一、如何分析排序算法

1、时间复杂度

1.1最好情况、最坏情况、平均情况

有的排序算法这几种情况区别比较大；

数据有序度不同，对排序时间有影响，数据有序度对不同排序算法的影响程度也不同。

1.2时间复杂度的系数、常数、低阶

小规模数据的排序需要考虑这些。

1.3比较和交换次数

基于比较的排序算法的执行过程，会涉及两种操作，一种是元素比较大小，另一种是元素交换或移动。

2、空间复杂度

原地排序（Sorted in place）：指空间复杂度是 O(1) 的排序算法。

3、稳定性

稳定性：如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

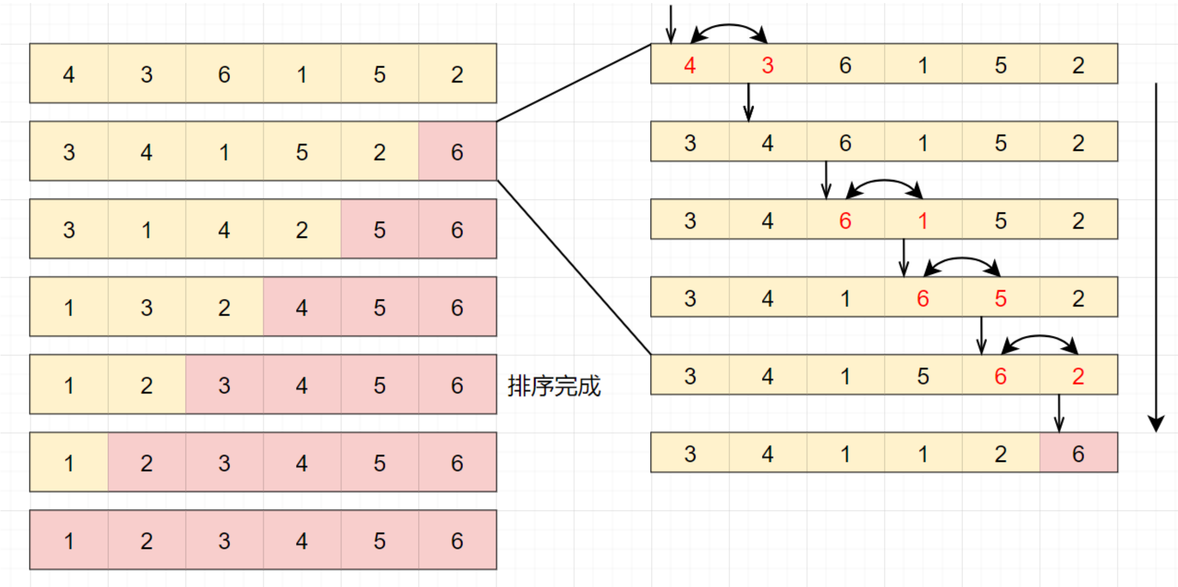
稳定的排序算法：相等的值在排序前后相对顺序没改变。

不稳定的排序算法：相等的值在排序前后相对顺序发生改变。

二、冒泡排序

1、原理与代码

冒泡排序只会操作相邻的两个数据。每次冒泡操作都会对相邻的两个元素进行比较，看是否满足大小关系要求。如果不满足就让它俩互换。如果是从小到大的排序，每次冒泡都会把最大的数放在最后。




```
func BubbleSort(a []int, n int) {
    if n <= 1 {
        return
    }

    for i := 0; i < n; i++ {
        for j := 0; j < n-i-1; j++ {
            // 将更大的元素交换到后面。
            if a[j] > a[j+1] {
                a[j], a[j+1] = a[j+1], a[j]
            }
        }
    }
}
```

1.1一个简单的优化

使用一个标志位，当一次没有元素交换的时候，说明数组已经有序了，这时就可以直接返回了。

```
func BubbleSort(a []int, n int) {
    if n <= 1 {
        return
    }
    for i := 0; i < n; i++ {
        // 提前退出标志
        flag := false
        for j := 0; j < n-i-1; j++ {
            if a[j] > a[j+1] {
                a[j], a[j+1] = a[j+1], a[j]
                flag = true      //此次冒泡有数据交换
            }
        }
        // 如果没有交换数据，提前退出
        if !flag {
            break
        }
    }
}
```

2、分析

是原地排序算法：只涉及相邻元素交换，不需要额外的空间。

是稳定的排序算法：当两个元素相等的时候，不交换就稳定了。

时间复杂度：最好O(n)；最坏O(n^2)；平均O(n^2)。

三、插入排序

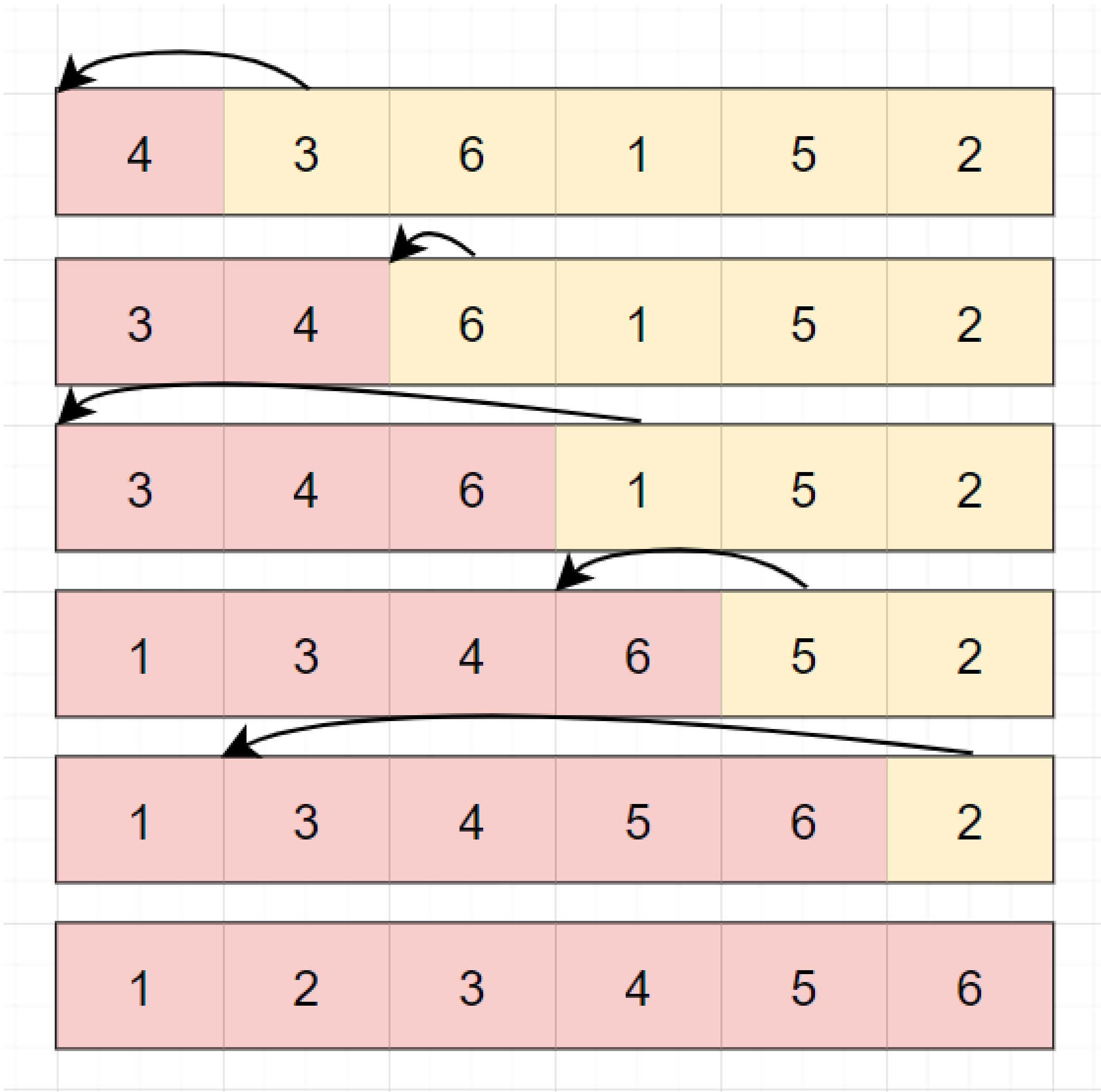
1、原理与代码

1.1算法过程：

1. 将数组中的数据分为两个区间，已排序区间和未排序区间。初始已排序区间只有数组的第一个元素。
2. 取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。
3. 重复第二步，直到未排序区间为空。

1.2第二步的实现：

当将一个数据 a 插入到已排序区间时，需要拿 a 与已排序区间的元素依次比较大小，找到合适的插入位置。找到插入点之后，还需要将插入点之后的元素顺序往后移动一位，这样才能腾出位置给元素 a 插入。



1.3代码：

```
func InsertionSort(a []int, n int) {
    if n <= 1 {
        return
    }
    for i := 1; i < n; i++ {
        value := a[i]
        // i后面是未排序部分，i前面是已排序的。
        // 从后向前遍历找到合适位置，因为每次向前找的时候，要把更大的元素向后移
        for j := i - 1; j >= 0; j-- {
            if a[j] > value {
                a[j+1] = a[j]
            } else {
                break
            }
        }
        // 找到位置之后，将value元素插到正确的位置。
        a[j+1] = value
    }
}
```

2、分析

是原地排序算法：只涉及相邻元素交换，不需要额外的空间。

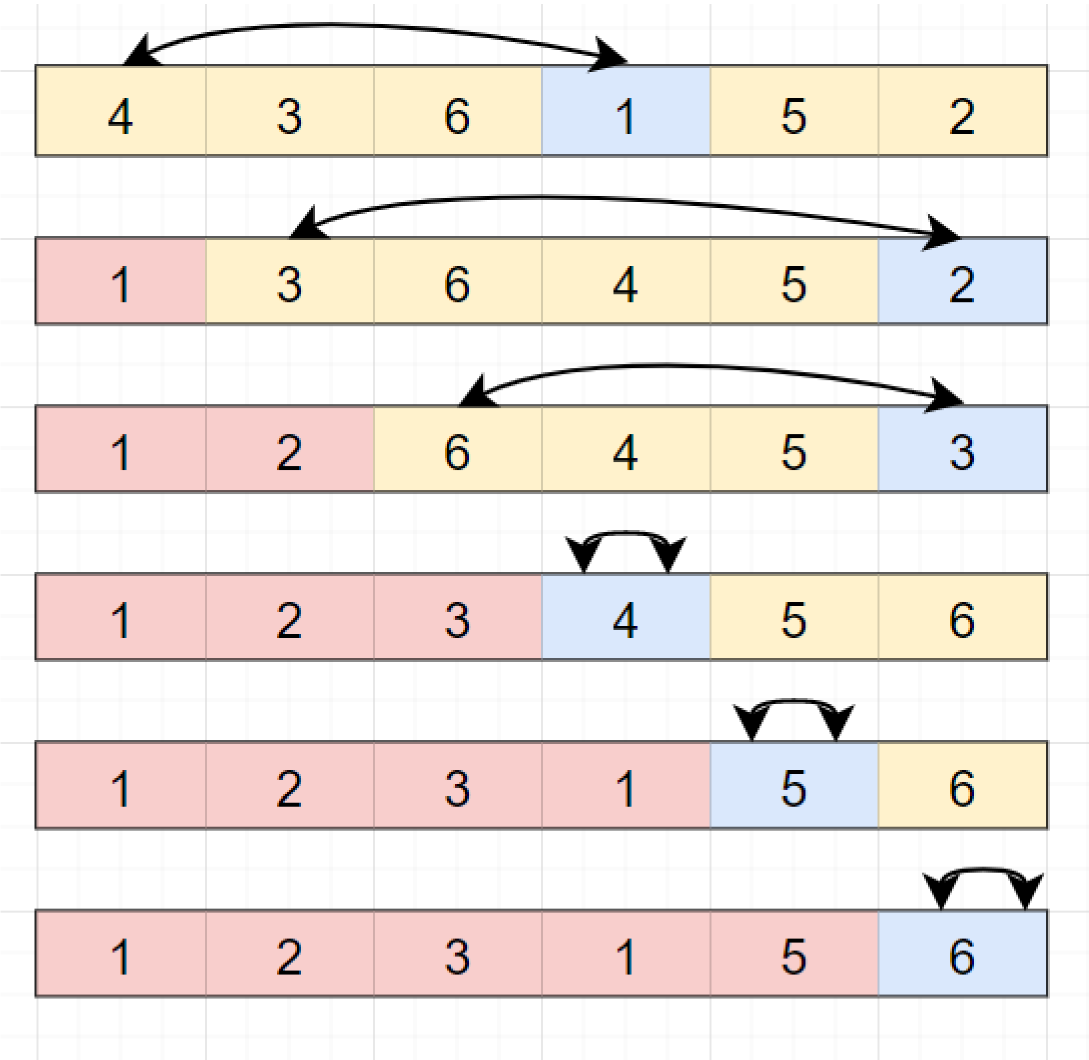
是稳定的排序算法：当两个元素相等的时候，将后面出现的元素，插入到前面出现元素的后面。

时间复杂度：最好O(n)；最坏O(n^2)；平均O(n^2)。

四、选择排序

1、原理与代码

将数组分为已排序区间和未排序区间。选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾。




```
func SelectionSort(a []int, n int) {
    if n <= 1 {
        return
    }

    for i := 0; i < n; i++ {
        // 查找最小值
        minIndex := i
        // i后面是未排序部分，i前面是已排序的。
        // 每次找到未排序部分中最小元素的索引
        for j := i + 1; j < n; j++ {
            if a[j] < a[minIndex] {
                minIndex = j
            }
        }
```

```
            // 交换
            a[i], a[minIndex] = a[minIndex], a[i]
        }
    }
```

2、分析

是原地排序算法：只涉及相邻元素交换，不需要额外的空间。

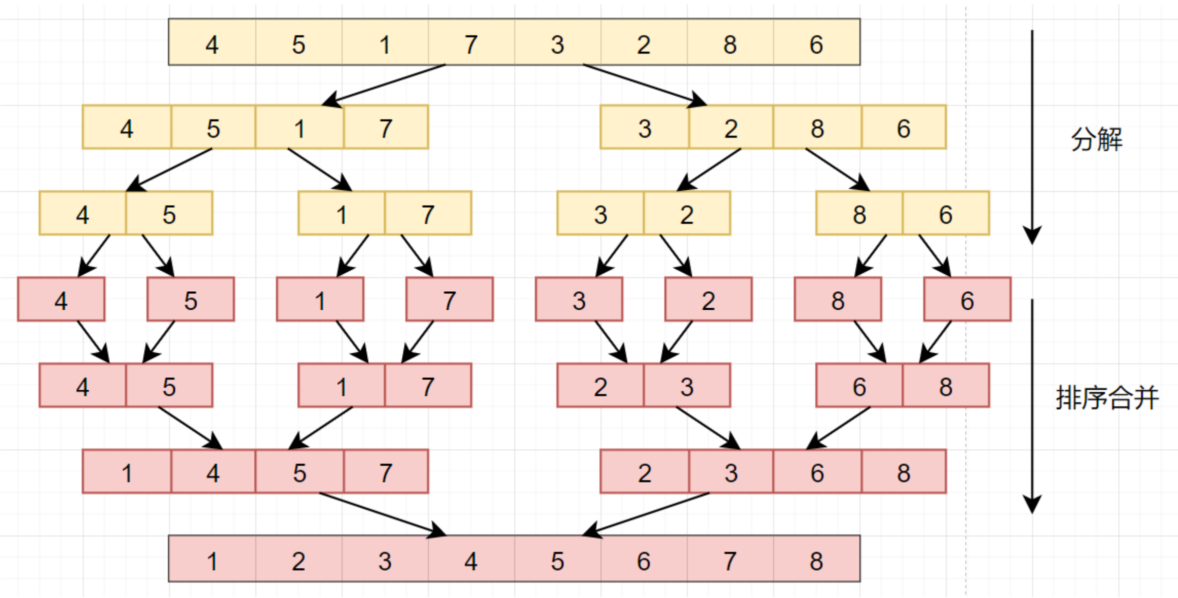
是不稳定的排序算法：每次找到未排序区间的最小值，与前面的元素交换位置，破坏了稳定性。

时间复杂度：最好O(n)；最坏O(n^2)；平均O(n^2)。

五、归并算法

1、归并算法思想

归并算法是典型的分治思想，将一个大问题分解成小的子问题来解决。放在排序中就是，先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。



2、代码

2.1递归以代码一般模板

```
void recursion(int level, int param) {
    // 递归终止条件
    if (level > MAX_LEVEL) {
        // 处理结果
        return ;
    }

    // 处理当前逻辑
    process(level, param);

    // 下探到下一层
    recursion(level + 1, param);
```

```
    // 恢复当前level状态，一般回溯算法需要添加逻辑
}
```

2.2衍生到分治代码模板

```
int divide_conquer(Problem *problem, int params) {
    // 递归终止条件
    if (problem == nullptr) {
        // 处理结果
        return result;
    }

    // 处理当前层（计算子问题）
    subproblems = split_problem(problem, data)

    // 下探到下一层，下探到子问题
    subresult1 = divide_conquer(subproblem[0], p1)
    subresult2 = divide_conquer(subproblem[1], p1)
    subresult3 = divide_conquer(subproblem[2], p1)

    // 合并结果
    result = process_result(subresult1, subresult2, subresult3)

    // 恢复当前level状态，一般回溯算法需要添加逻辑
}
```

2.3排序算法的实现

```
func MergeSort(arr []int) {
    arrLen := len(arr)
    if arrLen <= 1 {
        return
    }

    mergeSort(arr, 0, arrLen-1)
}

func mergeSort(arr []int, start, end int) {
    // 递归终止条件
    if start >= end {
        return
    }

    // 处理当前层，当前层只需要计算mid。
    mid := (start + end) / 2
    // 下探到下一层，下一层就是数组的左一半和右一半
    mergeSort(arr, start, mid)
    mergeSort(arr, mid+1, end)
    // 合并当前层，合并就是排序的过程。
    merge(arr, start, mid, end)
}

func merge(arr []int, start, mid, end int) {
    tmpArr := make([]int, end-start+1)
```

```
    i := start
    j := mid + 1
    k := 0

    // 两部分数组：start~mid和mid~end，将两部分排序后的元素放在tmpArr中，
    for ; i <= mid && j <= end; k++ {
        if arr[i] <= arr[j] {
            tmpArr[k] = arr[i]
            i++
        } else {
            tmpArr[k] = arr[j]
            j++
        }
    }
    // 把两部分（只可能剩下一部分）中剩下没存完的数据存到tmpArr
    for ; i <= mid; i++ {
        tmpArr[k] = arr[i]
        k++
    }
    for ; j <= end; j++ {
        tmpArr[k] = arr[j]
        k++
    }
    copy(arr[start:end+1], tmpArr)
}
```

3、性能分析

不是原地排序算法：合并函数无法在原地执行。临时内存空间最大也不会超过 n 个数据的大小，所以空间复杂度是 O(n)。

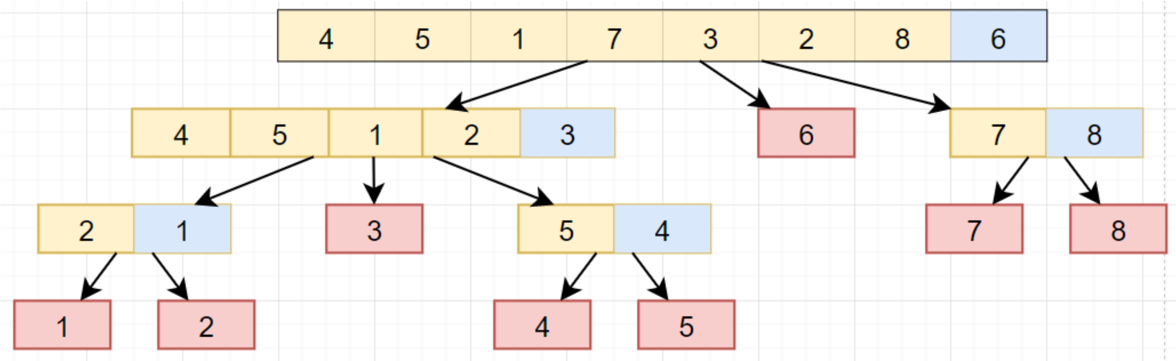
是稳定的排序算法：放入tmp数组的时候，保证相同元素前后顺序不变，保证了稳定。

时间复杂度：最好、最坏、平均都是O(nlogn)。

六、快速排序

1、快排思想

1. 如果要排序数组中下标从 p 到 r 之间的一组数据，选择 p 到 r 之间的任意一个数据作为 pivot（分区点）。
2. 遍历 p 到 r 之间的数据，将小于 pivot 的放到左边，将大于 pivot 的放到右边，将 pivot 放到中间。
3. 经过这一步骤之后，数组 p 到 r 之间的数据就被分成了三个部分，前面 p 到 q-1 之间都是小于 pivot 的，中间是 pivot，后面的 q+1 到 r 之间是大于 pivot 的。
4. 根据分治、递归的处理思想，我们可以用递归排序下标从 p 到 q-1 之间的数据和下标从 q+1 到 r 之间的数据，直到区间缩小为 1



2、代码

由于快排也是分支的思想，所以也可以用递归模板来写

```
func QuickSort(arr []int) {
    separateSort(arr, 0, len(arr)-1)
}

func separateSort(arr []int, start, end int) {
    // 递归终止条件
    if start >= end {
        return
    }
    // 处理当前层，完成当前层的分区
    i := partition(arr, start, end)

    // 下探到左子问题
    separateSort(arr, start, i-1)

    // 下探到右子问题
    separateSort(arr, i+1, end)
}

func partition(arr []int, start, end int) int {
    // 选取最后一位当对比数字
    pivot := arr[end]

    var i = start
    for j := start; j < end; j++ {
        if arr[j] < pivot {
            if !(i == j) {
                // 保证<pivot的数在前面，>pivot的数在后面
                arr[i], arr[j] = arr[j], arr[i]
            }
            i++
        }
    }

    arr[i], arr[end] = arr[end], arr[i]

    return i
}
```

- 归并排序的处理过程是由下到上的，先处理子问题，然后再合并；
- 快排正好相反，它的处理过程是由上到下的，先分区，然后再处理子问题。

3、快排分析

原地排序算法：使用原地分区的函数，就可以实现原地排序。空间复杂度就是O(1)

不稳定的排序算法：分区过程会打乱相同元素的相对顺序

时间复杂度：

- 当分区用的元素比较合理，能比较平均的进行分区，时间复杂度O(nlogn)
- 当每次选的元素都不合理，分区都极其不均衡，最坏时间复杂度为O(n^2)
- 但是大多数都是O(nlogn)，所以平均时间复杂度就是O(nlogn)

七、桶排序

1、核心思想

- 将要排序的数据分到几个有序的桶里，每个桶里的数据再单独进行排序。
- 桶内排完序之后，再把每个桶里的数据按照顺序依次取出，组成的序列就是有序的了。

2、时间复杂度

已知要排序的元素有n个，分到m个桶中，每个桶就k=n/m个元素。

- 如果每个桶内的元素用快速排序，时间复杂度为 O(k * logk)。
- m 个桶排序的时间复杂度就是 O(m * k * logk)，因为 k=n/m，所以整个桶排序的时间复杂度就是 O(n*log(n/m))。
- 当桶的个数 m 接近数据个数 n 时，log(n/m) 就是一个非常小的常量，这个时候桶排序的时间复杂度接近 O(n)。

3、桶排序的适用场景

3.1桶排序的局限

1. 要排序的数据需要很容易就能划分成 m 个桶，并且，桶与桶之间有着天然的大小顺序。这样桶与桶之间的数据不需要再进行排序。
2. 数据在各个桶之间的分布是比较均匀的。上述分析的O(n)的时间复杂度的前提是，数据较为平均，如果某一个桶的元素个数很多，极限情况是所有数据都分到1个桶里，就会退化为O(n * logn)。

3.2适用场景

- 适用在外部排序中，外部排序就是数据存储在外部磁盘中，数据量比较大，内存有限，无法将数据全部加载到内存中。
- MySQL中用B+树做索引也有桶排序的影子。

3.3代码

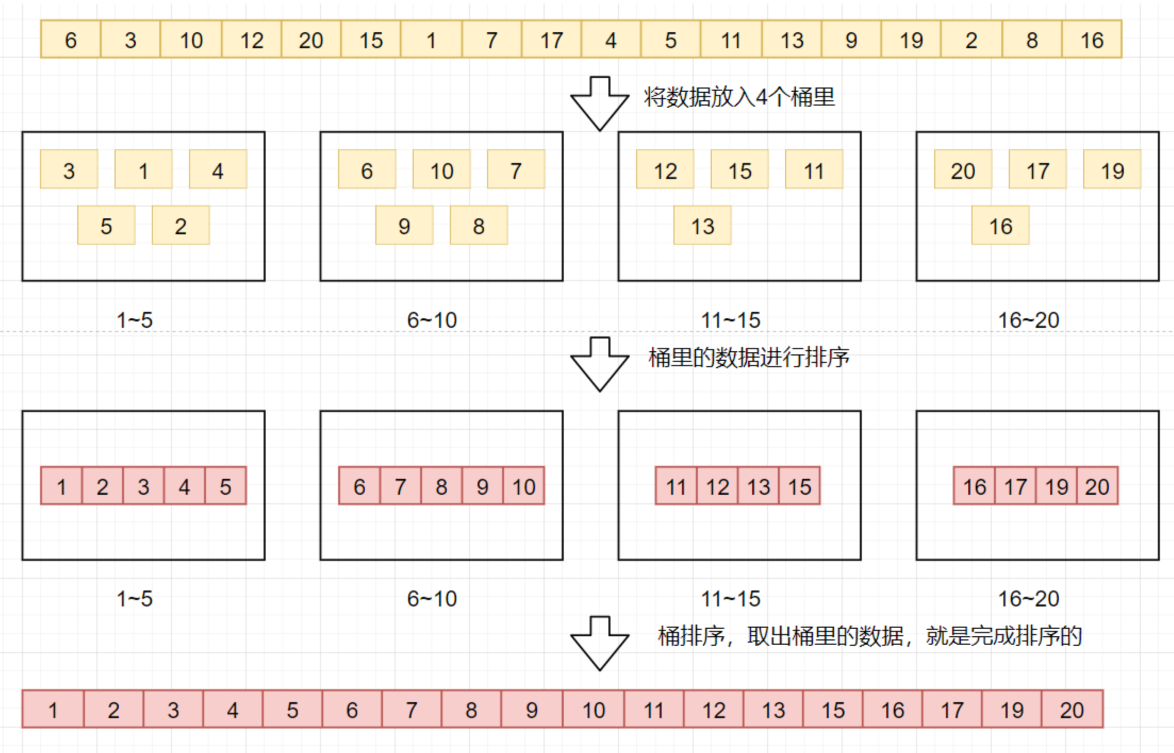
```
// 获取待排序数组中的最大值
func getMax(a []int)int{
    max := a[0]
    for i:=1; i<len(a); i++{
        if a[i] > max{
            max = a[i]
        }
    }
```

```go
    return max
}

func BucketSort(a []int)  {
    num := len(a)
    if num <= 1{
        return
    }
    max := getMax(a)
    buckets := make([][]int, num)  // 桶

    index :=0
    for i:=0; i < num; i++{
        index = a[i]*(num -1) / max  // 桶序号
        buckets[index] = append(buckets[index],a[i]) // 加入对应的桶中
    }

    tmpPos := 0  // 标记数组位置
    for i := 0; i < num; i++ {
        bucketLen := len(buckets[i])
        if bucketLen > 0{
            sort.Ints(buckets[i])         // 桶内排序
            copy(a[tmpPos:], buckets[i])  // 将桶中的元素存回a中
            tmpPos += bucketLen
        }
    }
}
```



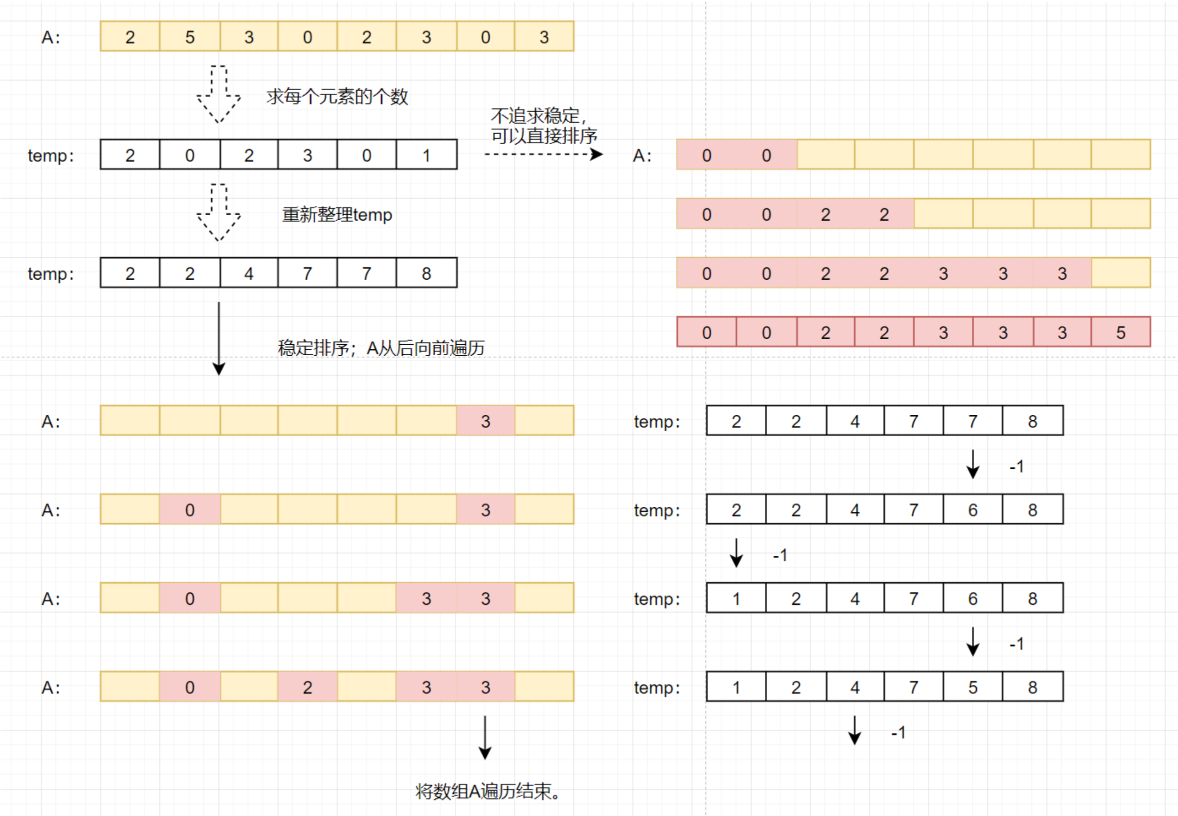
八、计数排序

可以看做是桶排序的一个特殊情况。当要排序的 n 个数据，所处的范围并不大的时候，比如最大值是 k，我们就可以把数据划分成 k 个桶。每个桶存1个数。

1、实现

假设数组A有8个元素，且有重复数据，具体的值为2，5，3，0，2，3，0，3

1. 首先，A的最大值为5，最小值为0，新建1个temp数组，6个元素，每个索引上的值是A中元素的个数。
2. 这时把temp的索引按元素个数取出来就已经完成排序了。这种排序方法不稳定，如果想满足稳定排序算法的条件需要更复杂。
3. 已知temp数组每个索引上的元素是A中元素的个数。然后对temp数组顺序求和，temp[i]里存的就是A中<=i的元素个数。
4. 从后向前扫描数组A，第一个扫到的元素为3，temp[3]=7，说明<=3的元素有7个，这个3的索引排序后的索引就是6，还要temp[3]--，因为<=3的数少了1个。
5. 扫到第二个3的时候，temp[3]=6，这个3的索引就是5。
6. 扫描结束，就排好序了，并且是稳定的排序算法。



2、代码实现

```
func CountingSort(a []int, n int) {
    if n <= 1 {
        return
    }

    // 求a中的最大值，默认a中的元素是0~max的。
    var max int = 0
    for i := range a {
        if a[i] > max {
            max = a[i]
        }
    }

    // 将元素个数存到temp中。temp[i]存的就是i的个数。
    temp := make([]int, max+1)
    for i := range a {
        temp[a[i]]++
    }
```

```
    // 累加，temp[i]中存的变为<=i的个数
    for i := 1; i <= max; i++ {
        temp[i] += temp[i-1]
    }

    // 排序，从后向前遍历
    r := make([]int, n)
    for i := n - 1; i >= 0; i-- {
        index := temp[a[i]] - 1 // 求i在排序后数组中的索引
        r[index] = a[i]
        temp[a[i]]--
    }

    copy(a, r)
}
```

3、应用

- 计数排序只能用在数据范围不大的场景中，temp数组如果比原数组大很多，就得不偿失。
- 计数排序只能给非负整数排序，如果数据是其他类型的，要转成非负整数，这些都是temp数组的局限性。

九、基数排序

1、数据的要求

可以分割出独立的“位”来比较，而且位之间有递进的关系；

每一位的数据范围不能太大，要可以用线性排序算法来排序；

比如电话号码，每位都是独立的，每位之间有递进关系，因为每位都是0-9的数字，范围也很小。

2、基数排序的过程

- 首先排倒数第一位，这个排序要使用稳定的排序算法，因为排高位的时候，如果相等，不稳定的排序算法会打乱低位顺序。
- 然后向前逐一进行排序。
- 如果排序的数据不等长，用0补齐，具体补高位还是低位，看要求。

基数排序的时间复杂度为O(k * n)的，k是每位的范围，当范围较小时，基数排序就是O(n)的了。