Course 2: 时间复杂度

【我们一般只关心最坏情况下的时间复杂度，即所谓的大O表示法】

一、前言部分

DSA（数据结构与算法）是两个分开但又相互联系的主题。数据结构是储存和组织数据的方法，主要的思想是最小化时间和空间复杂度，数据结构包括堆、栈、列表、字符串等；算法是一个过程或者是一系列指令的集合用以执行并解决特定的（一组）问题，算法包括搜获、排序、递归、贪心、动态规划等。

学习DSA，本质是学习时间和空间复杂度、基本的数据结构、基本算法，并练习去解决一些题目。

1. 复杂度

复杂度是用以判断程序有效性的度量方法。复杂度有两个维度：时间维度——执行代码需要花费多少时间；空间维度——执行代码需要耗费多少内存。在程序运行过程中，往往还需要申请除存储数据外额外的空间，我们称为Auxiliary Space。

在程序执行时，所花费的时间有很多因素，包括代码的长度、设备的速度和数据在线上平台传输的时间。为了使得我们对于代码复杂度的评价独立于机器及实际的操作（忽略以来系统的常量）、只与程序操作有关，我们经常使用三种渐进表示法（asymptotic notation）：

大O表示法（Big-O Notation）：描述了算法在最坏情况下的复杂度，给予了程序运行的上界 **Omega Notation**：描述了算法在最好情况下的复杂度 **Theta Notation**：描述了算法的平均复杂度

时间复杂度也称作算法分析。其以输入数据规模n为自变量，以其增长量级为表示。如在大O表示法中，包括O（1），O（logn）,O(nlogn),O(n^c),O(c^n),O(n!)

2.排序算法

排序算法是最常用的算法，其指的是在具体情况下重组、排序同类数据

二、大O表示法

1.一些定义

算法分析：我们只关注影响执行时间最重要的变量。

随机存取机器RAM（random-access machine）是一个用于分析算法的计算机模型。在这个模型里，程序不会并发执行，而是一个接着一个基本单元顺序执行。在RAM模型，包含的是实际计算机中常用的基本指令，包括算术、控制等指令，这些基本指令的执行时间都是常量时间。在RAM中，数据类型是整数型或浮点型（有字长【衡量计算机性能的指标之一，较大字长用于处理更大量级的数据、进行复杂的计算】限制，一般8的倍数）。RAM模型可以预测程序在一个机器上运行的实际性能，时间复杂度就是在RAM模型里统计出来的。

输入问题的规模取决于具体问题本身。如排序时，规模指的是排序的总的数量，两个数相乘时，输入问题的规模指的是二进制表示下总的位数。

算法的运行时间是基本操作或者步骤被执行的数量。基于RAM，我们可以假设每一行执行的时间是一个确定的常量（可能不同的行有不同的执行时间）。

2.基本数据类型的时间复杂度

注：list查找时间O（n）,set查找为常量时间

list：v = a[i];a[i] = v;lst.append(v)均为O（1），lst = lst + [v]为O（n+k）（k为增加的列表长度）；列表copy为O（n）

Operation	Big-O Efficiency
index []	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n+k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

字典：取值get、赋值set和contain in判断的时间复杂度均为O（1）

三、排序算法

1. 插入排序分析（insertion sort）

基本原理：在一个循环里，从2-n对各元素插入到左侧已经排好序的模块，直至全部排好序。在pick没有排好序的元素时，逐一比较，直至确定元素的位置。

eg:升序算法

```
def insertion_sort(arr):
    for i in range(1,len(arr)): #执行n次（包括检验退出循环一次），时间乘常量c1
        j = i #执行n-1次，时间乘常量c2
        while arr[j-1] > arr[j] and j>0: #执行t1+...+t_(n-1)次，时间乘常量c3
            arr[j-1],arr[j] = arr[j],arr[j-1] #执行t1+...+t_(n-1)-(n-1)次，时间乘常量c4
            j = j-1 #执行t1+...+t_(n-1)-(n-1)次，时间乘常量c5
#执行时间：最简单情况下为c1*n+c2*(n-1)+c3*(n-1);
#最坏情况下为c1*n+c2*(n-1)+c3*(0.5n^2+0.5n-1)+(c4+c5)*(0.5n^2-0.5^n)
```

由此得出，插入算法最坏情况下及平均时间复杂度为O（n^2）（我们要寻找的是基于rate of growth中最快的，同时忽略小项和常系数）。最好情况theta notation为theta(n)。平均情况本质上是基于概率分析求期望，但这往往比较困难。

注：经典的插入排序算法不每次做交换，当还需要向左移动时，把该位置的项向右移动，这时每次循环只需做一次赋值即可

插入排序不需要额外空间

2.冒泡排序（Bubble Sort）

基本原理：每次扫描全部序列。具体实现方式为：从左侧第一、第二各元素开始作比较，将大的放在右边(对升序算法而言)，然后扫描第二、第三个元素，直至推到最右边排序好的部分。每次保证剩余序列中最大的排好序，放置在右侧大数中排好序（最左侧）的位置。

eg：升序算法

```
# Optimized Python program for implementation of Bubble Sort
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place
        for j in range(0, n - i - 1): #range(0,0)不会报错
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if (swapped == False):
            break
# Driver code to test above
if __name__ == "__main__":
    arr = [64, 34, 25, 12, 22, 11, 90]
```

```
    bubbleSort(arr)
    print(' '.join(map(str, arr)))
```

时间复杂度：O（n^2），相对较差

冒泡排序不要求任何的额外空间(较差时间效率的补偿)。是**稳定排序**算法（相同value的key保持他们原先的顺序）

在排好序的情况下，冒泡排序的时间复杂度为O（n）

3.选择排序（Selection Sort）

基本原理：选择排序重复从未被排序的部分选择最小（最大）的元素和未被排序的第一个元素交换顺序，从而由左往右增加已被排序的部分

eg:升序算法

```
A = [64, 25, 12, 22, 11]
# Traverse through all array elements
for i in range(len(A)):
    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i + 1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j
        # Swap the found minimum element with
    A[i], A[min_idx] = A[min_idx], A[i]
# Driver code to test above
print(' '.join(map(str, A)))
# Output: 11 12 22 25 64
```

选择排序相较冒泡排序有改进，因为比较后每次只做一次交换。

但是从量级上来看，其时间复杂度没有发生变化，仍为O(n^2)。

选择排序不是稳定排序，是原地排序。

4.快速排序（Quick Sort）

基本原理：其选择一个pivot（枢纽）元素，比枢纽元素小的和比枢纽大的元素分别放置在枢纽元素两边。然后分别对两边再进行快速排序（递归）。

快排本质是一种分治算法。（注：二分法查找非分治，因为其进行了剪枝）

快速排序不需要额外空间 eg：

```python
# 使用双指针实现partition:
def quicksort(arr,left,right):
    if left<right:
        partition_pos = partition(arr,left,right)
        quicksort(arr,left,partition_pos-1)
        quicksort(arr,partition_pos+1,right)
def partition(arr,left,right):
    i = left
    j = right-1
    pivot = arr[right]
    while i<=j:
        while i<= right and arr[i]<pivot:
            i+=1
        while j>=left and arr[j]>=pivot:
            j-=1
        if i<j:
            arr[i],arr[j] = arr[j],arr[i]
    if arr[i]>pivot:
        arr[i],arr[right] = arr[right],arr[i]
    return i



#使用单指针实现快排
def partition(array,low,high):
    pivot = array[high]
    i = low - 1
    for j in range(low,high):
        if array[j]<=pivot:
            i = i+1
            (array[i],array[j]) = (array[j],array[i])
    (array[i+1],array[high]) = (array[high],array[i+1])
    return i+1
def quicksort(array,low,high):
    if low<high:
        pi = partition(array,low,high)
        quicksort(array,low,pi-1)
        quicksort(array,pi+1,high)
```

快速排序的时间复杂度：若partition总是在列表中间出现，时间复杂度为nlogn，其平均复杂度也为nlogn。但若pivot选的不好，最坏情况为O（n^2）

快速排序并非稳定排序。

所需额外空间最坏的情况为O（N）

分拣中值的选择 -- 三者取中原则：选择pivot时，选择第一、最后和中间元素的中间值（同第一个元素交换位置）

5.归并排序（Merge Sort）

基本原理：归并排序是持续对半分割序列直至单元素无法再分割的递归算法，最后，再将排序好的子序列进行合并。

归并排序本质上也是一种分治的方法

```
def mergeSort(arr):
        if len(arr) > 1:
                mid = len(arr)//2

                L = arr[:mid]   # Dividing the array elements
                R = arr[mid:] # Into 2 halves

                mergeSort(L) # Sorting the first half
                mergeSort(R) # Sorting the second half

                i = j = k = 0
                # Copy data to temp arrays L[] and R[]
                while i < len(L) and j < len(R):
                        if L[i] <= R[j]:
                                arr[k] = L[i]
                                i += 1
                        else:
                                arr[k] = R[j]
                                j += 1
                        k += 1

                # Checking if any element was left
                while i < len(L):
                        arr[k] = L[i]
                        i += 1
                        k += 1

                while j < len(R):
                        arr[k] = R[j]
                        j += 1
                        k += 1
```

归并排序的时间复杂度为O（nlogn）：T（n）=2T（n/2）+θ（n）。分裂的过程，时间复杂度为O（logn），归并的过程，时间复杂度为O（n）

所需要的额外空间为O（n）（1倍额外空间），较大

其可以并行进行处理。不是原地排序，是稳定排序算法，在小数据上不是最优的。

归并排序是外部排序算法，当主存储器RAM当前可用的存储空间较小时，适宜用这种算法。

注：笔试题目中，（从原理上）归并排序**第一轮**归并的结果应为两两归并后所得到的列表（二路归并）

6.希尔排序（Shell Sort）

基本原理：是插入排序的变种。在插入排序中，每次元素只向前移动一个单位。在希尔排序中，我们允许元素以一个gap值向前移动（缩小增量排序），然后不断减少gap直至1。

随着子列表的数量越来越少，无需表的整体越来越接近有序，从而减少整体排序的比对次数

一般子列表的间隔从n/2开始，每趟倍减

```python
def shellSort(arr, n):
    gap = n // 2
    while gap > 0:
        j = gap
        # Check the array in from left to right
        # Till the last possible index of j
        while j < n:
            i = j - gap  # This will keep help in maintain gap value

            while i >= 0:
                # If value on right side is already greater than left side value
                # We don't do swap else we swap
                if arr[i + gap] > arr[i]:

                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]

                i = i - gap  # To check left side also
            # If the element present is greater than current element
            j += 1
        gap = gap // 2
```

时间复杂度：希尔排序是对插入排序的优化。其算法分析较为困难，大致介于O（n）和O（n^2）之间，若将间隔保持在2^k-1，希尔排序的时间复杂度约为O（n^1.5）

希尔排序并不是一个稳定排序

注：其本质是对于给定的gap（不断倍减），在gap分割情况下的子序列使用插入排序

希尔排序不需要额外空间

6.排序算法间比较

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$nlog^2n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection	
Merge sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Merging	Highly parallelizable (up to *O*(log *n*) using the Three Hungarian's Algorithm)
Timsort	n	$nlogn$	$nlogn$	n	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Quicksort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning	Quicksort is usually done in-place with *O*(log *n*) stack space.
Shellsort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	n	n^2	n^2	1	Yes	Insertion	*O*(n + d), in the worst case over sequences that have *d* inversions.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

排序算法	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	不稳定
插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定
快速排序	O(nlogn)	$O(n^2)$	O(nlogn)	O(nlogn)	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定
希尔排序	O(nlogn)	O(ns)	O(n)	O(1)	不稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
计数排序	O(n+k)	O(n+k)	O(n+k)	O(n+k)	稳定
基数排序	O(N*M)	O(N*M)	O(N*M)	O(M)	稳定