Question 4: Collatz Conjecture

Reflection: Why is the Collatz Conjecture hard for both humans and computers to prove?
The Collatz Conjecture is one of the hardest problems in math because of the way it is structured and acts. The rules are very easy: if the number is even, divide by 2; if it's odd, use the method 3 times the number and plus 1. However, we cannot see if the number will eventually hit 1. Some of the numbers will decrease very fast, while others increase before dropping, which makes it harder to set an established idea. Normal methods of proofs like induction and modular numbers will not work because the changes in series are so random. Despite computer tests they still show the idea will work for numbers up to 2^68, but that does not mean it will be true for all of them. This topic is connected to a lot of chaos theory and just general undecidability. Which means we might need new math concepts to prove something, or it may just be impossible to prove with mathematics. The Collatz Conjecture is a difficult problem in number theory as well because no one has been able to prove and find a counter example to it. This conjecture highlights that there are very complicated math problems that appear very simple at first glance.

Approach: To explore this conjecture, I made a program to compute and analyze the sequence length for numbers up to a given range. The lengths were stored in a ADT to identify the numbers with the longest sequence, and also tracked the highest number encountered in each sequence using a variable that was updated during computations.

Optimizations: I tried to save computations by using a dictionary (cache) that was applied to store the previously computed sequence lengths. This helped my program to reuse results rather than just calculating them, which improved efficiency. Also by iterating only one time over the range and updating the stored results dynamically I minimized redundant calculations and reduced execution time. I also used sorting techniques to identify the top ten longest sequences.

Reflections: This made me realize that the difficulty of this conjecture arose from the exponential growth of numbers in a certain sequences, that required a lot of overall memory and computations. For humans the unpredictability of the progression makes it way harder to establish a pattern. After coding this I began to not only see the fascination of this all, but also the frustration.

Question 5: Decorator Extra Credit

1) What is a higher-order function and how is it different from a functor?
A higher order function is a function that either: takes another function as an argument or returns a function as a result itself. Python allows passing functions as arguments that enable functional programming techniques. A functor is a data structure that can be mapped over (Ex map() in python). This means it applies a function to every element inside it without changing its structure. Higher-order function deal with functions while functors operate on data structures:

```
#Example of a higher order function
def apply_twice(func, x):
   return func(func,x))
```

2)What are first-class objects? What is the significance of functions being first-class objects?
First class objects are entities that can be assigned to variables, passed as arguments, and returned from functions. In python specifically functions are first class objects , meaning they can be stored, modified and used dynamically. The significance of this is that they enable higher-order functions, support functional

programming, and allow decorators and callbacks. An example would be a function being assigned to a variable and printed as the variable.

3)What are inner functions? Why are they important for decorators?
An inner function is a function that is defined within another function(like def_wraps from above). It remains purely local to the function it is inside and is not accessible by anything else. They are important for decorators because it provides encapsulation so they are not accessible outside of their function, making sure there is better control overall. They also retain access to the enclosed function's variables even after the function has been executed. An example of this is the def_wraps from above.

4)Benefits and drawbacks of using python decorators
Benefits
-Code Reusability: Adds functionality through various functions without code duplications
-Cleaner Syntax: uses the @dectorator instead of having to manually wrap everything
-Automatic execution: Useful for logging, authentication, and tracking performance
Drawbacks
-Debugging: Errors within decorators can be way harder to trace
-Reduced Readability: Excessive decorators can make code harder to understand
-Preformence: Adds an extra layer of execution which can impact the overall performance

5)Why is @ called syntactic sugar? What's the advantage of using it for decorators?
The @ symbol is called syntactic sugar because it makes syntax way simpler for applying decorators without impacts on code functionality. The advantages are similar to question 4 but it helps the readability by making it explicit that a function is being modified, prevents manual wrapping, and has consistent behavior across a program. An example would be calling a function def say_hello(), that just prints hello. Without the decorator we would need to declare it like: say_hello = my_decorator(say_hello), but without it you just need to @my_dectorator before the given function.

6)How does the @property decorator help with Python's weak encapsulation?
Python lacks strict encapsulation overall. There are really no truly private variables. @property decorator provides a way to control the privacy and access of class attributes using a simple syntax. The benefits are that it prevents direct modification of attributes, and just improves overall data integrity by having validation rules. An example would be:

```
class Person:
   def__init__(self,name):
       self._name = name #private

@property
   def name(self):
       return self._name #read only
```

Used Sources in the section that were provided as well as :
Python Software Foundation - PEP 318: Decorators for Functions and Methods
https://peps.python.org/pep-0318/
Provides an official explanation of decorators and their application in Python.

Python Documentation - Function Objects

https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy
Covers first-class objects in Python and their significance.

GeeksforGeeks - Python Decorators
https://www.geeksforgeeks.org/decorators-in-python/
Provides an overview of inner functions and their role in decorators.hhhh