# COMP4500/7500 Advanced Algorithms & Data Structures
# Tutorial Exercises 3 (2014/2)*

School of Information Technology and Electrical Engineering, University of Queensland

August 27, 2014

This material aims to familiarise you with analysing programs to determine a formula that characterises their running times, and with graph representations and algorithms. It is important that you attempt to derive the required algorithms. A good treatment of elementary graph algorithms may be found in CLRS Chapter 22 [3rd, 2nd]; CLR Chapter 23 [1st].

1. Using $\Theta$-notation (rather than $O$-notation), give the worst-case execution time complexities of a call P($n$), where the argument $n$ is a positive integer.

```
P(n)
for i = 1 to n
    Q(i)
```

given that

(a) the execution time of Q($i$) is $\Theta(1)$

(b) the execution time of Q($i$) is $\Theta(n)$

(c) the execution time of Q($i$) is $\Theta(i)$

(d) the execution time of Q($i$) is $\Theta(n^2)$

(e) the execution time of Q($i$) is $\Theta(i^2)$

2. Using $\Theta$-notation, give the worst-case time complexity of a call P($n$), where the argument $n$ is a positive integer:

```
P(n)
for i = 1 to n
    if i mod 5 == 0
        for j = 1 to i
            for k = 1 to n
                // statements taking Θ(1) time
    else // i mod 5 ≠ 0
        for j = 1 to i
            // statements taking Θ(1) time
```

3. Consider the following merge procedure as used in implementing merge sort. It merges two segments of `array`: `array[left..middle]` and `array[middle+1..right]`. Each of these segments is assumed to be in order before the merge takes place.

```
//   This file contains the Java code from Program 15.15 of
//   "Data Structures and Algorithms
//    with Object-Oriented Design Patterns in Java"
//   by Bruno R. Preiss.
//
//   Copyright (c) 1998 by Bruno R. Preiss, P.Eng.
//   All rights reserved.
//
//   http://www.pads.uwaterloo.ca/Bruno.Preiss/books/opus5/
//          programs/pgm15_15.txt
```

```
//
public class TwoWayMergeSorter
    extends AbstractSorter
{
    Comparable[] tempArray;

    protected void merge (int left, int middle, int right)
    {
        int i = left;
        int j = left;
        int k = middle + 1;
        while (j <= middle && k <= right)
        {
            if (array [j].isLT (array [k]))
                tempArray [i++] = array [j++];
            else
                tempArray [i++] = array [k++];
        }
        while (j <= middle)
            tempArray [i++] = array [j++];
        for (i = left; i < k; ++i)
            array [i] = tempArray [i];
    }
    // ...
}
```

Carefully analyse the performance of procedure `merge` to give a worst case upper bound on its time complexity. All individual variable assignments take constant time as do comparisons and numeric operations. What is the space overhead of this procedure in $\Theta$ notation?

4. (See CLRS Exercise 22.1-2, p592 [3rd], p530 [2nd], CLR Exercise 23.1-2, p468 [1st])
   Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that the vertices are numbered 1 to 7 with 1 as the root of the tree, 2 and 3 as its children, 4 and 5 as the children of 2, and 6 and 7 as the children of 3.

5. (See CLRS Exercise 22.1-1, p592 [3rd], p530 [2nd], CLR Exercise 23.1-1, p468 [1st])

   (a) The out-degree of a vertex is the number of edges leaving the vertex. Give an abstract algorithm to compute the out-degrees of all vertices of a directed graph and for each vertex $u$ set $u.od$ to its out-degree. It is to be abstract in the sense that we do not want to worry about implementation details of the underlying data structures. You may use the abstract **for** loop

   **for** each vertex $u \in G.V$
   "process vertex $u$"

   to traverse the vertices of the graph $G$, and the abstract **for** loop

   **for** each vertex $v \in G.Adj[u]$
   "process edge $(u, v)$"

   to traverse the edges $(u, v)$ adjacent to $u$.

   (b) Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? Give your analysis in terms of the number of vertices, abbreviated $|V|$, and number of edges, abbreviated $|E|$, of the graph. What assumptions do you make in this analysis?

   (c) Give an algorithm using abstract **for** loops as above to determine the in-degrees of all the vertices of a directed graph. The in-degree of a vertex is the number of edges entering the vertex. How long does it take to compute the in-degrees?

(d) Outline a procedure to compute the out-degrees, but this time rather than using the abstract **for** loops (which are not available in most languages), do so in Java-like pseudocode. This will require you to design (but not necessarily implement) appropriate traversal procedures. You should indicate both algorithmic and data structure ideas that are used.

(e) Consider how the corresponding in-degree problem could be solved.

6. (See CLRS Exercise 22.1-3, p592 [3rd], p530 [2nd], CLR Exercise 23.1-3, p468 [1st])
The ***transpose*** of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where

$$E^T = \{(v, u) \in V \times V \mid (u, v) \in E\}.$$

Thus, $G^T$ is $G$ with all its edges reversed.

(a) Describe an algorithm for computing $G^T$ from $G$, for an adjacency-list representation of $G$. Assume the existence of a procedure $\textsc{InitGraph}(G)$ for initialising a graph $G$ to the empty graph, and methods $G.\textsc{AddVertex}(u)$ to add a vertex $u$ to a graph $G$, and $G.\textsc{AddEdge}(u, v)$ to add an edge from $u$ to $v$ to a graph $G$. Analyse the running time of your algorithm.

(b) Give a transpose algorithm, but this time using adjacency-matrix representation, where $G.\mathit{edge}$ is a boolean matrix with both the rows and columns indexed by vertices, so that $G.\mathit{edge}[u, v]$ is TRUE if and only if there is an edge from $u$ to $v$ in $G$.

Analyse the running time of your algorithm.