**Overview**

This assessment evaluates your ability to perform the following tasks in accordance with ICTPRG547 Apply advanced programming skills in another language:

**1. Performance elements**

* 1.4 Code sorting algorithm using programming techniques
* 3.2 Detect and resolve errors of syntactical, logical and design origin
* 3.3 Design and document required tests
* 4.1 Develop and document solution according to debugging test results

You will demonstrate your performance by providing evidence that you can code at least one sorting algorithm, and test and debug the code to resolve errors of a syntactical, logical, or design origin.

To succeed you must use a systematic, analytical processes in complex, non-routine situations, setting goals, gathering relevant information, and identifying, and evaluating, options against the agreed criteria

**2. General instructions**

CRITICAL: Failure to follow these instructions will lead to an NYC

* Copy this file into a docs folder in your assessment repo
* Add and commit this file to your repository and associate the tag por3-start :
  + Copy and add this file to your repository under the docs folder
  + `git commit -m "chore: add task overview to my repo"
  + git tag por3-start
  + git push origin main --tags
  + Optional: you may want to complete this work in a branch
  + On your last commit, add the tag por3-finish
* Commit changes after you complete each task
* Push changes to your GitHub repository
* Ensure you submit your git repo (.git/) along with your assessment submission

**3. Players have scores now**

**3.1. Task: Add scores to players**

Add a private instance variable to the Player class that will hold the score (a positive integer value).

Provide a getter (property) and a setter method for this value.

**3.1.1. Success criteria**

* Correct use of private instance variable
* Use of properties to create a getter and setter
* Raising ValueError if someone attempts to set a non-positive value

**4. Sorting players**

**4.1. Task: Add unit tests for sorting players**

Add the following unit tests to the test\_player.py file:

def test\_sort\_players(self):

players = [Player("Alice", uid='01', score=10), Player("Bob", uid='02', score=5), Player("Charlie", uid='03', score=15)]

# note: ensure initialization code is valid for \*\*your\*\* implementation

# do \*\*not\*\* change the following code:

sorted\_players = sorted(players)

# players must be sorted by score as shown here:

manually\_sorted\_players = [Player("Bob", uid='02', score=5), Player("Alice", uid='01', score=10), Player("Charlie", uid='03', score=15)]

self.assertListEqual(sorted\_players, manually\_sorted\_players)

**Note:** If you have made other changes to the initializer of your player update the above code to reflect this change - you must not make any other changes to the test code above.

**4.2. Task: Interpret unit tests**

What was the outcome of running the above unit test, copy paste the output **for just this particular test** below:

Ran 1 test in 0.032s

FAILED (errors=1)

Error

Traceback (most recent call last):

File "C:\Users\liutin\source\repos\SRUS-TL-Games\test\test\_player.py", line 10, in test\_sort\_players

sorted\_players = sorted(players)

TypeError: '<' not supported between instances of 'Player' and 'Player'

**4.3. Success criteria**

* Unit test added to test\_player.py
* Unit test output provided
* Unit test output reflects the error in sorted(players) (if you are getting another error read the instructions CAREFULLY)

**4.3.1. Question**

The tests checks that calling sorted on a list of players will sort them by score, what is the **only** magic method that must be implemented in the player class for the sorted function to succeed?

\_\_lt\_\_

**4.3.2. Task: Implement the magic method in the Player class**

Add a test case to test\_player to test the comparison operator you are about to add - ensure you do not test a dunder method directly!

def test\_players\_can\_be\_compared\_by\_score(self):

# note: ensure initialization code is valid for \*\*your\*\* implementation

alice = Player("Alice", uid='01', score=10)

bob = Player("Bob", uid='02', score=5)

# Add the appropriate expression to the following assert test

self.assertTrue(...)

Run the test and confirm that your error resembles the previous error

Error

Traceback (most recent call last):

File "C:\Users\liutin\source\repos\SRUS-TL-Games\test\test\_player.py", line 21, in test\_players\_can\_be\_compared\_by\_score

self.assertTrue(bob < alice)

^^^^^^^^^^^

TypeError: '<' not supported between instances of 'Player' and 'Player'

Error

Traceback (most recent call last):

File "C:\Users\liutin\source\repos\SRUS-TL-Games\test\test\_player.py", line 10, in test\_sort\_players

sorted\_players = sorted(players)

TypeError: '<' not supported between instances of 'Player' and 'Player'Implement the appropriate magic method in the Player class and ensure you pass this test (and only this test!).

**4.3.3. Success criteria**

* Unit test added to test\_player.py
* Magic method implemented in Player class
* Initial Failed Unit test output provided
* Unit test runs successfully with submitted code
* Dunder method not employed directly

**4.3.4. Task: Are we sorted yet?**

Rerun test\_sort\_players does the test pass? If not, include the output below:

Failure

Traceback (most recent call last):

File "C:\Users\liutin\source\repos\SRUS-TL-Games\test\test\_player.py", line 13, in test\_sort\_players

self.assertListEqual(sorted\_players, manually\_sorted\_players)

~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

AssertionError: Lists differ: [Player: Bob, Score: 5, Player: Alice, Score: 10, Player: Charlie, Score: 15] != [Player: Bob, Score: 5, Player: Alice, Score: 10, Player: Charlie, Score: 15]

First differing element 0:

Player: Bob, Score: 5

Player: Bob, Score: 5

[Player: Bob, Score: 5, Player: Alice, Score: 10, Player: Charlie, Score: 15]

Why did the test fail (note: if it doesn't fail, it means there is something you have already done before you were asked to - you need to figure out what that is!)?

because we didn’t define “ \_\_eq\_\_ “, If the “ \_\_eq\_\_ “ method is not defined, the lists will not be considered equal.

Add the necessary code to the Player class to ensure that the test\_sort\_players test passes.

**4.3.5. Success criteria**

* Correct explanation of why test\_sort\_players failed/passed
* Correct implementation of the magic method in the Player class
* test\_sort\_players passes when run against the submitted code

**5. Implement a custom sorting algorithm**

The senior developer on your team believes that a custom sorting algorithm would be more efficient than the built-in sorted function (you grit your teeth, sigh, and realize you need this job!). They have asked you to implement a custom sorting algorithm that will sort a list of players by score.

To help you get started they have provided you with some example code that they wrote in their undergraduate days:

def sort\_quickly(arr):

if len(arr) <= 1:

return arr

pivot = arr[0]

left = []

right = []

for x in arr[1:]:

if x < pivot:

left.append(x)

else:

right.append(x)

return sort\_quickly(left) + [pivot] + sort\_quickly(right)

**5.1. Question: complexity**

What is the expected time and space complexity of the above algorithm? You can answer using big O or in plain English but in both cases you MUST justify your answer.

When choose an element as the pivot, in the best case expected time complexity is O(n log n), space complexity is O(log n)

in the worse case , expected time complexity is O(n2), space complexity is O(n)

**5.2. Task: Implement the custom sorting algorithm**

**5.2.1. Create a new method in the Player class**

Use the sample above (and its algorithm) as a starting point to implement a classmethod in the Player class that takes a list of players and returns a list of players sorted by score in **descending** order. Top scores come first!

**5.2.2. Create a test cases**

Add a separate test case to test\_player.py to test your custom sorting algorithm

Include your code below:

@classmethod  
def sort\_descending(cls, players):  
 if len(players) <= 1:  
 return players  
 pivot = players[0]  
 left = [p for p in players[1:] if p.score > pivot.score] # Higher scores go left  
 right = [p for p in players[1:] if p.score <= pivot.score] # Lower scores go right  
 return cls.sort\_descending(left) + [pivot] + cls.sort\_descending(right)

def test\_class\_sort\_descending(self):  
 players = [Player("Alice", uid='01', score=10), Player("Bob", uid='02', score=5),  
 Player("Charlie", uid='03', score=15)]  
 sorted\_players = Player.sort\_descending(players)  
 manually\_sorted\_players = [Player("Charlie", uid='03', score=15),Player("Alice", uid='01', score=10),  
 Player("Bob", uid='02', score=5)]  
 self.assertListEqual(sorted\_players, manually\_sorted\_players)

**5.2.3. Success criteria**

* Custom sorting algorithm implemented in the Player class as classmethod
* Custom sorting algorithm sorts in descending order
* Custom sorting algorithm compares players using their score (via the rich comparison operators)
* Custom sorting algorithm tested in test\_player.py and tests passed

**5.3. Test your custom sorting algorithm at scale**

The senior developer is impressed with your work and asks you to test your custom sorting algorithm with a list of 1000 players. They provide you with a script that will generate a list of 1000 players with random scores.

import random

from player import Player

players = [Player(f"Player {i}", uid=f"{i:03}", score=random.randint(0, 1000)) for i in range(1000)]

**5.3.1. Task: Create a test case to sort 1000 players**

Using the code above as a starting point, create a test case to test your custom sort algorithm - you can test it against the sorted function to ensure it is working correctly.

Include your test case below:

def test\_sort\_descending\_random(self):  
 players = [Player(f"Player {i}", uid=f"{i:03}", score=random.randint(0, 1000)) for i in range(1000)]  
 sorted\_players = Player.sort\_descending(players)  
  
 expected\_order = sorted(players, key=lambda p: p.score, reverse=True)  
 self.assertEqual(sorted\_players, expected\_order)

**5.3.2. Success criteria**

* Test case added to test\_player.py
* Test case sorts 1000 players correctly when compared to sorted function
* Test case passes when run against the submitted code

**5.3.3. Task: Testing sorting sorted players**

You had a scary thought - and decided to test your custom sorting algorithm against a list of players that are already sorted by score. You are worried that your algorithm might not be efficient in this case.

**5.3.4. Task: Create a test case to sort 1000 sorted players**

Create a test case that tries to sort 1000 players that are already sorted.

If you get a failure, include the failure below:

Error

Traceback (most recent call last):

File "F:\Ting\tafe\SRUS-TL-Games\test\test\_player.py", line 54, in test\_sort\_descending\_random\_already\_sorted\_list

sorted\_players = Player.sort\_descending(players.copy())

File "F:\Ting\tafe\SRUS-TL-Games\app\player.py", line 46, in sort\_descending

return cls.sort\_descending(left) + [pivot] + cls.sort\_descending(right)

~~~~~~~~~~~~~~~~~~~^^^^^^

File "F:\Ting\tafe\SRUS-TL-Games\app\player.py", line 46, in sort\_descending

return cls.sort\_descending(left) + [pivot] + cls.sort\_descending(right)

~~~~~~~~~~~~~~~~~~~^^^^^^

File "F:\Ting\tafe\SRUS-TL-Games\app\player.py", line 46, in sort\_descending

return cls.sort\_descending(left) + [pivot] + cls.sort\_descending(right)

~~~~~~~~~~~~~~~~~~~^^^^^^

[Previous line repeated 983 more times]

File "F:\Ting\tafe\SRUS-TL-Games\app\player.py", line 44, in sort\_descending

left = [p for p in players[1:] if p.score > pivot.score] # Higher scores go left

^^^^^^^

RecursionError: maximum recursion depth exceededProvide a reason why this test failed (if you got recursion errors, you need to explain **why** they occurred).

If your implementation did not fail, you must explain what changes you made to the original algorithm given by the senior developer to ensure that it did not fail.

Answer here

Propose a fix to your sorting algorithm that fixes this issue.

Choose a random element as pivot from the list, it helps to distribute the elements more evenly and reduce the chance of worst-case scenarios.

# Highlight what the fix was

**5.3.5. Success criteria**

* Test case added to test\_player.py
* Test case passes only when changes above are added

**6. Task: Authenticity of in class work**

Complete the following snippet before you submit:

I, <name and student number>, completed this work in class <room number>, on <date>, under the supervision of <assessor's name>.

Or (if not completed in class):

I, <name and student number>, completed this work outside of the scheduled hours. I emailed <assessors name>, on <date>, along with my documented reason for non-attendance, and have scheduled a time to meet to discuss my work.

I understand that until I meet my assessor to confirm that this work is a valid and true representation of my abilities to write and debug a sorting algorithm in Python, this submission cannot be considered complete.

**7. Submit your work**

* Ensure all tasks are complete and tests pass
* Answer all questions in your own words
* Complete the statement of authenticity
* Include .git showing each task committed (you must show at least 5 commits)
* Tag your last commit as por3-finish
* Push your changes to your GitHub repository
* Submit a zip of your repository to the LMS (ensure you do not add the .venv or \_\_pycache\_\_ folders)

End of assessment task