# UoL logoUniversity of London

# Assessment Coversheet

Complete this coversheet and read the instructions below carefully.

**Candidate Number**:

WP1282

**Degree Title**:

BSc Computer Science

**Course/Module Title**:

Algorithms and data structures I

**Course/Module Code:**

CM1035

**Enter the numbers, and sub-sections, of the questions in the order in which you have attempted them:**

**Question 1: a, b, c, d, e, f, g**

**Question 2: a – i, ii , b, c – i, ii, d – i, ii**

**Date**: 18.03.2021

**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 1:**

1. **iii.** For this recursion to work, line 5 should be: return n \* Factorial(n-1).
2. **n + Sum(n-1)**.
3. L <- 1, R <- 7. L(1) ≤ R(7) => Vector is not empty => M <- floor((L+R)/2) = 4, we compare V[M] and item=> V[4] = 7, item = 8 => item > median => if item is in this vector, it must be to sub vector to the right of the current M => we shift our left boundary => L <- M+1 = 4+1 = 5. Now we repeat the process for this sub vector: L(5) ≤ R(7) => sub vector is not empty => new M <- floor((5+7)/2) = floor(6) = 6. Again, compare item against median: V[6] = 9, item = 8 => item < median => if item is there, it must be in the sub vector to the left of the median => shift right boundary => R <- M – 1 = 5. Again, repeat the process: L(5) ≤ R(5) => M <- floor((5+5)/2) = 5 => compare V[M] and item: V[5] = 8, item = 8 => item = V[M] => return the position of this element => **return 5**.

1. **Worst-case time complexity – O(log(n))** [When item is not in the vector]. **Best-case time complexity – O(1)** [When item is in the position of the initial median].
2. These two concepts are similar, but not equal. While both use relation between the bigger problem and smaller problem, they do it differently. Decrease and conquer algorithm uses the fact that solution to the bigger problem is tied to the solution of the basic case of the problem, so we need only to solve one instance of the problem and this solution is a key to the initial bigger problem. For example, binary search is an example of decrease and conquer algorithm, we decrease our vector, leaving some of the sub vectors unattended. And our search doesn’t require us to check all sub vectors. Divide and conquer algorithms uses the fact that solution to the bigger problem is equal to the combination of these “small” solutions. Merge sort is a good example of the Divide and conquer algorithm, that is similar to the binary search. But to sort initial vector, we need to divide it into “simple” instances and then sort them all between them (solution to the initial sorting is the combination of solutions of the smaller sorts). So, to use **Divide and conquer** we need to **solve multiple simple “subproblems”, to combine their results**, while in **Decrease and conquer** there is **only one simple problem** (base case), for which we need to find the solution.

1. 1: function BinarySearch(vector, item, left, right)

2: if (left ≤ right) then

3: M <- floor((left+right)/2)

4: if vector[M] > item then

5: right <- M – 1

6: else if vector[M] < item then

7: left <- M + 1

8: else

9: return TRUE

10: end if

11: BinarySearch(vector, item, left, right)

12: end if

13: return FALSE

14: end function

1. As we previously mentioned (Q1-d) the worst case for the binary search algorithm is when the item is not in the vector. Even when we use recursion, **worst-case time complexity of the binary search is O(log(n))**, because each recursive step will halve the previous vector in half and for each recursive step there are fixed number of instructions, that doesn’t depend on the size of n. So, to split initial vector of length n and check that our item not in this vector, we will need floor(log2(n)) + 1 steps.

**QUESTION 2:**

1. i. At the end of line 7, the value of **x = 3**.

ii. HEAD --> [ 5| -]-> [ 2| -]-> [ 3| -]-> NULL

^*tail pointer*

1. In vectors and dynamic arrays, we can access each element using appropriate index, but in the queues and vectors **we cannot access every element, but only specific ones** (head element for a queue and top element for a stack). So, if we want to check different element of a queue (or a stack) we need to free this specific area, so another element will take this place and become accessible. This creates problem of the data loss, because if we dequeue element from a queue, we will lose it, unless we use auxiliary procedure, to preserve dequeued values. But this is not a direct application of the linear search.
2. i. 1: function SearchQueue(Queue, x)

2: while EMPTY[Queue] = FALSE do

3: if HEAD[Queue] = x then

4: return TRUE

5: else

6: DEQUEUE[Queue]

7: end while

8: return FALSE

9: end function

ii. The problem is that if initial head of the Queue is not equal x, then **this algorithm will lead to a loss of data**, because dequeued elements will go nowhere. To **prevent this** from happening we can **enqueue values before dequeuing** them, but to **prevent from infinite cycle** (line 2) we need also enqueue some element that will work as a stop signal, and change the condition in the while loop to (EMPTY[Queue] = FALSE AND HEAD[Queue] != “Stop signal value”). This way, we will keep our initial data intact, we just need to add a simple function, that will cycle through the vector and dequeue “stop signal” that we inserted (before BOTH return lines).

1. i. I think that the **worst-case time complexity** of this algorithm is **O(n)**, because we initialize the loop on the line 4, that will make in worst case sqrt(n) iterations. And for each of these iterations we start another loop that will again make sqrt(n) iterations. Let us look at example: the worst case is that the value x is not in the vector V, so line 8 will never be executed and we need to end all loops to reach line 13, that will end the function and return FALSE. So, now let’s take a closer look at the nested loop construction. We take all integers in the range [1: floor(sqrt(n))] and for each of them we make skips of the length “m” = sqrt(n). So, our border cases will look like: i = 1, j = 1, and now we will make “y” skips, so j + y\*m > n. Or 1 + y \* sqrt(n) > n OR y > (n – 1)/sqrt(n) (\*) [sqrt is nonnegative, and n = 0 is empty vector]. And for i = floor(sqrt(n)), j = floor(sqrt(n)). Then j + y\*m > n can be written as floor(sqrt(n)) + y\*(sqrt(n)) > n [for large n we can omit floor function] => sqrt(n) \* (1 + y) > n => y > n/sqrt(n) – 1 (\*\*). When n is big, both (\*) and (\*\*) are approach sqrt(n). And now, we see that for the purpose of the asymptotic analysis the worst-case time complexity of this algorithm is O(n).

ii. Because **both** SkipSearch and LinearSearch **algorithms have worst-case time complexity O(n)**, we can say that **both** algorithms are **equally good** (from the perspective of the worst-case time complexity).