组，对于每个元素 value，计算其最终位置 pos = count[value] - 1，将 value 写入 output[pos]，然后 count[value]--。为什么倒序？倒序遍历可保证相等元素的相对位置与原数组一致（例如原数组中 [2, 1, 2]，倒序遍历第一个 2 时 pos = count[2]-1，第二个 2 时 pos = count[2]-1（此时 count[2] 已减 1），两个 2 的相对位置不变），实现稳定排序。
- 拷贝回原数组：将临时数组 output 中的有序元素拷贝回原数组 nums。

1.3 复杂度与适用场景

- 时间复杂度：O(n + max)，其中 n 是待排序数组长度，max 是数组中的最大值。统计次数、计算前缀和、倒序写入、拷贝数组均为 O(n) 或 O(max) 操作，整体为线性时间。
- 空间复杂度：O(n + max)，不稳定版为 O(max)（计数数组），稳定版为 O(n + max)（计数数组 + 临时数组 output）。
- 适用场景：元素为非负整数、取值范围较小的场景（max 不宜过大，否则计数数组占用空间过多）；稳定版适用于需要保持相等元素相对位置的场景（如基数排序的子排序步骤）；不适用于元素取值范围过大或包含负数的数据。

2. 基数排序（RadixSort）

2.1 核心思想

基数排序是“多轮桶排序”，基于元素的“位”进行排序。从元素的最低位（个位）开始，依次对每一位进行桶排序（通常使用计数排序作为子排序，保证稳定性）；每轮排序后，数组按当前位有序；当所有位都排序完成后，数组整体有序。代码中采用“最低位优先（LSD）”策略，使用 10 个桶（对应 0-9 的数字）。

2.2 代码逻辑拆解

```
public void radixSort(int[] arr) {
    if (arr == null || arr.length <= 1) return; // 边界处理

    List<Integer>[] buckets = new ArrayList[10]; // 10个桶，对应0-9的数字
    for (int i = 0; i < 10; i++) buckets[i] = new ArrayList<>(); // 初始化每个桶
```

```
    int divisor = 1; // 除数，用于提取当前位（1-个位，10-十位，100-百位...）
    boolean hasMoreDigit = true; // 标记是否还有更高位需要排序

    while (hasMoreDigit) {
        hasMoreDigit = false;

        // 第一步：将元素按当前位放入对应桶中
        for (int num : arr) {
            int digit = (num / divisor) % 10; // 提取当前位的数字（0-9）
            buckets[digit].add(num);

            // 若num/divisor > 0，说明存在更高位，需继续排序
            if (num / divisor > 0) hasMoreDigit = true;
        }

        // 第二步：将桶中的元素按顺序写回原数组
        int idx = 0;
        for (int i = 0; i < 10; i++) {
            for (int num : buckets[i]) {
                arr[idx++] = num;
            }
            buckets[i].clear(); // 清空桶，准备下一轮排序
        }

        divisor *= 10; // 除数乘10，准备提取更高位（个位→十位→百位...）
    }
}
```

- 边界处理：空数组或单元素数组直接返回。
- 桶初始化：创建 10 个 ArrayList 作为桶，对应数字 0-9；每个桶用于存放当前位数字为对应值的元素。
- 多轮排序循环：
    - 初始化 divisor = 1（提取个位），hasMoreDigit = true（标记是否需要继续排序）。
    - 元素入桶：遍历原数组，通过 (num / divisor) % 10 提取当前位数字 digit，将 num 放入 buckets[digit] 中；同时判断 num / divisor 是否大于 0，若大于 0，说明存在更高位，将 hasMoreDigit 设为 true，下一轮继续排序。
    - 元素回写：遍历 10 个桶，按桶的顺序（0-9）将桶中的元素依次写回原数组，此时数组按当前位有序；写回后清空桶，准备下一轮排序。
    - 更新 divisor：divisor *= 10，用于下一轮提取更高位（个位→十位→百位...），直到 hasMoreDigit = false（所有元素的最高位已排序完成）。
- 稳定性保证：桶排序的过程中，元素按入桶顺序出桶，相等位的元素相对位置不变，因此基数排序是稳定排序。稳定性是基数排序的关键，确保高位排序时，低位已排序的顺序不被打乱。

2.3 复杂度与适用场景

- 时间复杂度：O(d * (n + r))，其中 d 是元素的最大位数，n 是数组长度，r 是基数（此处 r=10）。每轮桶排序的时间复杂度为 O(n + r)，共需 d 轮，整体为线性时间。
- 空间复杂度：O(n + r)，主要消耗在 10 个桶的存储空间上（最坏情况下所有元素放入一个桶，空间为 O(n)）。
- 适用场景：非负整数、位数较少的大规模数据排序（如身份证号、手机号、年份等）；不适用于包含负数或位数过多的数据（位数过多会导致 d 增大，时间复杂度上升）。

四、总结：各排序算法核心特性对比

排序算法	时间复杂度（平均）	空间复杂度	稳定性	核心优势	适用场景
冒泡排序	$O(n^2)$	O(1)	稳定	实现简单，无需额外空间	小规模、接近有序数据
选择排序	$O(n^2)$	O(1)	不稳定	交换次数少	小规模、交换代价高的场景
插入排序	$O(n^2)$	O(1)	稳定	接近有序时效率高	小规模、接近有序数据
希尔排序	O(n^(3/2))	O(1)	不稳定	效率优于基础$O(n^2)$排序	中大规模数据
归并排序	O(n log n)	O(n)	稳定	效率稳定，无最坏情况	大规模、要求稳定的场景
快速排序	O(n log n)	O(log n)	不稳定	原地排序，实际效率高	大规模、无稳定性要求的场景
堆排序	O(n log n)	O(1)	不稳定	原地排序，可快速获取最值	大规模、空间敏感的场景
计数排序	O(n + max)	O(n + max)	稳定（版本）	线性时间，无元素比较	非负整数、取值范围小的场景
基数排序	O(d*(n + r))	O(n + r)	稳定	线性时间，适用于多位数数据	非负整数、位数少的大规模数据