

排序算法

回顾

- 散列表/哈希表
 - 哈希函数
 - 哈希冲突
- 查找算法
 - 顺序查找Sequential search
 - 二分查找Binary search
 - 插值查找Interpolation search
 - 哈希查找Hash search

Python输入

读取键盘输入

Python提供了 `input()` 内置函数从标准输入读入一行文本，默认的标准输入是键盘。

`input` 可以接收一个Python表达式作为输入，并将运算结果返回。

实例

```
#!/usr/bin/python3

str = input("请输入: ");
print ("你输入的内容是: ", str)
```

这会产生如下的对应着输入的结果：

请输入：菜鸟教程

你输入的内容是： 菜鸟教程

排序

- 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”的记录序列调整为“有序”的记录序列。
- 分内部排序和外部排序，若整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序。反之，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。

冒泡排序

- 重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢"浮"到数列的顶端。

冒泡排序-具体步骤

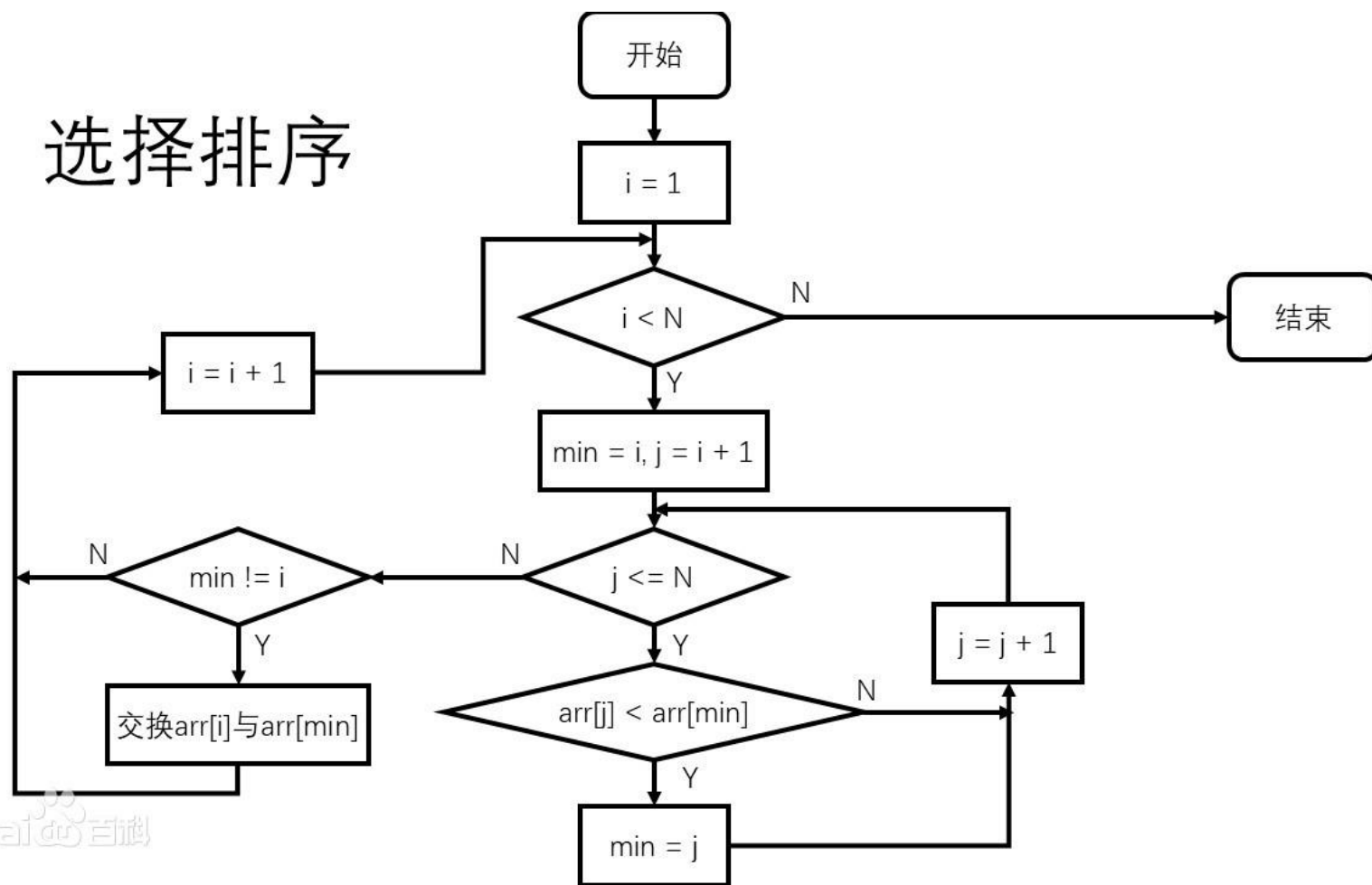
- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

选择排序

- 第一次从待排序的数据元素中选出最小的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。
- 具体步骤：
- 首先在未排序序列中找到最小元素，存放到排序序列的起始位置。
- 再从剩余未排序元素中继续寻找最小元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕。

选择排序-流程图

选择排序



插入排序

- 通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。
- 具体步骤：
 - 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
 - 从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

课堂作业1

代码实现以上三种排序算法


Bubble Sort

```
def bubbleSort(arr):  
    for i in range(1, len(arr)):  
        for j in range(0, len(arr)-i):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  
    return arr
```

Selection Sort

```
def selectionSort(arr):  
    for i in range(len(arr) - 1):  
        # 记录最小数的索引  
        minIndex = i  
        for j in range(i + 1, len(arr)):  
            if arr[j] < arr[minIndex]:  
                minIndex = j  
        # i 不是最小数时, 将 i 和最小数进行交换  
        if i != minIndex:  
            arr[i], arr[minIndex] = arr[minIndex], arr[i]  
    return arr
```

Insertion Sort

```
def insertionSort(arr):  
    for i in range(len(arr)):  
        preIndex = i-1  
        current = arr[i]  
        while preIndex >= 0 and arr[preIndex] > current:  
            arr[preIndex+1] = arr[preIndex]  
            preIndex -= 1  等价于 preIndex = preIndex - 1  
        arr[preIndex+1] = current  
    return arr
```

归并排序/合并排序

- 分治法 (Divide and Conquer)
 - 分--将问题分解为规模更小的子问题;
 - 治--将这些规模更小的子问题逐个击破;
 - 并/合--将已解决的子问题合并, 最终得出“母”问题的解;
- 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。
- 作为一种典型的分而治之思想的算法应用, 归并排序的实现由两种方法:
 - 自上而下的递归 (所有递归的方法都可以用迭代重写, 所以就有了第 2 种方法);
 - 自下而上的迭代;

归并排序/合并排序-具体步骤

- 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列；
- 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
- 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，并移动指针到下一位置；
- 重复步骤 3 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

- 快速排序是对冒泡排序的一种改进，由 C.A.R.Hoare（Charles Antony Richard Hoare，东尼·霍尔）在 1962 年提出。
- 通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另一部分的所有数据要小，再按这种方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，使整个数据变成有序序列。
- 也运用到了分治思想

快速排序-具体步骤

- 从数列中挑出一个元素，称为 "基准" (pivot) ；
- 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（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区 (partition) 操作；
- 递归地 (recursive) 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，直到每个分区都只有一个数为止。

课堂作业

代码实现以上2种排序算法

Merge Sort

```
def mergeSort(arr):  
    import math  
    if(len(arr)<2):  
        return arr  
    middle = math.floor(len(arr)/2)  
    left, right = arr[0:middle], arr[middle:]  
    return merge(mergeSort(left), mergeSort(right))  
  
def merge(left,right):  
    result = []  
    while left and right:  
        if left[0] <= right[0]:  
            result.append(left.pop(0))  
        else:  
            result.append(right.pop(0));  
    while left:  
        result.append(left.pop(0))  
    while right:  
        result.append(right.pop(0));  
    return result
```

Quick Sort

```
def quickSort(arr, left=None, right=None):  
    left = 0 if not isinstance(left,(int, float)) else left  
    right = len(arr)-1 if not isinstance(right,(int, float)) else right  
    if left < right:  
        partitionIndex = partition(arr, left, right)  
        quickSort(arr, left, partitionIndex-1)  
        quickSort(arr, partitionIndex+1, right)  
    return arr
```

```
def partition(arr, left, right):  
    pivot = left  
    index = pivot+1  
    i = index  
    while i <= right:  
        if arr[i] < arr[pivot]:  
            swap(arr, i, index)  
            index+=1  
        i+=1  
    swap(arr,pivot,index-1)  
    return index-1
```

```
def swap(arr, i, j):  
    arr[i], arr[j] = arr[j], arr[i]
```

时间复杂度总结

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定

名称	数据对象	稳定性	时间复杂度		额外空间复杂度	描述
			平均	最坏		
冒泡排序	数组	✓	$O(n^2)$		$O(1)$	（无序区，有序区）。 从无序区透过交换找出最大元素放到有序区前端。
选择排序	数组	✗	$O(n^2)$		$O(1)$	（有序区，无序区）。 在无序区里找一个最小的元素跟在有序区的后面。对数组：比较得多，换得少。
	链表	✓				
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	（有序区，无序区）。 把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。
堆排序	数组	✗	$O(n \log n)$		$O(1)$	（最大堆，有序区）。 从堆顶把根卸出来放在有序区之前，再恢复堆。
归并排序	数组	✓	$O(n \log^2 n)$		$O(1)$	把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。 可从上到下或从下到上进行。
			$O(n \log n)$		$O(n) + O(\log n)$ 如果不是从下到上	
	链表				$O(1)$	
快速排序	数组	✗	$O(n \log n)$	$O(n^2)$	$O(\log n)$	（小数，基准元素，大数）。 在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。

参考资料

- <https://baike.baidu.com/pic/%E9%80%89%E6%8B%A9%E6%8E%92%E5%BA%8F/9762418/0/a9d3fd1f4134970a3be0b86f98cad1c8a6865dda?fr=lemma&ct=single#aid=0&pic=a9d3fd1f4134970a3be0b86f98cad1c8a6865dda>
- <https://www.runoob.com/w3cnote/ten-sorting-algorithm.html>