练识课堂 -- 作业 -- 排序算法

要求

> 对于大公司面试来说，排序算法是基本的要求，还会根据排序的数据进行位图操作，折半查找操作，约瑟夫环操作，博弈论操作等等面试题。
>
> 先根据以给定的内容先学习，并了解排序的操作和复杂度分析后实现7种排序。
>
> 提交时间：2019年12月21日20:00:00（星期六）

排序的概念

概念

> 排序是计算机内经常进行的一种操作，其目的是将一组无序的数据元素调整为有序的数据元素的过程。

操作

> **比较**：任意两个数据元素通过比较操作确定先后次序。

```
//比较数组中两个数据
if (arr[i] > arr[j])
  //arr[i]大于arr[j]
else
  //arr[i]小于arr[j]
```

> **交换**：数据元素之间需要交换才能得到预期结果。

```
//数据交换函数
void swap(int *a, int *b)
{
      int tmp = *a;
      *a = *b;
      *b = tmp;
}
```

数据量分析

> **内部排序**：

> 若整个排序过程不需要访问外存，仅在内存中完成数据的调整，则称此类排序问题为内部排序。

> **外部排序：**
>
> 若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为外部排序。

稳定性分析

> **前提**：一组数据中出现多个相同的数据
>
> ```
> //一组数据中出现多个相同的数据
> arr:=[]int{9,1,5,6,4,10,5,8,7,3};
> ```
>
> 若在原始记录序列中， ai 和 aj 的关键字相同， ai 出现在 aj 之前，经过某种方法排序后，ai的位置仍在 aj之前，则称这种排序方法是稳定的；
>
> 反之，若经过该方法排序后， ai的位置在 aj 之后，即相同关键字记录的领先关系发生变化，则称这种排序方法是不稳定的。

冒泡排序

原理

> **冒泡排序（Bubble Sort）**排列的序列，较大（或较小）的数据会“浮”到序列的顶端（或底部）。
>
> 冒泡排序原则：
>
> 比较两个相邻的数组元素，使起满足条件交换元素位置，直到n-1轮循环操作结束。

实现

> 1. 从头部开始，比较相邻的两个元素arr[j]和arr[j+1]，如果前一个元素比后一个元素大，进行数据交换。
> 2. 下标向后移动，即使j=j+1，再次比较元素arr[j]和arr[j+1]，判断是否需要交换数据。
> 3. 针对序列中每一对两两相邻的数据重复以上步骤，直到下标指向最后一个位置。
> 4. 在每一轮循环中重复以上步骤(1)(2)(3)，直到len-1轮循环执行完毕。

代码

```
//冒泡排序
func BubbleSort(slice []int){

}
```

复杂度分析

> ### 时间复杂度：
>
> 平均时间复杂度：**O(n²)**
>
> 最好时间复杂度：**O(n)** 数组中数据有序，遍历一次，不需要交换。
>
> 最坏时间复杂度：**O(n²)**

> ### 空间复杂度：
>
> **O(1)**，只需要一个额外空间用于交换。

> ### 稳定性：
>
> **稳定排序**

选择排序

原理

> **选择排序（Selection Sort）**是从待排序的序列中选出最大值（或最小值），交换该元素与待排序序列头部元素，直到所有待排序的数据元素排序完毕为止。

实现

> 1. 第一趟从len个元素的数据序列中选出关键字最小（或最大）的元素并放到最前（或最后）位置。
> 2. 下一趟再从len-1个元素中选出最小（大）的元素并放到次前（后）位置。
> 3. 以此类推，经过len-1趟完成排序。

代码

```
func SelectSort(slice []int){

}
```

复杂度分析

> ### 时间复杂度：
>
> 平均时间复杂度：**O(n²)**
>
> 最好时间复杂度：**O(n²)**
>
> 最坏时间复杂度：**O(n²)**

> **选择排序**最大的特点就是交换移动数据次数比较少，尽管与冒泡排序同为O(n²)，但性能上略优于冒泡排序。

> **空间复杂度：**
>
> **O(1)**，只需要一个额外空间用于交换。

> **稳定性：**
>
> **不稳定排序**

插入排序

原理

> **直接插入排序（Straight Insertion Sort）** 基本操作是：将一个记录插入到已经排好序的有序数据中，从而得到一个新的、记录数增加1的有序表。

实现

> 把待排序序列视为两部分：
>
> 1. 一部分为有序序列，通常在排序开始之时将序列中的第一个数据视为一个有序序列；
> 2. 另一部分为待排序序列，有序序列之后的数据视为待排序序列。
> 3. 在排序开始之时，从序列头部到尾部逐个选取数据，与有序序列中的数据，按照从尾部到头部的顺序逐个比较，直到找到合适的位置，将数据插入其中。

代码

```
func InsertSort(slice []int){

}
```

复杂度分析

> **时间复杂度：**
>
> 平均时间复杂度：**O(n²)**
>
> 最好时间复杂度：**O(n)**
>
> 最坏时间复杂度：**O(n²)**
>
> 如果排序的数据是随机的，根据概率相同原则，平均比较和移动的次数应为n²/4次，得出**直接插入排序**的时间复杂度为**O(n²)**。在同样的时间复杂度中直接插入排序要优于选择排序和冒泡排序。

> **空间复杂度：**
>
> **O(1)**，只需要一个额外空间用于交换。

> **稳定性：**
>
> **稳定排序**

希尔排序

原理

> **希尔排序（Shell Sort）**的基本思想是：先取定一个小于序列元素个数的整数作为增量，把序列的全部元素分成增量个组，所有相互之间距离为增量整数倍的元素放在同一个组中，在各组内进行直接插入排序。

实现

> 1. 将一个数据序列按照增量进行分组。
> 2. 将各个分组的数据进行直接插入排序。
> 3. 更新增量，同时增量大于零在进行分组并排序。

代码

```
func ShellSort
```

复杂度分析

> **时间复杂度：**
>
> 平均时间复杂度：**O(n log n)**
>
> 最好时间复杂度：**O(n log² n)**
>
> 最坏时间复杂度：**O(n log² n)**

> **空间复杂度：**
>
> **O(1)**，只需要一个额外空间用于交换。

> **稳定性：**
>
> **不稳定排序**

堆排序

原理

> **堆排序（Heaps Sort）**是指利用堆这种数据结构所设计的一种排序算法。
>
> 堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。
>
> 堆排序可以说是一种利用堆的概念来排序的选择排序。分为两种方法：

> - **大顶堆（大根堆）**：每个节点的值都大于或等于其子节点的值，在堆排序算法中用于升序排列；
> - **小顶堆（小根堆）**：每个节点的值都小于或等于其子节点的值，在堆排序算法中用于降序排列；

实现

> 1. 创建一个堆，将数据放在堆中存储。
> 2. 按大顶堆构建堆，其中大顶堆的一个特性是数据将被从大到小取出，将取出的数据元素按照相反的顺序进行排列，数据元素就完成了排序。
> 3. 然后从左到右，从上到下进行调整，构造出大顶堆。
> 4. 入堆完成之后，将堆顶元素取出，将末尾元素置于堆顶，重新调整结构，使其满足堆定义。

代码

> 构建对数据结构

```
func HeapSort(slice []int){

}
```

复杂度分析

> **时间复杂度：**
>
> 平均时间复杂度：**O(n log n)**
>
> 最好时间复杂度：**O(n log n)**
>
> 最坏时间复杂度：**O(n log n)**

> **空间复杂度：**
>
> **O(1)**，只需要一个额外空间用于交换。

> **稳定性：**
>
> **不稳定排序**

递归排序

原理

> **归并排序(Merge Sort)**的基本思想是：将两个序列合并在一起，并且使之有序。
>
> 该算法是采用**分治法（Divide-and-Conquer）**的经典的应用。
>
> 归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并

实现

1. 把长度为n的输入序列分成两个长度为n/2的子序列；
2. 对这两个子序列分别采用归并排序；
3. 将两个排序好的子序列合并成一个最终的排序序列。
4. 在2-路归并排序算法中，由于需要进行递归调用，为了保证递归的顺利执行，按照一定的方法划分序列，直到子序列成为单个的元素，才开始对相邻的序列进行排序与归并。

代码

```
func MergeSort(slice []int){

}
```

复杂度分析

> **时间复杂度：**
>
> 平均时间复杂度：**O(n log n)**
>
> 最好时间复杂度：**O(n log n)**
>
> 最坏时间复杂度：**O(n log n)**

> **空间复杂度：**
>
> **O(n)**，需要数据元素大小的额外空间用于交换。

> **稳定性：**
>
> **稳定排序**

快速排序

原理

> **快速排序（Quick Sort）**是对冒泡排序的改进。
>
> 快速排序的基本思想是：通过一趟排序，将序列中的数据分割为两部分，其中一部分的所有数值都比另一部分的小；然后按照此种方法，对两部分数据分别进行快速排序，直到参与排序的两部分都有序为止。

实现

> 将序列划分为如上所述的两部分：
>
> 1. 需要在开始的时置一个参考值，通过与参考值的比较来划分数据；
> 2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面；
> 3. 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。

代码

```
//快速排序
void QuickSort(slice []int){
}
```

复杂度分析

> ### 时间复杂度：
>
> 平均时间复杂度：**O(n log n)**
>
> 最好时间复杂度：**O(n log n)**
>
> 最坏时间复杂度：**O(n²)**

> ### 空间复杂度：
>
> **O(log n)**

> ### 稳定性：
>
> **不稳定排序**