数据科学导论作业

复习题3：

[时间复杂度表示算法运行所需的时间，而空间复杂度表示算法在执行过程中临时占用存储空间的大小](https://cloud.baidu.com/article/3352841" \t "https://cn.bing.com/_blank)。

****时间复杂度****：它描述了算法执行时间随输入数据规模增长的变化趋势。它通常用大O表示法来表示，例如O(n)、O(n^2)、O(log n)等。时间复杂度关注的是算法执行步骤的多少，与具体执行时间无关。

****空间复杂度****：它描述了算法执行过程中所需的存储空间随输入数据规模增长的变化趋势。同样用大O表示法来表示，例如O(1)、O(n)、O(n^2)等。

复习题4：

算法是解决特定问题的一系列定义明确的计算步骤，它通常用于计算机科学和数学领域。算法的作用包括但不限于以下几点：

解决问题：算法提供了一种系统化的方法来解决各种问题，无论是简单的数学计算还是复杂的数据分析。

提高效率：通过精心设计的算法，可以显著提高解决问题的速度，减少资源消耗，如时间、内存等。

确保正确性：算法的步骤通常是确定性的，这意味着在相同的输入下，算法总是产生相同的输出，这有助于确保结果的正确性。

自动化：算法使得计算机程序能够自动执行任务，无需人工干预，这在大规模数据处理和自动化系统中尤为重要。

优化决策：在运筹学和经济学中，算法可以帮助找到最优或近似最优的解决方案，如最短路径问题、资源分配问题等。

模拟和预测：算法可以模拟复杂系统的行为，帮助预测未来的趋势或结果。

复习题5：

评判算法效率的常用方法主要包括以下几种：

事后统计法：通过实际运行算法并记录其运行时间和内存使用情况来评估效率。这种方法简单直接，但受测试环境影响较大，如硬件配置、数据输入等，因此可能无法准确反映算法的真实效率

。

事前分析估算法：在算法实际运行前，通过分析算法的理论性能来预测其效率。这种方法不依赖于具体的运行环境，可以更准确地比较不同算法的效率。

渐近复杂度分析：使用大O表示法来描述算法的时间复杂度和空间复杂度。例如，一个算法的时间复杂度为O(n log n)，表示随着输入规模n的增长，算法的执行时间将按对数级别增长。这种方法关注的是算法执行时间或所需存储空间随输入规模增长的变化趋势，而不是具体数值

。

实验测试：通过在不同规模的输入数据上运行算法，记录其时间与空间的使用情况，从而评估算法的效率。

理论分析：通过数学工具和模型来分析算法的执行步骤，如利用概率论来分析随机化算法的效率。

比较分析：将算法与已知效率的其他算法进行比较，或者使用一些已知的基准问题来测试算法的性能。

举例说明：

冒泡排序的时间复杂度为O(n^2)，在数据量较大时，其效率较低，不适用于处理大规模数据集。

快速排序的平均时间复杂度为O(n log n)，通常情况下比冒泡排序更高效，更适合处理大规模数据集

。

在实际应用中，通常会根据算法的最坏情况、最优情况和平均情况来进行效率分析，以确保算法在各种情况下都能有可接受的性能表现。

复习题6：

评判一个算法的复杂度通常涉及时间复杂度和空间复杂度两个方面：

时间复杂度：衡量算法执行所需时间的增长趋势。它与算法执行过程中进行的基本操作次数有关。常见的时间复杂度有：

O(1)：常数时间复杂度，与输入规模无关。

O(log n)：对数时间复杂度，每次输入规模增加一个数量级，执行时间增加一个常数倍。

O(n)：线性时间复杂度，执行时间与输入规模成正比。

O(n log n)：线性对数时间复杂度，常见于高效的排序算法。

O(n^2)：二次时间复杂度，执行时间与输入规模的平方成正比。

O(2^n)：指数时间复杂度，非常慢的增长趋势。

O(n!)：阶乘时间复杂度，用于描述诸如旅行商问题这类的组合爆炸问题。

空间复杂度：衡量算法执行过程中所需的存储空间的增长趋势。它与算法执行过程中分配的额外存储空间有关。常见的空间复杂度有：

O(1)：常数空间复杂度，与输入规模无关。

O(n)：线性空间复杂度，存储空间与输入规模成正比。

O(n^2)、O(n^3)：多项式空间复杂度，通常用于描述递归算法的空间需求。

O(log n)：对数空间复杂度，较少见，通常用于描述某些高效的树形结构。

评判算法复杂度的方法包括：

确定输入规模：明确算法的输入是什么，以及如何定义输入规模。

识别基本操作：找出算法中影响复杂度的关键操作，如循环次数、递归调用次数等。

分析最坏情况：通常分析算法的最坏情况复杂度，以保证算法在任何情况下的表现。

去除常数项与低阶项：在大O表示法中，忽略这些项，只关注最高阶项。

复习题7：

算法通常被认为具有以下五个基本属性，它们共同定义了算法的质量和效率：

有穷性（Finiteness）：

算法必须在执行有限步骤之后结束。这意味着算法不能包含无限循环，且每一步都必须在有限时间内完成。

确定性（Determinism）：

算法的每一步操作都必须有明确的定义，相同的输入总是产生相同的输出。算法的执行不依赖于任何随机因素或非确定性过程。

可行性（Feasibility）：

算法描述的操作必须是可执行的，即每一步都必须是可以通过已经实现的基本运算执行的，不包含无法执行或无法实现的步骤。

输入（Input）：

一个算法可以有一个或多个输入。输入是算法执行所需的初始数据或信息，是算法加工和处理的对象。

输出（Output）：

算法必须有一个或多个输出，作为算法执行的结果。输出是算法处理输入后产生的数据，是算法目的的体现。

践行题1：

a = int(input(""))

isprime = True

if a < 5:

    if a == 1 or a == 4:

        isprime = False

else:

    for i in range(2,a//2+1):

        if a % i ==0:

            isprime = False

print(isprime)

践行题6：

import random

import time

def selection\_sort(arr):

    for i in range(len(arr)):

        min\_index = i

        for j in range(i+1, len(arr)):

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

                min\_index = j

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

    return arr

def generate\_random\_array(length):

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

def calculate\_sort\_time(arr\_length):

    array = generate\_random\_array(arr\_length)

    start\_time = time.time()

    selection\_sort(array)

    end\_time = time.time()

    return end\_time - start\_time

lengths = [100, 1000, 5000, 10000]

for length in lengths:

    time\_taken = calculate\_sort\_time(length)

    print(f"Length: {length}, Time taken: {time\_taken:.6f} seconds")

践行题7:

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

    """

    解决汉诺塔问题的递归函数

    :param n: 盘子的数量

    :param source: 起始柱子

    :param target: 目标柱子

    :param auxiliary: 辅助柱子

    """

    if n > 0:

        # 将 n-1 个盘子从 source 移动到 auxiliary

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

        # 将最大的盘子从 source 移动到 target

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

        # 将 n-1 个盘子从 auxiliary 移动到 target

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

# 盘子的数量

n = 3

# 调用函数开始解决汉诺塔问题

hanoi(n, 'A', 'C', 'B')  # A为起始柱子，C为目标柱子，B为辅助柱子

关于改进：

非递归实现：虽然递归实现很简洁，但它可能会导致大量的函数调用，在盘子数量较多时可能会引起栈溢出。可以使用迭代方法来避免这个问题。

使用记忆化：对于递归实现，可以使用记忆化技术来存储已经解决的子问题的解，避免重复计算。

践行题8：

class TreeNode:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, value):

        if self.root is None:

            self.root = TreeNode(value)

        else:

            self.\_insert\_recursive(self.root, value)

    def \_insert\_recursive(self, node, value):

        if value < node.value:

            if node.left is None:

                node.left = TreeNode(value)

            else:

                self.\_insert\_recursive(node.left, value)

        elif value > node.value:

            if node.right is None:

                node.right = TreeNode(value)

            else:

                self.\_insert\_recursive(node.right, value)

    def inorder\_traversal(self, node, result=None):

        if result is None:

            result = []

        if node is not None:

            self.inorder\_traversal(node.left, result)

            result.append(node.value)

            self.inorder\_traversal(node.right, result)

        return result

# 使用二叉搜索树进行排序

def sort(values):

    bst = BinarySearchTree()

    for value in values:

        bst.insert(value)

    return bst.inorder\_traversal(bst.root)