数据科学导论 第三次作业

厉帅 10235501418

作业内容：
完成“数据科学与工程导论-04”课后题的复习题的第 3-7 题;践习题 1、6、7、8 题。

复习题：

3.

算法的时间复杂度是对算法运行时间的度量，通常用大 O 记号（Big O notation）表示。它描述了随着输入数据规模的增大，算法的运行时间是如何增长的。
空间复杂度是对算法运行过程中所需内存空间的度量，描述了随着输入数据规模的增大，算法的内存需求如何增长。空间复杂度也用大 O 记号表示。

4.

算法算法是解决特定问题的一系列明确的步骤或指令。它定义了从输入到输出的过程，可以是某个计算问题、数据处理问题或逻辑推理问题。
算法的作用体现在多个方面,尤其是在计算机科学和日常生活中,它们起着至关重要的作用：提高计算效率；提供问题解决方案：算法是问题解决的核心；优化资源使用；自动化操作；保证结果的准确性；处理大规模数据。

5.

- 时间复杂度分析

示例：对于一个输入为 n 的数组的线性查找算法，需要依次遍历所有元素，查找到目标元素。其时间复杂度为 O(n)，表示随着输入规模 n 的增长，算法运行时间呈线性增长。

- 空间复杂度分析

示例：递归算法，如斐波那契数列的递归解法，由于每次递归调用都需要存储中间状态，因此其空间复杂度为 O(n)。

- 渐进分析

示例：插入排序的时间复杂度为 $O(n^2)$，但在输入数据接近有序的情况下，其实际表现接近 O(n)。

- 最坏、最好和平均情况分析

示例：对于快速排序，在最坏情况下（即每次选择的基准值导致极度不平衡的划分），时间复杂度为 $O(n^2)$。但在平均情况下，其时间复杂度为 O(n log n)，这也是其实际表现更优的原因。

- 实际性能分析（实验分析）

示例：运行归并排序和快速排序并比较它们在不同规模的数据集上的实际表现，可以通过时间测量函数（如 time()）来得出结论。

- 问题规模和增长率

示例：比较 $O(n^2)$ 和 O(n log n)算法的增长率：假设数据规模为 10，000，当使用 $O(n^2)$ 算法时，需要的时间为 100,000,000，而使用 O(n log n)算法时，需要的时间约为 132,877。可以明显看出，在大规模输入下，O(n log n)算法表现更优。

6.

- 时间复杂度评判

时间复杂度是指算法在运行过程中随着输入规模增大，所需执行步骤或时间的增长情况。评判时间复杂度可以通过忽略低阶项和常数系数得到渐进的结果。

不同情况下的复杂度：最坏情况：输入导致算法需要最长时间执行；最好情况：输入使算法执行时间最短；平均情况：算法在所有可能的输入下的平均执行时间。

- 空间复杂度评判

空间复杂度衡量的是算法在运行时所需的额外内存空间。和时间复杂度类似，空间复杂度也是输入规模的函数，用大 O 表示。空间复杂度包含两部分：固定空间：算法执行时所需要的常量空间，独立于输入规模，如函数调用时的局部变量、常量等；动态分配的空间：算法根据输入规模动态分配的额外空间，如递归调用栈、动态分配的数组或哈希表等。

7.

- 有穷性

有穷性是指算法在执行有限的步骤之后必须终止，不能进入无限循环。换句话说，算法的每一个步骤都清晰明确，执行过程在有限的时间内结束。

- 确定性

确定性指的是算法的每一步操作在每一种情况下都是明确的，没有二义性。也就是说，对于相同的输入，每一次运行算法时，都会执行完全相同的步骤，并得到相同的输出。

- 可行性

可行性意味着算法的每一个步骤都是可以在有限的时间内通过某种已知的机制完成的，操作的执行要简单且有效。算法中的每个步骤必须足够明确，以便可以通过物理设备（例如计算机）执行。

- 多个输入

多个输入意味着一个算法应该能够接受多个输入变量，甚至是复杂的数据结构作为输入。输入可以是数值、字符、列表、树、图等多种数据类型。

- 一个或多个输出

一个或多个输出意味着算法必须提供至少一个结果，或产生一个可识别的终止状态，代表问题已经解决。输出可以是数值、布尔值、结构化数据等。

践习题 1、6、7、8 题：

1.

```python
import math

def is_prime(a):
    if a < 2:
        return False  # 小于 2 的数不是质数

    for i in range(2, int(math.sqrt(a)) + 1):
        if a % i == 0:
            return False  # 如果 a 能被 i 整除，则 a 不是质数
    return True  # 如果没有找到能整除 a 的数，则 a 是质数

a = int(input("请输入一个整数: "))
```

```
if is_prime(a):
    print(f"{a} 是质数")
else:
    print(f"{a} 不是质数")
```

6.

```
import random
import time
import pandas as pd
from IPython.display import display

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
def test_selection_sort(lengths):
    results = []
    for length in lengths:
        random_array = random.sample(range(1, 10000), length)
        start_time = time.time()
        selection_sort(random_array.copy())
        end_time = time.time()
        execution_time = (end_time - start_time) * 1000  # Convert to
milliseconds
        results.append(execution_time)
    return results
array_lengths = [10, 100, 500, 1000, 2000]
sort_times = test_selection_sort(array_lengths)

# Prepare data for DataFrame
data = {
    'Array Length': array_lengths,
    'Execution Time (ms)': sort_times
}
df = pd.DataFrame(data)

display(df)
```

运行结果：

	Array Length	Execution Time (ms)
0	10	0.000000
1	100	0.000000
2	500	6.996393
3	1000	24.002075
4	2000	91.669798

7. 用迭代法而非层层递归，减少深度递归的浪费

```
def hanoi_iterative(n, source='X', target='Y', auxiliary='Z'):
    stack = []
    stack.append((n, source, target, auxiliary))

    while stack:
        n, source, target, auxiliary = stack.pop()

        if n == 1:
            print(f"Move disk from {source} to {target}")
        else:
            stack.append((n-1, auxiliary, target, source))
            stack.append((1, source, target, auxiliary))
            stack.append((n-1, source, auxiliary, target))

# Example usage
hanoi_iterative(3)  # Solves the problem for 3 disks
```

8.

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def inorder_traversal(root):
    if root is None:
        return

    # Visit left subtree
    inorder_traversal(root.left)
```

```
    # Visit node
    print(root.value)

    # Visit right subtree
    inorder_traversal(root.right)


root = TreeNode(1)

root.left = TreeNode(2)

root.right = TreeNode(3)

inorder_traversal(root)
```