排序 递归

吴益强

Python 版

日期：2022 年 4 月 8 日

1 排序

1.1 排序的分类

在内存中进行的排序，叫内排序，简称排序复杂度不可能优于 O(nlog(n))
对外存（硬盘）上的数据进些排序，叫外排序。数据量较大。

1.2 如何评价排序算法

```
时间复杂度
    平均复杂度
    最坏情况复杂度
    最好情况复杂度

空间复杂度
    需要多少额外辅助空间

是否稳定:
    同样大小的元素, 排序前和排序后是否先后次序不变
```

表 1: 排序总结

排序方法	平均情况	最好情况	最坏情况	空间	稳定性
冒泡	O(n^2)	O(n)	O(n^2)	O(1)	稳定
简单选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定
直接插入排序	O(n^2)	O(n)	O(n^2)	O(1)	稳定
希尔排序	O(nlogn)—O(n^2)	O($n^{1.3}$)	O(n^2)	O(1)	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
快速排序	O(nlogn)	O(nlogn)	O(n^2)	O(logn)-O(n)	不稳定

1.3 Python 的排序函数

Timsort, 蒂姆排序

一种混合了归并排序和插入排序的算法

稳定

最坏时间复杂度 O(nlog(n))

最好时间复杂度接近 O(n)

额外空间：最坏 O(n)，但通常较少

是目前为止最快的排序算法

1.4 冒泡排序

基本思想：

1、将序列分成有序的部分和无序的部分。有序的部分在右边，无序的部分在左边。开始有序部分没有元素

2、每次从左到右，依次比较无序部分相邻的两个元素。如果右边的小于左边的，则交换它们。做完一次后，无部分最大元素即被换到无序部分最右边，有序部分元素个数 +1 。

3、2 做 n-1 次，排序即完成。

代码示例：

```
def bubbleSort(a):
    n = len(a)
    for i in range(1,n):
        for j in range(n-i):
            if a[j+1] < a[j]:
                a[j+1],a[j] = a[j],a[j+1]
```

- 无论最好、最坏、平均，语句 1）必定执行 (n 1)+...+3+2+1 次，复杂度 O(n^2)
- 稳定
- 额外空间 O(1)

1.5 选择排序

基本思想：

1、将序列分成有序的部分和无序的部分。有序的部分在左边，无序的部分在右边。开始有序部分没有元素

2、每次找到无序部分的最小元素（设下标为 i），和无序部分的最左边元素（设下标为 j）交换。有序部分元素个数 +1 。

3、2 做 n-1 次，排序即完成

代码示例：

```
def selectionSort(a):
    n =len(a)
    for i in range(n-1):
        minPos = i 最小元素位置
        for j in range(i+1,n):
            if a[j] < a[minPos]:
```

```
                minPos = j
        if minPos != i:
            a[minPos], a[i] =a[i], a[minPos]
```

- 无论最好、最坏、平均，必定执行 (n-1)+...+3+2+1 次，复杂度 O(n^2)
- 稳定性：不稳定，因 a[i] 被交换时，可能越过了其后面一些和它相等的元素
- 额外空间：O(1)
- 平均效率低于插入排序，没啥实际用处

1.6 插入排序

基本思想:

1、将序列分成有序的部分和无序的部分。有序的部分在左边，无序的部分在右边。开始有序部分只有 1 个元素

2、每次找到无序部分的最左元素（设下标为 i），将其插入到有序部分的合适位置（设下标为 k, 则原下标为 k 到 i-1 的元素都右移一位），有序部分元素个数 +1

3、直到全部有序

代码示例:

```
def insertionSort(a):
    for i in range(1,len(a)):
        e,j= a[i],i
        while j > 0 and e < a[j 1]:
            a[j] = a[j-1]
            j -= 1
        a[j] = e
```

- 规模很小的排序可优先选用, 比如元素个数 10 以内
- 特别适合元素基本有序的情况复杂度接近 O(n)
- 许多算法会在上述两种情况下采用插入排序。例如改进的快速排序算法、归并排序算法，在待排序区间很小的时候就不再递归快排或归并，而是用插入排序

1.7 快速排序

基本思想:

1、从数列中挑出一个元素，称为" 基准"（pivot）；

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（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；

3、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；

代码示例:

```
def quick_sort(data):
    """快速排序"""
```

```
    if len(data) >= 2:  # 递归入口及出口
        mid = data[len(data)//2]  # 选取基准值，也可以选取第一个或最后一个元素
        left, right = [], []  # 定义基准值左右两侧的列表
        data.remove(mid)  # 从原始数组中移除基准值
        for num in data:
            if num >= mid:
                right.append(num)
            else:
                left.append(num)
        return quick_sort(left) + [mid] + quick_sort(right)
    else:
        return data
```

1.8 归并排序

基本思想:

数组排序任务可以如下完成:

1、把前一半排序

2、把后一半排序

3、把两半归并到一个新的有序数组，然后再拷贝回原数组，排序完成。代码示例:

```
def MergeSort(lists):
    if len(lists) <= 1:
        return lists
    num = int( len(lists) / 2 )
    left = MergeSort(lists[:num])
    right = MergeSort(lists[num:])
    return Merge(left, right)
def Merge(left,right):
    r, l=0, 0
    result=[]
    while l<len(left) and r<len(right):
        if left[l] <= right[r]:
            result.append(left[l])
            l += 1
        else:
            result.append(right[r])
            r += 1
    result += list(left[l:])
    result += list(right[r:])
    return result
print MergeSort([1, 2, 3, 4, 5, 6, 7, 90, 21, 23, 45])
```

1.9 堆排序

基本思想:

1、创建一个堆 H[0⋯⋯n-1]；

2、把堆首（最大值）和堆尾互换；

3、把堆的尺寸缩小 1，并调用 $shiftdown(0)$，目的是把新的数组顶端数据调整到相应位置；

4、重复步骤 2，直到堆的尺寸为 1。

代码示例：

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]  # 交换

        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # 一个个交换元素
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # 交换
        heapify(arr, i, 0)

arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("排序后")
for i in range(n):
    print ("%d" %arr[i]),
```

1.10 希尔排序

基本思想：

1、选择一个增量序列 t1，t2，⋯⋯，tk，其中 ti > tj, tk = 1；

2、按增量序列个数 k，对序列进行 k 趟排序；

3、每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

代码示例：

```
def shell(nums):
    n = len(nums)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            while i >= gap and nums[i - gap] > nums[i]:
                nums[i - gap], nums[i] = nums[i], nums[i - gap]
                i -= gap
        gap //= 2
    return nums
```

2 递归

2.1 递归的作用

1) 替代多重循环进行枚举

2) 解决本来就是用递归形式定义的问题

3) 将问题分解为规模更小的子问题进行求解

2.2 全排列

从 n 个不同元素中任取 m（mn）个元素，按照一定的顺序排列起来，叫做从 n 个不同元素中取出 m 个元素的一个排列。当 m=n 时所有的排列情况叫全排列。

输入：nums = [1,2,3]

输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

```
力扣递归解法
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        if len(nums)==1:
            return [nums]
        if len(nums)==2:
            return [nums,nums[::-1]]
        ans = []
        # a + [permute(nums-a)]
        for a in nums:
            tmp = copy.deepcopy(nums)
            tmp.remove(a)
            for i in self.permute(tmp):
                ans.append([a]+i)
```

```
        return ans

dfs解法
s = list(input())
s.sort()
N = len (s)
result = [0 for i in range(N )] #存最新找到的一个排列
used = [False for i in range(N )]# used[i] 表示字母 s[i] 是否用过
def dfs (n): #摆放第 n 个位置及其右边的字母
    if n == N:
        print("".join(result))
    for i in range(N):
    if not used[i ]: # i 这个字母没用过
        result[n] = s[i]
        used[i] = True
        dfs(n+1)
        used[i] = False
dfs(0)
```

2.3 爬楼梯

树老师爬楼梯，他可以每次走 1 级或者 2 级，输入楼梯的级数，求不同的走法数

例如：楼梯一共有 3 级，他可以每次都走一级，或者第一次走一级，第二次走两级，也可以第一次走两级，第二次走一级，一共 3 种方法。

输入：输入包含若干行，每行包含一个正整数 N，代表楼梯级数 1<=N<=30 输出不同的走法数，每一行输入对应一行

输出: 不同的走法数，每一行输入对应一行输出

```
样例输入
5
8
10
样例输出
8
34
89
```

n 级台阶的走法 = 先走一级后，n-1 级台阶的走法 + 先走两级后, n-2 级台阶的走法

f(n) = f(n-1)+f(n+2) 边界: n < 0 0 ,n = 0 1

```
def stairs( n ):
    if n < 0:
        return 0
    if n == 0:
        return 1
    return stairs( n-1 ) + stairs( n-2 )
```

```
try:
    while True:
    N = int(input())
    print( stairs( N ))
except EOFError:
    pass
```

2.4 汉诺塔问题递归解法

```
def Hanoi(n, src,mid,dest):
    if( n == 1) :
        print(src + "-->" + dest)
        return
    Hanoi(n-1,src,dest,mid)
    print(src + "-->" + dest)
    Hanoi(n-1,mid,src,dest)
n = int(input())
Hanoi(n, 'A', 'B', 'C')
```

2.5 2 的幂次方表示

任何一个正整数都可以用 2 的幂次方表示。例如：

137=$2^7+2^3+2^0$

同时约定方次用括号来表示，即 a^b 可表示为 a(b)。由此可知，137 可表示为：

2(7)+2(3)+2(0)

进一步：7=2^2+2+2^0（2^1 用 2 表示）

3=$2+2^0$

所以最后 137 可表示为：

2(2(2)+2+2(0))+2(2+2(0))+2(0)

又如：

1315=$2^{10}+2^8+2^5+2+1$

所以 1315 最后可表示为：

2(2(2+2(0))+2)+2(2(2+2(0)))+2(2(2)+2(0))+2+2(0)

输入一个正整数 n（n<=20000）。输出一行，符合约定的 n 的 0，2 表示（在表示中不能有空格）。

```
n=int(input())
def M(n):
    if n<=0:
        raise TypeError
    if n==1:
        return "2(0)"
    if n==2:
```

```
        return "2"
    if n==3:
        return "2+2(0)"
    else:
        s=len(bin(n))-3
        if n==2**s:
            return "2(%s)"%M(s)
        return "2(%s)+"%M(s) + M(n-2**s)
print(M(n))
```

2.6 二叉树的最大深度 (Leetcode 104)

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例：给定二叉树 [3,9,20,null,null,15,7]，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

代码示例:

```
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root:
            return 0
        return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

2.7 两两交换链表中的节点 (Leetcode 24)

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

输入：head = [1,2,3,4]

输出：[2,1,4,3]

```
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if not head or not head.next:
            return head
        newHead = head.next
        head.next = self.swapPairs(newHead.next)
        newHead.next = head
```

```
        return newHead
```