排序算法

O(n^2)级排序算法

冒泡排序

通常来讲, 冒泡排序有三种写法

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

一边比较一边向前两两交换

```go
func bubbleSort(nums []int) {
    swapped := true
    for i := 0; i < len(nums); i++ {
        if !swapped {
            break
        }
        swapped = false
        for j := 1; j < len(nums)-i; j++ {
            if nums[j] < nums[j-1] {
                nums[j], nums[j-1] = nums[j-1], nums[j]
                swapped = true
            }
        }
    }
}
```

优化: 使用一个变量记录当前轮次的比较是否发生过交换，如果没有发生交换表示已经有序，不再继续排序；

```java
public static void bubbleSort(int[] arr) {
    boolean swapped = true;
    // 最后一个没有经过排序的元素的下标
    int indexOfLastUnsortedElement = arr.length - 1;
    // 上次发生交换的位置
    int swappedIndex = -1;
    while (swapped) {
        swapped = false;
        for (int i = 0; i < indexOfLastUnsortedElement; i++) {
            if (arr[i] > arr[i + 1]) {
                // 如果左边的数大于右边的数，则交换，保证右边的数字最大
                swap(arr, i, i + 1);
                // 表示发生了交换
                swapped = true;
                // 更新交换的位置
                swappedIndex = i;
            }
        }
        // 最后一个没有经过排序的元素的下标就是最后一次发生交换的位置
        indexOfLastUnsortedElement = swappedIndex;
    }
}
// 交换元素
private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

再优化: 除了使用变量记录当前轮次是否发生交换外，再使用一个变量记录上次发生交换的位置，下一轮排序时到达上次交换的位置就停止比较。

选择排序

选择排序的思想是：双重循环遍历数组，每经过一轮比较，找到最小元素的下标，将其交换至首位

```go
/*
 * 方法一：选择排序
 * 此题属于部分排序，可以使用选择排序或者堆排序来做
 * 选择 k 次数组中的最大元素，将其交换到数组前面，然后返回数组的第 k 个元素即可
 */
func findKthLargest(nums []int, k int) int {
    var maxIndex int
    for i := 0; i < k; i++ {
        maxIndex = i
        for j := i+1; j < len(nums); j++ {
            if nums[j] > nums[maxIndex] {
                maxIndex = j
            }
        }
        nums[i], nums[maxIndex] = nums[maxIndex], nums[i]
    }
    return nums[k-1]
}
```

插入排序

交换法：在新数字插入过程中，不断与前面的数字交换，直到找到自己合适的位置。

移动法：在新数字插入过程中，与前面的数字不断比较，前面的数字不断向后腾出位置，当新数字找到自己的位置后，插入一次即可。

O(nlogn)级排序算法

希尔排序

堆排序

堆:

符合以下两个条件之一的完全二叉树：

根节点的值 ≥ 子节点的值，这样的堆被称之为最大堆，或大顶堆；

根节点的值 ≤ 子节点的值，这样的堆被称之为最小堆，或小顶堆。

堆排序过程

用数列构建出一个大顶堆，取出堆顶的数字；

调整剩余的数字，构建出新的大顶堆，再次取出堆顶的数字；

循环往复，完成整个排序。

```go
func heapSort(nums []int) {
    n := len(nums)
    buildHeap(nums, n)
    for i := n - 1; i >= 0; i-- {
        nums[0], nums[i] = nums[i], nums[0] // 把根节点和最后一个节点交换
        heapify(nums, i, 0)                 // 砍断最后一个节点，重新调整堆
    }
    return
}

func heapify(nums []int, n, i int) {
    if i >= n {
        return
    }
    c1, c2, max := 2*i+1, 2*i+2, i
    if c1 < n && nums[c1] > nums[max] {
        max = c1
    }
    if c2 < n && nums[c2] > nums[max] {
        max = c2
    }
    if max != i {
        nums[i], nums[max] = nums[max], nums[i]
        heapify(nums, n, max)
    }
}

func buildHeap(nums []int, n int) {
    lastNode := n - 1
    parent := lastNode - 1>>1
    for i := parent; i >= 0; i-- {
        heapify(nums, n, i)
    }
}
```

完全二叉树性质

对于完全二叉树中的第 i 个数，它的左子节点下标：left = 2i + 1

对于完全二叉树中的第 i 个数，它的右子节点下标：right = left + 1

对于有 n 个元素的完全二叉树(n≥2)(n≥2)，它的最后一个非叶子结点的下标：n/2 - 1

快速排序

```go
func quickSort(arr []int, left, right int) {
    if left >= right {
        return
    }
    pivot := partition(arr, left, right)
    quickSort(arr, left, pivot-1)
    quickSort(arr, pivot+1, right)
}

func partition(arr []int, left, right int) int {
    p := arr[left]
    for left < right {
        for left < right && p <= arr[right] {
            right--
        }
        arr[left] = arr[right]
        for left < right && p >= arr[left] {
            left++
        }
        arr[right] = arr[left]
    }
    arr[left] = p
    return left
}
```

基本思想

从数组中取出一个数，称之为基数（pivot）

遍历数组，将比基数大的数字放到它的右边，比基数小的数字放到它的左边。遍历完成后，数组被分成了左右两个区域

将左右两个区域视为两个数组，重复前两个步骤，直到排序完成

复杂度分析

平均时间复杂度为 O(nlogn)

最坏时间复杂度为 O(n^2)　　当 pivot 取极值时

归并排序

O(n)级排序算法

计数排序

有特定的使用场景, 利用数字本身的属性完成排序

通过计数的结果，计算出每个元素在排序完成后的位置，然后将元素赋值到对应位置。

计数排序算法比基于比较的排序算法更快的根本原因

计数排序时申请了长度为 k 的计数数组，在遍历每一个数字时，这个数字落在计数数组中的可能性共有 k 种，但通过数字本身的大小属性，我们可以「一次」把它放到正确的位置上。相当于一次排除了 (k - 1)/k(k-1)/k 种可能性。

基数排序

桶排序

计数排序的优化版本，用有限数量的桶存放数据(存放规则由自定义函数来确定), 对每个桶进行排序。

流程

初始状态下，整个序列R[1,2,.....,n]处于无序状态，且大小在[a,a+k]范围内

设置桶的数量为bucketNum，则数据可以划分为[0,bucketNum]、[bucketNum,2*bucketNum-1]、.....、[n*(bucketNum-1)/bucketNum,n]，数组中数据分别分配到相应桶中

再对每个非空桶中的元素进行排序

所有的非空桶依次合并即得到排序好的数据

复杂度分析

时间复杂度：遍历序列O(n)，因此桶排序耗时主要取决于每个桶排序用时O(k)，总共耗时O(n+k)

稳定性：桶排序是稳定的，相同数的顺序没有改变。