CODE FIRST GIRLS

CFGDEGREE

SOFTWARE ASSESSMENT MATERIAL RELEASE

THEORY QUESTIONS

SECTION	MARK
1. Theory Questions	26
2. Coding Questions	24
3. Theory Challenge	25
4. Coding Challenge	25
TOTAL	100

Important notes:

- This document shares the first section of the Software Assessment which is composed of 6 Software Theory Questions
- It is worth just over a quarter of your assessment mark
- You have 24 hours before the assessment to prepare.
- If any plagiarism is found in how you choose to answer a question you will receive a 0 and the instance will be recorded. Consequences will occur if this is a repeated offence. You can remind yourself of the plagiarism policy here.
- You are allowed to use any online images to support your answers.

Section 1: Theory Questions [26 points]

Please note you will **not** need to provide any code for these answers.

1.1 The deque module is part of which python library?	**1 point**

Deque is part of the Collections Python library.

1.2 What are 2 differences that distinguish a tree from a graph?	**2 points**

A graph does not always have root nodes, and a graph has a wider range of relationships such as cyclic or acyclic. A tree however, typically has a root node and has a parent child relationship.

A graph can have loops, meaning you can traverse through edges and end up at the same node (cyclic). Whereas a tree does not include cycles or loops and only has one path between two nodes (acylic).

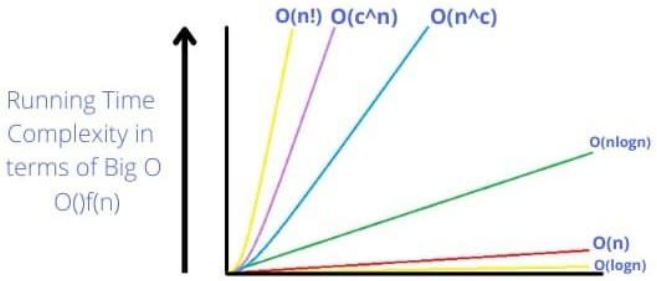
1.3 Give the definitions of time complexity and space complexity	**2 points**

Space complexity by definition is the total space or memory taken by an algorithm, with respect to the input size. It includes both the input size and auxiliary space. Auxiliary space is temporary memory used when functions are called, returned when the program is being executed. Space complexity is measured using big O notations, as is time complexity.

Time complexity is the measure of how long an algorithm takes to execute as the input grows. Using big O, you can predict how long it will take for your code to implement based on input size. You can also predict how long it would take based on bigger input sizes as you scale.

Time and space complexities are used to predict the performance of algorithms. In addition, big O notations are used to predict the worst case scenario and are not an exact measurement of usage space or time.

You can use the following graph to assess time complexity:



O(n!), O(c^n), O(n^c), Worst
O(nlogn - Bad)
O(n) - Fair
O(logn)- Good
O(1) - Best

Taken from: https://flexiple.com/algorithms/big-o-notation-cheat-sheet/.

1.4 Describe the bubble sort algorithm and its complexity. What is guaranteed at the end of the first pass?	**5 points**

Bubble sort is an algorithm based on comparisons. The algorithm will start at the beginning of the list and repeatedly compare and swap adjacent elements if they are in the wrong order in a list until all elements are in order. The algorithm will place the smaller element to the left and the higher element is moved to the right.

Bubble sorts have a time complexity of O(n2) in the worst case. N stands for the number of elements. If an array had 7 elements, you would have to do n-1 passes through the list in the case of the list being unsorted. So 7 -1 = 6 comparisons.

The best case scenario would be if the list was already sorted, therefore no swaps are needed, then the bubble sort would have a time complexity of linear O(n).

Bubble sort has a space complexity of O(1). This means it only needs a fixed or constant amount of extra space (auxiliary space) for flags and temporary variables.

Bubble sort is not recommended for larger data sets and is typically used for smaller lists.

At the end of the first pass, imagine a bottle of coke and the bubbles rising to the top, similarly the largest element in a list will move to its correct position at the end.

The largest element of an unsorted list will ‘bubble up’ to the end if sorting in an ascending order. If you are sorting in a descending order, the largest element will instead ‘bubble down’ to the beginning of the list. The remainder of the list will remain unsorted.

To conclude, bubble sort algorithms are efficient when used for smaller lists but become inefficient when dealing with larger data sets.

1.5 Explain what LIFO and FIFO are and how each works in practice with a named data structure	**8 points**

LIFO stands for last in, first out. The principle can be seen in python using lists in a stack data structure.

Imagine stacking a pile of books, the last one to be added to the stack will be the first one to be removed. Similarly, in a python list the last element added to a list will also be the first to be removed. You can use the append() method to add elements to the stack and pop() method to remove the last element.

This principle is popularly used for expression evaluation for mathematics expressions, operands such as + can be pushed onto a stack and then the calculations can be performed in order (reference: (taken from https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/), also to implement undo/redo functions.

FIFO stands for first in, first out, just like if you were queuing in a shop.

In the context of Python and queue data structures, the first element that is added to a queue is the first to be removed.

This principle is effective for controlling the order of data processing and is commonly used when scheduling tasks and request handling to ensure fairness of execution.

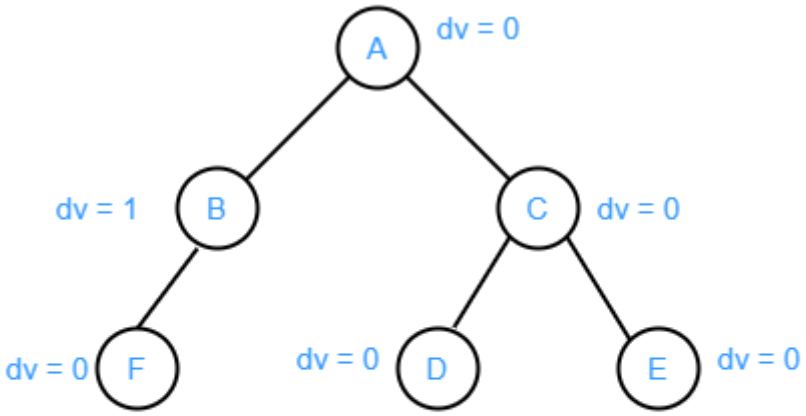
1.6 What is a Balanced Binary Tree and what would be the best root? Walkthrough how you search using this structure.	**8 points**

A balanced binary tree means that the difference between the height of the left and right subtree nodes differ by at most one. The balanced nature of the tree allows for logarithmic search in relation to the number of nodes. The worst case space time complexity of a balanced binary tree is usually root(n) or log(n).

The best way to search this structure would be to start at the root node then compare the value of the current node with the one you are searching for, if this is the correct node the search stops, If not, you then compare the left and right node. You would move to the left if the current value of the node is less than your searching value and right if the value is greater. This process is repeated to either the left of the right with comparison of values until you find your value.

A binary tree is an effective and quick way of searching, inserting or deleting elements due to its shallow nature. Examples of balanced binary trees are the AVL tree, b trees and red black tree. The fact that they are balanced makes them more efficient for data organisation in the memory and have a better time complexity.

Structure example (reference https://www.shiksha.com/online-courses/articles/about-balanced-binary-tree/) :



Balanced Binary Search Tree