THE UNIVERSITY OF THE WEST INDIES

Semester I ☐ Semester II ■ Supplemental/Summer School ☐

Examinations of December ☐ April/May ■ July ☐ 2019

Originating Campus: **Cave Hill** ☐ **Mona** ■ **St. Augustine** ☐ **Open** ☐

Mode: **On Campus** ■ **By Distance** ☐

Course Code and Title: **COMP2211 – Analysis of Algorithms**

Date:	**May 9, 2019**	Time:	**4pm**
Duration:	**2 Hours**	Paper No:	**1**

Materials required:

Answer booklet:	**Normal**	■	**Special**	☐	**Not required**		☐
Calculator: (where applicable)	**Programmable**	☐	**Non-Programmable**	☐	**Not required**		■
Multiple Choice answer sheets:	**numerical**	☐	**alphabetical**	☐	**1-20** ☐	**1-100**	☐
Auxiliary/Other material(s):	None						

Instructions to Candidates: This paper has 6 page(s) and 4 questions

Candidates are reminded that the examiners shall take into account the proper use of the English Language in determining the mark for each response.

Answer Question one (1) and any other two (2).

The total number of marks for the entire paper is **50**. The number in [] by each question indicates the number of marks allotted to the question. Justify all your answers; full credit will be given only for properly supported answers, partial credit will be given where applicable. Good skill!

Question 1 *General* **[20]**

a. Simplify the expression below to find its order of growth. Be sure to simplify your final answer as far as possible. **[3]**

$$\sum_{k=1}^{n^2}(\lg n^k)$$

b. In each of the recurrences below, $T(n) = 1$ for $n \leq 1$. Determine for each recurrence whether the Master Theorem is applicable, and if so, whether it would succeed at solving the recurrence. Copy the table into your answer booklet, and fill in the missing columns. If the Master Theorem is applicable, then fill in the E column, otherwise write "NA" in it. If the Master Theorem cannot solve the recurrence (in spite of E being well defined), write "NA" in the *Case* column. **[12]**

Recurrence	E	Case	Order of Growth
$T(n) = 2T((2n-3)/4) + n \lg n$			
$T(n) = 2T(n/3) + n \lg n$			
$T(n) = 3T(n/2) + n \lg n$			
$T(n) = T((2n-3)/2) + 1$			
$T(n) = 2T((2n-1)/4) + n$			

c. The code fragment below computes a table of maximum values over fixed sub-ranges of a given list.

```
1  def mkMaxTable(vec):
2      n = len(vec)
3      tbl = [vec]
4      def helper(width):
5          if width <= 1:
6              return vec
7          else:
8              hw = width // 2
9              prevRow = helper(hw)
10             row = []
11             for i in range(n - width):
12                 row.append(max(prevRow[i], prevRow[i + hw]))
13             tbl.append(row)
14             return row
15     helper(n)
16     return tbl
```

Let $T(n)$ represent the (worst case) time complexity of `mkMaxTable(vec)`, `vec` is some vector (list) and n is its length. Let $T_h(w)$ represent the worst case time complexity of `helper(`w`)`. Use the set {`+,-,*,//,%, <=, max, append`} as the basic operations under consideration.

(i) Derive a relationship between T and T_h. **[2]**

(ii) Derive a recurrence relation defining $T_h(w)$. **[3]**

Question 2 *Dynamic Data Structures* **[15]**

a. Show the changes in structure of an initially empty *binary heap* after inserting each of the following keys, in the order given: **[5]**

12, 7, 3, 5, 14, 10

b. Explain why searching a binary heap is (generally) less efficient than searching a binary search tree for a given key. **[3]**

c. Explain how one could exploit the heap ordering property to abort a search for a given value in a (minimising) heap. **[2]**

d. In this part you are asked to complete the implementation of `heapSearch` below (in spite of its relative inefficiency).

Assume that the heap is embedded in a vector (Python list) according to the description in the coursework; namely the size is stored at index 0, and the root is stored at index 1, and the left and right children of a node at index i are stored at indexes $2i$ and $2i + 1$ respectively.

As an example of the function's use, suppose that `h` represented the heap of integers that you were asked to illustrate in part(a). Then invoking: `heapSearch(h, 1, 5)` would return `True` (since we inserted 5 into it).

```
1   def heapSearch(vec, index, val):
2       size = vec[0]                # size stored at slot 0
3       if index > size:
4           return [      A      ]
5       else:
6           v = vec[index]
            [                    ]
            [                    ]
7           [          B         ]
```

Complete the preceding code fragment for performing a search on a binary heap, making sure to implement the exploit referenced in the previous part. You may write only the portions labeled `A` and `B`, or if you choose to rewrite the whole function in your answer booklet, you should be sure to highlight the sections of your code corresponding to the labels `A` and `B`. **[5]**

Question 3 *Sorting and Hashing* **[15]**

a. Show the operation of the `mergeSort` algorithm on the list `[12, 7, 19, 43, 5, 8, 15]` **[4]**

b. Complete the following implementation of the `merge` procedure so that it behaves as would be expected for use in an implementation of `mergeSort` for vectors (Python lists). (Please provide only the code indicated as missing in your answer booklet) **[6]**

```
1   def merge(vec1, vec2):
2       n1, n2 = len(vec1), len(vec2)
3       i, j = 0, 0
4       result = []
5       while i < n1 and j < n2:
6           e1 = vec1[i]
7           e2 = vec2[j]
8           if e1 <= e2:
9                   A
10          else:
11                  B
12      if i == n1:
13          C
14      else:
15          D
16      return result
```

c. A hash table uses an open-address hashing scheme with probing function, $g(i) = 3i^2 + 2i$. Each key k is a two digit number k_1k_2 (e.g. if $k = 19$ then $k_1 = 1, k_2 = 9$), and the hash function is defined as:

$$f(k_1k_2) = (k_1 + 5k_2) \bmod m$$

where m is the (current) size of the array backing the hash table.

Suppose that $m = 11$, and the following key-value pairs are added, in the order given, to the hash table:

$$\langle 21, \texttt{a}\rangle, \langle 33, \texttt{b}\rangle, \langle 45, \texttt{c}\rangle, \langle 71, \texttt{d}\rangle$$

Illustrate how these key-value pairs would be stored in the array backing the hash table, indicating any addresses that were probed when a key was added. Make sure to show the final address for each key-value pair. **[5]**

Question 4 *Graphs* **[15]**

a. Show the operation of Dijkstra's algorithm on the graph shown in Figure 1, to find the shortest weight path from v_0 to v_5. If, at any step, there is more than one choice of the next vertex to be explored, choose the *lower* numbered vertex. **[5]**

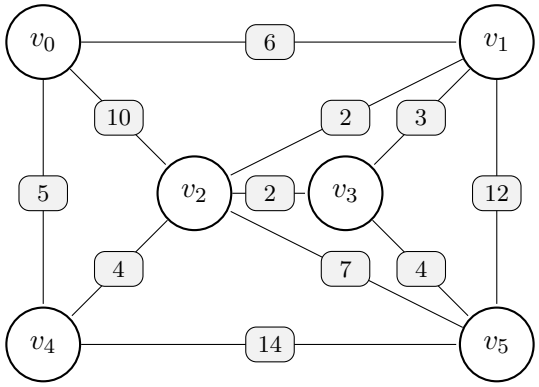

Figure 1: Graph for Question 4

b. The code fragment below is an implementation of Prim's algorithm. It uses a custom implementation of a priority queue, represented by the class `PriQueue` to manage discovered nodes. Its methods are standard, as discussed in the coursework and text book, but it also supports an efficient `update` operation to change the priority and any other data associated with a node.

```
1        UNSEEN = 0
2        DISCOVERED = 1
3        VISITED = 2
4
5        def primMST(graph, start):
6            """Graph is given as an adjacency list, represented by a
7               list of lists of (nbr, weight) pairs.
8               Each node is specified by its index, starting from 0.
9            """
10           n = len(graph)              # number of nodes in graph
11           bestWeights = [float('inf')] * n # initially all infinity
12           bestSources = [None] * n
13           status = [UNSEEN] * n
14           q = PriQueue()
15           q.insert(0, start, None)    # wt, node, src
16           while not q.isEmpty():
17               weight, node, src = q.deleteMin()
18               status[node] = VISITED
19               bestSources[node] = src
20               for nbr, wt in graph[node]:
21                  if status[nbr] == UNSEEN:
22                      status[nbr] = DISCOVERED
23                      q.insert(wt, nbr, node)
24                      bestWeights[nbr] = wt
25                  elif status[nbr] == DISCOVERED:
26                      if wt < bestWeights[nbr]:
27                          q.update(wt, nbr, node)
28                          bestWeights[nbr] = wt
29                  else:
```

```
30                          pass
31              return (bestWeights, bestSources)
```

(i) Suppose that `g` is defined to be the graph shown in Figure 1. Show what `print(g)` would display. (Assume that since `g` is undirected, each edge is actually represented twice, once for each direction.) **[3]**

(ii) It turns out that after the code is executed, the value stored in `bestWeights[start]` is infinity; but it should have been 0. Explain how to fix this bug (give line number(s) in your explanation). **[2]**

(iii) Although `primMST` implements Prim's algorithm, on completion, it returns a pair of lists, instead of an actual tree. Complete the implementation of `getMST` below so that it returns a list of edges in the minimum spanning tree computed by `primMST`. Each edge should be represented as a pair of indexes (e.g. the pair `(0, 2)` would represent the edge from node v_0 to v_2. It is acceptable to represent only one direction for each edge. (That is, if you include `(0, 2)` in your result, you do not also need to include `(2, 0)`.

```
1               def getMST(graph):
2                   bw, bs = primMST(graph, 0)
3                   tree = []
                    [                                  ]
                    [                                  ]
                    [                                  ]
4                   [                                  ]
5                   return tree
```

[3]

(iv) Suppose that we wanted to modify this program to compute Dijkstra's algorithm instead. Use the line numbers to indicate the sections of code that would need to be modified, and indicate what modifications are needed to accomplish this objective. **[2]**

END