**ASSESSMENT 1:**

**Programming Tasks**

**COMP3002**

**Alternative Paradigms**

**2022/2023**

Table of Contents

[Q1. Recursion and Dynamic Programming (in Python) 2](#_Toc130796321)

[Q2. Randomized Algorithms 3](#_Toc130796322)

[Q3. Functional Programming (in Python) 4](#_Toc130796323)

# Q1. Recursion and Dynamic Programming (in Python)

**a. Explain, using the data-structure of a graph as an example, what is meant by a recursive data type and describe three others.**

A recursive data type can be defined as a data type that is defined by itself or can contain itself. Recursive data types are described as self-referential. Graphs are an example of this wherein nodes are connected to each other via one or more edges. Graphs are recursive because they are formed of graphs that connect to other graphs. For example, there is G1 and G2 and they can be connected by an edge via one or many nodes inside each graph. G1 and G2 being connected creates a larger graph G3. It can therefore be determined that G3 contains references to other graphs (such as G1 and G2) within itself. Essentially, nodes within a graph have targets to other nodes via which they are connected.

Linked lists are another recursive data type wherein a node within a list has both a value and a node that points to the next node in the list – this gives linked lists an order.

Binary trees are also recursive are they are made from nodes that point to either an empty tree or two children which are also binary trees. One way in which trees differ from graphs is that they have a starting point at the root of the graph.

Sets are also recursive each element of a set is a set itself – sets are subsets of another set. Put simply, sets can contain other sets and are therefore recursive.

**b. Explain what is meant by single and multiple function recursion and give examples using the factorial function and the Fibonacci numbers.**

Single function recursion is a classification of recursion wherein there can only be one possible self-reference. An example of this is the factorial function. The factorial function is an algorithm upon which you start with an integer N and you multiply N with N-1. This process is repeated with N repeatedly getting smaller each time, and will terminate when the condition of N becoming 0 is met. The factorial function is therefore the process of multiplying a selected integer and all subsequent whole numbers down to 1 together. This example is single-function recursion because the function only references itself as N once.

Multiple function recursion means that there can be more than one self-reference. An example of this is the Fibonacci sequence. This sequence follows the rule wherein the next number in the sequence can be found by adding the two subsequent numbers in the sequence. For example, the first 9 numbers in this sequence are:

0, 1, 1, 2, 3, 5, 8, 13, 21

This sequence can be implemented as an algorithm that is classified as multiple-function recursion as each time the algorithm is called it will need to find the value of the node one position behind as well as the value of the node two positions behind to calculate the next value within the sequence. The sequence can be stored as a list.

**c. Explain what is meant by tail recursion and why it theoretically allows to write more efficient code as compared to general recursion. Why does tail recursion not work efficiently in Python?**

Tail recursion is defined as recursion wherein the last statement executed by the function is the recursive call. Normal recursion methods are theoretically less efficient and dangerous because each level of recursion is creating a stack frame on the call stack, which for large recursive problems can cause a stack overflow and result in the execution being terminated. These stack frames are created because the recursive call occurs whilst there are still other statements within that function that need to be executed so the stack frame keeps track of where that function stopped executing. In a non-tail recursive function, the code has to recurse until the base case and then go back to finish each function.

For tail recursion, however, the recursive call is the last statement within each function so upon hitting the base case, all functions terminate. There is no need to add to the stack – instead the current stack frame is just replaced. Therefore, there is minimal use of the call stack which makes tail recursion theoretically more efficient, as this overall means that less memory is being used.

However, within Python the process described above does not happen and tail recursive calls are still added to the stack frame as within Python the compiler does not optimize the process. Overall, within Python these additional stack frames are not eliminated – the Python compiler is not optimized to automatically remove them if tail recursion is detected.

**d. Explain what is meant by a divide-and-conquer algorithm. Explain in detail how one could use the divide-and-conquer principle to sort a set of integer numbers by the product of their digits. You don’t need to provide code.**

A divide-and-conquer algorithm is an algorithm that is used to recursively break down a primary problem into a two or more subsets of sub-problems with some relation. These sub problems will be broken down into even simpler problems until eventually they can’t be broken down anymore, upon which they are solved. Combining the solutions of each sub-problem then allows for the desired output for the initial primary problem to be found.

For the supplied problem, a divide-and-conquer algorithm might recognise that there are different patterns that occur within the problem, such as the task requiring multiplications and sorting.

For multiplication of digits, a dvide-and-conquer algorithm could be implemented so that the total number of multiplications is reduced by recognising repetitions in multiplications, for example:

S = { 23 , 132}

Set S can be broken down into the following multiplications: (2\*3) and (1\*3\*2) and these multiplications can be further expressed as (2\*3) and (1\*3) and (1\*2) and (3\*2) wherein the algorithm could recognise that 2\*3 and 3\*2 are the same and this multiplication only needs to be calculated once for both expressions.

Likewise, if set S was larger and had 8 elements, a common search algorithm (such as merge sort) could be used to repeatedly half the set to create smaller sets until the set has only 1 element. These set will then be merged again but each merge will determine whether the first set is smaller or larger than the second. This algorithm will call itself recursively until there is one final set again, but this time it should be arranged, which in this case would go from lowest value to highest value integer.

**e. Explain in detail how a general recursive function can be processed using a run-time stack.**

A general recursive function can be described as a function wherein the recursive call is executed before the last statement of the function is reached. Therefore, this means that the function will recess before fully executing all of the function’s statements and after processing the recursive call will need to return to compute the statements that occur after the call.

For example, once the algorithm has hit the base case, it will need to return to the layer of recursion from which it was just called and compute the trailing statements. Within this process, the run-time stack is needed so that the program can keep track of where it stopped processing that function. Each time an additional layer of recursion is called, a new stack frame is created. The stack will continually grow in memory until the base case is reached, upon which each frame will be read for the program to get the information needed to resume processing the rest of the function, and the frame will be popped from the stack. Overall, the stack stores the context of all of the paused functions – these functions are paused because a recursive call has been made to the function and the program must then start computing the new function.

**f. Write a Python function to calculate the greatest common divisor of a list of numbers. Avoid loops if you can. Explain in detail how your solution works.**

Text

Description automatically generated

This function has been implemented to work for any number length of a list. The function has one parameter which is the list of numbers. In the use case example where the function is only provided with a list of length 1 (meaning that there is only one integer in the list), the function returns the value that is given, as the GCD of a single number is the number itself.

The function will check if the list has exactly two integers, upon which these values are stored within x and y variables. If the second value, which is assigned to Y, is equal to 0 then the value of X will be returned as it is the GCD value. This if statement is the base case which will stop the recursion.

Within the else that precedes the base case, a recursive call is made wherein the GCD function calls itself, with the Y variable as its first argument and the result of X modulus Y as its second argument. This calculation of X mod Y provides a remainder value. Upon calling itself, when the base case checks if the Y value is equal to zero it is actually checking whether there is a remainder or not. In the case where there is not a remainder and Y == 0, the function can end because the GCD has been found.

In the instance that there are more than two integers within the list, the function gets more complex. Within the else that establishes that there are more than two numbers, the GCD function is called with the first element of the number list as the first argument, and a recursive call to the GCD function itself as a second argument. This recursive call takes the numbers list as an argument, except it does not include the first element of the list. Therefore, within each subsequent level of recursion the numbers list will thin out and become smaller until there are two numbers left, upon which it will execute the code for if the length of numbers is equal to 2. The function will return the GCD of those two values, which will then be used to calculate the GCD of this returned value and the integer from the list that preceded the two values that were just compared.

An example of finding the GCD of a list of three numbers is provided below:

Original list: [6, 9, 15]

Step 1: List now becomes [9, 15] so calculate GCD of 9 and 15 which is 3

Step 2: GVC is called with [6, 3] as numbers argument.

Step 3: GVC of [6, 3] is 3 and function is broken from. Final returned GVC value is 3.

Essentially, this function works by comparing GVC’s in pairs and then comparing the GVC values of these pairs until one final pair remains, and then the final GVC value is found.

**g. Write a Python class that finds the longest common substring of a set of strings. Use memoization. Use comments in the code to explain how your code works.**

# **Q2. Randomized Algorithms**

**a. Assume you have a biased random bit generator (in Python) called biased\_bits that when called returns bit 1 with probably 0 < p < 0.5 and a 0-bit else. Using biased\_bits, write a function random\_float, that generates floating point numbers equally distributed on the interval [0,1[ . Explain why and how your solution works.**

**b. Write a function that returns a random node of a linked list such that any node has the same probability of being returned.**

**c. Write a function that returns a random node of a tree (not necessarily binary) such that any node has the same probability of being returned.**

# Q3. Functional Programming (in Python)

**a. Explain the main idea behind functional programming by contrasting it with Object-Orientated Programming, as well as its advantages and disadvantages.**

Functional programming is a paradigm where all code is written inside of pure functions. These functions are designed to be solely dependent on the input parameters of the function, and the operation of this function have no external impact. These functions can however return values, which may then be used externally. Functional programming avoids unwanted side effects of having variables changing values, because within FP variables never change once defined – rather new ones are created as outputs. FP therefore ensures immutability.

On the other hand, OOP is a paradigm wherein code is represented by objects, which have attributes and methods. Methods differ from pure functions as they are often used to change the state of an object – state is manipulated via the object’s attributes. Therefore, whilst FP does not modify external values, OOP does.

Pure functions are very simple as they retrieve inputs, process them, and return an output value. The same output will always be generated for the same input arguments. As these functions are independent of state they can provide the advantage of being highly readable and comprehensible as well as being easy to test. Inputs simply need to be fed into the argument and the outputs can be measured. Pure functions are regularly used because of their simplicity and directness.

Object oriented methods are more complex than pure functions in the sense that they are often used to manipulate data. Pure functions however can be implemented within OOP – making OOP very versatile. In contrast to FP where functions don’t have side effects, OOP methods can result in many side effects that are hard to debug. As previously mentioned, because of FP’s simplicity it makes it very easy to debug code.

Overall, FP is all about performing an activity, whilst OOP is concerned with adjusting data and state. This also makes FP safer to use and test, but less versatile and more direct/technical.

**b. Explain using Python examples what higher order functions are and what “functions are treated as first-class citizens” means.**

A higher order function is a function that takes one or more functions as an argument, or returns the function as an output. Higher order functions have many usages.

One usage is to assign a function to a variable. Doing so within Python will not call the function, however it will create a reference to the function. Using this reference, the original function can then be called. Another usage is that functions can take another function as a parameter.

First-class citizens describe a function as capable of being passed as an argument to another function, returning them as the value of another function, and being able to get assigned to a variable or stored within a data structure. Function names are treated as if they are variables if they are First class citizens.

**c. Explain the concepts, commonalities and differences of generators, iterators and closures using Python examples.**

Iterators within FP are objects that represent a stream of data, such as: lists, tuple, dictionaries, and files. The iterator object is used to return a single element at a time, and it does this via the \_\_next\_\_() method. This method will simply return the next element within the stream, until it reaches the end and raises an exception that specifies that there is no next element, and the stream therefore cannot be iterated over anymore. Iterators also have an \_\_iter\_\_() method that is used to return the object itself. This is used in conjunction with for loops etc.

Within an iterator you can only proceed forward and cannot return to a previous iteration as the iterator only specifies a the \_\_next\_\_() and \_\_iter\_\_() methods.

A generator is type of function that returns an iterator (which will store a sequence of values as described above). However, in Python generator functions are written using the yield statement, rather than the return statement that Iterator objects use. The yield statement allows generators to return a sequence of values as yield can be used to return multiple values without terminating the function, unlike the return statement. The yield statement works by temporarily suspending the functions execution and sending a value back to the caller, except the yield statement will then carry on executing the rest of the function, restarting from where it stopped. Multiple yields can be called within a function.

Generators are considered to be advantageous over iterators because they are quicker to write and simpler, especially as they are a function, and an iterator is a class. Another advantage of generators is that generators will dynamically generate their values on demand and one at a time – this is considered more memory efficient as the entire sequence is not being stored in memory. This makes generators theoretically more desirable when ‘iterating’ over a large sequence of values. The term iterating is used here as generators are defined as an iterator.

Generators could be used to, for example, create a sequence of numbers that follow a pattern. This pattern could be all even numbers up until 100 or every multiple of 5 for a specific range.

Overall, the main difference between a generator and an iterator is that the former does not store the values, whereas an iterator does.

Within Python, a nested function is a function with another function defined inside of it. These inner functions can access variables within the same scope as the outer function. Closures build upon this concept wherein they are function objects that can remember values in enclosing scopes. Closures allow functions to access the closure’s copy of values or references.

Closures are used as call-back functions and are generally used to reduce the usage of global variables and provide data-hiding.

**d. Write a function called ‘compress\_keys’ that takes a dictionary with string keys and returns a new dictionary with the vowels removed from the keys. The function should use a list comprehension nested inside a dict comprehension.**

Text

Description automatically generated

**e. Write a Python function that maps a lambda expression for the square-root function to a list of integers.**

Text

Description automatically generated

**f. Write a Python decorator that wraps the function from part e) and checks if any of the arguments in the list is smaller or equal to zero. If so, an error message should be printed. This should use a comprehension.**

**g. Explain the concept of Monads as well as their advantages.**

A side effect is defined as when a function modifies or relies upon something outside of its parameters. However, FP is all about writing pure functions that do not have side effects. Put simply, monads are a component of Functional Programming where they enable side effect behaviour to be computed, ensuring that functions can remain pure.

Side effects that monads might compute are state and IO, such as when printing to the console. Sometimes the computations within functions are not as pure as desired and can: result in errors, or even come after determining a state, of which the specific computation is reliant on.

Monads are introduced into FP as a theoretical container that separates these side effects from the actual pure computations. Monads provide a way to model or describe behaviour without affecting the computation. An example when to use monads is when a function might need to access a value that may not exist or be known yet.

Monads are beneficial because they abstract the details of computations and make the computations more readable – developers can easily look at functions and tell what they are doing. Monads keep functions pure because the side effects are wrapped into their own wrapper functions. Within FP, monads are a structural tool that can be used to simplify complex sequences of functions.