CURTIN UNIVERSITY
DEPARTMENT OF COMPUTING

Test 2 – S1/ 2017

SUBJECT: Design and Analysis of Algorithms Unit Code COMP3001

TIME ALLOWED:

55 minutes test. The supervisor will indicate when answering may commence.

AIDS ALLOWED:

To be supplied by the Candidate: Nil

To be supplied by the University: Nil

Calculators are NOT allowed.

GENERAL INSTRUCTIONS:

This paper consists of Two (2) questions with a total of 50 marks.

ATTEMPT ALL QUESTIONS

Name: ____

Student No: ____

Tutorial Time/Tutor: ________________________________

QUESTION ONE (total: 20 marks): Graph and Heap

a) **(Total: 8 marks).** Consider a list A = <6, 4, 0, 15, 7, 2>, and the following algorithm to build a max-heap from list A. **Note** that list A starts from index 1.

Build-Max-Heap1 (*A*)
Input: An array *A* of size $n = A.length$
Output: A max-heap of size *n*

```
A.heap_size ← 1
for i ← 2 to A.length
    do Max-Heap-Insert (A, A[i])
```

// The following function is from the lecture slide
Max-Heap-Insert (*A*, *key*)
Input: heap ($A[1\ldots n]$), *key* - the new element
Output: heap ($A[1\ldots n+1]$) with *key* in the heap

```
A.heap_size = A.heap_size + 1;
i = A.heap_size;
while i > 1 and A[PARENT(i)] < key
    A[i] = A[PARENT(i)];
    i = PARENT(i);
A[i] = key
```

(i) **(5 marks).** Use Build-Max-Heap1 (*A*) to construct the max heap from *A*. Show your detailed steps.

(ii) **(3 marks).** How does the running time complexity of the Build-Max-Heap1 (*A*) as compared to that of the following Build-Max-Heap (*A*), i.e., is it faster, slower, or the same? Justify your answer by giving the time complexity of the two algorithms.

// The following function is from the lecture slide
Build-Max-Heap (*A*)
Input: An array *A* of size $n = A.length$
Output: A max-heap of size *n*

```
A.heap_size = A.length
for i = ⌊A.length/2⌋ downto 1
    do MAX-HEAPIFY(A, i)
```

<u>Answer:</u>

(i) Build-Max-Heap1 (A)

(ii)

b) **(Total: 6 marks).** Consider a graph $G(V, E)$ that contains n nodes and m edges, and its adjacency list L[x] of each node $x \in V$.

(i) **(4 marks).** Write the pseudocode of an $O(m+n)$ algorithm that returns a node with the maximum degree and its node degree.

(ii) **(2 marks).** Analyse the time complexity of your algorithm.

Answer:

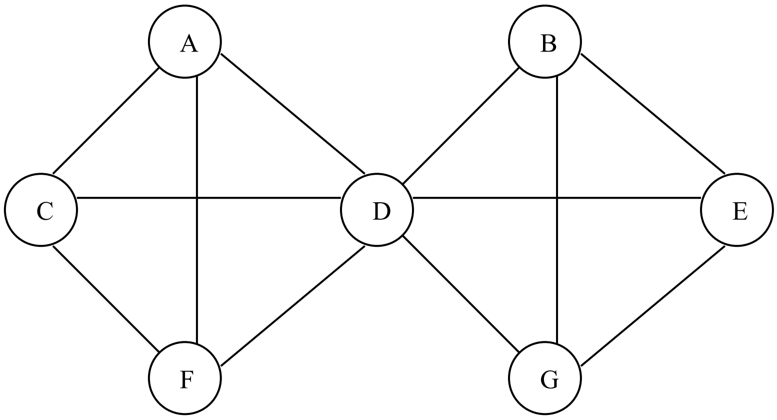
(i) Pseudocode

(ii)

c) **(Total: 6 marks).** Consider the following graph $G(V, E)$.

(i) **(3 marks).** Draw the depth first search tree of the graph. Assume the root of the tree is **node B**. You have to traverse the nodes in alphabetical order whenever possible.

(ii) **(3 marks).** Draw the breadth first search tree of the graph. Assume the root of the tree is **node B**. You have to traverse the nodes in alphabetical order whenever possible.



<u>Answer:</u>

(i) Depth first search tree

(ii) Breadth first search tree

END OF QUESTION ONE

QUESTION TWO (total: 30 marks): Greedy Algorithms

a) **(10 marks).** Suppose there are n sorted lists $L_1, L_2, \ldots, L_n$ of sizes $S_1, S_2, \ldots, S_n$, respectively, which need to be merged into one combined sorted list but we can merge them only two at a time, e.g., using the merge function that we discussed for the merge sort. We aim to find a merge ordering to generate a combined sorted list with minimized total number of comparisons.

As an example, consider L_1, L_2, and L_3 of sizes $S_1 = 30$, $S_2 = 20$, and $S_3 = 10$. One possible merge order is 1) merge L_1 and L_2; this step needs in the worst case $30 + 20 = 50$ comparisons, and produces a list of size 50, and 2) merge the resulting list and L_3; this step needs $50 + 10 = 60$ comparisons. The total number of comparisons is thus $50 + 60 = 110$.

Alternatively, 1) merge L_2 and L_3; this step requires $20 + 10 = 30$ comparisons, and 2) merge the resulting list (size 30) with L_1; this step needs $30 + 30 = 60$ comparisons. The total number of comparisons for this alternative is $30 + 60 = 90$, which is better as compared to the first attempt. So, the better pattern is $L_1 + (L_2 + L_3)$, where "+" means "merge".

Your task: Generate an optimal merge ordering and its total number of comparisons for six (6) sorted lists L_1, L_2, L_3, L_4, L_5, and L_6 of size $S_1 = 30$, $S_2 = 20$, $S_3 = 10$, $S_4 = 5$, $S_5 = 25$, and $S_6 = 15$. Show your detailed steps to obtain the number of comparisons, and the list order. Your solution must also state the greedy property to use to design a greedy algorithm for the optimal merge ordering. E.g., the greedy property in Kruskal's algorithm is to greedily select a link with minimum weight that does not create a cycle.

<u>**Answer:**</u>

<u>Greedy Property:</u>

<u>Detailed Steps:</u>

List order:

Number of comparions:

b) **(4 marks).** Consider the following Greedy Activity Selector algorithm.

GREEDY_ACTIVITY_SELECTOR (*s*, *f*)

```
n ← length [S]
A ← {1}
j ← 1;
for i ← 2 to n
        do if s_i ≥ f_j
              then A ← A ∪ {i}
                   j ← i
return A
```

Generate the maximum-size set of mutually compatible activities for the following activities (A_i denotes activity i). Show your steps.

S = { $A_1 = (0, 4)$, $A_2 = (4, 6)$, $A_3 = (1, 4)$, $A_4 = (12, 14)$, $A_5 = (4, 7)$, $A_6 = (3, 8)$, $A_7 = (7, 9)$, $A_8 = (8, 12)$, $A_9 = (6, 8)$, $A_{10} = (9, 11)$, $A_{11} = (0, 13)$ }

Hint: the input S to the greedy algorithm must first be sorted as required.

Answer:

c) **(Total: 16 marks).** Consider the following Dijkstra's algorithm to be implemented using the **binary min-heap**.

Single-source shortest path_G (*V*, *E*, *u*)
Input: $G = (V, E)$, the weighted directed graph and u the source vertex
Output: for each vertex, v, $d[v]$ is the length of the shortest path from u to v.

1. mark vertex u
2. $d[u] \leftarrow 0$
3. **for** each unmarked vertex $v \in V$ **do**
4.   **if** edge (u, v) exists **then** $d[v] \leftarrow weight\,(u, v)$
5.   **else** $d[v] \leftarrow \infty$
6. **while** there exists an unmarked vertex **do**
7.   let v be an unmarked vertex such that $d[v]$ is minimal
8.   mark vertex v
9.   **for** all edges (v, x) such that x is unmarked **do**
10.   **if** $d[x] > d[v] + weight[v, x]$ **then**
11. $d[x] \leftarrow d[v] + weight[v, x]$

(i) **(2 marks).** Which line or lines use function Build-Min-Heap ()? Justify your answer.

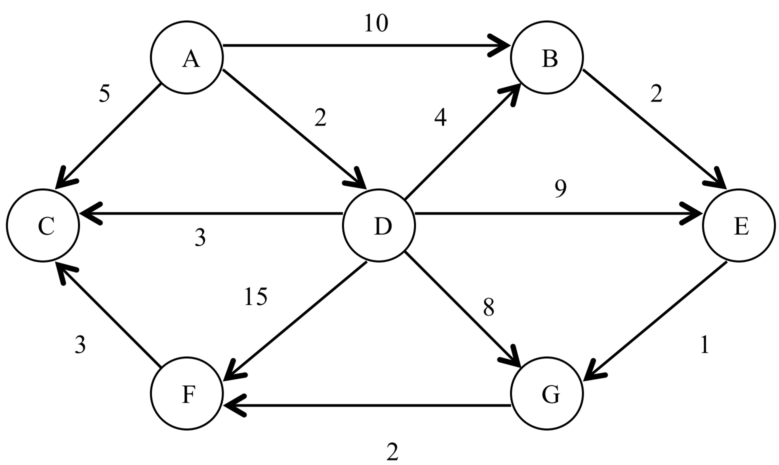
(ii) **(2 marks).** Which line or lines use function Heap-Extract-Min ()? Justify your answer.

(iii) **(4 marks).** Explain how to implement **Line 11** for the binary min heap. What is the time complexity of the implementation?

Note: you **are not required** to write any pseudocode in your explanation.

(iv) **(6 marks).** Use the Dijkstra's algorithm to generate the shortest paths from node **A** of the following graph.

(v) **(2 marks).** From your solution in part (iv), give the **shortest path** from node **A** to node **F,** and its **minimum distance**.



Answer:

(i)

(ii)

(iii)

(iv) Dijkstra – shortest paths.

Step#	**Vertex to be marked**	**Distance to vertex**						
		A	**B**	**C**	**D**	**E**	**F**	**G**
0								
1								
2								
3								
4								
5								
6								

(v)

END OF QUESTION TWO

Attachment

Procedure DFS_Tree_G (V,E)
Input: G = (V,E) in adjacency list format;. x refers to the value on top of stack; L[x] refers to the adjacency list of x.
Output : The DFS tree T

```
Mark all vertices "new" and set T ← {0}
Mark any one vertex v ← old
push (S,v)
while S is nonempty do
    while exists a new vertex w in L[x] do
        T ← T ∪ (x,w)
        w ← old
        push w onto S
    pop S
```

BFS_Tree_G (*V*,*E*)
Input: *G* = (*V*, *E*). *L*[*x*] refers to the adjacency list of *x*.
Output: The BFS tree *T*;

```
Mark all vertices new and set T = { }
Mark the start vertex v =old
insert (Q, v) // Q is a queue
 while Q is nonempty do
    x = dequeue (Q)
    for each vertex w in L[x] marked new do
        T = T ∪ {x,w}
        Mark w = old
        insert (Q,w)
```

BUILD-MIN-HEAP (*A*)
Input: An array *A* of size *n*= *length*[*A*]; *heap_size*[*A*]
Output: A min-heap of size *n*

```
heap_size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do MIN-HEAPIFY(A, i)
```

MIN-HEAPIFY (*A*, *i*)

```
l ← LEFT_CHILD (i)
r ← RIGHT_CHILD (i)
if l ≤ heap_size[A] and A[ l ] < A[ i ]
   then smallest ← l
   else smallest ← i
if r ≤ heap_size[A] and A[r] < A[smallest]
   then smallest ← r
if smallest ≠ i
   then exchange A[i] ↔ A[smallest]
        MIN-HEAPIFY (A, smallest)
```

HEAP_EXTRACT_MIN (*A*[1…*n*])

```
if heap_size[A] ≥ 1 then
    min ← A[1];
    A[1] ← A[heap_size[A]];
    heap_size[A] ← heap_size[A]-1;
    MIN-HEAPIFY(A, 1)
    return min
```

HEAP_INSERT (*A*, *key*)

```
heap_size[A] ← heap_size[A]+1;
i ← heap_size[A];
while i > 1 and A[PARENT(i)] > key
    A[i] ← A[PARENT(i)];
    i ← PARENT(i);
A[i] ← key[1]
```

For a node with *index i*:

PARENT(*i*) is the *index* of the parent of *i*
LEFT_CHILD(*i*) is the *index* of the left child of *i*
RIGHT_CHILD(*i*) is the index of the right child of *i*

END OF TEST PAPER