常见的七种排序算法解析

选择排序

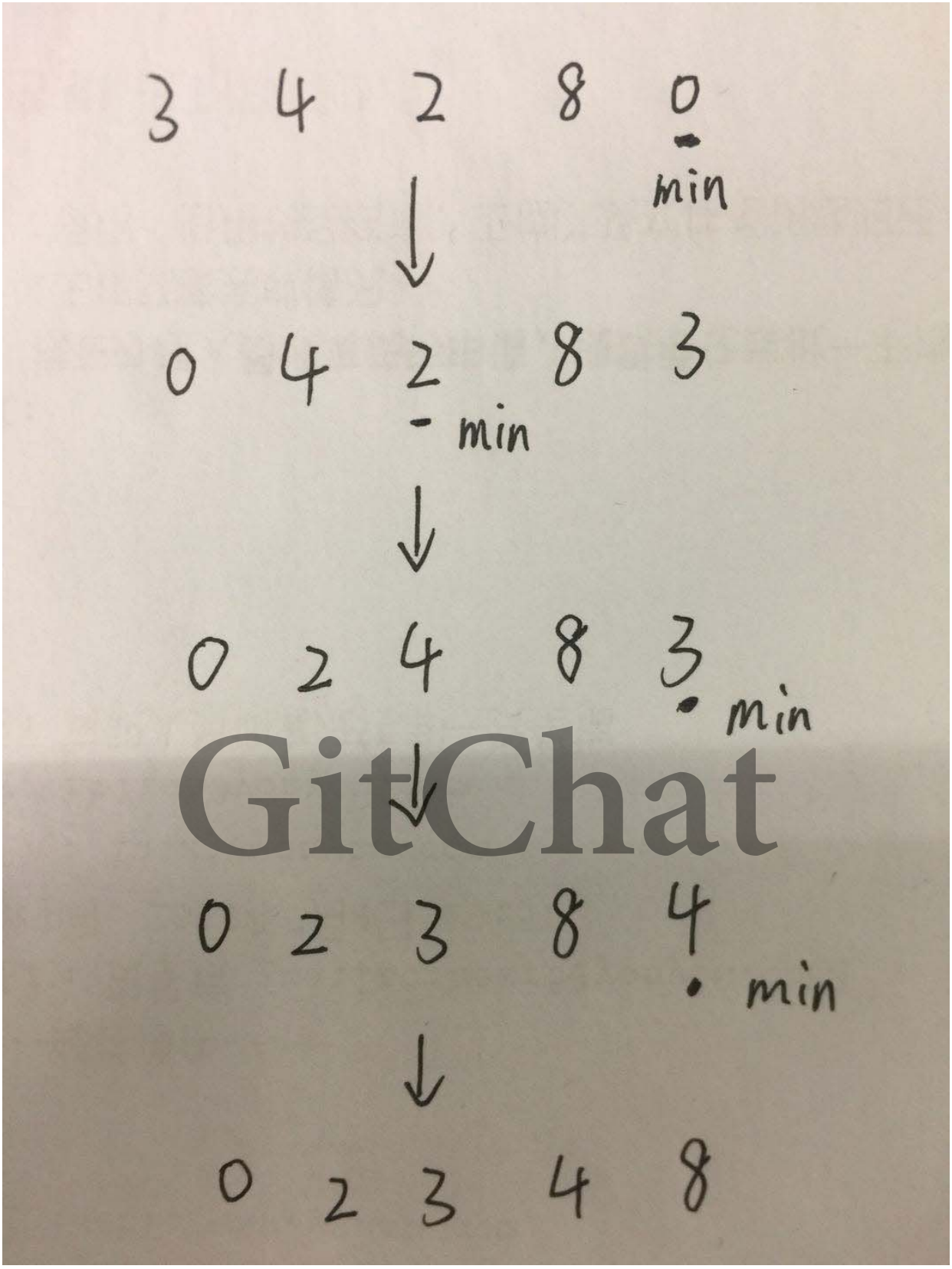
实现原理

首先从未排序序列中找到最小的元素，放置到排序序列的起始位置，然后从剩余的未排序序列中继续寻找最小元素，放置到已排序序列的末尾。所以称之为选择排序。

代码实现

```
public static int[] selectionSort(int[] arr){
    if (null == arr || arr.length == 0){
        return null;
    }

    int length = arr.length;
    for (int i = 0; i < length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < length; j++) {
            if (arr[j] < arr[min]){
                min = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
    return arr;
}
```


案例分析



时间复杂度与空间复杂度

每次要找一遍最小值，最坏情况下找n次，这样的过程要执行n次，所以时间复杂度还是O(n^2)。空间复杂度是O(1)。

快速排序

实现原理

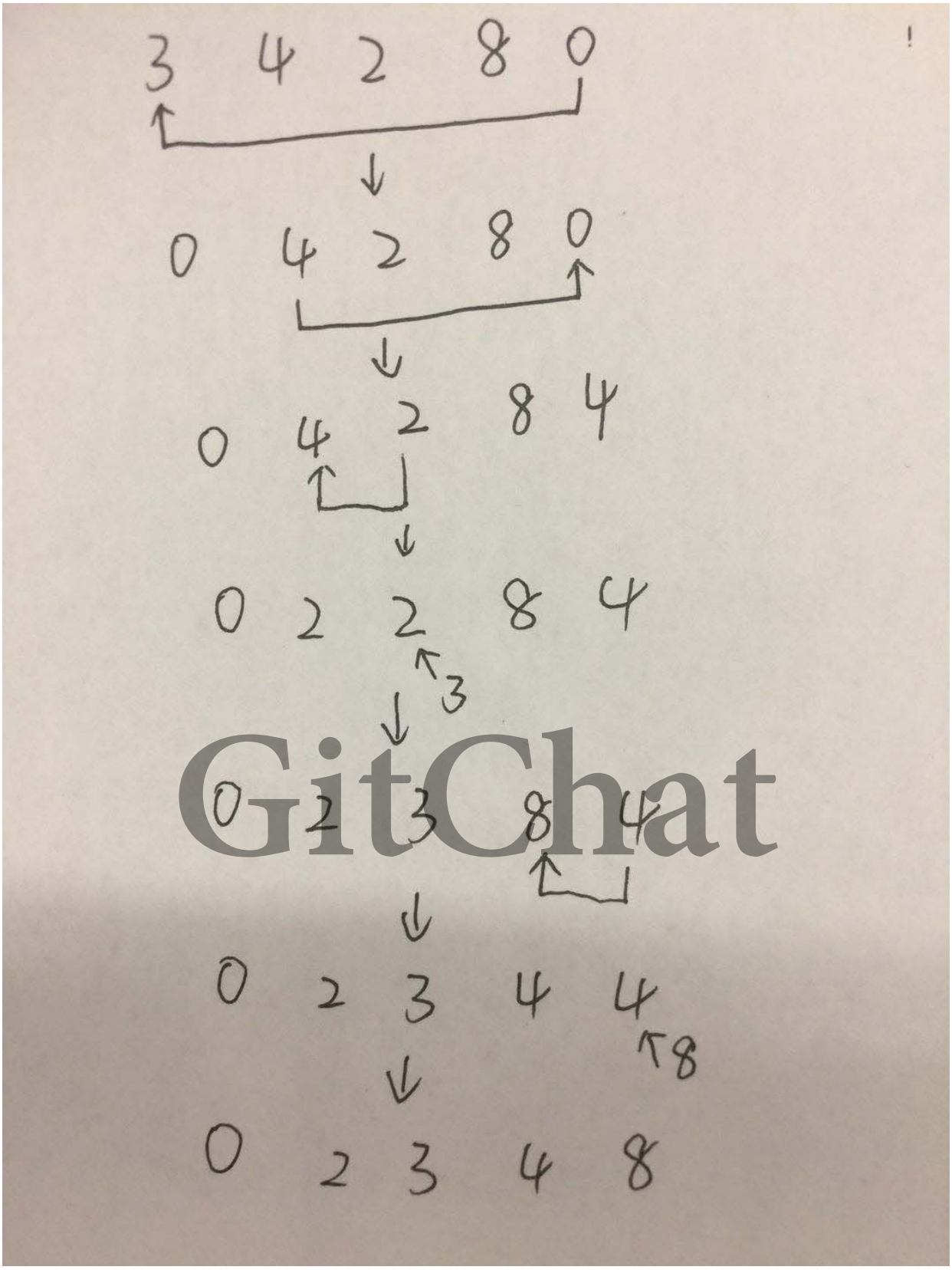
- 在数据集之中，选择一个元素作为”基准”（ pivot ）。
- 所有小于”基准”的元素，都移到”基准”的左边；所有大于”基准”的元素，都移到”基准”的右边。这个操作称为分区 (partition)。

  操作，分区操作结束后，基准元素所处的位置就是最终排序后它的位置。
- 对”基准”左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。

代码实现

```
        public static int partition(int[] array, int lo, int hi)
{
            // 固定的切分方式
            int key = array[lo];
            while (lo < hi) {
                while (array[hi] >= key && hi > lo) {// 从后半部分
向前扫描
                    hi--;
                }
                array[lo] = array[hi];
                while (array[lo] <= key && hi > lo) {// 从前半部分
向后扫描
                    lo++;
                }
                array[hi] = array[lo];
            }
            array[hi] = key;
            return hi;
        }

        public static int[] sort(int[] array, int lo, int hi) {
            if (lo >= hi) {
                return array;
            }
            int index = partition(array, lo, hi);
            sort(array, lo, index - 1);
            sort(array, index + 1, hi);
            return array;
        }
```


案例分析



时间复杂度与空间复杂度

快速排序也是一个不稳定排序，平均时间复杂度是O(nlogn)。空间复杂度是O(logn)。

冒泡排序

实现原理

依次比较相邻的两个元素，如果第一个元素大于第二个元素就交换它们的位置。这样比较一轮之后，最大的元素就会跑到队尾。然后对未排序的序列重复这个过程，最终转换成有序序列。

代码实现

```
public static int[] bubbleSort(int[] arr){
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i -1; j++) {
            if (arr[j] > arr[j + 1]){
                arr[j] = arr[j] + arr[j + 1];
                arr[j + 1] = arr[j] - arr[j + 1];
                arr[j] = arr[j] - arr[j + 1];
            }
        }
    }
    return arr;
}
```


案例分析

以数组 arr = [3 4 2 8 0] 为例说明，加粗的数字表示每次循环要比较的两个数字：

第一次外循环

(**3 4** 2 8 0) → (**3 4** 2 8 0)， 4 > 3 位置不变
(3 **4 2** 8 0) → (3 **2 4** 8 0)， 4 > 2 交换位置
(3 2 **4 8** 0) → (3 2 **4 8** 0)， 8 > 4 位置不变
(3 2 4 **8 0**) → (3 2 4 **0 8**)， 8 > 0 交换位置

第二次外循环（除开最后一个元素8，对剩余的序列）

(**3 2** 4 0 8) → (**2 3** 4 0 8)， 3 > 2 交换位置
(2 **3 4** 0 8) → (2 **3 4** 0 8)， 4 > 3 位置不变
(2 3 **4 0** 8) → (2 3 **0 4** 8)， 4 > 0 交换位置

第三次外循环（除开已经排序好的最后两个元素，对剩余的循环，直到剩余的序列为1）

(**2 3** 0 4 8) → (**2 3** 0 4 8)，3 > 2 位置不变
(2 **3 0** 4 8) → (2 **0 3** 4 8)，3 > 0 交换位置

第四次外循环（最后一次）

(**2 0** 3 4 8) → (**0 2** 3 4 8)，2 > 0 交换位置

时间复杂度与空间复杂度

由于我们要重复执行n次冒泡，每次冒泡要执行n次比较（实际是1到n的等差数列，也就是(a1 + an) * n / 2），也就是 O(n^2)。 空间复杂度是O(1)。

插入排序

实现原理

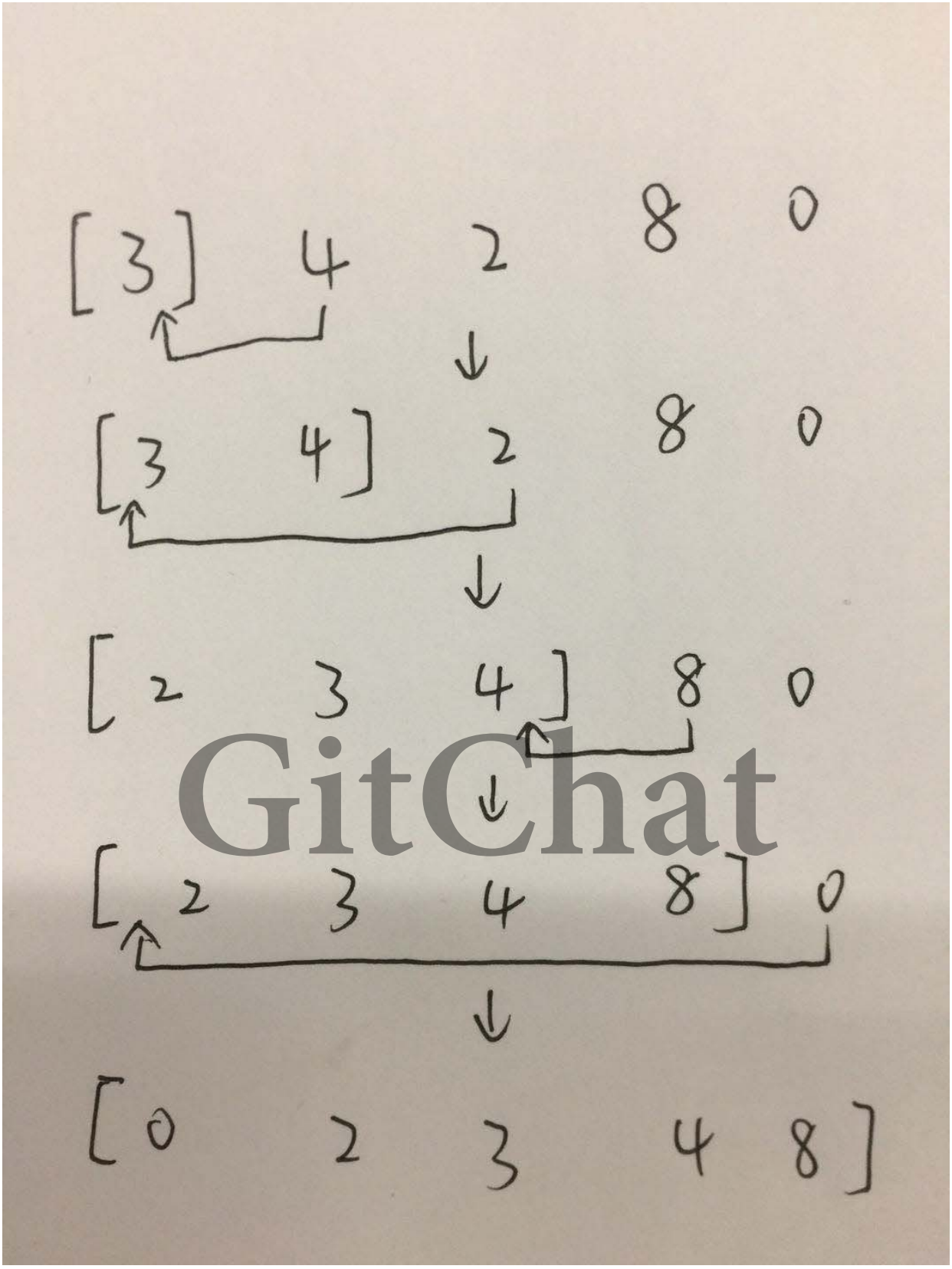
- 认为第一个元素是排好序的，从第二个开始遍历。
- 拿出当前元素的值，从排好序的序列中从后往前找。
- 如果序列中的元素比当前元素大，就把它后移。直到找到一个小的。
- 把当前元素放在这个小的后面（后面的比当前大，它已经被后移了）。

代码实现

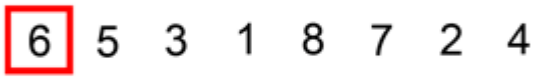

```
public static int[] insertionSort(int[] arr){
    for (int i = 1; i < arr.length; i++) {
        for (int j = i; j > 0; j--) {
            if (arr[j] < arr[j - 1]){
                int temp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = temp;
            }
        }
    }
    return arr;
}
```

原理图解

案例1



案例2



时间复杂度与空间复杂度

因为要选择n次，而且插入时最坏要比较n次，所以时间复杂度同样是O(n^2)。空间复杂度是O(1)。

希尔排序

实现原理

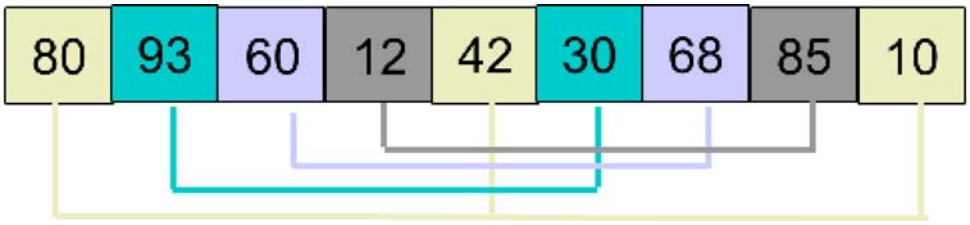
- 先取一个正整数 d1(d1 < n)，把全部记录分成 d1 个组，所有距离为 d1 的倍数的记录看成一组，然后在各组内进行插入排序
- 然后取 d2(d2 < d1)
- 重复上述分组和排序操作；直到取 di = 1(i >= 1) 位置，即所有记录成为一个组，最后对这个组进行插入排序。一般选 d1 约为 n/2，d2 为 d1 /2， d3 为 d2/2 ，... ， di = 1。

代码实现

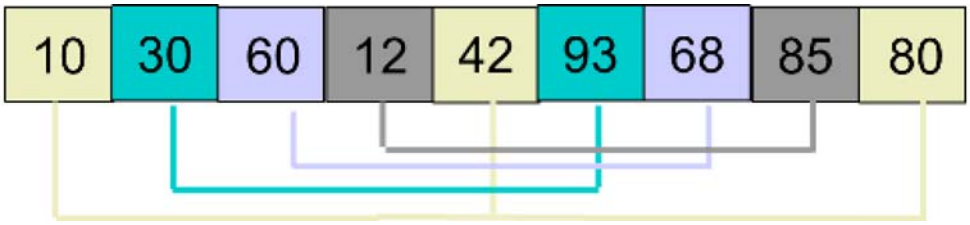
```
public static int[] shellSort(int[] arr){
    for (int gap = arr.length/2; gap > 0 ; gap/=2) {
        for (int i = gap; i < arr.length; i++) {
            int j = i;
            while (j-gap>=0 && arr[j] < arr[j-gap]){
                int temp = arr[j];
                arr[j] = arr[j-gap];
                arr[j-gap] = temp;
                j -= gap;
            }
        }
    }
    return arr;
}
```

案例分析

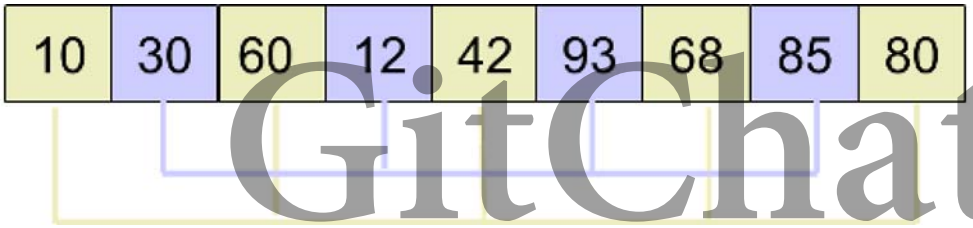
假设有数组 array = [80, 93, 60, 12, 42, 30, 68, 85, 10]，首先取 d1 = 4，将数组分为 4 组，如下图中相同颜色代表一组：



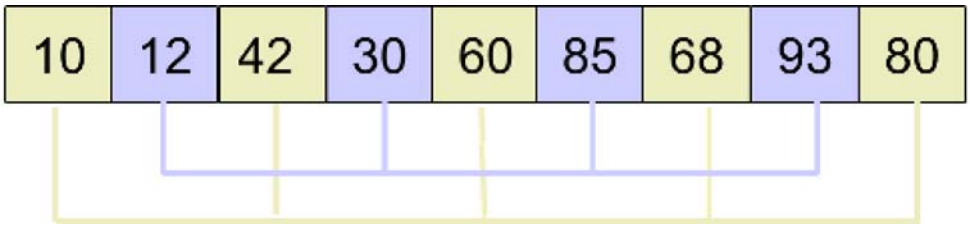
然后分别对 4 个小组进行插入排序，排序后的结果为：



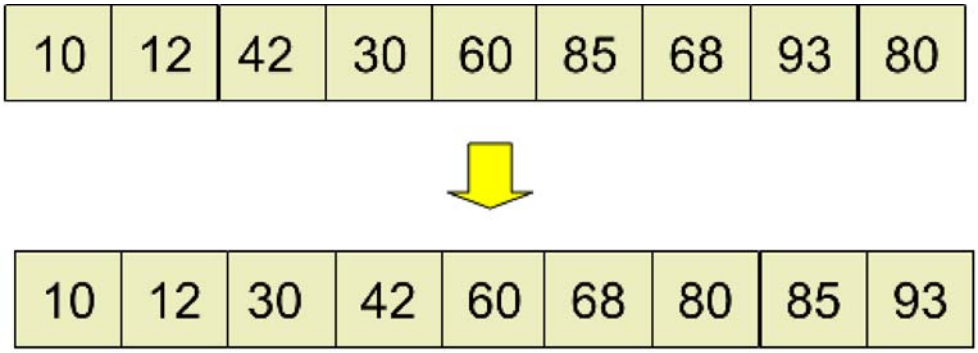
然后，取 d2 = 2，将原数组分为 2 小组，如下图：



然后分别对 2 个小组进行插入排序，排序后的结果为：



最后，取 d3 = 1，进行插入排序后得到最终结果：



时间复杂度与空间复杂度

希尔排序的时间复杂度受步长的影响，平均时间复杂度是O(n log2 n),空间复杂度是O(1)。

归并排序

实现原理

- 把 n 个记录看成 n 个长度为 l 的有序子表
- 进行两两归并使记录关键字有序，得到 n/2 个长度为 2 的有序子表
- 重复第 2 步直到所有记录归并成一个长度为 n 的有序表为止。

总而言之，归并排序就是使用递归，先分解数组为子数组，再合并数组。

代码实现


```
    public static int[] mergeSort(int[] arr){
        int[] temp =new int[arr.length];
        internalMergeSort(arr, temp, 0, arr.length-1);
        return temp;
    }
    private static void internalMergeSort(int[] a, int[] b, int
left, int right){
        //当left==right的时，已经不需要再划分了
        if (left<right){
            int middle = (left+right)/2;
            internalMergeSort(a, b, left, middle);          //左
子数组
            internalMergeSort(a, b, middle+1, right);       //右
子数组
            mergeSortedArray(a, b, left, middle, right);    //合
并两个子数组
        }
    }
    // 合并两个有序子序列 arr[left, ..., middle] 和 arr[middle+1,
..., right]。temp是辅助数组。
    private static void mergeSortedArray(int arr[], int temp[],
int left, int middle, int right){
        int i=left;
        int j=middle+1;
        int k=0;
        while ( i<=middle && j<=right){
            if (arr[i] <=arr[j]){
                temp[k++] = arr[i++];
            }
```

```
            else{
                temp[k++] = arr[j++];
            }
        }
        while (i <=middle){
            temp[k++] = arr[i++];
        }
        while ( j<=right){
            temp[k++] = arr[j++];
        }
        //把数据复制回原数组
        for (i=0; i<k; ++i){
            arr[left+i] = temp[i];
        }
    }
```

案例分析

案例1

以数组 array = [4 2 8 3 5 1 7 6] 为例，首先将数组分为长度为 2 的子数组，并使每个子数组有序：

[4 2] [8 3] [5 1] [7 6]
↓
[2 4] [3 8] [1 5] [6 7]

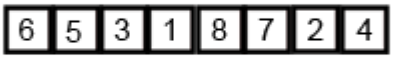
然后再两两合并：

[2 4 3 8] [1 5 6 7]
↓
[2 3 4 8] [1 5 6 7]

最后将两个子数组合并：

[2 3 4 8 1 5 6 7]
↓
[1 2 3 4 5 6 7 8]

案例2



时间复杂度与空间复杂度

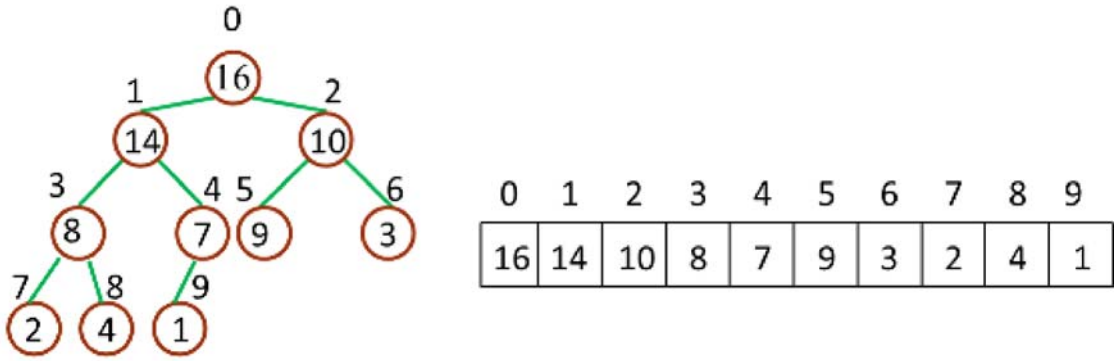
在合并数组过程中，实际的操作是当前两个子数组的长度，即2m。又因为打散数组是二分的，最终循环执行数是logn。所以这个算法最终时间复杂度是O(nlogn)，空间复杂度是O(1)。

堆排序

实现原理

堆排序就是把最大堆堆顶的最大数取出，将剩余的堆继续调整为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。在堆中定义以下几种操作：

- 最大堆调整（Max-Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
- 创建最大堆（Build-Max-Heap）：将堆所有数据重新排序，使其成为最大堆
- 堆排序（Heap-Sort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算



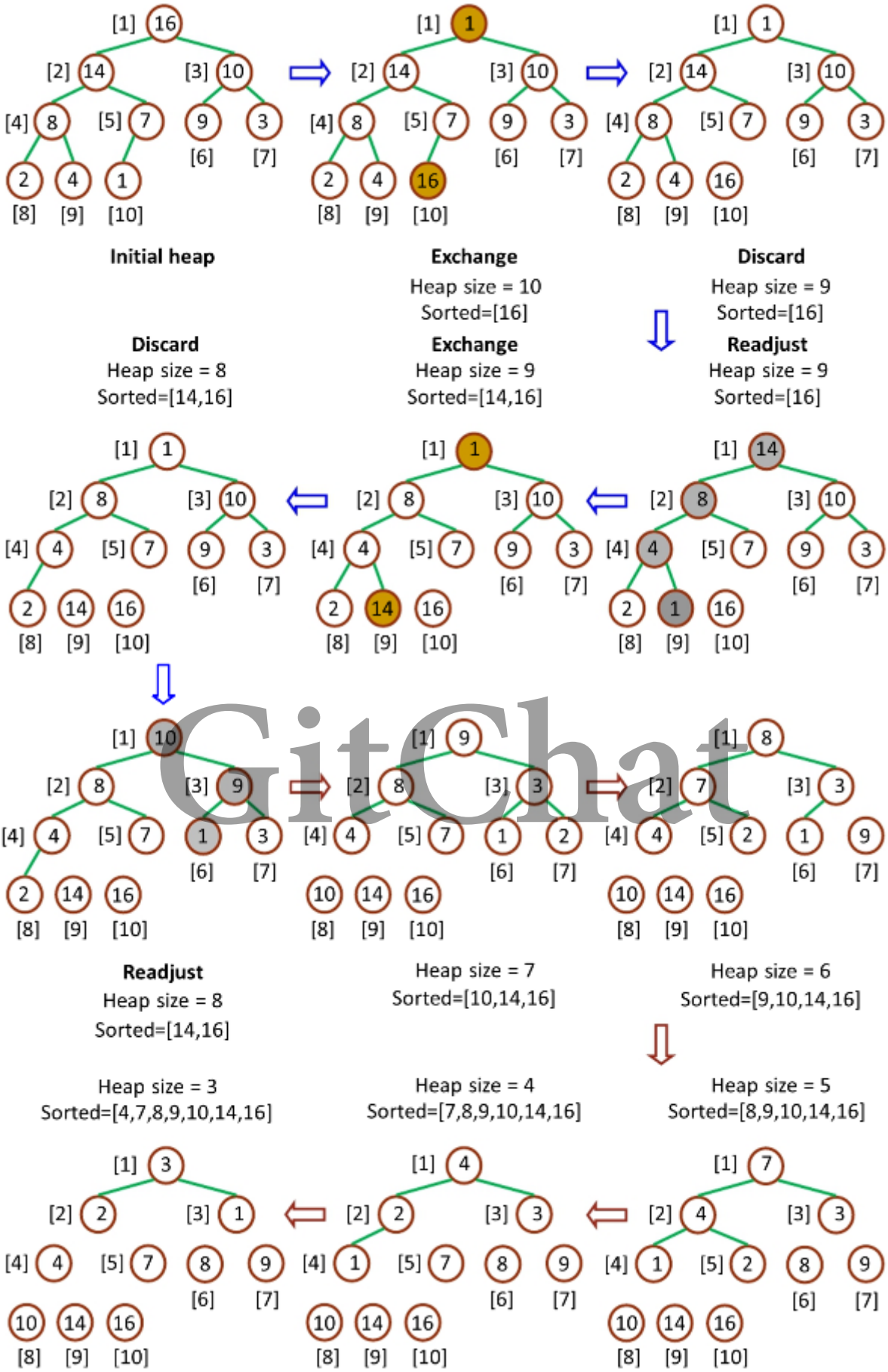
- Parent(i) = floor((i-1)/2)，i 的父节点下标
- Left(i) = 2i + 1，i 的左子节点下标
- Right(i) = 2(i + 1)，i 的右子节点下标

代码实现

```
    /**
     * 堆排序
     */
    public static int[] heapSort(int[] arr) {
        // 将待排序的序列构建成一个大顶堆
        for (int i = arr.length / 2; i >= 0; i--){
            heapAdjust(arr, i, arr.length);
        }

        // 逐步将每个最大值的根节点与末尾元素交换，并且再调整二叉树，使其成为大顶堆
        for (int i = arr.length - 1; i > 0; i--) {
            swap(arr, 0, i); // 将堆顶记录和当前未经排序子序列的最后一个记录交换
            heapAdjust(arr, 0, i); // 交换之后，需要重新检查堆是否符合大顶堆，不符合则要调整
        }
        return arr;
    }

    /**
     * 构建堆的过程
     * @param arr 需要排序的数组
     * @param i 需要构建堆的根节点的序号
     * @param n 数组的长度
     */
    private static void heapAdjust(int[] arr, int i, int n) {
        int child;
        int father;
        for (father = arr[i]; leftChild(i) < n; i = child) {
            child = leftChild(i);

            // 如果左子树小于右子树，则需要比较右子树和父节点
            if (child != n - 1 && arr[child] < arr[child + 1]) {
                child++; // 序号增1，指向右子树
            }

            // 如果父节点小于孩子结点，则需要交换
            if (father < arr[child]) {
                arr[i] = arr[child];
            } else {
                break; // 大顶堆结构未被破坏，不需要调整
            }
        }
        arr[i] = father;
    }

    // 获取到左孩子结点
```

```
    private static int leftChild(int i) {
        return 2 * i + 1;
    }

    // 交换元素位置
    private static void swap(int[] arr, int index1, int index2) {
        int tmp = arr[index1];
        arr[index1] = arr[index2];
        arr[index2] = tmp;
    }
```

案例分析



时间复杂度与空间复杂度

堆执行一次调整需要O(logn)的时间，在排序过程中需要遍历所有元素执行堆调整，所以最终时间复杂度是O(nlogn)。空间复杂度是O(1)。