1. 冒泡排序

```
// 标准版冒泡排序
func BubbleSortI(nums []int) []int {
    l := len(nums)
    for i := 0; i < l; i++ { // 这个 i 表示，数组会被排序多少次， 0 ~ i 表示 冒泡要进行 i +
        for j := 0; j < l-1-i; j++ { // 这一层表示，在这一轮里的数据交换 j 表示 在某一趟排序
            if nums[j] > nums[j+1] {
                nums[j], nums[j+1] = nums[j+1], nums[j]
            }
        }
    }

    return nums
}

// 优化版冒泡排序一，加入一个哨兵，如果没有发生交换，说明排序已经完成，无需再排序里
// 在外层 for 循环处优化
func BubbleSortII(nums []int) []int {
    l := len(nums)
    var flag int

    for i := 0; i < l; i++ {
        flag = 0
        for j := 0; j < l-1-i; j++ {
            if nums[j] > nums[j+1] {
                nums[j], nums[j+1] = nums[j+1], nums[j]
                flag = 1
            }
        }

        if flag == 0 {
            return nums
        }
    }

    return nums
}

// 优化版冒泡排序二，记录最后一次发生交换的位置，表明之前的还是未无序的，之后的都是已经有序了的
// 在内层 for 循环处优化
func BubbleSortIII(nums []int) []int {
    l := len(nums)
    var flag int
    k, pos := l-1, 0 // 记录最后一次交换的位置

    for i := 0; i < l; i++ {
        flag = 0
        for j := 0; j < k; j++ {
            if nums[j] > nums[j+1] {
                nums[j], nums[j+1] = nums[j+1], nums[j]
                flag = 1
                pos = j
            }
        }
```

```
        k = pos

        if flag == 0 {
            return nums
        }
    }

    return nums
}
```

2. 选择排序

```
func SelectionSort(nums []int) []int {
    l := len(nums)
    var min int // 用来记录每轮最小的数出现的位置

    for i := 0; i < l-1; i++ { // 控制比较的轮次
        min = i                      // 从i开始，就会把之前已经排序的元素给排除掉，不会被重复
        for j := i + 1; j < l; j++ { // 控制每一轮里找到最小的值
            if nums[min] > nums[j] {
                min = j
            }
        }

        // 一轮结束，找到最小的数的位置了，然后进行插入
        nums[min], nums[i] = nums[i], nums[min]
    }

    return nums
}
```

3. 插入排序

```
func InsertionSort(nums []int) []int {
    var preIndex int
    var current int

    for i := range nums { // 这里是在从未筛选的序列里取数字
        preIndex = i - 1  // 这个控制位置，方便插入
        current = nums[i] // 这是当前的未排序元素的首位元素

        // 这里是在找到要插入的位置
        for preIndex >= 0 && current < nums[preIndex] {
            // preIndex >= 0 ? 因为 index 要从0开始
            // current < nums[preIndex] ?
            // current 的含义是指当前已经选用了一个未排序序列里的首位数
            // nums[preIndex] 是已经拍好序序列里的最后一个数
            nums[preIndex+1] = nums[preIndex]
            preIndex--
        }

        nums[preIndex+1] = current
```

```
    }

    return nums
}
```

4. 希尔排序

```
func ShellSort(nums []int) []int {
    l := len(nums)

    var gap int

    // 假定增量为3
    t := 3

    // 先计算 gap
    for gap < l/t {
        gap += gap*t + 1
    }

    // 开始增量插入
    for gap > 0 {
        for i := gap; i < l; i++ { // 比较当前
            temp := nums[i]
            j := i - gap // 从没有排好序的序列里，选取一个数
            // j 的变化依赖于i，而 i 是在递增，所以没有明显的看到j的增加
            for j >= 0 && temp < nums[j] { // 这里的 for 就是找到插入位置
                nums[j+gap] = nums[j]
                j -= gap
            }
            nums[j+gap] = temp

        }
        gap = gap / t
    }

    return nums
}
```

5. 归并排序

```
func MergeSort(nums []int) []int {
    // 递归终止条件
    if len(nums) < 2 {
        return nums
    }

    // 单层递归
    middle := len(nums) / 2
    left := nums[:middle]
    right := nums[middle:]

    return merge(MergeSort(left), MergeSort(right))
```

```
}

// 这里处理合并和排序
func merge(left, right []int) []int {
    var result []int // 开辟额外的空间处理

    for len(left) != 0 && len(right) != 0 {
        if left[0] < right[0] {
            result = append(result, left[0])
            left = left[1:] // 删除已经排序完的元素
        } else {
            result = append(result, right[0])
            right = right[1:] // 删除已经排完序的元素
        }
    }

    // 处理还没有并入的元素
    result = append(result, left...)

    result = append(result, right...)

    return result
}
```

6. 快速排序

```
func QuickSort(nums []int) []int {
    return quickSort(nums, 0 , len(nums) - 1)
}

// 快排的具体实现 - 递归
func quickSort(nums []int, left, right int) []int {
    // 递归退出条件
    if left < right {
        // 找到一趟快排后的左右两个区间的分界线
        partitionIndex := partition(nums, left, right)
        // -1 和 + 1 表示要跳过基准的元素
        quickSort(nums, left, partitionIndex - 1)
        quickSort(nums, partitionIndex + 1, right)
    }

    return nums
}

// 这里是在移动左右指针，找到按照基准划分得到的区域
func partition(nums []int, left, right int) int {
    pivot := left // 这里选择的基准数的下标
    index := pivot + 1 // 从已经选择的基准后面开始排序

    for i := index ; i <= right; i++ { // i++ 左指针的移动
        if nums[i] < nums[pivot] {
            // index 记录的是比 nums[pivot] 大的元素的位置
            // 当找到 nums[i] < nums[pivot] 的地方时，要交换位置
            // 这里的交换位置是把小的元素(nums[i]) 移动到index的位置
            // 然后 index 要 ++，即当前位置的元素已经改变，需要移动到下一个位置来交换
```

```
            swap(nums, i, index)
            index += 1
        }
    }

    // 这里是把选出来的基准元素放到中间的位置
    swap(nums, pivot, index - 1)
    return index - 1
}

func swap(nums []int, i, j int) {
    nums[i], nums[j] = nums[j], nums[i]
}
```

7. 堆排序

```
func HeapSort(nums []int) []int{
    return headSort(nums)
}

func headSort(nums []int) []int {
    length := len(nums)
    buildMaxHeap(nums, length)

    for i := length - 1; i >= 0 ; i-- {
        // 堆顶 即 nums[0], 放到末尾
        swap(nums, 0, i)
        // nums 前面有序的元素已经放到末尾了，所以这里的length 需要--
        // 也可以理解成删减了一个节点
        length -= 1
        heapify(nums, 0, length) // 调整大顶堆，
    }

    return nums
}

// 构建大顶堆
func buildMaxHeap(nums []int, length int) {
    for i := length / 2 ; i >= 0 ; i-- {
        heapify(nums, i , length)
    }
}

func heapify(nums []int, i, length int) {
    // 这里是利用了完全二叉树的性质
    left := 2*i + 1
    right := 2*i + 2
    largest := i // 因为构建的是大顶堆，所以中间要放最大的值

    // 寻找左子堆比当前中间结点大的值
    if left < length && nums[left] > nums[largest] {
        largest = left
    }

    if right < length && nums[right] > nums[largest] {
```

```
        largest = right
    }

    if largest != i {
        swap(nums, i , largest)
        heapify(nums, largest, length) // 这里可以理解成处理下一层
        // 这里隐含了递归的终止条件
    }
}

func swap(nums []int, i, j int) {
    nums[i], nums[j] = nums[j], nums[i]
}
```

8. 计数排序

```
func CountingSort(nums []int, maxValue int) []int {
    return countingSort(nums, maxValue)
}

// maxValue 是指要排序的数据中最大的哪一个
func countingSort(nums []int, maxValue int) []int {
    bucketLen := maxValue + 1
    bucket := make([]int, bucketLen)

    sortedIndex := 0
    length := len(nums)

    // 这个是计数，计算每个值出现多少次，统计过程
    for i := 0 ; i < length; i++ {
        bucket[nums[i]]++
    }

    // 这个是
    for j := 0 ; j < bucketLen; j++ {
        for bucket[j] > 0 { // j 这个值在nums里面出现了
            nums[sortedIndex] = j
            sortedIndex += 1 // 当前位置已经放置元素，需要填充下一个位置的元素了
            bucket[j] -= 1 // 累计的数减去1
        }
    }

    return nums
}
```

9. 桶排序

```
func BucketSort(nums []int, bucketSize int) []int {
    return bucketSort(nums, bucketSize)
}

func bucketSort(nums []int, bucketSize int) []int {
    length := len(nums)
```

```
    if length == 0 {
        return nums
    }

    minValue := nums[0]
    maxValue := nums[0]

    for i := 1; i < length; i++ {
        if nums[i] < minValue { // 找到要排序元素中的最小值
            minValue = nums[i]
        } else if nums[i] > maxValue { // 找到要排序元素中的最大值
            maxValue = nums[i]
        }
    }

    // 初始化桶
    bucketCount := caculateFloor(maxValue, minValue, bucketSize) + 1

    buckets := make([][]int, bucketCount)

    // 将数据分别放入到桶中
    for i := 0 ; i < length; i++ {
        index := caculateFloor(nums[i], minValue, bucketSize)
        buckets[index] = append(buckets[index], nums[i])
    }

    var sortedIndex int
    // 对桶进行排序
    for _, item := range buckets {
        if len(item) < 1 {
            continue
        }

        // 对每个桶进行排序，这里使用了插入排序
        item = InsertionSort(item)
        for _, v := range item {
            nums[sortedIndex] = v
            sortedIndex++
        }
    }

    return nums
}

func caculateFloor(maxValue, minValue, bucketSize int) int {
    count := float64((maxValue - minValue) / bucketSize)
    res := math.Floor(count)
    return int(res)
}
```

10. 基数排序

```
func RadixSort(nums []int) []int {
    return radixSort(nums)
```

```
}

func radixSort(nums []int) []int {
    length := len(nums)

    maxBit := getMaxBit(nums, length)

    base := 1 // 取余基数，用于取出每个元素的倒数第 i + 1 位的值，计算公式 v / base % 10
    buckets := make([][]int, 10) // 基数桶 10 个

    for i := 0 ; i < maxBit; i++ { // 依次遍历每个数的每一位
        for _, v := range nums { // 遍历待排序的数
            d := v / base % 10 // 获取每个数字当前位的值
            buckets[d] = append(buckets[d], v) // 存入对应的桶中
        }

        // 按照本次排序，将数据填充到当前的数组里，从左至右，从上到下的顺序
        sortedIndex := 0

        // 遍历当前已经完成一趟排序的桶
        for k, bucket := range buckets {
            if len(bucket) == 0 {
                continue
            }

            for _, v := range bucket {
                nums[sortedIndex] = v
                sortedIndex++
            }

            // 清空桶
            buckets[k] = []int{}
        }

        base *= 10 // 基数进一位
    }

    return nums
}

// 获取待排序元素中的最大值和其最大位数
func getMaxBit(nums []int, length int) (int) {
    maxValue := nums[0]

    // 先找到最大的数
    for i := 1 ; i < length; i++ {
        if maxValue < nums[i] {
            maxValue = nums[i]
        }
    }

    var bit int // 记录需要进行几趟排序，也就是说最大的数字有几位

    for maxValue > 0 {
        maxValue /= 10
        bit++
```

```
    }

    return bit
}
```