编码实现、分析和测试作业

题目:最长上升子序列的编程实现、代码分析、单元测试

给定一个长度为 N 的数列，求数值严格单调递增的子序列的长度最长是多少(子序列元素在原数列中可以不连续)

示例 1:输入:nums=[3,1,2,1,8,5,6]

输出:4

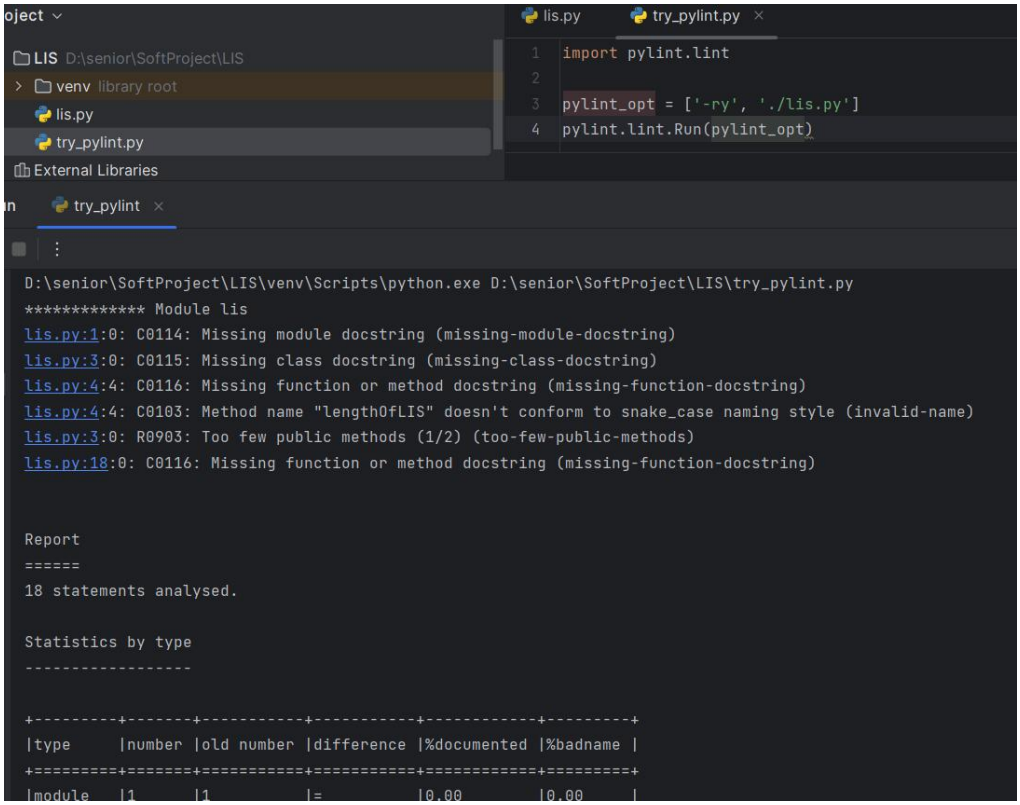
零：要求分析

动态规划是利用一个数组 dp 来记录以每个元素结尾的最长递增子序列的长度。通过迭代计算，最终可以得到整个数列的最长递增子序列的长度。
状态定义：定义一个长度与输入数列 nums 相同的动态规划数组 dp，其中 dp[i] 表示以 nums[i] 结尾的最长递增子序列的长度。
初始化：将 dp 数组的所有元素初始化为 1，因为任何单个元素都可以形成一个长度为 1 的递增子序列。
状态转移：对于每个元素 nums[i]，我们再次遍历之前的元素 nums[j]（j < i），如果 nums[i] 大于 nums[j]，则表示可以将 nums[i] 加入到以 nums[j] 结尾的子序列中，从而形成更长的子序列。因此，更新 dp[i] = max(dp[i], dp[j] + 1)。
结果计算：最终，dp 数组中的最大值即为最长递增子序列的长度。时间复杂度为 O(N^2)，其中 N 是输入数列的长度

一、用 pylint 进行代码分析

try_pylint.py

```
1.  import pylint.lint
2.
3.  pylint_opt = ['-ry', './lis.py']
4.  pylint.lint.Run(pylint_opt)
```



Missing module docstring: 模块缺少文档字符串。建议在模块的开头添加文档字符串，描述模块的功能和使用方法。

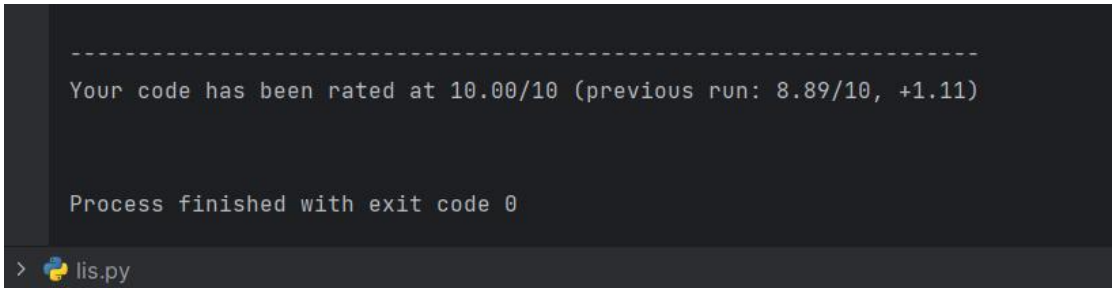
Missing class docstring: 类缺少文档字符串。建议在类的开头添加文档字符串，描述类的功能和使用方法。

Missing function or method docstring: 函数或方法缺少文档字符串。建议在函数或方法的开头添加文档字符串，描述函数或方法的功能、参数和返回值。

Method name "lengthofIls" doesn't conform to snake_case naming style: 方法名不符合蛇形命名风格。Python 通常使用 snake_case 命名风格，建议将方法名改为小写字母，并使用下划线分隔单词。

Too few public methods (1/2): 类中公共方法过少。这个警告表示类中公共方法的数量较少，可能导致类的职责不够清晰。如果确实需要更多的公共方法，可以考虑添加更多的方法来增强类的功能和灵活性。

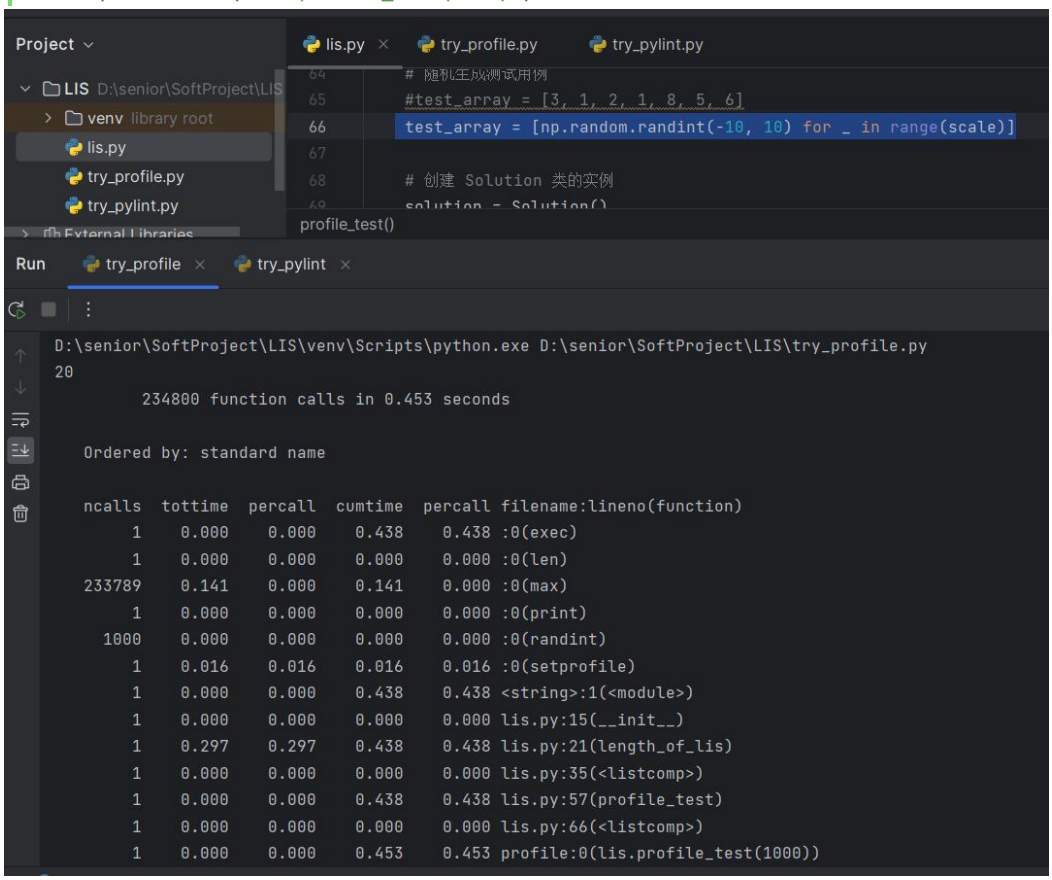
按照给出的提示修改，如下：



二、用 profile 进行性能分析

try_profile:

```
import profile
import lis as lis

profile.run("lis.profile_test(1000)")
```



240354 function calls in 0.219 seconds：总共执行了 240354 个函数调用，并花费了 0.219 秒的时间。

ncalls：函数调用的次数。

tottime：函数的总时间。

percall：每次函数调用的平均时间。

cumtime：函数及其所有子函数的总时间。

percall：每次函数及其所有子函数调用的平均时间。

filename:lineno(function)：文件名、行号和函数名。

分析可知，

lis.py:21(length_of_lis) 函数占用了大部分的时间，总时间为 0.219 秒中的 0.125 秒。这是因为在这个函数中进行了大量的计算，包括循环遍历、比较和更新数组等操作。

max() 函数在总时间中占据了一定比例，共执行了 239343 次，总时间为 0.094 秒。这是因为在动态规划的过程中，需要频繁地比较更新最大值。

基于以上分析，性能瓶颈主要集中在 length_of_lis 方法中的计算过程，特别是在使用动态规划求解最长递增子序列的过程中。

改进思路：

```
    def length_of_lis(self, nums: List[int]) -> int:
        if not nums:
```

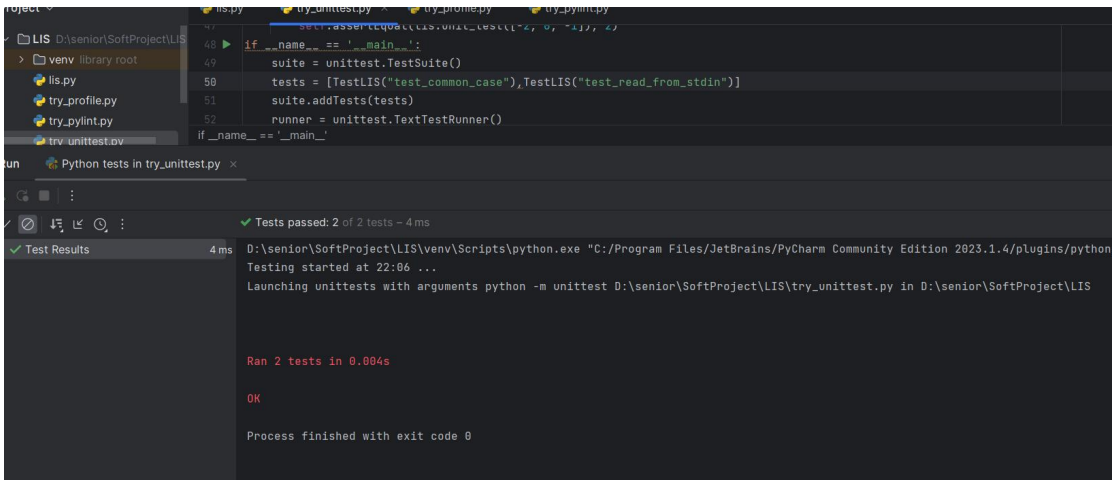
```
            return 0

        piles = []
        for num in nums:
            # 在堆栈中找到 num 应该放置的位置
            left, right = 0, len(piles) - 1
            while left <= right:
                mid = (left + right) // 2
                if piles[mid] >= num:
                    right = mid - 1
                else:
                    left = mid + 1

            # 如果堆栈中没有比 num 大的元素，则新建一个堆栈
            if left == len(piles):
                piles.append(num)
            else:
                piles[left] = num

        return len(piles)
```

```
D:\senior\SoftProject\LIS\venv\Scripts\python.exe D:\senior\SoftProject\LIS\try_profile.py
20
         3030 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       20    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.000    0.000 :0(exec)
     2001    0.000    0.000    0.000    0.000 :0(len)
        1    0.000    0.000    0.000    0.000 :0(print)
     1000    0.000    0.000    0.000    0.000 :0(randint)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 lis.py:15(__init__)
        1    0.000    0.000    0.000    0.000 lis.py:44(length_of_lis)
        1    0.000    0.000    0.000    0.000 lis.py:80(profile_test)
        1    0.000    0.000    0.000    0.000 lis.py:90(<listcomp>)
        1    0.000    0.000    0.000    0.000 profile:0(lis.profile_test(1000))
        0    0.000             0.000          profile:0(profiler)
```

三、用 unittest 进行单元测试

对于数据的正负值，我们设计了全为正数、全为负数以及正数和负数混合的测试样例
对于数据类型，我们设计了全为整数、全为浮点数以及整数和浮点数混合的测试样例
对于数组长度，我们设计了数组长度为 1 的极端数据情况

```
import unittest
import lis as lis
from unittest.mock import patch
class TestLIS(unittest.TestCase):
    """单元测试类"""
    def test_common_case(self):
        #数组长度为1
        self.assertEqual(lis.unit_test([2]),1)
        self.assertEqual(lis.unit_test([-2]),1)
        #全正整数
        self.assertEqual(lis.unit_test([3,4,5]),3)
        #全负整数
        self.assertEqual(lis.unit_test([-3,-4,-5]),1)
        #正负整数混合
        self.assertEqual(lis.unit_test([2,3,-4,5]),3)
        #全浮点数
        self.assertEqual(lis.unit_test([2.0,-3.0,4.0,5.0]),3)
        #整数浮点数混合
        self.assertEqual(lis.unit_test([2,-3.0,4,5.0]),3)
        #带0
        self.assertEqual(lis.unit_test([-2,0,-1]),2)

if __name__ == '__main__':
    suite = unittest.TestSuite()
    tests = [TestLIS("test_common_case")]
    suite.addTests(tests)
    runner = unittest.TextTestRunner()
    runner.run(suite)
```



程序正确。

1. 算法效率

算法复杂度是多少?如何对代码性能进行分析?分析的结果如何?你是如何进行优化的?

改进：
遍历原始数组 nums，将每个元素插入到一个有序的堆栈中。
如果当前元素大于堆栈中的最大元素，则将其放入新的堆栈。
如果当前元素小于或等于堆栈中的最大元素，则在堆栈中找到第一个大于等于当前元素的元素，并将其替换为当前元素。
通过这种方式，最终堆栈的个数即为最长递增子序列的长度。
维护一个有序的堆栈，当遍历到一个新元素时，根据其大小将其插入到堆栈中的适当位置。整个过程只需要遍历一次数组，因此时间复杂度为 o(nlogn)

2. 扩展

是否考虑算法的可扩展性

程序的可扩展性是指程序在面对变化和增长时，能够以一种简单、灵活和可持续的方式进行修改、扩展和维护的能力。可扩展性是一个重要的软件设计目标，它确保程序在需求变化和系统演化的情况下能够保持高效、可靠和易于维护。一个可扩展的程序具有以下特征：

- 模块化设计：程序应该被分解为独立的、可重用的模块，每个模块都专注于特定的功能。模块化设计使得在需要扩展或修改程序时，可以更容易地理解和操作代码，而无需改动整个程序。
- 松耦合：模块之间应该尽可能地解耦，即减少模块之间的依赖关系。通过定义明确的接口和使用适当的抽象层，可以降低模块之间的耦合性，使得对一个模块的修改不会对其他模块产生不必要的影响。
- 可替换性：程序应该提供一种简单的方式来替换或添加新的功能模块，而无需修改现有代码。这可以通过接口的设计和依赖注入等技术来实现。可替换性使得在需求变化时，可以快速地引入新的功能或替换旧的实现，而不会对整个程序产生破坏性的影响。
- 可配置性：程序应该提供一些配置选项，使得用户能够根据需求和环境来自定义程序的行为。通过将程序的行为参数化，可以更容易地适应不同的使用场景和需求变化。
- 扩展点：程序应该提供一些明确的扩展点，允许其他开发者或用户通过插件、模块或扩展来增加新的功能。这些扩展点应该具有良好的接口定义和规范，以便其他人能够简单地添加新功能而无需修改核心代码。

数据扩展：数据方面，我们主要考虑了数据来源的扩展、数据规模的扩展以及数据形式的扩展。
数据来源：考虑我们如何得到数据，我们可以从用户输入、文件、网络、数据库等来源处得到我们需要处理的数组，因此这里我们将得到数据的这一行为封装成 *set_array* 成员函

数，该函数根据传入参数的不同调用不同的函数，从不同的源头处获得数据。如代码实现部分所展示，我们目前实现了从用户输入、文件、函数参数处获取待处理的数组，之后如果想对获取数组这一行为进行扩展，我们只需要实现新的相应的数据读取函数，并将该函数添加到 *set_array* 函数的一个新的分支中即可。

数据规模：这里为了支持对任意长度的数组以及多维数组的处理，我们选择使用 *list* 类型来存储数组，使用 *list* 类型在理论上可以对数组的维度和数组长度进行无限延伸，从而支持不同数据规模的处理。
考虑到在未来，我们还有可能对最长增长子序列算法的具体实现方式进行更改，我们定义成员函数提供统一的抽象，当我们修改或者添加新的算法实现方式的时候，我们只需要在函数中调用新的处理函数即可，而不需要更改其他的函数的代码。

结果呈现扩展: 除了返回数组的最长增长子序列的长度值之外，我还实现了得到该最长增长子序列的具体内容，用户调用函数便可以得到最长增长子序列内容，未来可以考虑将结果可视化地呈现给用户

3.两个原则

单一职责原则(SingleResponsibility Principle,SRP)

单一职责原则指的是一个类或模块应该只有一个单一的责任或功能。换句话说，一个类应该只有一个引起它变化的原因。如果一个组件有多个职责，则会导致代码耦合度增加，难以维护和扩展。相反，如果一个组件只负责一个职责，则可以使其更加专注、可测试和易于修改。单一职责原则是许多其他原则和设计模式的基础，如依赖反转原则、工厂模式等。这样做的好处是：

- 高内聚：类的职责单一，类内部的各个方法和属性都围绕着同一个关注点。这样可以提高类的内聚性，减少代码的复杂性。
- 易于理解和维护：由于类的职责明确，代码的功能和行为更加清晰，使得其他开发者更容易理解和维护。
- 可扩展性：当需求变化时，只需要修改与该职责相关的类，而不影响其他类。这样可以减少代码的修改范围，降低引入错误的风险。

要遵守和应用单一职责原则，可以采取以下措施：

- 分析类的职责：仔细分析每个类的职责，并确保每个类都专注于一个单一的责任。如果一个类承担了多个职责，应该考虑将其拆分为多个类。
- 提取通用功能：当发现多个类有相似的功能时，可以将这些功能提取出来，形成一个单独的类或模块，以避免功能重复。
- 使用组合和委托：当一个类需要实现多个职责时，可以使用组合和委托的方式，将职责委托给其他类来处理。这样可以保持类的单一职责，同时协同工作完成多个功能。

我们类的成员函数的设计符合单一职责原则，如 *def __init__(self);*

def length_of_lis(self, nums: List[int]) -> int 负责不同的功能

且对于不同数组读取方式以及不同的问题求解算法也编写到了不同的函数中，从而保证对某一个功能的修改只会导致单一函数的代码发生改动，符合单一职责原则。

开放-封闭原则(Open-ClosePrinciple ， OCP)

开放封闭原则指的是软件实体（类、模块、函数等）应该对扩展开放，对修改封闭。换句话说，软件实体应该通过扩展来改变其行为，而不是通过修改其源代码来实现，而是应该通过增加新代码来实现。这样可以避免对原有功能产生影响，并且可以更好地维护和测试代码。为了实现开放封闭原则，可以使用接口、抽象类、继承、多态等技术。开放封闭原则也是面向对象设计中最重要的原则之一。这样做的好处是：

- 可扩展性：通过扩展现有的代码，可以引入新的功能和行为，而无需修改已经稳定的代码。这样可以减少对现有功能的影响和风险。
- 可维护性：由于不修改现有的代码，所以不会破坏已经测试和验证过的功能。这样可以降低维护成本，提高系统的稳定性。

要遵守和应用开放封闭原则，可以采取以下措施：

- 使用抽象和接口：通过定义抽象类和接口，可以定义稳定的协议，供其他模块或类来扩展和实现。这样可以保持现有代码的稳定性，同时允许引入新的实现。
- 使用多态性：通过利用多态性，可以将扩展的具体实现替换为抽象的接口，而不影响现有代码的调用。这样可以实现新的功能，同时不修改已有的代码。
- 使用设计模式：一些设计模式，如装饰器模式、策略模式等，提供了一种实现开放封闭原则的方法。通过应用适当的设计模式，可以使系统更加灵活和可扩展。
- 编写可插拔的代码：通过使用插件、扩展点和配置文件等机制，使得系统的功能可以通过外部的组件进行扩展。这样可以在不修改现有代码的情况下，添加新的功能。

LIS 类成员函数的作用是求出一维数组的最长上升子序列，这一部分内容是不必进行更改的，对于更高维度的数组的求解可以通过调用该函数来实现，是可以扩展的，满足开放-封闭原则。

4.错误与异常处理

错误是导致程序崩溃的问题，例如 *Python* 程序的语法错误 (解析错误) 或者未捕获的异常(运行错误) 等. 异常是运行时期检测到的错误，即使一条语句或者表达式在语法上是正确的，当试图执行它时也可能会引发错误。异常处理是用于管理程序运行期间错误的一种方法，这种机制将程序中的正常处理代码和异常处理代码显式地区别开来，提高了程序的可读性。

良好的错误和异常处理对于构建健壮、可靠的软件系统至关重要。以下是在 *Python* 中实现

良好错误和异常处理的一些关键概念和技术：

• 异常类和异常处理语句：Python 中的异常是通过抛出异常对象来表示的。异常对象是由内置的异常类或自定义的异常类实例化得到的。通过使用 raise 语句抛出异常，可以将异常传递给调用者或处理该异常的代码块。在处理异常时，可以使用 try-except 语句来捕获和处理特定类型的异常。try 块中包含可能引发异常的代码，而 except 块则定义了如何处理捕获到的异常。可以使用多个 except 块来处理不同类型的异常，也可以使用 else 块定义在 try 块中没有发生异常时执行的代码，还可以使用 finally 块定义无论是否发生异常都必须执行的代码。

• 异常处理的层级结构：Python 中的异常类是通过形成层级结构来组织的，这样可以根据异常的类型进行更精细的处理。所有的异常类都是 BaseException 类的子类。常见的内置异常类包括 Exception、TypeError、ValueError 等。可以根据具体情况选择捕获特定类型的异常或捕获更通用的异常。

• 适当捕获异常：捕获和处理可能发生的异常，避免程序因为异常而中断或崩溃。根据具体情况选择捕获特定类型的异常或捕获更通用的异常。

• 提供有意义的错误信息：在捕获异常时，可以使用 except 块来记录或打印有意义的错误信息，以便于调试和排查问题。错误信息应该清晰明确，指示发生了什么问题和如何解决。

• 避免捕获过宽的异常：避免捕获过于宽泛的异常，因为这可能会隐藏潜在的问题或导致错误的处理。尽可能捕获特定的异常类型，以便有针对性地处理和恢复。

• 不要忽略异常：避免将异常处理代码

• 留空或简单地忽略异常。至少应该记录异常，以便追踪和排查问题。根据具体情况，可以选择重新抛出异常、执行备选操作或中断程序的执行。

• 使用 finally 块进行清理操作：finally 块中的代码总是会被执行，无论是否发生异常。这可以用于进行清理操作，如关闭文件、释放资源等。

• 定义异常：在 Python 中，可以通过继承内置的异常类来创建自定义的异常类。自定义异常类可以根据具体的应用场景和需求来定义，以便更好地表示特定的异常情况。通过自定义异常类，可以提供更具体和有意义的错误信息，以及针对特定异常的处理逻辑。

针对我们的程序，首先我们应该规定要处理的数组的格式，我们规定数组应该以列表的形式输入，数组元素为 int 或 float 类型。同时目前只支持一维数组的求解，对于多维数组应当给出报错提

示。

5. 软件编程规范

是否遵守编程规范，参考的哪个规范如何检查是否遵守编程规范的 ？

代码规范是一套约定俗成的准则，用于指导程序员在编写代码时遵循统一的风格和结构。它的目的是提高代码的可读性、可维护性和可扩展性，以便多个人能够协同开发并理解代码。在 Python 中，PEP8（Python Enhancement Proposal 8）是一份广泛接受的代码规范，它提供了关于代码布局、命名约定、注释等方面的建议。

PEP8 规范主要包括以下几个方面的内容：

- 缩进：使用四个空格作为缩进的标准，不要使用制表符（Tab）。
- 行长度：每行代码应尽量保持在 79 个字符以内，超过这个限制时应进行换行处理。对于长的表达式或注释，可以使用括号或反斜杠进行换行，以提高可读性。
- 命名规范：变量、函数、类等命名应具有描述性，并采用小写字母和下划线的组合。类名应采用驼峰命名法（每个单词首字母大写），模块名应尽量短小，并避免与 Python 标准库模块重名。

空格使用：在运算符、逗号、冒号、分号等符号周围应添加适当的空格，以增加代码的可读性。例如，赋值操作符前后应添加空格，函数参数之间应添加逗号和空格。

- 导入规范：每个导入语句应独占一行，按照标准库、第三方库和本地库的顺序进行分组。每个分组之间应留有一个空行。避免使用通配符导入（如 from module import *），而是明确导入所需的模块或函数。
- 注释规范：使用注释来解释代码的功能、实现细节等重要信息。注释应该清晰明了，并且应该与代码保持同步更新。对于非常重要或复杂的功能，可以使用多行注释进行详细说明。
- 函数和类的定义：函数和类的定义之间应留有两个空行，类中的方法定义之间应留有一个空行。函数和方法的参数列表应放在括号内，并在逗号后添加一个空格。
- 其他规范：避免使用多个语句或表达式在同一行上，除非是在列表、字典等简单结构中。对于条件语句和循环语句，要使用适当的缩进，并在必要时使用括号来明确代码块的范围。

为了编写符合 PEP8 规范的 Python 代码，可以使用一些工具来自动检查和格式化代码，例如 pylint 等。这些工具可以帮助发现不符合规范的代码，并提供修复建议。当然，最重要的是我们要养成良好的习惯。

6. 代码性能分析与优化

前：

遍历列表 nums 中的每个元素 nums[i]，对于每个元素，再次遍历其前面的所有元素 nums[j]，其中 j < i。

如果发现 nums[i] 大于 nums[j]，意味着可以将 nums[i] 加入到以 nums[j] 结尾的递增子序列中，这时就更新 dp[i] 的值为 dp[j] + 1，表示以 nums[i] 结尾的递增子序列长度可能比之前的长度要更长，返回 dp 中的最大值。

时间复杂度为 O(n^2)，其中 n 是列表 nums 的长度，因为需要遍历每个元素，并在每个元素处再次遍历其前面的所有元素。

后：

初始化一个空的堆栈 piles，用于存储递增序列的堆栈。
对于列表 nums 中的每个元素 num，我们希望将其按照递增顺序放入堆栈 piles 中。为了实现这一点，我们采用了二分查找的方法来确定 num 应该放置在堆栈中的位置。
在堆栈 piles 中查找 num 应该插入的位置。这里采用的是二分查找的方法，不断缩小查找范围直到找到合适的位置。
如果找不到比 num 大的元素，则将 num 添加到堆栈的末尾，表示新建一个堆栈。
否则，将 num 放置在堆栈中找到的位置上，覆盖掉原有的值，因为我们只关心最终堆栈的长度，而不关心堆栈中的具体元素。
最后，返回堆栈 piles 的长度，即为整个列表 nums 的最长递增子序列长度。
时间复杂度为 O(nlogn)，其中 n 是列表 nums 的长度，因为对于每个元素，需要进行二分查找来确定其插入位置，而二分查找的时间复杂度为 O(logn)。

7.单元测试

测试用例设计思路测试用例表测试覆盖率测试通过率缺陷报告

statements	*missing*	*excluded*	*coverage*
41	16	0	61%
37	0	0	100%
78	**16**	**0**	**79%**

代码中未被覆盖的语句主要为用于性能测试的 profile_test 函数以及 main 函数，这两个函数并不需要被测试，在单元测试时也没有被调用，因此未被覆盖。其余部分的代码大致均已被覆盖。