**沈思妤-10235501458-数据科学导论第四次作业**

**复习题**

1. **算法的时间复杂度和空间复杂度分别是什么？**

时间复杂度：算法的时间特性可以用时间复杂度来表示，通常将每个程序所需要的执行次数称为该程序的时间复杂度。一个算法的耗费时间是该算法所有语句的执行时间之和，而每条语句的执行时间是该语句的执行次数和执行一次所需时间的乘积。讨论算法的时间复杂度，也是在讨论程序使用该算法运行的时间。

空间复杂度：算法的空间特性可以用时间复杂度来表示，指程序运行算法时所消耗的空间资源，这包括了寄存器、内存和磁盘等。在算法执行过程中，除了输入和输出所必需的空间外，还包括了程序中声明的变量、动态分配的数组、递归调用的栈空间等辅助空间。

1. **算法是什么，有什么作用？**

答：算法是对可机械执行的一系列步骤精准而明确的规范。算法是解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述问题的策略机制。它是一种系统的方法，通过有限的步骤来处理输入数据，产生所需的输出结果。

算法的作用非常广泛，算法能够为人们解决生活中的问题。在计算机科学中，算法是软件和应用程序的核心，它们决定了程序的运行方式和性能。算法在数学、工程、生物学、经济学等多个学科中都有重要应用，在日常生活中也无处不在，提高我们的生活质量和工作效率。

1. **算法分析的方法是多种多样的，常用的评判算法效率的方法有哪些？请举例说明。**

答：

1. 渐近复杂度分析：使用大O表示法来描述算法的运行时间和空间需求随输入规模增长的上界。这包括最坏情况、最优情况和平均情况的分析，帮助理解算法在不同情况下的性能表现。比如冒泡排序算法的时间复杂度为O(n2），而快速排序算法的平均时间复杂度为O(n log n)，其中n是输入数据的规模。
2. 理论分析：通过数学工具和模型来预测算法的运行时间和内存使用，主要关注算法的时间复杂度和空间复杂度。这种方法不依赖于实际的执行环境，可以提供算法随输入规模增长的效率趋势。比如归并排序算法，其时间复杂度为O(n log n)，可以通过分析归并排序的递归过程和合并过程来得出。
3. 实验测试：通过在特定的硬件和软件环境下实际运行算法，记录其运行时间和资源消耗，从而评估算法的实际性能。这种方法可以提供非常直观的性能数据，但结果可能受到测试环境的影响。比如可以在同一台计算机上，使用相同的数据集（如随机生成的整数数组）来测试冒泡排序、快速排序和归并排序等算法的执行时间。通过比较这些算法在相同条件下的执行时间，可以直观地评估它们的性能差异。
4. 基准测试：利用一组标准数据集或测试用例来评估和比较不同算法的性能。这种方法有助于在相同条件下公平地评价算法的效率。
5. **如何去评判一个算法的复杂度？**

答：评判一个算法的复杂度主要涉及到两个方面：时间复杂度和空间复杂度。这两个方面分别衡量了算法执行所需的时间和空间资源。时间复杂度衡量的是算法执行所需时间的增长趋势，通常表示为输入规模（例如，数组中的元素数量、图中的节点数等）的函数。空间复杂度衡量的是算法执行过程中所需的额外存储空间（不包括输入数据所占用的空间）。

1. **算法在一般情况下被认为有五个基本属性，它们分别是什么？请简要说明。**

答：算法的五个基本属性分别是：有穷性、确定性、可行性、有0个或多个输入、有一个或多个输出。

输入（Input）：算法具有零个或多个输入，这些输入取自于某个特定对象的集合。输入是算法开始执行时所需要的数据或信息，它们可以是数值、字符串、文件或其他数据类型的集合。

输出（Output）：算法至少具有一个或多个输出，这些输出同输入有着某些特定关系的量。输出是算法执行完毕后产生的结果，它可以是计算结果、决策、分类、排序后的数据等。

有穷性（Finiteness）：一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。这意味着算法的执行时间是有限的，它不能陷入无限循环中，而且每一步操作都应在合理的时间内完成。

确定性（Definiteness）：算法中每一条指令必须有确切的含义，不存在二义性。这意味着算法的描述是清晰且明确的，对于算法中的每一个操作，执行者都能准确地理解其意图并正确地执行。

可行性（Effectiveness）：一个算法是可行的，意味着算法描述的操作是可以通过已经实现的基本运算执行有限次来实现的。换句话说，算法中的每一步操作都必须是实际可行的，即它们可以通过现有的计算机硬件和软件来执行。

这五个基本属性共同定义了算法的本质特征，确保了算法的正确性、有效性和可执行性。在实际应用中，设计算法时需要充分考虑这些属性，以确保算法能够满足实际需求并高效运行。

**践习题**

1. **编写 Python 程序，判断输人a是否为质数。**

答：def is\_prime(a):

if a <= 1:

return False

if a == 2:

return True

if a % 2 == 0:

return False

for i in range(3, int(a \*\* 0.5) + 1, 2):

if a % i == 0:

return False

return True

a = int(input("请输入一个数字："))

if is\_prime(a):

print(f"{a} 是质数")

else:

print(f"{a} 不是质数")

1. **用 Python 实现选择排序，并尝试对不同长度的随机数组排序，并计算出程序执行**

**时间。**

答：import random

import time

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_index = i

for j in range(i+1, n):

if arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

def test\_sorting(length):

arr = [random.randint(0, 10000) for \_ in range(length)]

start\_time = time.time()

selection\_sort(arr)

end\_time = time.time()

print(f"排序{length}个元素的数组所需时间: {end\_time - start\_time:.5f} 秒")

test\_sorting(10)

test\_sorting(100)

test\_sorting(1000)

test\_sorting(10000)

运行后得到结果：

排序10个元素的数组所需时间: 0.00000 秒

排序100个元素的数组所需时间: 0.00017 秒

排序1000个元素的数组所需时间: 0.01633 秒

排序10000个元素的数组所需时间: 1.63104 秒

1. **4.3.2小节详细介绍了汉诺塔问题的函数方式的实现,用Python 实现能够运行的代码。该算法有没有可以改进的地方？**

答：def hanoi(n, source, target, auxiliary):

if n == 1:

print(f"Move disk from {source} to {target}")

else:

hanoi(n - 1, source, auxiliary, target)

print(f"Move disk from {source} to {target}")

hanoi(n - 1, auxiliary, target, source)

num\_disks = int(input("请输入盘子的数量: "))

hanoi(num\_disks, 'A', 'C', 'B')

汉诺塔问题通常用递归解决，但可以使用迭代算法实现。虽然它不一定能显著降低复杂度，但可以避免递归带来的栈溢出问题。同时，还可以提升一些细节：

状态存储：将每一步的状态保存，可以在后续使用中快速访问，减少重复计算。

并行处理：对于每个盘子的移动，如果在某些条件下可以独立处理，尝试使用并行处理来提高效率。

可视化：增强用户体验，可以增加图形界面或简单的文本图形表示来更直观地展示盘子的移动过程，帮助理解和教学。

1. **4.3.3小节详细介绍了树排序的实现方法，用Python 实现能够运行的代码。**

答：代码实现如下：

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

def second\_visit\_traversal(node):

if node is None:

return

second\_visit\_traversal(node.left)

print(node.value) # 输出当前节点的值

second\_visit\_traversal(node.right)

# 测试代码

if \_\_name\_\_ == "\_\_main\_\_":

root = TreeNode(1)

root.left = TreeNode(2)

root.right = TreeNode(3)

root.left.left = TreeNode(4)

root.left.right = TreeNode(5)

root.right.left = TreeNode(6)

root.right.right = TreeNode(7)

print("第二次访问遍历结果：")

second\_visit\_traversal(root)