THIS PAPER IS NOT TO BE REMOVED FROM THE EXAMINATION HALL

UNIVERSITY OF LONDON

CM1035

BSc EXAMINATION

COMPUTER SCIENCE

Algorithms and Data Structures I

Time allowed: 2 hours

DO NOT TURN OVER UNTIL TOLD TO BEGIN

INSTRUCTIONS TO CANDIDATES:

This examination paper is in two parts: Part A and Part B. You should answer **ALL** of question 1 in Part A and **TWO** questions from Part B. Part A carries 40 marks, and each question from Part B carries 30 marks. If you answer more than **TWO** questions from **Part B** only your first **TWO** answers will be marked.

All answers must be written in the answer books; answers written on the question paper will not be marked. You may write notes in your answer book. Any notes or additional answers in the answer book(s) should be crossed out.

The marks for each part of a question are indicated at the end of the part in [.] brackets. There are 100 marks available on this paper.

Calculators are not permitted in this examination.

© University of London 2020

PART A

Candidates should answer **ALL** of Question 1 in Part A.

Question 1 If instructed to choose only one option, do not choose multiple options. If you select multiple options, your answer will be marked as 0. If otherwise instructed to select all options that apply, there will be multiple correct options, but you can lose marks for selecting incorrect options.

(a) Consider the following problem:

Given a non-negative integer n, is n^2 odd?

Which of the following is the type of the expected output for this problem? Choose only one option. [3]

i. Integer
ii. A fraction
iii. Boolean
iv. None of the above.

iii

(b) Which of the following statements describes what an algorithm is? There is only one correct answer. [2]

i. An algorithm is only a method for checking if computer programs are correct.
ii. An algorithm is only a method for finding if a number is an integer.
iii. An algorithm is a method of step-by-step basic instructions which, if followed, solves a problem.
iv. An algorithm is a set of problems that can be solved by a computer.

iii

(c) Consider the following piece of pseudocode:

```
1: x ← 2
2: x ← x + 2
3: function CHECK(n)
4:     if n = 2 then
5:         n ← n + 1
6:     end if
7:     return n − 1
8: end function
9: x ← CHECK(x)
10: x ← CHECK(x)
```

i. What is the value of x at the end of line 1? [1]

2

ii. What is the value of x at the end of line 2? [1]

4

iii. What is returned by CHECK(2)? [1]

2

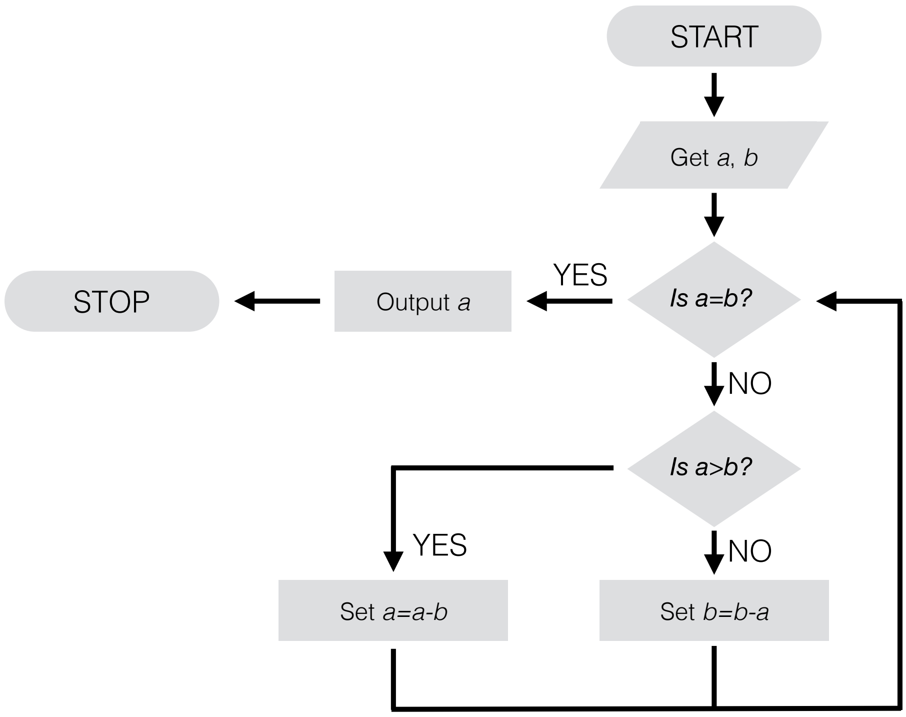
iv. What is the value of x at the end of line 9? [1]

3

v. What is the value of x at the end of line 10? [1]

2

(d) Consider the following flowchart:



The following incomplete pseudocode should implement the algorithm in this flowchart:

```
function EUCLID(a,b)
    while $a \neq b$ do
        if MISSING then
            $a \leftarrow a - b$
        else
            $b \leftarrow b - a$
        end if
    end while
    return a
end function
```

Which of the following should go in the place of MISSING? Choose only one option. [3]

i. $b < a$

ii. $b \neq a$

iii. $b > a$

i

(e) From where are elements removed in a stack? Choose only one option. [3]

i. Pop
ii. Head
iii. Tail
iv. Top

iv

(f) Consider the following piece of pseudocode:

```
new DynamicArray d
store ← 0
for 1 ≤ i ≤ 5 do
    d[i] ← i
end for
store ← store + LENGTH[d]
store ← store + d[4] + d[5]
```

i. What is the value of *store* at the end of line 6? [3]

5

ii. What is the value of *store* at the end of line 7? [3]

14

(g) Consider the following piece of incomplete Javascript:

```
var d = [];
var store = 0;
for (var i = 0; i < 5; i++) {
  MISSING1;
}
store = MISSING2;
store = store + d[3] + d[4];
```

When completed, this code should implement the pseudocode in question 1(f). What should go in the place of `MISSING1` to complete this code? Choose only one option. [2]

i. `d[i] = i;`

ii. `d(i) = i;`

iii. `d.i = i;`

i

(h) What should go in the place of `MISSING2` to complete the code in part (g) of this question? Choose only one option. [2]

i. `store + d[4].length`

ii. `store + d.length`

iii. `store + d[4]`

ii

(i) Which of the following describes the difference between a dynamic array and a stack? Choose only one option. [3]

i. The number of elements in a stack is fixed, but in a dynamic array the number of elements can change

ii. A stack has the LENGTH operation, but a dynamic array does not

iii. The number of elements in a dynamic array is fixed, but in a stack the number of elements can change

iv. A dynamic array has the LENGTH operation, but a stack does not *iv*

(j) Which of the following is the worst-case time complexity in ‘Big O’ notation of the linear search algorithm in n for a vector of length n? Choose only one option. [4]

i. O(log n)
ii. O(n^2)
iii. O(n)
iv. None of the rest

iii

(k) Consider the following piece of incomplete pseudocode for a function SUM(n), which takes the number n as an input parameter, and should return the sum of all integers from 0 to n.

```
function SUM(n)
    if n = 0 then
        return 0
    end if
    MISSING
end function
```

Which of the following should replace MISSING to complete this recursive function? Choose only one option. [4]

i. **return** n + SUM(n − 1)
ii. **return** n + SUM(n)
iii. **return** SUM(n)
iv. **return** SUM(n − 1)

i

(l) You have a large set of data with n elements, where n will tend to get very large. There is an algorithm that takes at most $0.1n^2 + 10n \log n$ time-steps to complete a task. Which of the following is the worst-case time complexity of this algorithm? Choose only one option. [3]

i. O($n \log n$)
ii. O(n^2)

iii. None of the above

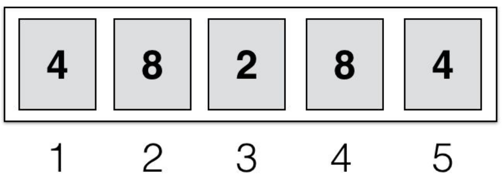
ii

PART B

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

Question 2 This question is about checking if a vector is a palindrome. A vector is a palindrome if it is exactly the same after the order of the elements is reversed (the vector is flipped). An empty vector is a palindrome.

(a) The following vector is an example of a palindrome:



Give an example of a vector that is **not** a palindrome. [3]

[3, 2, 1]

(b) To check if a vector of length n is a palindrome, one can first create an empty stack, and then push the first floor(n/2) values in the vector to it. Here floor(x) means the largest integer smaller than or equal to x. Consider the following piece of pseudocode:

```
function MAKESTACK(vector)
    new Stack s
    n ← LENGTH[vector]
    for 1 ≤ i ≤ floor(n/2) do
        PUSH[vector[i], s]
    end for
    return s
end function
```

Draw a picture of the returned stack after calling this function MAKESTACK on the example palindrome vector provided in part (a) of this question. [4]

Something that looks like `top->8,4`

(c) A linked list can be used to implement the stack in part (b) of this question. Draw the corresponding linked list of the final stack in part (b). [4]

Something that looks like `head->8->4->null`

(d) To complete the algorithm for deciding if a vector is a palindrome, after creating the stack using MAKESTACK, the contents of the stack are compared with the remaining values in the vector, starting from index ceil(n/2) + 1. Here ceil(x) is the smallest integer larger than or equal to x (the ceiling). Consider the following piece of incomplete pseudocode:

```
function IsPalindrome(vector)
    if LENGTH[vector] = 1 then
        return TRUE
    end if
    new Stack $s \leftarrow$ MakeStack(vector)
    n $\leftarrow$ LENGTH[vector]
    for ceil(n/2) + 1 $\leq$ i $\leq$ n do
        if TOP[$s$] $\neq$ vector[i] then
            return MISSING1
        end if
        MISSING2
    end for
    return TRUE
end function
```

This function should return TRUE if the input vector is a palindrome, or return FALSE otherwise. What should go in the place of MISSING1 and MISSING2 to complete this function? [4]

MISSING1 = FALSE; MISSING2 = POP[s]

(e) Given an argument for why the worst-case time complexity of implementing IsPalindrome is O(n) for an input vector of length n. [5]

An answer would look like this: The function MakeStack *has around n/2 iterations of a for loop, and then in* IsPalindrome *there are another n/2 iterations of a for loop performed after the first loop. All other operations are done in constant time so the total number of operations is O(n).*

(f) Another method for deciding if an input vector is a palindrome is through recursion. Consider the following piece of incomplete pseudocode:

```
function IsPalNew(vector)
    n $\leftarrow$ LENGTH[vector]
    return RecPalindrome(vector, 1, n)
end function
function RecPalindrome(vector, left, right)
    if right $\leq$ left then
        return TRUE
    end if
    if vector[left] $\neq$ vector[right] then
```

```
10:         return MISSING1
11:     end if
12:     MISSING2
13: end function
```

The function IsPalNew should return TRUE if the input vector is a palindrome, and FALSE otherwise - the function calls RecPalindrome to do this.

i. What should go in the place MISSING1? [2]

FALSE

ii. What should go in the place of MISSING2? [4]

RecPalindrome*(vector, left + 1, right - 1)*

(g) Briefly explain why the worst-case time complexity of implementing IsPalNew is also O(n) for an input vector of length n. [4]

An answer would look like this: Every time there is a recursive function call, a new element is pushed to the call stack. The worst-case input is if the vector is a palindrome and so while left ¡ right, vector[left] and vector[right] will be compared: and for each comparison there is a recursive call. The number of such calls is around n/2, so the call stack will have at most O(n) elements. Pushing and popping the elements will take O(n) operations and thus the worst-case time complexity is O(n).

Question 3 This question is about sorting algorithms.

(a) Consider the following vector of integers:

4	**2**	**12**	**1**	**7**	**9**	**9**
1	2	3	4	5	6	7

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]

First pass:

2,4,12,1,7,9,9

2,4,1,12,7,9,9

2,4,1,7,12,9,9

2,4,1,7,9,12,9

2,4,1,7,9,9,12

Second pass:

2,1,4,7,9,9,12

Third pass:

1,2,4,7,9,9,12

Fourth pass or returning it sorted

ii. What is the worst-case time complexity in n of the Bubble Sort algorithm for a vector of length n? [3]

$O(n^2)$

(b) Consider the following vector of integers:

6	**2**	**3**	**5**	**4**
1	2	3	4	5

i. By hand, work through the Quicksort algorithm on this vector where the pivot is the mid-point (calculated as the floor of half the sum of the left index plus the right index). 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. [6]

Calculate the pivot using floor of (left + right)/2 = 3

Partition vector to get [2, 3, 6, 5, 4]

Sort the following vector on the right of final location of pivot: [6, 5, 4]

Calculate new pivot of this smaller vector using floor of (3 + 5)/2 = 4

Partition this right vector to get: [2, 3, 4, 5, 6]

All sub-vectors are now of length 1 so we can return sorted vector [2, 3, 4, 5, 6]

ii. Give an example of a worst-case input vector of four elements for Quicksort where the pivot is the mid-point. Each element should store an integer and assume sorting in ascending order. Very briefly explain why it is a worst-case input. [5]

An example is [3,1,2,4] since every choice of mid-point is the next smallest value in the vector. Initially the value is 1, then it is 2, and then 3.

iii. Briefly explain the role of recursion in the Quicksort algorithm. [4]

An answer would look something like this: Quicksort is a divide-and-conquer algorithm where the input is divided into smaller inputs that can be solved and the solutions combined. The smaller inputs in the case of Quicksort are the vectors of length 1 or 0. In Quicksort the division is done by partitioning vectors, and then running Quicksort on the divided vectors. Recursion is then used to call Quicksort on the divided, smaller inputs. The base case are the then the vectors of length 1 or 0.

(c) Consider the following piece of incomplete JavaScript:

```
function swap(array, i, j) {
  var store = array[i];
  array[i] = array[j];
  array[j] = store;
  return array;
}

function sort(array, right) {
  if (right <= 0) {
    return MISSING1;
  }
  for (var i = 0; i < right; i++) {
    if (array[i+1] < array[i]){
      swap(array, i, i+1);
    }
  }
  return MISSING2;
}

function sorter(array) {
  return sort(array, array.length - 1);
}
```

When completed this code should implement a recursive version of a sorting algorithm on a JavaScript array. The function `sorter` will return the sorted version of the argument `array`.

i. What should go in the place of `MISSING1`? [2]

`array`

ii. What should go in the place of `MISSING2`? [3]

`sort(array, right - 1)`

Question 4 This question is about determining if a number is the sum of two square numbers. A square number is the square of an integer. The number 0 and 1 are both square numbers.

(a) The following vector stores all square numbers from 1^2 to 7^2:

1	**4**	**9**	**16**	**25**	**36**	**49**
1	2	3	4	5	6	7

i. You are tasked with algorithmically searching this vector to see if it has an element with the value 1. Directly run through the binary search algorithm by hand on this vector, and return the index where the value is stored. Show explicitly each step taken in the algorithm. [7]

Calculate the midpoint as either the floor of (1+7)/2, which will be 4

Update the right variable to be 3

Calculate the new midpoint to be 2

Update the right variable to be 1

Calculate the new midpoint to be 1

Return 1

ii. Briefly explain why the vector above with the value 1 is an example of a worst-case input for the binary search algorithm? [3]

An answer would look like this: The value is in the very first element so it will not be inspected until the mid-point is 1, which is when we are only considering a single-element vector (or when left pointer equals right). Thus the vector needs to be divided completely until we have just one element, thus requiring the maximum number of iterations.

(b) One method to determine if an integer n is the sum of two square numbers is to try all combinations of the sum of two square numbers. Consider the following piece of incomplete pseudocode:

```
function SumSquares(n)
    if n = 0 then
        return TRUE
    end if
    for 0 ≤ i ≤ n do
        for 0 ≤ j ≤ n do
            if MISSING = n then
                return TRUE
            end if
        end for
    end for
    return FALSE
end function
```

This function, when completed, should return TRUE if n is the sum of two square numbers, and FALSE otherwise.

i. What should go in the place of MISSING to complete this function? [3]

   $i^2 + j^2$*, or equivalent*

ii. What is the worst-case time complexity of this function? [4]

   $O(n^2)$

(c) Another method to solve the same problem as in part (b) of this question is to first create a vector of all square numbers from 0 to n^2 and then for each value i^2 stored in the vector, search the vector for the value $n - i^2$. Since the values in the vector will be sorted, the binary search algorithm can be used. Consider the following piece of pseudocode:

```
function BinarySquares(n)
    if $n = 0$ then
        return TRUE
    end if
    new Vector(n+1) squares
    for $0 \leq j \leq n$ do
        squares[j] $\leftarrow j^2$
    end for
    for $0 \leq i \leq n$ do
        if BinarySearch(squares, $n - i^2$) = TRUE then
            return TRUE
        end if
    end for
    return FALSE
end function
```

This function will return TRUE if n is the sum of two squares, and FALSE otherwise. It assumes that the binary search algorithm is implemented by a function BinarySearch(vector, x), which returns TRUE if x is stored in the input vector, and FALSE otherwise.

i. Write the pseudocode function called BinarySearch(vector, x), which will implement the binary search algorithm. [9]

```
function BinarySearch(vector, x)
    right $\leftarrow$ LENGTH[vector]
    left $\leftarrow$ 1
    while left $\leq$ right do
        mid $\leftarrow$ floor((left + right/2))
        if vector[mid] = x then
            return TRUE
        else if vector[mid] > x then
            right $\leftarrow$ mid $-$ 1
        else
            left $\leftarrow$ mid + 1
        end if
    end while
```

```
14:     return FALSE
15: end function
```

ii. Explain why the worst-case time complexity of implementing the function BinarySquares(n) is O(nlog n). [4]

An answer would look like this: The worst case input n to BinarySquares *is one that is not a sum of two square numbers since all possible iterations will be carried out. The worst-case time complexity of binary search is O(log n). There are n iterations each of the first for loop in* BinarySquares *and the second for loop. For each iteration of the second for loop* BinarySquares *there is an implementation of binary search, thus the second loop will have O(nlog n) iterations in total. Since O(nlog n) grows faster than n, the worst-case time complexity is O(nlog n).*

END OF PAPER

.