UNIVERSITY OF LONDON

CM1035

BSc EXAMINATION

COMPUTER SCIENCE

Algorithms and Data Structures I

Release date: Wednesday 17 March 2021 at 12 midday Greenwich Mean Time

Submission date: Thursday 18 March 2021 by 12 midday Greenwich Mean Time

Time allowed: 24 hours to submit

INSTRUCTIONS TO CANDIDATES:

Section A of this assessment paper consists of a set of **TEN** Multiple Choice Questions (MCQs) which you will take separately from this paper. You should attempt to answer **ALL** the questions in Section A. The maximum mark for Section A is 40.

Section A will be completed online on the VLE. You may choose to access the MCQs at any time following the release of the paper, but once you have accessed the MCQs you must submit your answers before the deadline or within **4 hours** of starting, whichever occurs first.

Section B of this assessment paper is an online assessment to be completed within the same 24-hour window as Section A. We anticipate that approximately **1 hour** is sufficient for you to answer Section B. Candidates must answer **TWO** out of the THREE questions in Section B. The maximum mark for Section B is **60**.

Calculators are not permitted in this examination. Credit will only be given if all workings are shown.

You should complete **Section B** of this paper and submit your answers as **one document**, if possible, in Microsoft Word or a PDF to the appropriate area on the VLE. You are permitted to upload 30 documents. However, we advise you to upload as few documents as possible. Each file uploaded must be accompanied by a coversheet containing your **candidate number**. In addition, your answers must have your candidate number written clearly at the top of the page before you upload your work. Do not write your name anywhere in your answers.

SECTION B

Candidates should answer any **TWO** questions from Section B.

Question 1 This question is about recursion and the binary search algorithm.

(a) Consider the following piece of pseudocode:

```
1: function FACTORIAL(n)
2:     if n = 0 then
3:         return 1
4:     end if
5:     return n × FACTORIAL(n)
6: end function
```

This function is supposed to return the factorial of a non-negative integer n. The factorial is the product of all integers from 1 to n, and is equal to 1 for n equal to 0. For which of the following reasons will it not do this correctly? There is only one correct answer. [2]

i. The base case is incorrect

ii. The function does not return anything for all arguments n

iii. There will be infinite recursion for n not equal to 0

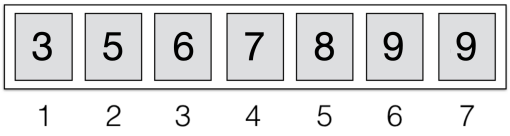
iv. The function will not return anything for n equal to 0

(b) Consider the following piece of incomplete pseudocode function, which takes the number n as an argument, and should return the sum of all integers from 0 to n:

```
1: function SUM(n)
2:     if n = 0 then
3:         return 0
4:     end if
5:     return MISSING
6: end function
```

What expression should go in the place of MISSING? [3]

(c) Consider the following vector of integers:



You are tasked with algorithmically searching this vector to see if it is storing the value 8. By hand, directly run through an implementation of the binary search algorithm on this vector to search for this value. [7]

(d) What is the worst-case and best-case time complexity (in n) for the binary search algorithm searching a sorted vector of length n? [4]

(e) Binary search is a ”decrease and conquer” algorithm, but not a ”divide and conquer” algorithm. Explain this statement, and make a distinction between these two types of algorithm. [4]

(f) Consider the following piece of incomplete pseudocode:

```
1: function BINARYSEARCH(vector, item, left, right)
2:     ...
3: end function
4:
5: function SEARCH(vector, item)
6:     n ← LENGTH[vector]
7:     if n = 0 then
8:         return FALSE
9:     end if
10:    return BINARYSEARCH(vector, item, 1, n)
11: end function
```

This code, when completed, should describe a recursive version of the binary search algorithm. Complete the function BINARYSEARCH such that it returns TRUE if the value item is stored in the vector, and FALSE otherwise. Do not use any form of iteration in the function. [6]

(g) What is the worst-case time complexity (in n) of the recursive version of binary search in Question 2(f)? Briefly explain your answer. [4]

Question 2 This question is about queues and searching.

(a) Consider the following pseudocode:

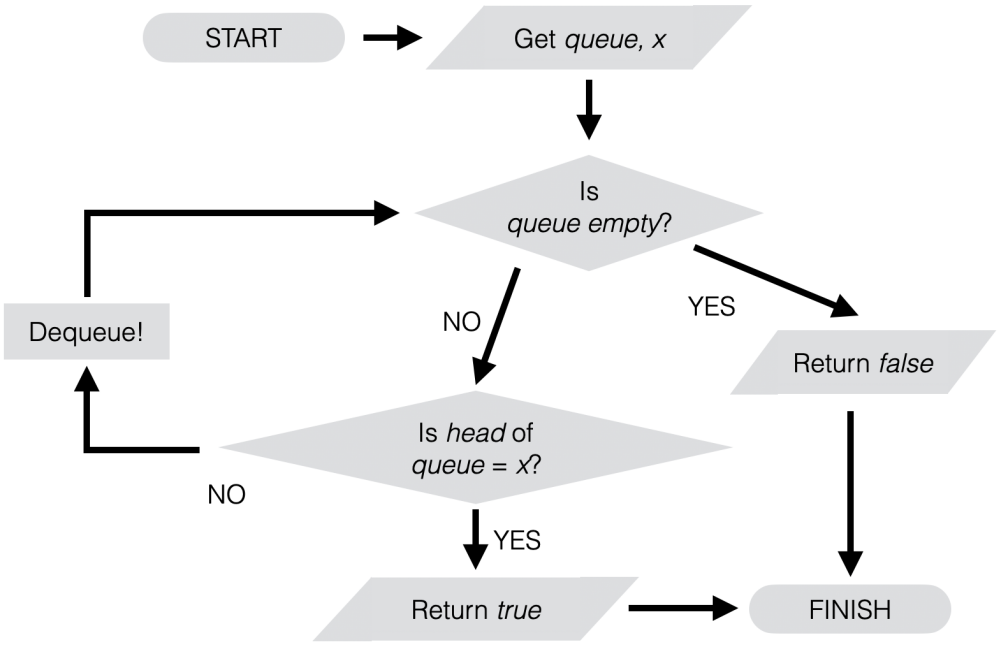
```
1: new Queue q
2: ENQUEUE[2,q]
3: ENQUEUE[3,q]
4: ENQUEUE[HEAD[q],q]
5: DEQUEUE[q]
6: ENQUEUE[5,q]
7: x ← HEAD[q]
```

i. At the end of this pseudocode, what is the final value of x? [2]

ii. A linked list is used to implement this pseudocode. Draw the final linked list after implementing all of the pseudocode above. [4]

(b) For vectors and dynamic arrays storing integers, the linear search algorithm can be used to find an integer value stored in these abstract data structures. Briefly explain why the linear search algorithm cannot be directly applied to search a queue for a particular integer value. [4]

(c) Consider the following flowchart:



This is a proposed algorithm to search a queue for the value *x*. It should return TRUE if the value *x* is in the queue, and FALSE otherwise.

i. Write a new pseudocode function which will implement the algorithm in this flowchart. The function should have a queue called *queue* and the value *x* as input parameters, and return a Boolean. [7]

ii. There is a problem with this proposed algorithm. Very briefly explain what the problem is and how you could solve it. [6]

(d) A new algorithm is proposed to search for a value in an arbitrary vector. If the vector is of length n, then create the integer m $= \lfloor\sqrt{\mathrm{n}}\rfloor$ where $\lfloor y \rfloor$ is the floor of y, which is the largest integer smaller or equal to y. The idea is to start at the first element and if the value is not stored there, then inspect the element m places to the right until; this continues until there are no more elements m places to the right. When the end is reached, this process is repeated starting at the second element, if possible.

Here is the pseudocode for this algorithm:

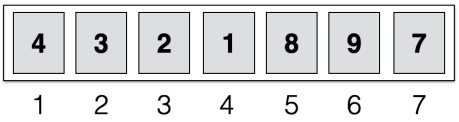
```
1: function SkipSearch(vector, x)
2:     n ← LENGTH[vector]
3:     m ← ⌊√n⌋
4:     for 1 ≤ i ≤ ⌊√n⌋ do
5:         j ← i
6:         while j ≤ n do
7:             if vector[j] = x then
8:                 return TRUE
9:             end if
10:            j ← j + m
11:        end while
12:    end for
13:    return FALSE
14: end function
```

i. What is the worst-case time complexity (in n) of the algorithm described by SkipSearch for a vector of length n? Briefly explain your answer. [4]

ii. From the perspective of worst-case time complexity, is this algorithm better than, worse than, or equally as good as the linear search algorithm? Very briefly explain your answer. [3]

Question 3 This question is about sorting algorithms.

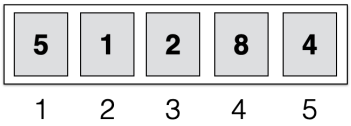
(a) Consider the following vector of integers:



i. By hand, work through the standard bubble sort algorithm on this vector, explicitly showing how it changes in the algorithm. You should sort the vector in ascending order so the lowest value is in the first element. [7]

ii. Give an example of a worst-case input vector of length 4 for the bubble sort algorithm. [4]

(b) Consider the following vector of integers:



i. By hand, work through the merge sort algorithm on this vector. Explicitly show how the vector changes in the algorithm. You should sort the vector in ascending order so the lowest value is in the first element. [7]

ii. What is the worst-case space complexity (in n) of the merge sort algorithm for a vector of length n? [3]

iii. From the point of view of worst-case time complexity, would you prefer to use merge sort or Quicksort? Briefly explain your answer. [4]

(c) Consider the following piece of incomplete JavaScript:

```
1  function swap(array, i, j) {
2    var store = array[i];
3    array[i] = array[j];
4    array[j] = store;
5    return array;
6  }
7
8  function partition(array, left, right) {
9    var m = Math.floor((left + right) / 2);
10   var final = m;
11   var pivot = array[m];
12   while (left < right) {
13     while (array[left] < pivot) {
14       left++;
15     }
16     while (array[right] > pivot) {
17       right--;
18     }
19     if (left < right) {
20       swap(array, left, right);
21       if (left === final) {
22         final = right;
23         left++;
24       } else if (right === final) {
25         final = left;
26         right--;
27       } else {
28         left++;
29         right--;
30       }
31     }
32   }
33   return final;
34 }
35
36 function quicksort(array, left, right) {
37   if (right <= left) {
38     return array;
39   }
40   MISSING
41 }
```

When completed this code should be a JavaScript implementation of the Quicksort algorithm on a JavaScript array (called `array`). Write the multiple lines of code that should go in the place of `MISSING` to complete this implementation of Quicksort. [5]

END OF PAPER