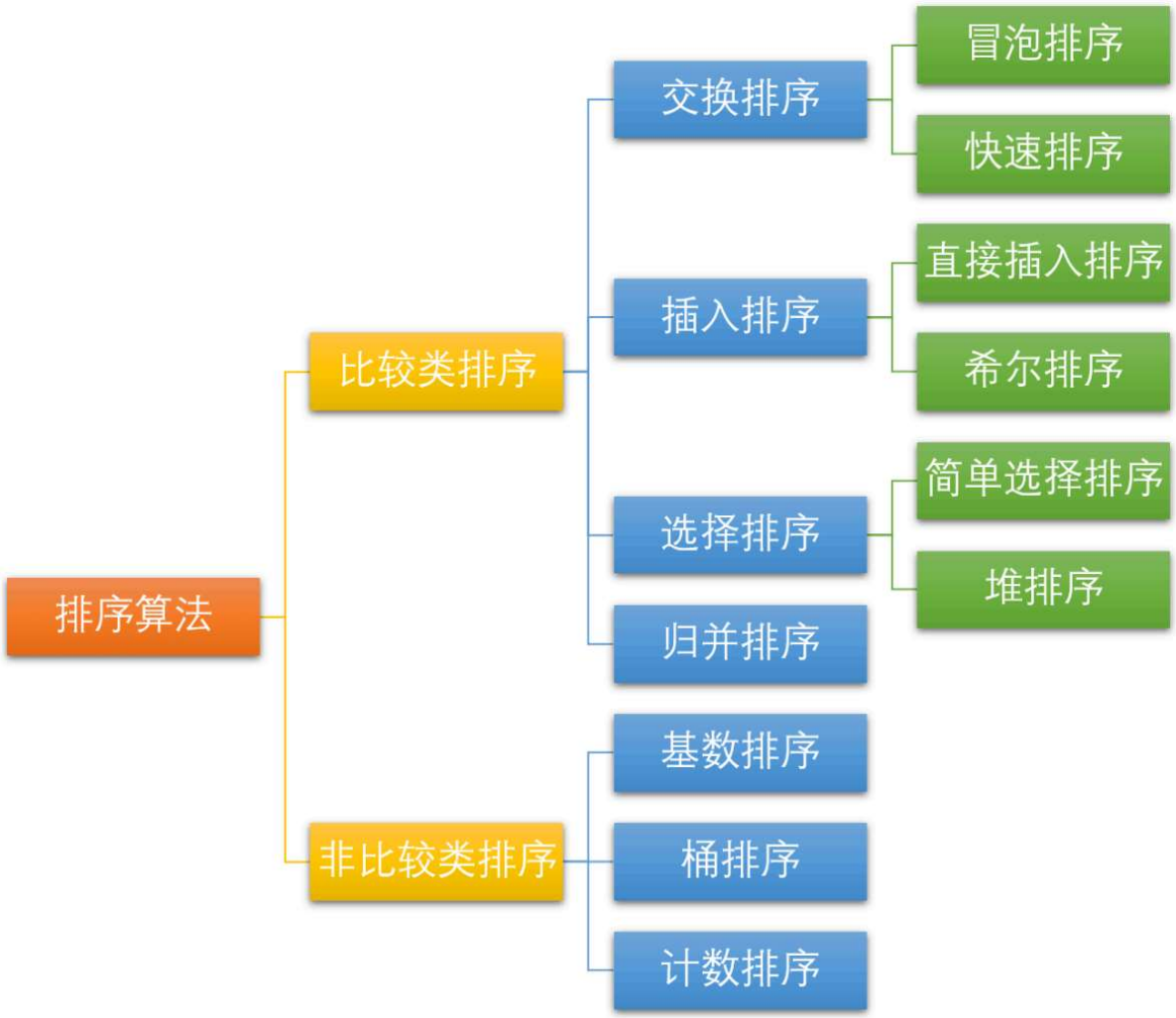
排序算法分类


排序算法
比较类排序
非比较类排序
交换排序
插入排序
选择排序
归并排序
基数排序
桶排序
计数排序
冒泡排序
快速排序
直接插入排序
希尔排序
简单选择排序
堆排序

排序算法比较

排序算法	时间复杂度（平均）	时间复杂度（最差）	时间复杂度（最好）	空间复杂度	排序方式	稳定性
冒泡排序	O(n^2)	O(n^2)	O(n)	O(1)	内部排序	稳定
选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	内部排序	不稳定
插入排序	O(n^2)	O(n^2)	O(n)	O(1)	内部排序	稳定
希尔排序	O(nlogn)	O(n^2)	O(nlogn)	O(1)	内部排序	不稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	外部排序	稳定
快速排序	O(nlogn)	O(n^2)	O(nlogn)	O(logn)	内部排序	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	内部排序	不稳定
计数排序	O(n+k)	O(n+k)	O(n+k)	O(k)	外部排序	稳定
桶排序	O(n+k)	O(n^2)	O(n+k)	O(n+k)	外部排序	稳定
基数排序	O(n×k)	O(n×k)	O(n×k)	O(n+k)	外部排序	稳定

术语解释：

- **n**：数据规模，表示待排序的数据量大小。
- **k**：“桶”的个数，在某些特定的排序算法中（如基数排序、桶排序等），表示分割成的独立的排序区间或类别的数量。
- **内部排序**：所有排序操作都在内存中完成，不需要额外的磁盘或其他存储设备的辅助。这适用于数据量小到足以完全加载到内存中的情况。
- **外部排序**：当数据量过大，不可能全部加载到内存中时使用。外部排序通常涉及到数据的分区处理，部分数据被暂时存储在外部磁盘等存储设备上。
- **稳定**：如果 A 原本在 B 前面，而 A=B，排序之后 A 仍然在 B 的前面。
- **不稳定**：如果 A 原本在 B 的前面，而 A=B，排序之后 A 可能会出现在 B 的后面。
- **时间复杂度**：定性描述一个算法执行所耗费的时间。
- **空间复杂度**：定性描述一个算法执行所需内存的大小

算法代码

冒泡排序

```
package main
```

```go
import "fmt"

func main() {
    nums := []int{1, 24, 35, 343, 463, 46, 34, 35, 12, 123, 245, 413, 5, 132}
    bubbleSort(nums)
    fmt.Println(nums)
}

//每轮把 最小的 放到 当前i位置
func bubbleSort(nums []int) {
    //len(nums)个数需要遍历len(nums)-1轮
    //1 2 3 ... len(nums)-1
    for i := 1; i <= len(nums)-1; i++ {
        //该轮有交换，则flag置为false
        flag := true
        //每轮遍历需要比较len(nums)-i次
        //len(nums)-1 len(nums)-2 len(nums)-3 ... 1
        for j := 0; j < len(nums)-i; j++ {
            if nums[j] > nums[j+1] {
                nums[j], nums[j+1] = nums[j+1], nums[j]
                flag = false
            }
        }
        //后面本来就排好序了
        //该轮没有交换，说明前面也排好序了
        //可以提前退出
        if flag {
            break
        }
    }
}
```

选择排序

```go
package main

import "fmt"

func main() {
    nums := []int{1, 24, 35, 343, 463, 46, 34, 35, 12, 123, 245, 413, 5, 132}
    selectSort(nums)
    fmt.Println(nums)
}

func selectSort(nums []int) []int {
    for i := 0; i < len(nums)-1; i++ {
        minIndex := i
        for j := i + 1; j < len(nums); j++ {
            if nums[j] < nums[minIndex] {
                minIndex = j
            }
        }
        nums[i], nums[minIndex] = nums[minIndex], nums[i]
```

```
    }
    return nums
}
```

归并排序

https://zhuanlan.zhihu.com/p/343986766外部排序归并算法

当输入序列仅有一个元素时，直接返回

如果输入内只有一个元素，则直接返回，否则将长度为n的输入序列分成两个长度为n/2的子序列

分别对这两个子序列进行归并排序，使子序列变为有序状态

设定两个指针，分别指向两个已经排序子序列的起始位置

比较两个指针所指向的元素，选择相对小的元素放入到合并空间（用于存放排序结果），并移动指针到下一位置

重复上面2个步骤直到某一指针达到序列尾

将另一序列剩下的所有元素直接复制到合并序列尾

快速排序

```go
// https://www.acwing.com/video/227/
package main

import "fmt"

func main() {
    nums := []int{1, 24, 35, 343, 463, 46, 34, 35, 12, 123, 245, 413, 5, 132}
    quickSort(nums, 0, len(nums)-1)
    fmt.Println(nums)
}

func quickSort(nums []int, left, right int) {
    if left >= right {
        return
    }
    pivot := nums[(left+right)/2]
    i, j := left-1, right+1
    for i < j {
        for {
            i++
            if nums[i] >= pivot {
                break
            }
        }
        for {
            j--
            if nums[j] <= pivot {
                break
            }
        }
        if i < j {
            nums[i], nums[j] = nums[j], nums[i]
```

```
        }
    }
    quickSort(nums, left, j)
    quickSort(nums, j+1, right)
}
```

堆排序

大根堆

```
// https://www.acwing.com/video/263/
package main

import "fmt"

// 要求排序为升序，一般采用大顶堆，每次将堆顶元素(最大值)与未排序的最后一个元素交换，再调整堆
// 要求排序为降序，一般采用小顶堆，每次将堆顶元素(最小值)与未排序的最后一个元素交换，再调整堆
func main() {
    nums := []int{1, 24, 35, 343, 463, 46, 34, 35, 12, 123, 245, 413, 5, 132}
    heapSort(nums)
    fmt.Println(nums)
}

// 大根堆排序
func heapSort(nums []int) {
    var down func(x, length int)
    // 小的元素下沉(相当于大的元素上浮),向下调整堆以保持堆的性质
    // length:当前堆的元素个数(每次确定一个最大值,需要构建堆的大小都会-1,因此length是会变的)
    down = func(x, length int) {
        y := x
        // 如果左孩子存在,找到"nums[x]和左孩子nums[2*x+1]中大的那个"
        if 2*x+1 < length && nums[2*x+1] > nums[y] {
            y = 2*x + 1
        }
        // 如果右孩子存在,找到比"nums[x]和左孩子nums[2*x+1]中大的那个"还要大的那个
        if 2*x+2 < length && nums[2*x+2] > nums[y] {
            y = 2*x + 2
        }
        // 如果找到了需要交换的孩子
        if x != y {
            nums[x], nums[y] = nums[y], nums[x]
            // nums[y]替换nums[x]后,继续进行下沉,直到下沉到叶子节点
            down(y, length)
        }
    }
    // floyd构建初始堆的时间复杂度为O(n)
    // 从倒数第二层最后一个可能有孩子的节点开始，向上遍历到根节点
    // 在down的时候会遍历它的左右孩子,因此能遍历到所有元素
    for i := len(nums)/2 - 1; i >= 0; i-- {
        down(i, len(nums))
    }
    // 每次将堆顶元素nums[0](最大值)与未排序数组的末尾元素nums[i]交换,使结尾是最大值
    for i := len(nums) - 1; i > 0; i-- {
        /*
```

```
            第一次循环:
            将最大值nums[0]与nums[len(nums)-1]交换后
            nums[len(nums)-1]的值确定为nums[0]~nums[len(nums)-1]的最大值
            nums[0]~nums[len(nums)-2]不再是大根堆
            让nums[0]下沉,使nums[0]~nums[len(nums)-2]还是大根堆
            nums[0]~nums[len(nums)-2]共len(nums)-1个数
            因此是down(0, len(nums)-1)
            第二次循环:
            将最大值nums[0]与nums[len(nums)-2]交换后
            nums[len(nums)-2]的值确定为nums[0]~nums[len(nums)-2]的最大值
            nums[0]~nums[len(nums)-3]不再是大根堆
            让nums[0]下沉,使nums[0]~nums[len(nums)-3]还是大根堆
            nums[0]~nums[len(nums)-3]共len(nums)-2个数
            因此是down(0, len(nums)-2)
            依此类推......
        */
        nums[0], nums[i] = nums[i], nums[0]
        down(0, i)
    }
}
```

小根堆

```
//大根堆
if 2*x+1 < length && nums[2*x+1] > nums[y] {
if 2*x+2 < length && nums[2*x+2] > nums[y] {
//小根堆
if 2*x+1 < length && nums[2*x+1] < nums[y] {
if 2*x+2 < length && nums[2*x+2] < nums[y] {
```

```
package main

import "fmt"

func main() {
    nums := []int{1, 24, 35, 343, 463, 46, 34, 35, 12, 123, 245, 413, 5, 132}
    heapSort(nums)
    fmt.Println(nums)
}

func heapSort(nums []int) {
    var down func(x, length int)
    down = func(x, length int) {
        y := x
        if 2*x+1 < length && nums[2*x+1] < nums[y] {
            y = 2*x + 1
        }
        if 2*x+2 < length && nums[2*x+2] < nums[y] {
            y = 2*x + 2
        }
        if x != y {
            nums[x], nums[y] = nums[y], nums[x]
            down(y, length)
        }
```

```
    }
    for i := len(nums)/2 - 1; i >= 0; i-- {
        down(i, len(nums))
    }
    for i := len(nums) - 1; i > 0; i-- {
        nums[0], nums[i] = nums[i], nums[0]
        down(0, i)
    }
}
```