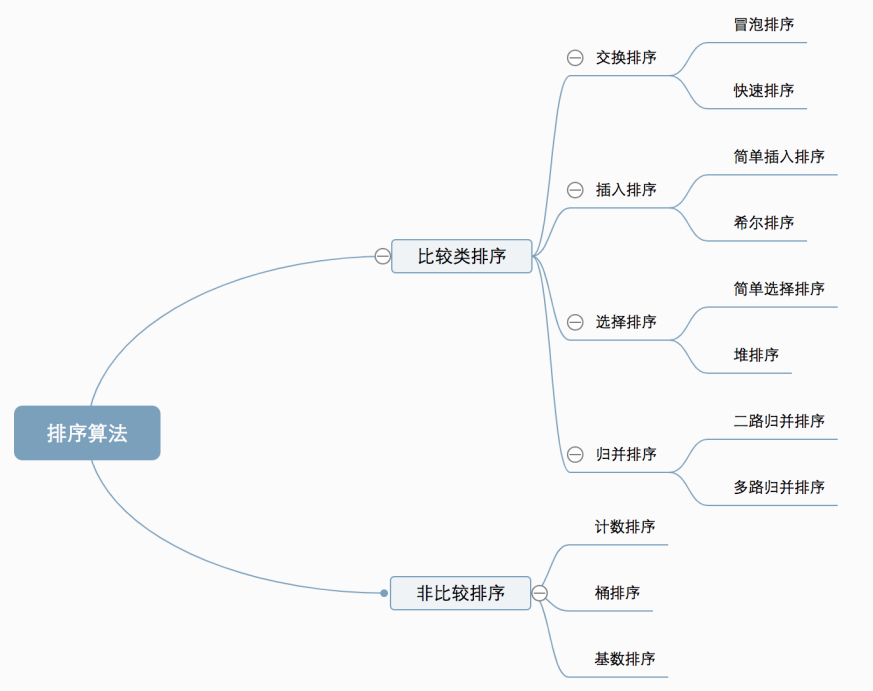
1. 概述

1.1 分类

- 比较类排序：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破O(nlogn)，因此也称为非线性时间比较类排序。
- 非比较类排序：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排序。

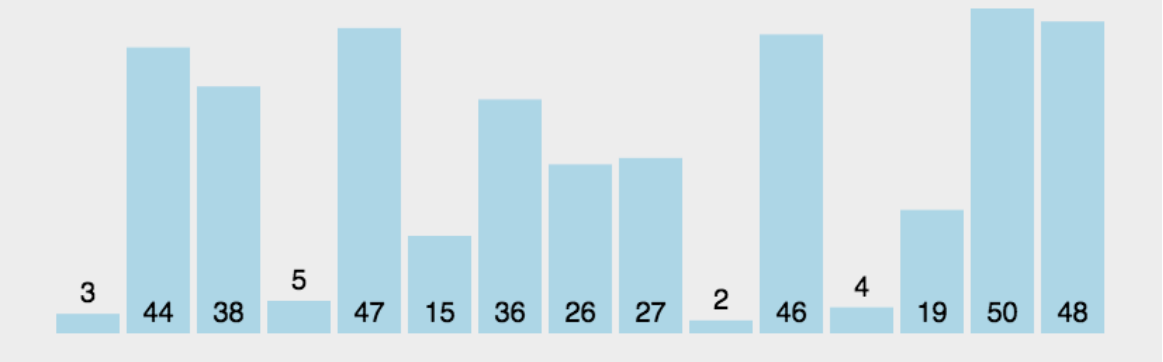


1.2 复杂度及稳定性

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

2. 排序算法

2.1 冒泡排序 (Bubble Sort)



- Python 版

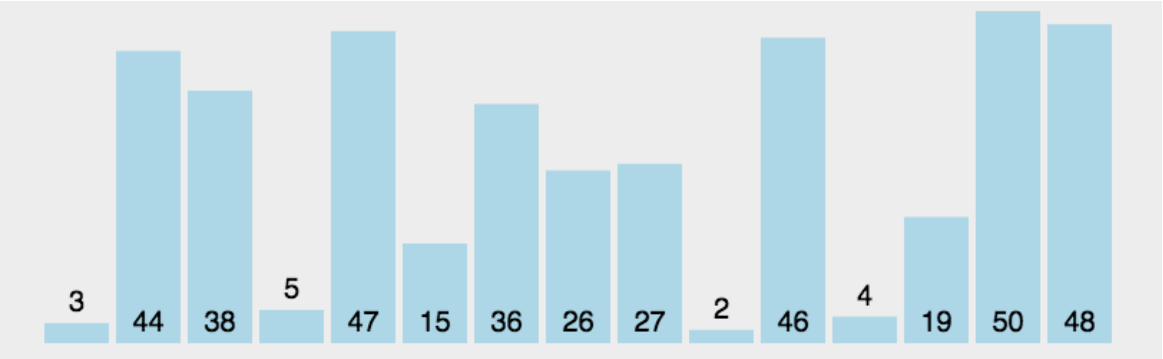
```
def bubbleSort(array):
    n = len(array)
    for i in range(n):
        flag = True     # 标记
        for j in range(1, n - i):
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
                flag = False
        # 某一趟遍历如果没有数据交换，则说明已经排好序了，因此不用再进行迭代了
        if flag:
            break
    return array
```

- Golang 版

```
func bubbleSort(array []int) []int {
    for i := 0; i < len(array); i++ {
        flag := true
        for j := 0; j < len(array)-i-1; j++ {
            if array[j] > array[j+1] {
                array[j], array[j+1] = array[j+1], array[j]
                flag = false
            }
        }
        if flag {
            break
        }
    }
    return array
}
```

- 最好情况：我们只需要进行一次冒泡操作，没有任何元素发生交换，此时就可以结束程序，所以最好情况时间复杂度是O(n).
- 最坏情况：要排序的数据完全倒序排列的，我们需要进行n次冒泡操作，每次冒泡时间复杂度为O(n),所以最坏情况时间复杂度为O(n^2)
- 平均复杂度： `O(n^2)`
- 原地排序：冒泡的过程只涉及相邻数据之间的交换操作而没有额外的内存消耗，故冒泡排序为原地排序算法。
- 稳定排序：在冒泡排序的过程中，只有每一次冒泡操作才会交换两个元素的顺序。所以我们为了冒泡排序的稳定性，在元素相等的情况下，我们不予交换，此时冒泡排序即为稳定的排序算法。

2.2 选择排序 (Selection Sort)



- Python 版

```python
def selectSort(ary):
    n = len(ary)
    for i in range(0,n):
        index = i                               #最小元素下标标记
        for j in range(i+1,n):
            if ary[j] < ary[index] :
                index = j                       #找到最小值的下标
        ary[index],ary[i] = ary[i],ary[index]   #交换两者
    return ary
```

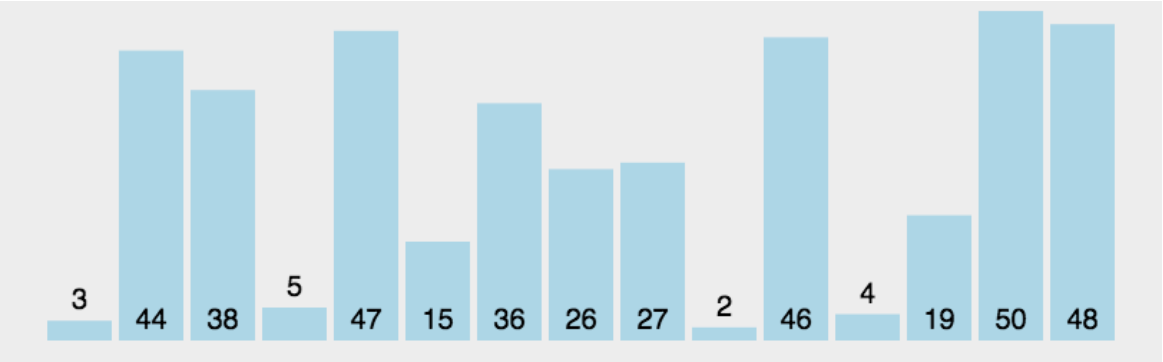
- Golang 版

```go
func selectSort(array []int) []int {
    for i := 0; i < len(array); i++ {
        min := i
        for j := i + 1; j < len(array); j++ {
            if array[j] < array[min] {
                min = j
            }
        }
        array[i], array[min] = array[min], array[i]
    }
    return array
}
```

- 最好情况： O(n^2)
- 最坏情况： O(n^2)
- 平均复杂度： O(n^2)
- 原地排序
- 非稳定排序 因为每次都要在未排序区间找到最小的值和前面的元素进行交换，这样如果遇到相同的元素，会使他们的顺序发生交换

2.3 快速排序 (Quick Sort)

快速排序的主要思想是通过划分将待排序的序列分成前后两部分，其中前一部分的数据都比后一部分的数据要小，然后再递归调用函数对两部分的序列分别进行快速排序，以此使整个序列达到有序。



- Python 版

```python
class Solution:
    def randomized_partition(self, nums, l, r):
        pivot = random.randint(l, r)
        nums[pivot], nums[r] = nums[r], nums[pivot]
        i = l - 1
        for j in range(l, r):
            if nums[j] < nums[r]:
                i += 1
                nums[j], nums[i] = nums[i], nums[j]
        i += 1
        nums[i], nums[r] = nums[r], nums[i]
        return i

    def randomized_quicksort(self, nums, l, r):
        if r - l <= 0:
            return
        mid = self.randomized_partition(nums, l, r)
        self.randomized_quicksort(nums, l, mid - 1)
        self.randomized_quicksort(nums, mid + 1, r)

    def sortArray(self, nums: List[int]) -> List[int]:
        self.randomized_quicksort(nums, 0, len(nums) - 1)
        return nums
```

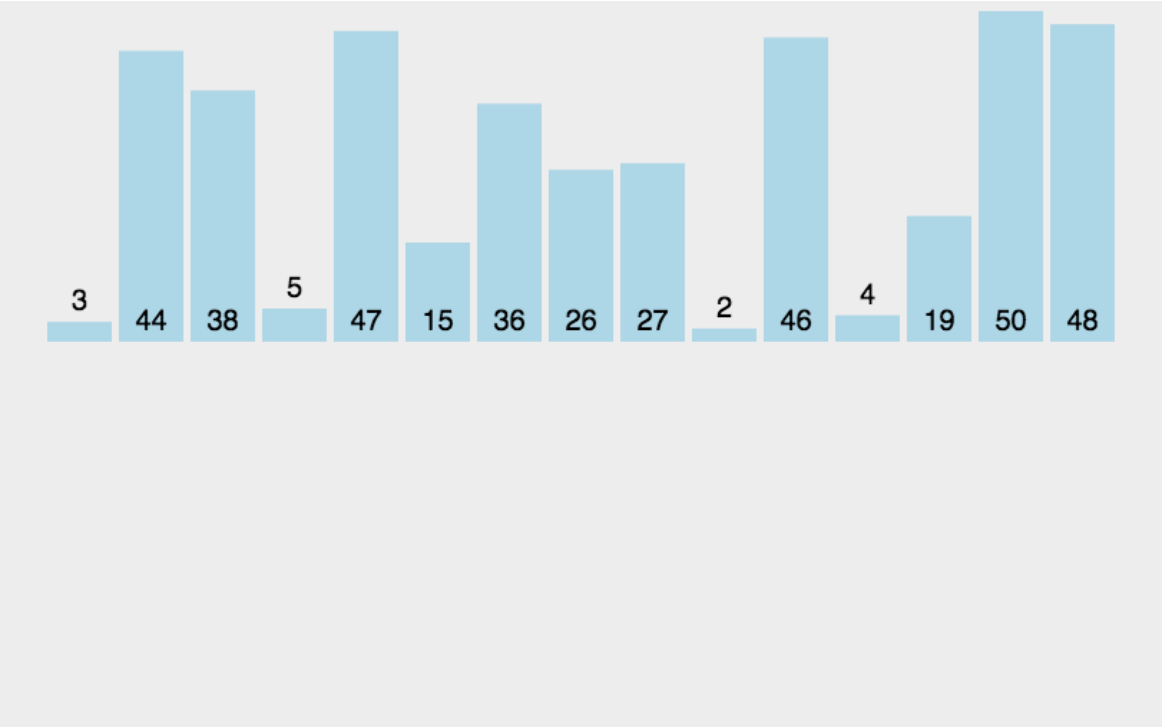
- Golang 版

```golang
func sortArray(nums []int) []int {
    // 快速排序，基于比较，不稳定算法，时间平均O(nlogn)，最坏O(n^2)，空间O(logn)
    // 分治思想，选主元，依次将剩余元素的小于主元放其左侧，大的放右侧
    // 然后取主元的前半部分和后半部分进行同样处理，直至各子序列剩余一个元素结束，排序完成
    // 注意：
    // 小规模数据(n<100)，由于快排用到递归，性能不如插排
    // 进行排序时，可定义阈值，小规模数据用插排，往后用快排
    // golang的sort包用到了快排
    // (小数，主元，大数)
    var quick func(nums []int, left, right int) []int
    quick = func(nums []int, left, right int) []int {
        // 递归终止条件
        if left > right {
            return nil
        }
        // 左右指针及主元
        i, j, pivot := left, right, nums[left]
        for i < j {
            // 寻找小于主元的右边元素
            for i < j && nums[j] >= pivot {
                j--
            }
            // 寻找大于主元的左边元素
            for i < j && nums[i] <= pivot {
                i++
            }
            // 交换i/j下标元素
            nums[i], nums[j] = nums[j], nums[i]
        }
        // 交换元素
        nums[i], nums[left] = nums[left], nums[i]
        quick(nums, left, i-1)
        quick(nums, i+1, right)
        return nums
    }
    return quick(nums, 0, len(nums)-1)
}
```

- 时间复杂度：基于随机选取主元的快速排序时间复杂度为期望 $O(logn)$，其中 n 为数组的长度。
- 空间复杂度：$O(h)$，其中 h 为快速排序递归调用的层数。我们需要额外的 $O(h)$ 的递归调用的栈空间，由于划分的结果不同导致了快速排序递归调用的层数也会不同，最坏情况下需 $O(n)$ 的空间，最优情况下每次都平衡，此时整个递归树高度为 $nlogn$，空间复杂度为 $O(logn)$

2.4 插入排序

插入排序是前面已排序数组找到插入的位置



- Python 版

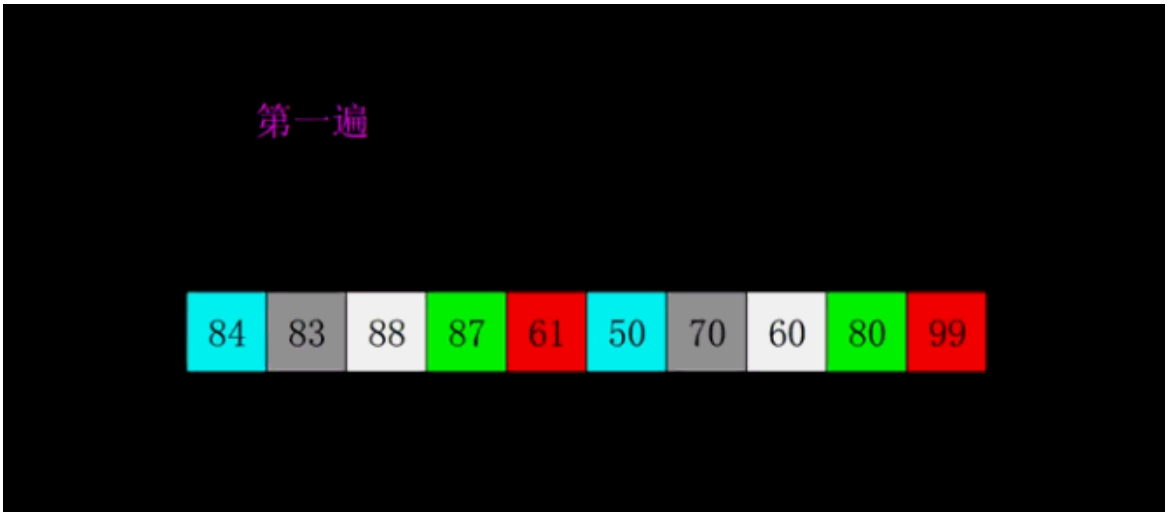
```python
def insertSort(nums):
    n = len(nums)
    for i in range(1, n):
        while i > 0 and nums[i - 1] > nums[i]:
            nums[i - 1], nums[i] = nums[i], nums[i - 1]
            i -= 1
    return nums
```

- Golang 版

```golang
func sortArray(nums []int) []int {
    // 插入排序，比较交换，稳定算法，时间O(n^2)，空间O(1)
    // 0->len方向，每轮从后往前比较，相当于找到合适位置，插入进去
    // 数据规模小的时候，或基本有序，效率高
    for i:=0; i<len(nums); i++ {
        // 从后往前比较，找到位置插入
        for j:=i; j>0; j-- {
            if nums[j-1] > nums[j] {
                nums[j-1], nums[j] = nums[j], nums[j-1]
            }
        }
    }
    return nums
}
```

2.5 希尔排序 (Shell Sort)

非稳定排序，内排序
希尔排序的时间复杂度和增量序列是相关的



- Python 版

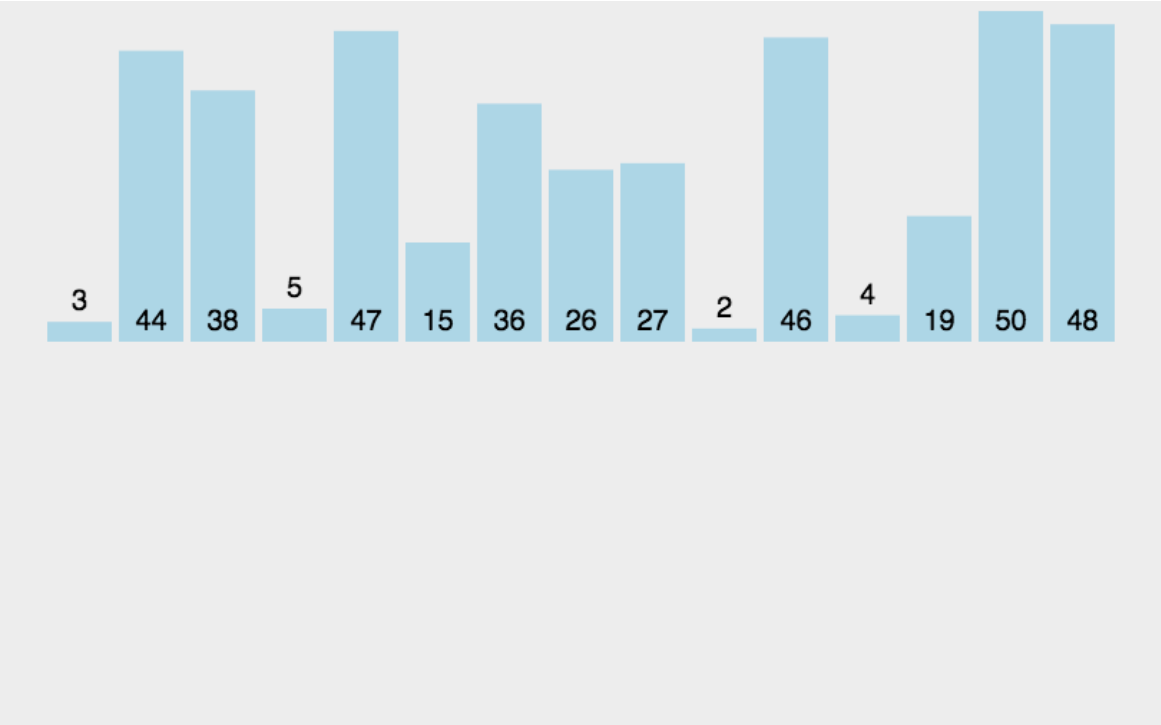
```python
def shellSort(nums):
    n = len(nums)
    gap = n // 2
    while gap:
        for i in range(gap, n):
            while i - gap >= 0 and nums[i - gap] > nums[i]:
                nums[i - gap], nums[i] = nums[i], nums[i - gap]
                i -= gap
        gap //= 2
    return nums
```

- Golang 版

```go
func sortArray(nums []int) []int {
    // 希尔排序，比较交换，不稳定算法，时间O(nlog2n)最坏O(n^2)，空间O(1)
    // 改进插入算法
    // 每一轮按照间隔插入排序，间隔依次减小，最后一次一定是1
    /*
    主要思想：
    设增量序列个数为k，则进行k轮排序。每一轮中，
    按照某个增量将数据分割成较小的若干组，
    每一组内部进行插入排序；各组排序完毕后，
    减小增量，进行下一轮的内部排序。
    */
    gap := len(nums)/2
    for gap > 0 {
        for i:=gap;i<len(nums);i++ {
            j := i
            for j-gap >= 0 && nums[j-gap] > nums[j] {
                nums[j-gap], nums[j] = nums[j], nums[j-gap]
                j -= gap
            }
        }
        gap /= 2
    }
    return nums
}
```

2.6 归并排序 (Merge Sort)

稳定排序，外排序（占用额外内存），时间复杂度 $O(nlogn)$



- Python 版

```python
def merge_sort(nums):
    if len(nums) <= 1:
        return nums
    mid = len(nums) // 2
    # 分
    left = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])
    # 合并
    return merge(left, right)

def merge(left, right):
    res = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            res.append(left[i])
            i += 1
        else:
            res.append(right[j])
            j += 1
    res += left[i:]
    res += right[j:]
    return res
```

- Golang 版

```go
// 递归实现归并算法
func sortArray(nums []int) []int {
    // 归并排序，基于比较，稳定算法，时间O(nlogn)，空间O(logn) | O(n)
    // 基于递归的归并-自上而下的合并，另有非递归法的归并(自下而上的合并)
    // 都需要开辟一个大小为n的数组中转
    // 将数组分为左右两部分，递归左右两块，最后合并，即归并
    // 如在一个合并中，将两块部分的元素，遍历取较小值填入结果集
    // 类似两个有序链表的合并，每次两两合并相邻的两个有序序列，直到整个序列有序
    merge := func(left, right []int) []int {
        res := make([]int, len(left)+len(right))
        var l,r,i int
        // 通过遍历完成比较填入res中
        for l < len(left) && r < len(right) {
            if left[l] <= right[r] {
                res[i] = left[l]
                l++
            } else {
                res[i] = right[r]
                r++
            }
            i++
        }
        // 如果left或者right还有剩余元素，添加到结果集的尾部
        copy(res[i:], left[l:])
        copy(res[i+len(left)-l:], right[r:])
        return res
    }
    var sort func(nums []int) []int
    sort = func(nums []int) []int {
        if len(nums) <= 1 {
            return nums
        }
        // 拆分递归与合并
        // 分割点
        mid := len(nums)/2
        left := sort(nums[:mid])
        right := sort(nums[mid:])
        return merge(left, right)
    }
    return sort(nums)
}

// 非递归实现归并算法
func sortArray(nums []int) []int {
    // 归并排序-非递归实现，利用变量，自下而上的方式合并
    // 时间O(nlogn)，空间O(n)
    if len(nums) <= 1 {return nums}
    merge := func(left, right []int) []int {
        res := make([]int, len(left)+len(right))
        var l,r,i int
```

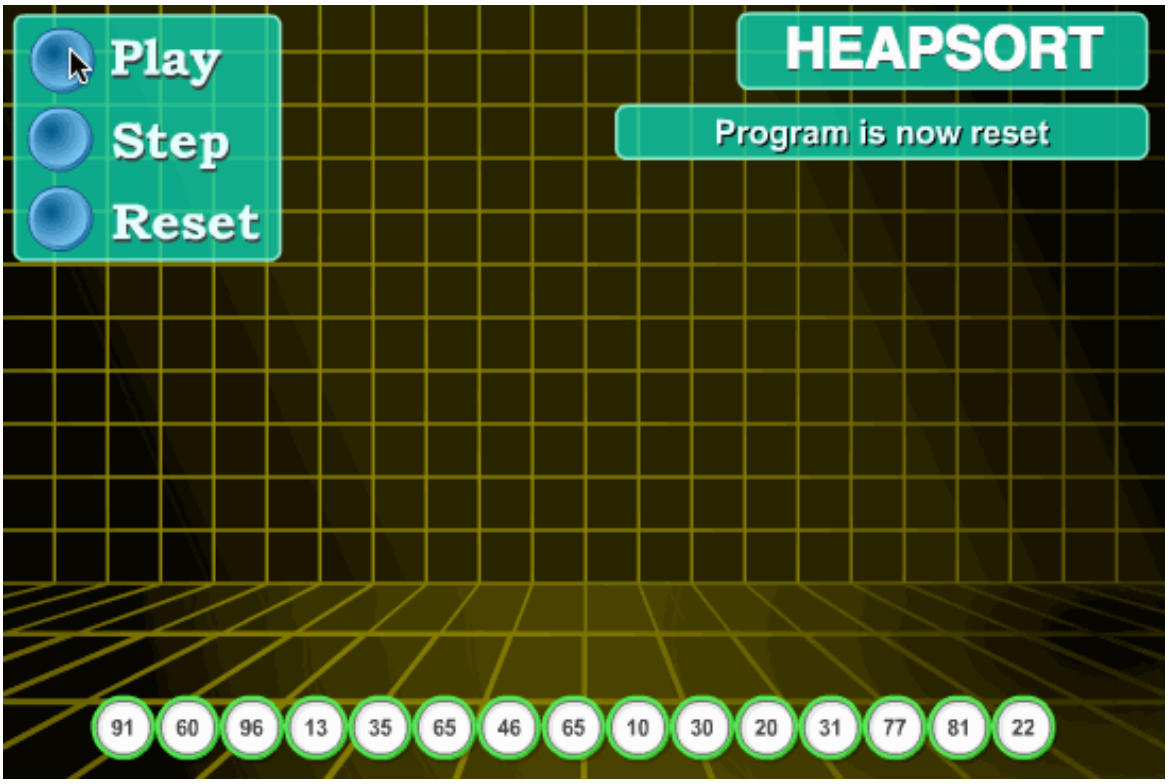
```
        // 通过遍历完成比较填入res中
        for l < len(left) && r < len(right) {
            if left[l] <= right[r] {
                res[i] = left[l]
                l++
            } else {
                res[i] = right[r]
                r++
            }
            i++
        }
        // 如果left或者right还有剩余元素，添加到结果集的尾部
        copy(res[i:], left[l:])
        copy(res[i+len(left)-l:], right[r:])
        return res
    }
    i := 1 //子序列大小初始1
    res := make([]int, 0)
    // i控制每次划分的序列长度
    for i < len(nums) {
        // j根据i值执行具体的合并
        j := 0
        // 按顺序两两合并，j用来定位起始点
        // 随着序列翻倍，每次两两合并的数组大小也翻倍
        for j < len(nums) {
            if j+2*i > len(nums) {
                res = merge(nums[j:j+i], nums[j+i:])
            } else {
                res = merge(nums[j:j+i], nums[j+i:j+2*i])
            }
            // 通过index控制每次将合并的数据填入nums中
            // 重填入的次数和合并及二叉树的高度相关
            index := j
            for _, v := range res {
                nums[index] = v
                index++
            }
            j = j + 2*i
        }
        i *= 2
    }
    return nums
}
```

2.7 堆排序

堆排序是利用堆这个数据结构设计的排序算法

算法描述:

1. 建堆，从底向上调整堆，使得父亲节点比孩子节点值大，构成大顶堆；
2. 交换堆顶和最后一个元素，重新调整堆。



- Python 版

```python
def heapSort(nums):
    # 调整堆
    # 迭代写法
    def adjust_heap(nums, startpos, endpos):
        newitem = nums[startpos]
        pos = startpos
        childpos = pos * 2 + 1
        while childpos < endpos:
            rightpos = childpos + 1
            if rightpos < endpos and nums[rightpos] >= nums[childpos]:
                childpos = rightpos
            if newitem < nums[childpos]:
                nums[pos] = nums[childpos]
                pos = childpos
                childpos = pos * 2 + 1
            else:
                break
        nums[pos] = newitem

    # 递归写法
    def adjust_heap(nums, startpos, endpos):
        pos = startpos
        childpos = pos * 2 + 1
        if childpos < endpos:
            rightpos = childpos + 1
            if rightpos < endpos and nums[rightpos] > nums[childpos]:
                childpos = rightpos
            if nums[childpos] > nums[pos]:
                nums[pos], nums[childpos] = nums[childpos], nums[pos]
                adjust_heap(nums, pos, endpos)

    n = len(nums)
    # 建堆
    for i in reversed(range(n // 2)):
        adjust_heap(nums, i, n)
    # 调整堆
    for i in range(n - 1, -1, -1):
        nums[0], nums[i] = nums[i], nums[0]
        adjust_heap(nums, 0, i)
    return nums
```

- Golang 版

```go
func sortArray(nums []int) []int {
    // 堆排序-大根堆，升序排序，基于比较交换的不稳定算法，时间O(nlogn)，空间O(1)-迭代建堆
    // 遍历元素时间O(n)，堆化时间O(logn)，开始建堆次数多些，后面次数少
    // 主要思路：
    // 1.建堆，从非叶子节点开始依次堆化，注意逆序，从下往上堆化
    // 建堆流程：父节点与子节点比较，子节点大则交换父子节点，父节点索引更新为子节点，循环操作
    // 2.尾部遍历操作，弹出元素，再次堆化
    // 弹出元素排序流程：从最后节点开始，交换头尾元素，由于弹出，end--，再次对剩余数组元素建堆，循环操作
    // 建堆函数，堆化
    var heapify func(nums []int, root, end int)
    heapify = func(nums []int, root, end int) {
        // 大顶堆堆化，堆顶值小一直下沉
        for {
            // 左孩子节点索引
            child := root*2 + 1
            // 越界跳出
            if child > end {
                return
            }
            // 比较左右孩子，取大值，否则child不用++
            if child < end && nums[child] <= nums[child+1] {
                child++
            }
            // 如果父节点已经大于左右孩子大值，已堆化
            if nums[root] > nums[child] {
                return
            }
            // 孩子节点大值上冒
            nums[root], nums[child] = nums[child], nums[root]
            // 更新父节点到子节点，继续往下比较，不断下沉
            root = child
        }
    }
    end := len(nums)-1
    // 从最后一个非叶子节点开始堆化
    for i:=end/2;i>=0;i-- {
        heapify(nums, i, end)
    }
    // 依次弹出元素，然后再堆化，相当于依次把最大值放入尾部
    for i:=end;i>=0;i-- {
        nums[0], nums[i] = nums[i], nums[0]
        end--
        heapify(nums, 0, end)
    }
    return nums
}
```