University of London

Assessment Coversheet

Complete this coversheet and read the instructions below carefully.

Candidate Number: LX0549
Refer to your Admission Notice

Degree Title:
BSc Computer Science

Course/Module Title:
Algorithms & Data Structures II

Course/Module Code:
CM2035

Enter the numbers, and sub-sections, of the questions in the order in which you have attempted them:
Question 2 a b c d e f g
Question 3 a b c d e f g

Date: September 15, 2021

Instructions to Candidates

1. Complete this coversheet and begin typing your answers on the page below, or, submit the coversheet with your handwritten answers (where handwritten answers are permitted or required as part of your online timed assessment).
2. Clearly state the question number, and any sub-sections, at the beginning of each answer and also note them in the space provided above.
3. For typed answers, use a plain font such as Arial or Calibri and font size 11 or larger.
4. Where permission has been given in advance, handwritten answers (including diagrams or mathematical formulae) must be done on light coloured paper using blue or black ink.
5. Reference your diagrams in your typed answers. Label diagrams clearly.

The Examiners will attach great importance to legibility, accuracy and clarity of expression.

Begin your answers on this page

Question 2

(a) The worst case inputs for the arrays A and B are two fully filled arrays of length N [1,1,1,1,….,1] as these would cause the most executions of both the while and for loop to carry over numbers as the summation is done. The worst-case time complexity is Θ(n) as the number of executions of the loops grow linearly with the size of the input, and the execution of the second loop just extends the runtime linearly as well.

(b) The worst-case time complexity of the function Multiply is $\Theta(n^2)$ as there is a nested loop that executes the outer and inner loop in relation to N, so for each execution of the outer loop, the inner loop is executed another N times, making the runtime grow exponentially with the size of N.

(c) Because we are observing bit arrays, we need to inspect every element of the arrays to perform the summation. Hence, it is not possible for the execution to become more efficient with additional elements in the array, limiting the running time to be at best linearly correlated with the array size. A better runtime, for example O(log n) would require the runtime to decrease as we make the input arrays larger. However, the only way to sum up two bit arrays is to use the “carry” method, which goes through the arrays bit-by-bit.

(d) All instructions until the first for loop can be executed in constant time. The first for loop is limited in executions to n/2 times, as it executed between x and h. x is always 0 or 1, and h is always at most n/2. Hence the worst case execution of this loop is n/2.
The second loop will execute over the second half of the numbers, between h and N. Since h is at least n/2, and N is N, this loop will execute n/2 times.

So the first part executed at O(n), however the function is recursive and calls itself on subarrays. Since these subarrays get smaller and smaller with each execution, the running time in relation to N is Θ(log n).

(e) I agree. The only better time complexity would be O(1) which is not achievable given the variable size of N and the need to divide and conquer the arrays bit by bit.

Question 3

(a) It returns the value at A[3] = 6.

(b) The algorithms create an array with the contents of the BST sorted in ascending order and returns the value at the midpoint of the resulting sorted array (assuming our binary tree is not empty). However Algorithm 1 doesn't perform the sorting step since the BST already has implicit sorting.

(c) The algorithm consists of two parts: A breadth-first traversal of the BST which has time complexity O(n), and an insertion sort algorithm, which has time complexity O(n^2). Hence the overall time complexity of A2 is O(n^2).

(d) The worst-case time complexity of A1 is O(n) as it only performs a depth-first traversal of the BST with no additional sorting.

(e) Given that we have a proper BST, we could just use a standard depth-first in-order traversal using recursion and get rid of the stack. This means we would traverse the nodes of the tree and add them to an array, which would result in a sorted array.

(f) I would use a helper function to achieve this, assuming I can pass the array by reference for manipulation.

```
function InOrder(T, A, i)

     if !IsEmpty(T) then

     InOrder(Left(T))

     A[i++] = VISIT(ROOT(T))

     InOrder(Right(T))

     end if

end function


function A1(root, N, T)
     A = new Array(N) of zeroes
     i = 0
     InOrder(T, A, i)
     return A[n/2]
end function
```

(g) Assuming the task is to order the array and then return the midpoint, we could use a more efficient sorting algorithm such as binary search to achieve a time complexity of O(log n).