COMP4500/7500 Advanced Algorithms & Data Structures
Sample Solution to Tutorial Exercise 3 (2014/2)*

School of Information Technology and Electrical Engineering, University of Queensland

August 27, 2014

1. Using Θ-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)
1  for $i = 1$ to $n$
2      Q($i$)
```

given that

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

Sample solution. $\Theta(n)$ because the execution time complexity of P(n) is given by

$$\sum_{i=1}^{n} \Theta(1) = \Theta(\sum_{i=1}^{n} 1) = \Theta(n).$$

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

Sample solution. $\Theta(n^2)$ because the execution time complexity of P(n) is given by

$$\sum_{i=1}^{n} \Theta(n) = \Theta(\sum_{i=1}^{n} n) = \Theta(n\sum_{i=1}^{n} 1) = \Theta(n^2).$$

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

Sample solution. $\Theta(n^2)$ because the execution time complexity of P(n) is given by

$$\sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(\frac{n(n+1)}{2}) = \Theta(n^2).$$

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

Sample solution. $\Theta(n^3)$ because the execution time complexity of P(n) is given by

$$\sum_{i=1}^{n} \Theta(n^2) = \Theta(\sum_{i=1}^{n} n^2) = \Theta(n^2\sum_{i=1}^{n} 1) = \Theta(n^3).$$

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

*Copyright © reserved August 27, 2014

Sample solution. $\Theta(n^3)$ because the execution time complexity of P(n) is given by

$$\sum_{i=1}^{n} \Theta(i^2) = \Theta(\sum_{i=1}^{n} i^2) = \Theta(\frac{n(2n+1)(2n+2)}{12}) = \Theta(n^3).$$

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

```
P(n)
1  for $i = 1$ to $n$
2      if $i$ mod 5 == 0
3          for $j = 1$ to $i$
4              for $k = 1$ to $n$
5                  // statements taking $\Theta(1)$ time
6      else // $i$ mod $5 \neq 0$
7          for $j = 1$ to $i$
8              // statements taking $\Theta(1)$ time
```

Sample solution. The innermost **for** loop of the first branch of the **if** command is executed n times and its body is $\Theta(1)$. Hence the innermost loop has cost $\Theta(n)$. The **for** loop that immediately encloses it is executed i times. Hence its complexity is $\Theta(i\ n)$. This loop is executed when i is divisible by 5 for i in the range 1 to n. That is, it is executed for

$$\begin{aligned} i &= 5, 10, 15, 20, \ldots, 5 \times \left\lfloor \frac{n}{5} \right\rfloor \\ &= 5 \times 1, 5 \times 2, 5 \times 3, 5 \times 4, \ldots, 5 \times \left\lfloor \frac{n}{5} \right\rfloor \end{aligned}$$

or if we introduce a new variable m such that $i = 5m$ then m ranges over all the values between 1 and $\lfloor \frac{n}{5} \rfloor$ and the complexity of the loops within the "then" part is $\Theta(i \times n) = \Theta(5m \times n)$. The complexity of the outer loop due to the first branch of the **if** command is

$$\sum_{m=1}^{\lfloor \frac{n}{5} \rfloor} \Theta(5mn) = \Theta\left(5n \sum_{m=1}^{\lfloor \frac{n}{5} \rfloor} m\right) = \Theta\left(5n \frac{\lfloor \frac{n}{5} \rfloor (\lfloor \frac{n}{5} \rfloor + 1)}{2}\right) = \Theta(n^3)$$

The complexity of the loop in the **else** branch of the **if** command is $\Theta(i)$. Hence the contribution to the outer **for** loop of the **else** branch is bounded above by

$$\sum_{i=1}^{n} O(i) = O\left(\frac{n(n+1)}{2}\right) = O(n^2).$$

As this is strictly less than $\Theta(n^3)$, the overall complexity is determined by the **then** branch and is $\Theta(n^3)$.

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 Θ notation?

Sample solution. For the size of the input to `merge` we use the total size of the array segment, i.e., $n =$ `right` $-$ `left` $+1$.

So we just have to work out upper bounds on each of the three loops. It is clear that each loop causes at most $O(n)$ iterations. Hence the combination of everything is $O(n)$. Is the complexity of `merge` $\Theta(n)$ here in the context of the O's above? (There is a subtle bug in this code related to stability; can you find it? A sort algorithm is stable if elements with the same key remain in the same relative order after sorting.)

The space overhead is for variables dynamically allocated by the procedure. That is, for `tempArray, i, j, k`, which take space $\Theta(n)$. (Why is it OK to ignore the code space for the procedure?)

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.

   Sample solution. No solution provided.

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.

Sample solution.

```
OUT-DEGREE($G$)
1  for each vertex $u \in G.V$
2      $u.od = 0$
3      for each vertex $v \in G.Adj[u]$
4          $u.od = u.od + 1$
```

(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?

Sample solution. Each edge leaving a vertex u is represented by an entry in the adjacency list for u. The out-degree of a vertex is just the length of its adjacency list (given that there are no duplicates in the adjacency list). To determine the length of the adjacency list we need to scan it. If we do this for all vertices then we scan every entry in all the adjacency lists of the graph. As there is one entry in an adjacency list for every edge in a directed graph, the total number of entries scanned is $|E|$. We need to consider all $|V|$ vertices in the graph (even if there are no edges). Hence, the time taken is $\Theta(|V| + |E|)$.

Notice that we have used a more sophisticated analysis than simply multiplying together upper bounds for the nested loops. Had we done that we would have obtained the poor upper bound $O(|V|^2)$ since the worst case for the inner loop is $O(|V|)$. It is sometimes necessary to consider the number of executions of the body of a nested loop over all executions of both the outer and inner loops to get a good bound. This leads on to the idea of amortised analysis.

This analysis assumes a standard scenario. There is no field per vertex giving the out-degree of the vertex (otherwise we could solve the problem in $\Theta(|V|)$ time, using $\Theta(|V|)$ extra space). Also, it assumes our traversal procedures can iterate in constant time per iteration, which any competent implementation would achieve.

(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?

Sample solution.

```
IN-DEGREE($G$)
1  for each vertex $u \in G.V$
2      $u.id = 0$
3  for each vertex $u \in G.V$
4      for each vertex $v \in G.Adj[u]$
5          $v.id = v.id + 1$
```

We can determine the in-degree of all vertices in a graph in the same amount of time as for computing the out-degrees. We initialise in-degree counts for all vertices to zero ($\Theta(|V|)$). We then scan all the adjacency lists in the graph and, for each entry in an adjacency list, we increment the count of the target vertex entered by the corresponding edge ($\Theta(|E|)$). The total time is $\Theta(|V| + |E|)$.

(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.

Sample solution. Java is a good language for this task using the Java `for` loop based on objects which implement the `Iterable` interface. Implementing such abstract loops is more tedious in many other languages where you have to initiallise suitable data structures, work through them yourself, and possibly terminate them. Here we simply assume that `G.V` and `G.Adj(u)` are suitable collections that provide a method `iterator` that returns a suitable iterator.

```
void outDegree(Graph G) {
    for (Vertex u : G) {
        u.od = 0;
        for (Edge e : G.adjacent(u)) {
            u.od = u.od + 1;
        }
    }
}
```

Implementors of the `Iterator` interface hopefully solve the following issues well: How much space do we need? How efficient are the operations?

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

Sample solution.

```
void inDegree(Graph G) {
    for (Vertex u : G) {
        u.od = 0;
    }
    for (Vertex u : G) {
        for (Edge e : G.adjacent(u)) {
            e.getDestination().od++;
        }
    }
}
```

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 INITGRAPH(G) for initialising a graph G to the empty graph, and methods G.ADDVERTEX(u) to add a vertex u to a graph G, and G.ADDEDGE(u, v) to add an edge from u to v to a graph G. Analyse the running time of your algorithm.

Sample solution. For an adjacency-list representation we can build a new graph corresponding to G^T by scanning the adjacency lists of G once, and for each entry adding a new edge, in the opposite direction, to G^T. Pseudocode follows.

```
TRANSPOSE-ADJ-LIST($G, GT$)
1  INITGRAPH($GT$)
2  for each vertex $u \in G.V$
3      $GT$.ADDVERTEX($u$)
4  for each vertex $u \in G.V$
5      for each vertex $v \in G.Adj[u]$
6          $GT$.ADDEDGE($v, u$)
```

The first loop is $\Theta(|V|)$. The second loop is executed $|V|$ times but the body of the **while** loop is only executed once for each edge, during all executions of the **for** loop. Hence the time complexity of TRANSPOSE is $\Theta(|V| + |E|)$.

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

Analyse the running time of your algorithm.

Sample solution. For the adjacency-matrix representation $GT.\,edge$ is assigned the transpose of $G.\,edge$. This is done by swapping edges so that for all vertices u and v, $G.edge[u, v] = GT.edge[v, u]$. Code for this follows.

```
TRANSPOSE-ADJ-MATRIX$(G, GT)$
1  for each vertex $u \in G.\,V$
2      for each vertex $v \in G.\,V$
3          $GT.edge[v, u] = G.edge[u, v]$
```

The execution time of this is $\Theta(|G.\,V|^2)$. You might care to consider various methods for in-place transposition. Is in-place transposition desirable? Is physical transposition really necessary?