3 搜索和排序

3.1 搜索

- 搜索时指从元素集合中找到某个特定元素的算法过程。

3.1.1 顺序搜索

- 无序列表的顺序搜索

```
def sequentialSearch(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos += 1

    return found
```

- 有序列表的顺序搜索

```
def orderedSequentialSearch(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
```

```
            else:
                pos += 1

    return found
```

3.1.2 二分搜索

- 二分搜索不是从第一个元素开始搜索列表，而是从中间的元素着手。
- 如果这个元素就是目标元素，那就立即停止搜索；如果不是，则可以利用列表有序的特性，排除一半的元素。
- 如果目标元素比中间的元素大，就可以直接排除列表的左半部分和中间元的元素。
- 接下来，针对右半部分重复二分过程。

```
def binarySearch(alist, item):
    first = 0
    last = len(alist) - 1
    found = False

    while first <= last and not found:
        midpoint = (first + last) // 2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint - 1
            else:
                first = midpoint + 1

    return found
```

- 二分搜索的递归版本

```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist) // 2
        if alist[midpoint] == item:
            return True
        else:
            if item < alist[midpoint]:
                return binarySearch(alist[:midpoint], item)
            else:
                return binarySearch(alist[midpoint+1:], item)
```

3.2 排序

- 排序是指将集合中的元素按某种顺序排列的过程。

3.2.1 冒泡排序

- **冒泡排序**多次遍历列表。
- 它比较相邻的元素，将不合顺序的交换。
- 每一轮遍历都将下一个最大值放到正确的位置上。

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1, 0, -1):
        for i in range(passnum):
            if alist[i] > alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
```

- 冒泡排序通常被认为是效率最低的排序算法，因为在确定最终的位置前必须交换元素。
- 由于冒泡排序要遍历列表中未排序的部分，因此它具有其他排序算法没有的用途。
- 特别是，如果在一轮遍历中没有发生元素交换，就可以确定列表已经有序。
- **短冒泡**

```
def bubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
        exchanges = False
        for i in range(passnum):
            if alist[i] > alist[i+1]:
                exchanges = True
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
        passnum -= 1
```

- 冒泡排序的时间复杂度是$O(n^2)$。

3.2.2 选择排序

- **选择排序**在冒泡排序的基础上做了改进，每次遍历列表时只做一次交换。
- 选择排序在每次遍历时寻找最大值，并在遍历完之后将它放到正确位置上。

```python
def selectionSort(alist):
    for fillslot in range(len(alist)-1, 0, -1):
        positionOfMax = 0
        for location in range(1, fillslot+1):
            if alist[location] > alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
```

- 可以看出，选择排序算法和冒泡排序算法的比较次数相同，所以时间复杂度也是$O(n^2)$。
- 但是，由于减少了交换次数，因此选择排序算法通常更快。

3.2.3 插入排序

- **插入排序**的时间复杂度也是$O(n^2)$，但原理稍有不同。
- 插入排序在列表较低的一端维护一个有序的子列表，并逐个将每个新元素"插入"这个子列表。

```python
def insertionSort(alist):
    for index in range(1, len(alist)):

        currentvalue = alist[index]
        position = index

        while position > 0 and alist[position-1] > currentvalue:
            alist[position] = alist[position-1]
            position -= 1

        alist[position] = currentvalue
```

- 插入排序的时间复杂度是$O(n^2)$。
- 交换操作的处理时间大约是移动操作的3倍，因为后者只需要进行一次赋值。
- 在基准测试中，插入排序算法的性能很不错。

3.2.4 希尔排序

- **希尔排序**对插入排序做了改进，将列表分成数个子列表，并对每一个子列表应用插入排序。
- 如何切分列表是希尔排序的关键——并不是连续切分，而是使用增量`i`（有时称作步长）选取所有间隔为`i`的元素组成子列表。

```python
def shellSort(alist):
    sublistcount = len(alist) // 2
    while sublistcount > 0:

        for startposition in range(sublistcount):
            gapInsertionSort(alist, startposition, sublistcount)

```

```
        print(f'after increments of size {sublistcount}, the list is
{alist}')

        sublistcount = sublistcount // 2

def gapInsertionSort(alist, start, gap):
    for i in range(start+gap, len(alist), gap):

        currentvalue = alist[i]
        position = i

        while position >= gap and alist[position-gap] > currentvalue:
            alist[position] = alist[position-gap]
            position -= gap

        alist[position] = currentvalue
```

```
>>> alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
>>> shellSort(alist)
after increments of size 4, the list is [20, 26, 44, 17, 54, 31, 93, 55, 77]
after increments of size 2, the list is [20, 17, 44, 26, 54, 31, 77, 55, 93]
after increments of size 1, the list is [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

- 希尔排序的时间复杂度大概介于$O(n)$和$O(n^2)$之间。
- 通过采用2^k-1（1, 3, 7, 15, 31, ...）增量，希尔排序的时间复杂度可以达到$O(n^{\frac{3}{2}})$。

3.2.5 归并排序

- **归并排序**是递归算法，每次将一个列表一分为二。
- 如果列表为空或只有一个元素，那么从定义上来说它就是有序的（基本情况）。
- 如果列表不止一个元素，就将列表一分为二，并对两部分都递归调用归并排序。
- 当两部分都有序后，就进行**归并**这一基本操作。
- 归并是指将两个较小的有序列表归并为一个有序列表的过程。

```
def mergeSort(alist):
    print(f'Splitting {alist}')
    if len(alist) > 1:
        mid = len(alist) // 2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] <= righthalf[j]:
```

```
                alist[k] = lefthalf[i]
                i = i+1
            else:
                alist[k] = righthalf[j]
                j = j+1
            k=k+1

        while i < len(lefthalf):
            alist[k] = lefthalf[i]
            i = i+1
            k = k+1

        while j < len(righthalf):
            alist[k] = righthalf[j]
            j = j+1
            k = k+1
    print(f'Merging {alist}')
```

```
>>> alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
>>> mergeSort(alist)
Splitting [54, 26, 93, 17, 77, 31, 44, 55, 20]
Splitting [54, 26, 93, 17]
Splitting [54, 26]
Splitting [54]
Merging [54]
Splitting [26]
Merging [26]
Merging [26, 54]
Splitting [93, 17]
Splitting [93]
Merging [93]
Splitting [17]
Merging [17]
Merging [17, 93]
Merging [17, 26, 54, 93]
Splitting [77, 31, 44, 55, 20]
Splitting [77, 31]
Splitting [77]
Merging [77]
Splitting [31]
Merging [31]
Merging [31, 77]
Splitting [44, 55, 20]
Splitting [44]
Merging [44]
Splitting [55, 20]
Splitting [55]
Merging [55]
Splitting [20]
Merging [20]
Merging [20, 55]
```

```
Merging [20, 44, 55]
Merging [20, 31, 44, 55, 77]
Merging [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

- 归并排序算法的时间复杂度是$O(nlogn)$。

3.2.6 快速排序

- **快速排序**算法首先选出一个**基准值**。
- 尽管有很多种选法，但为简单起见，选取列表中的第一个元素。
- 基准值的作用是帮助切分列表。
- 在最终的有序列表中，基准值的位置通常被称作**分割点**，算法再分割点切分列表，以进行对快速排序的子调用。

1. 首先加大`leftmark`，直到遇到一个大于基准值的元素。
2. 然后减小`rightmark`，直到遇到一个小于基准值的元素。
3. 互换这两个元素的位置，然后重复上述过程。
4. 当`rightmark`小于`leftmark`时，过程终止。
5. 此时，`rightmark`的位置就是分割点。
6. 将基准值与当前位于分割点的元素互换，即可使基准点位于正确位置。
7. 分割点左边的所有元素都小于基准值，右边的所有元素都大于基准值。
8. 因此，可以在分割点处将列表一分为二，并针对左右两部分递归调用快速排序函数。

```
def quickSort(alist):
    quickSortHelper(alist, 0, len(alist)-1)

def quickSortHelper(alist, first, last):
    if first < last:

        splitpoint = partition(alist, first, last)

        quickSortHelper(alist, first, splitpoint-1)
        quickSortHelper(alist, splitpoint+1, last)


def partition(alist, first, last):
    pivotvalue = alist[first]

    leftmark = first+1
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark += 1

        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark -= 1
```

```
27
28          if rightmark < leftmark:
29              done = True
30          else:
31              temp = alist[leftmark]
32              alist[leftmark] = alist[rightmark]
33              alist[rightmark] = temp
34
35      temp = alist[first]
36      alist[first] = alist[rightmark]
37      alist[rightmark] = temp
38
39      return rightmark
```

- 快速排序的时间复杂度是$O(nlogn)$。
- 另外，快速排序算法不需要像归并排序算法那样使用额外的存储空间。
- 不幸的是，最坏情况下，分割点不在列表的中部，而是偏向某一端，这会导致切分不均匀。