排序篇

交换排序

交换排序之冒泡排序

- 算法思路

  i. 将序列当中的左右元素，依次比较，保证右边的元素始终大于左边的元素；（ 第一轮结束后，序列最后一个元素一定是当前序列的最大值）
  ii. 对序列当中剩下的n-1个元素再次执行步骤1。
  iii. 对于长度为n的序列，一共需要执行n-1轮比较

- 复杂度与稳定性

  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：T(n) = O(n) 当输入的数据已经是正序时
  - 最差情况：T(n) = O(n2) 当输入的数据是反序时
  - 平均情况：T(n) = O(n2)
  - 属于稳定性排序

- 代码

```
def bubble_sort(collection):
    """冒泡排序"""
    length = len(collection)
    for i in range(length - 1):
        swapped = False
        for j in range(length - 1 - i):
            if collection[j] > collection[j + 1]:
                swapped = True
                collection[j], collection[j + 1] = collection[j + 1], collection[j]
        # 当某趟遍历时未发生交换，则已排列完成，跳出循环
        if not swapped:
            break
    return collection
```

- 代码优化方向之一：设置一标志性变量pos,用于记录每趟排序中最后一次进行交换的位置。由于pos位置之后的记录均已交换到位,故在进行下一趟排序时只要扫描到pos位置即可。

```

```

- 代码优化方向之二：传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值,我们考虑利用在每趟排序中进行正向和反向两遍冒泡的方法一次可以得到两个最终值(最大者和最小者) , 从而使排序趟数几乎减少了一半

```

```

交换排序之快速排序

- 算法思路：快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）

  i. 从数列中挑出一个元素，称为 "基准"（pivot）；
  ii. 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；

iii. 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

- 复杂度与稳定性
  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：T(n) = O(nlogn)
  - 最差情况：T(n) = O(n2)
  - 平均情况：T(n) = O(nlogn)
  - 属于不稳定性排序
- 代码

```
```

选择排序

选择排序之简单选择排序

- 算法思路：
  i. 初始状态：无序区为R[1..n]，有序区为空；
  ii. 第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序 从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
  iii. n-1趟结束，数组有序化了
- 主要的两层循环
  - 第一层循环：依次遍历序列当中的每一个元素
  - 第二层循环：将遍历得到的当前元素依次与余下的元素进行比较，符合最小元素的条件，则交换
- 复杂度与稳定性
  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：T(n) = O(n2)
  - 最差情况：T(n) = O(n2)
  - 平均情况：T(n) = O(n2)
  - 属于稳定性排序
  - 优缺点及应用场景
- 代码

```
def select_sort(L):
    '''简单选择排序'''
    # 依次遍历序列中的每一个元素，n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果
    for x in range(0, len(L) - 1):
        # 将当前位置的元素定义此轮循环当中的最小值
        minimum = L[x]
```

```
            # 将该元素与剩下的元素依次比较寻找最小元素
            for i in range(x + 1, len(L)):
                # 当小时才交换，则是稳定排序
                if L[i] < minimum:
                    L[i], minimum = minimum, L[i]

            # 遍历比较后，将比较后得到的真正的最小值赋值给当前位置
            L[x] = minimum

        return L
```

选择排序之堆排序

- 算法思路：

  i. 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；
  ii. 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1] <=R[n]；
  iii. 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后 再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元 素个数为n-1，则整个排序过程完成。

- 复杂度与稳定性

  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：T(n) = O(nlogn)
  - 最差情况：T(n) = O(nlogn)
  - 平均情况：T(n) = O(nlogn)
  - 属于稳定性排序

- 优缺点及应用场景

- 代码

```
def heapify(unsorted, index, heap_size):
    '''构建大顶堆'''
    # 定义保存当前系列最大值的下标largest
    largest = index
    # 获取当前节点下标(index)的左右子节点下标
    left_index = 2 * index + 1
    right_index = 2 * index + 2

   # 在不超过堆容量时，进行左节点与父节点的比较，若比父节点大则更新最大值的下标
   if left_index &lt; heap_size and unsorted[left_index] &gt; unsorted[largest]:
       largest = left_index

   # 在不超过堆容量时，进行右节点与父节点的比较，若比父节点大则更新最大值的下标
   if right_index &lt; heap_size and unsorted[right_index] &gt; unsorted[largest]:
       largest = right_index

   # 若当前节点并不是最大值，则进行交换调整
   if largest != index:
       # 将当前节点的值与最大值进行互换
       unsorted[largest], unsorted[index] = unsorted[index], unsorted[largest]
       heapify(unsorted, largest, heap_size)



def heap_sort(unsorted):
    '''堆排序'''
    n = len(unsorted)
    # 从n // 2 - 1节点处依次构建堆
    for i in range(n // 2 - 1, -1, -1):
        heapify(unsorted, i, n)

   # 执行循环：
```

```
        # 1. 每次取出堆顶元素置于序列的最后(len - 1, len - 2, len - 3...)
        # 2. 调整堆，使其继续满足大顶堆的性质，注意实时修改堆中序列的长度
        for i in range(n - 1, 0, -1):
            # i为序列下标从后往前移动，所以最大值为于列表的最后
            # 将抽象数据结构堆的最大值即索引0位置的数值移动i位置
            unsorted[0], unsorted[i] = unsorted[i], unsorted[0]
            heapify(unsorted, 0, i)
        return unsorted</code></pre></li>
```

插入排序

插入排序之简单插入排序

- 算法思路：

  i. 从第一个元素开始，该元素可以认为已经被排序；
  ii. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
  iii. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
  iv. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
  v. 将新元素插入到该位置后；
  vi. 重复步骤2~5。

- 主要的两层循环：

  - 第一层循环：遍历待比较的所有数组元素（从第二个元素开始）
  - 第二层循环：将上层循环选择的元素（selected）与已经排好序的所有元素（ordered）相比较，从选择元素的前面一个开始直到数组的起始位置。如果selected < ordered，那么将二者交换位置，继续遍历；反之，留在原地，选择下一个元素

- 复杂度与稳定性

  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：输入数组按升序排列。T(n) = O(n)
  - 最坏情况：输入数组按降序排列。T(n) = O(n2)
  - 平均情况：T(n) = O(n2)
  - 属于稳定性排序

- 优缺点及应用场景

- 代码

```
def insert_sort(nums):
    '''直接插入排序'''
    # 遍历数组中的所有元素，其中0号索引元素默认已排序，因此从1开始
    # 当nums元素数量为空或者1时，不会进入for循环
    for i in range(1, len(nums)):
        # 将该元素与已排序好的前序数组依次比较，如果该元素小，则交换
        # range(x,0,-1):从x倒序循环到0，依次比较，
        # 每次比较如果小于会交换位置，正好按递减的顺序
        for j in range(i, 0, -1):
            # 判断：如果符合条件则交换,并由此处可见直接插入排序为稳定性排序
            if nums[j] < nums[j - 1]:
                nums[j], nums[j - 1] = nums[j - 1], nums[j]
            else:
                break
    return nums
```

插入排序之折半插入排序

- 在直接插入排序中，查找插入位置时使用二分查找的方式

插入排序之希尔排序

- 学习基础：希尔排序是基于简单插入排序的
- 算法思路：
- 复杂度与稳定性
  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：T(n) = O(nlog2 n)
  - 最坏情况：T(n) = O(nlog2 n)
  - 平均情况：T(n) =O(nlog n)
  - 属于不稳定性排序
- 优缺点及应用场景
- 代码

```
def shell_sort(nums):
    # 初始化gap值，此处利用序列长度的一半为其赋值
    gap = len(nums) // 2
    # 第一层循环：依次改变gap值对列表进行分组,当gap等于1时进行最后一遍循环
    while gap >= 1:
        # 下面：利用直接插入排序的思想对分组数据进行排序
        # range(gap, len(L)):由于数组下标从0开始，所以从gap开始依次比较
        # gap分组比较，同等的数值可能分部不同组而导致后一组比较后交换，而前一组比较后不交换
        # 故而破坏稳定性，成为不稳定性排序
        for i in range(gap, len(nums)):
            # range(i, 0, -gap):从i开始与选定元素开始倒序比较
            # 每个比较元素之间间隔gap
            for j in range(i, 0, -gap):
                # 如果该组当中两个元素满足交换条件，则进行交换
                if nums[j] < nums[j-gap]:
                    nums[j],nums[j-gap] = nums[j-gap],nums[j]
                else:
                    break
        gap = gap // 2
    return nums
```

####