



**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 [h ere](https://drive.google.com/file/d/1k9UaGOR7hx54QRZ8jvp2jtC4P-8_Rs4F/view?usp=sharing).
* You are **not** allowed to use any online images to support your answers.

**Section 1: Theory Questions [26 points]**

**1 point**

* + **The deque module is part of which python library?**
  + **collctions**

**2 points**

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

**2 points**

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

**5 points**

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

**8 points**

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

**8 points**

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

* 1. Collections
  2. - A graph has vertices as well as nodes and edges, while a tree has just nodes and edges, and one unique root (not present in a graph).   
     - A tree cannot have cycles, while graph can have cycles.
  3. Time complexity: the time it will take the algorithm to finish as a function of an input size   
     Space complexity: specifies how much space is needed for an algorithm to run as a function of an input size
  4. Bubble sort makes multiple passes through an array multiple times, compares pairs of adjacent elements and swaps them if they are out of order. After the first pass it is guaranteed that the highest number will be in place and with each next pass, the next largest element in a list “bubbles” up and gets placed in its proper place. In the worst-case scenario, the function will need to traverse the array n-1 times (where n = number of elements in the array and minus 1 because it works with pairs) to sort the whole list, and for each of this times also perform a comparison of the two adjacent elements, which gives the function complexity of O(n2). When swapping the two elements, some programs will require creating a temporary variable to prevent overriding, thus the space complexity for this algorithm is O(1). Bubble sort, in general, is considered the least efficient sort algorithms, because of the need to go over the array n-1 times and continuously swapping the items which is a costly operation, this can however be improved by adding logic that will stop the algorithm if no swaps are made (short bubble), making it a good option for checking if the list is already partially sorted or is not very large.
  5. LIFO (Last In First Out) and FIFO (First In Last Out) both refer to the way data is added and removed/accessed in the data structure.  
     LIFO appears in **stacks**, a structure where the last item added is also the first one to be retrieved/removed. This structure is akin to stacks of books or plates, where we would grab the first item from its top and add a new one on its top as well. It is used e.g., to store browser history, when creating undo and redo functionality in a software etc.   
     FIFO on the other hand appears in **queues**, where the first added item will be also the first one to retrieve/remove. Used e.g., in waiting lists for booking systems such as for gym classes or concert tickets, managing call center incoming call queues, payments or to manage internet traffic trying to reach servers.
  6. Balanced binary tree is a tree in which the difference in height between its subtrees is not more than one and each of the parent nodes has up to two children. Its best root is its middle (or median) value – this way the tree becomes as balanced as can be.   
     E.g. taking a list of following values: 5, 16, 3, 8, 12, 10, 20 to build an efficient binary tree we would start with 10 (middle value) and then assign values below 10 (5, 3, 8) to the left subtree and above 10 (16, 12, 20) the right subtree and repeat this process recursively for the other values, which would result in a tree looking like so:

To find value 12, e.g., we would then first compare the value we are looking for to the root, which is 10. Since 12 is greater than 12 we will disregard the left subtree and check the next node in the subtree on the right, which is 16. Since 16 is greater than 12, we will then move to the left of this subtree and compare the value we are looking for with the value in this node, which in this case is 12, thus completing the search in 3 steps.