第二次作业

by 吴天韵10235304421

复习题

3 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"。（书上P115）我们以通常用最坏情况下的时间复杂度来衡量算法的时间复杂度。（P116）它通常用大O符号表示，如 `O(n)` 、 `O(log n)` 等。时间复杂度考虑了基本操作的数量，随着输入数据量的增加，算法所需的时间如何变化。例如， `O(n)` 表示算法的执行时间与输入大小成线性关系， `O(n^2)` 表示时间复杂度随输入大小的平方增长。

空间复杂度:
空间复杂度代表了“程序处理的空间”，即“程序运行算法时所需的那部分空间”（P116）。
它同样用大O符号表示，反映了算法在处理数据时所需的临时空间。例如， `O(1)` 表示算法使用的额外空间是常数， `O(n)` 则表示所需空间与输入规模成线性关系。空间复杂度考虑了算法所需的额外空间，包括变量、数据结构等，而不只是输入数据的大小。

4 算法是什么？有什么作用?

算法是对可机械执行的一系列步骤精准而明确的规范（P105），是一系列明确、有序、有限的指令集合，用于解决特定问题或执行特定任务。这些指令被设计来让计算机或其他可执行的机器按照预定的逻辑步骤进行操作，以达到预定的目标或结果。算法是计算机科学和信息技术领域中的核心概念，也是编程和软件开发的基础。

算法的作用主要包括:

1. **解决问题:**
   算法是解决特定问题的详细步骤或过程。通过定义明确的算法，我们可以将复杂的问题分解为一系列简单的、可管理的任务，从而更容易地找到解决方案。
2. **提高效率:**
   优秀的算法能够显著减少解决问题所需的时间、空间或其他资源消耗。在处理大规模数据时，高效的算法尤为重要，因为它们能够在合理的时间内提供结果，而不会造成资源浪费或系统崩溃。
3. **自动化:**
   算法是实现自动化的关键。通过将算法编程为计算机程序，我们可以让计算机自动执行繁琐、重复或复杂的任务，从而释放人类的劳动力，提高工作效率和准确性。

4. **决策支持：**
   算法可以分析大量数据并提取有用信息，为决策提供科学依据。在商业、科学、医疗等领域，算法被广泛用于预测趋势、识别模式、评估风险等，以支持更加明智的决策。
5. **创新推动：**
   算法的发展推动了科学技术的进步和创新。随着算法的不断优化和改进，我们能够解决以前无法解决的问题，实现以前无法想象的功能和应用。例如，机器学习算法的发展推动了人工智能领域的飞速发展。

5 常用的评判算法效率的方法有哪些？请举例说明

1. **时间复杂度分析：**
   通过分析算法的基本操作数量，评估算法在最坏、平均和最好情况下的运行时间。例如，排序算法的时间复杂度可为 `O(n log n)`（归并排序）或 `O(n^2)`（冒泡排序）。
2. **空间复杂度分析：**
   评估算法在运行时所需的内存空间，包括输入数据和额外空间。例如，递归算法通常需要 `O(n)` 的空间用于栈存储，而迭代算法可能只需要O(1)。
3. **渐进分析：**
   使用大O符号、Ω符号和Θ符号描述算法的性能，从而忽略常数和低阶项。比如，对于一个 `O(n^2)` 的算法，当n非常大时，它的效率主要由 `n^2` 主导。
4. **实验测量：**
   通过实际运行算法并测量其执行时间和内存使用情况。比如，使用不同规模的数据集运行排序算法，记录其运行时间，从而获取实测的效率数据。
5. **最坏情况和平均情况分析：**
   分析算法在特定输入下的性能，最坏情况表示在最不利条件下的表现，而平均情况则考虑所有可能输入的平均表现

6 如何去评判一个算法的复杂度？

评判一个算法的复杂度主要从两个方面来进行：时间复杂度和空间复杂度。
除此以外，还有：并发性和并行性、I/O 复杂度、网络通信开销、实现复杂度……

7 算法在一般情况下被认为有五个基本属性。请简要说明。

算法应具有**指定输入**、**指定输出**、**确定性**、**有效性**和**有限性**五个基本属性。（P105）

1. **指定输入**
   指定输入意味着算法在开始执行之前必须明确知道它需要哪些输入数据。输入可以是一个或多个，也可以是

没有输入（即空输入）。输入数据是算法处理的对象，决定了算法执行的具体行为和结果。

2. **指定输出**
   算法必须有至少一个输出结果。输出是算法执行完毕后产生的结果，它应当满足一定的条件或解决特定的问题。输出可以是一个单一的结果，也可以是多个结果，或者是某种状态的变化。
3. **确定性**
   算法的每一步都应该是明确无误的，即每一步骤都必须有清晰的定义，不能存在模糊或歧义。给定相同的输入，算法应当总是产生相同的结果。换句话说，算法的每一步都应该能够精确地执行，并且在执行过程中不会出现随机或不确定的行为。
4. **有效性**
   有效性意味着算法中的每一步操作都应该简单到足以被有效地执行。这意味着每一步都应该能够由某个人或机器在有限的时间内完成。如果一个步骤无法实际执行，那么算法就失去了其实用价值。
5. **有限性**
   算法必须在有限的步骤内结束，也就是说，算法不应该陷入无限循环或无休止的计算之中。算法的设计应该保证它能在合理的时间内给出结果，即使对于非常大的输入数据集也是如此。

践习题

1 编写Python程序。判断输入a是否为质数

```python
import math

def is_prime(a):
    if a < 2:
        return False
    for i in range(2, int(math.sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True

if __name__ == "__main__":
    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")
```

结果1:

> 请输入一个整数: 1
> 1 不是质数

结果2:

> 请输入一个整数: 10
> 10 不是质数

结果3:

> 请输入一个整数: 97
> 97 是质数

6 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。

```
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_sort(sort_func, sizes=[100, 1000, 10000]):
    for size in sizes:
        arr = [random.randint(0, 10000) for _ in range(size)]
        start_time = time.time()
        sort_func(arr)
        end_time = time.time()

        print(f"数列长度{size}, 用时 {end_time - start_time:.2f} seconds")

test_sort(selection_sort)
```

结果1:

> 数列长度100, 用时 0.00 seconds
> 数列长度1000, 用时 0.01 seconds
> 数列长度10000, 用时 0.97 seconds

结果2:

> 数列长度100, 用时 0.00 seconds
> 数列长度1000, 用时 0.01 seconds
> 数列长度10000, 用时 1.03 seconds

7 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?

```
def hanoi(n, source, target, med):
    """
    递归解决汉诺塔问题
    : n: 盘子的数量
    : source: 初始柱子
    : target: 目标柱子
    : med: 辅助柱子
    """
    if n == 1:
        print(f"Move {source} to {target}")
        i=1
    else:
        # 将 n-1 个盘子从 source 移动到 med，使用 target 作为辅助
        a=hanoi(n-1, source, med, target)
        # 将第 n 个盘子从 source 移动到 target
        print(f"Move {source} to {target}")
        # 将 n-1 个盘子从 med 移动到 target，使用 source 作为辅助
        b=hanoi(n-1, med, target, source)
        i=a+b+1
    return i

def solve_hanoi(n):
    print(f"解决 {n} 个盘子的汉诺塔问题：")
    total_steps=hanoi(n, 'A', 'B', 'C')
    print(f"总共需要 {total_steps} 步。")

n = int(input("请输入盘子的数量："))
solve_hanoi(n)
```

结果1:

> 请输入盘子的数量：3
> 解决 3 个盘子的汉诺塔问题：
> Move A to B
> Move A to C
> Move B to C
> Move A to B
> Move C to A
> Move C to B
> Move A to B
> 总共需要 7 步。

结果2:

> 请输入盘子的数量：5
> 解决 5 个盘子的汉诺塔问题：

> Move A to B
> Move A to C
> Move B to C
> Move A to B
> Move C to A
> Move C to B
> Move A to B
> Move A to C
> Move B to C
> Move B to A
> Move C to A
> Move B to C
> Move A to B
> Move A to C
> Move B to C
> Move A to B
> Move C to A
> Move C to B
> Move A to B
> Move C to A
> Move B to C
> Move B to A
> Move C to A
> Move C to B
> Move A to B
> Move A to C
> Move B to C
> Move A to B
> Move C to A
> Move C to B
> Move A to B
> 总共需要 31 步。

值得改进的地方：
虽然递归解法是经典的解决方法，但对于非常大的 n，递归调用的深度可能会达到 Python 的递归限制。可以考虑通过动态规划或者迭代的方式来避免递归深度限制。

8 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。

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

def insert(node, data):
    if node is None:
        return Node(data)
    else:
        if data <= node.data:
            node.left = insert(node.left, data)
        else:
            node.right = insert(node.right, data)
    return node


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

    second_visit_traversal(root.left)
    print(root.data,end=" ")
    second_visit_traversal(root.right)


# 创建二叉搜索树
input_sequence = input("请输入一个由空格分隔的数字序列来构建二叉搜索树: ").split()
sequence = [int(num) for num in input_sequence]
root = None
for data in sequence:
    root = insert(root, data)


second_visit_traversal(root)
```

结果1 :

> 请输入一个由空格分隔的数字序列来构建二叉搜索树: 2 3 17 8 34 18 19 12 1 5 9 7
> 1 2 3 5 7 8 9 12 17 18 19 34

结果2:

> 请输入一个由空格分隔的数字序列来构建二叉搜索树: 45 17 3 22 35 34 42 29 26 43 31 40 14 2 33 4 30 12 47 21 23 1 25 24 10 13 48 19 41 36 39 27 15 32 37 46 28 44 49 11 16 18 5 20 9 6 7 8
> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 39 40 41 42 43 44 45 46 47 48 49