The University of Melbourne
School of Computing and Information Systems
COMP90038 Algorithms and Complexity

Assignment 1, Semester 2, 2017

Student Name: Xiaowei Xu
Student No.: 843643

My Answers for the Challenges

1. (a) According to the Master Theorem, with $T(n) = 2 \cdot T(n/2) + 1$, we have $a = 2, b = 2, f(n) = 1$, thus we have $d = 0$. With $T(1) = 1$, we compare a with b^d, then we have $a > b^d$. Thus, $T(n) \in \Theta(n^{log_2 2}) = \Theta(n)$.

   (b) According to telescoping, we have $T(n) = 2 \cdot T(n) + 1 = 2 \cdot (2 \cdot T(n/4) + 1) + 1 = 4 \cdot (2 \cdot T(n/8) + 1) + 2 + 1$. Thus, according to smoothness rule, we conclude a similar form of the equation, that is: $T(n) = 2^k \cdot T(n/2^k) + 2^k - 1, (k \in [0, log_2 n])$. When $n = 2^k$, we have $T(1) = 1$, thus we have $T(n) = 2 \cdot n - 1$.

   (c) I don't agree with the argument.

   There's a hidden assumption in the question that $T(n) = k \cdot n$. Thus, it can have $T(n) = 2 \cdot k \cdot n/2 + 1$. We can find that $T(n) = 2 \cdot k \cdot n/2 + 1 \leq k \cdot n$ is clearly false. Thus, the hidden assumption $T(n) = k \cdot n$ is false. According to the question, from $T(n) \in \Theta(n)$, we should have $T(n) \leq k \cdot n$ but not $T(n) = k \cdot n$. Therefore, we can't have $T(n) = 2 \cdot k \cdot n/2 + 1$ and also $2 \cdot k \cdot n/2 + 1 \leq k \cdot n$. Hence we can't have $T(n) \notin \Theta(n)$.

2. (a)

```
function CountSubstring(S[0..n − 1])
    //Brute-force version
    //Input: String S[0..n − 1]
    //Output: The number of the substring "AC"
    count ← 0
    for i ← 0 to n − 2 do
        if s[i] = "A" then
            for j ← i + 1 to n − 1 do
                if s[j] = "C" then
                    count ← count + 1
    return count
```

   (b)

```
function CountSubstring(S[0..n − 1])
    //Linear version
    //Input: String S[0..n − 1]
    //Output: The number of the substring "AC"
    countA ← 0
    countAC ← 0
    for i ← 0 to n − 1 do
        if s[i] = "A" then
            countA ← countA + 1
        if s[i] = "C" then
            countAC ← countAC + countA
    return countAC
```

Complexity Analysis: In the worst case, the basic operation in this algorithm is the comparisons: whether $s[i]$ = "A" and $s[i]$ = "C" or not. The comparisons will conduct once each time when i increment 1 in the for loop. The for loop only iterate once from 0 to n-1. Thus, $C(n) \in O(n), n = |s|$ which is the size of the string, the complexity of the algorithm is linear in $|s|$.

3. (a)

function DISTANCE($\langle V, E\rangle, v$)
  //Use adjacency list in this algorithm
  //**Input:** Graph $G = \langle V, E\rangle$, node v
  //**Output:** An array *dist* recoding all the other nodes' distance from v
  creat an array *dist* of length n, index the array with all the nodes' indexes
  for each nodes in V **do**
    $dist[i] \leftarrow \infty$
    $dist[v] \leftarrow 0$
  inject(*queue*, v)
  while *queue* **is not empty do**
    $a \leftarrow$ **eject**(*queue*)
    for each $edge(a, b)$ **do**
      if $dist[b] = \infty$ **then**
        $dist[b] \leftarrow dist[a] + 1$
        inject(*queue*, b)
  return *dist*

Complexity Analysis: To find the distance of all other nodes to v, one have to traverse all the nodes and edges in the graph. Considering we are using an adjacency list, which contain only an array of nodes and each node's edge to their neighbour, the size of the adjacency list should be $|V| + |E|$. Considering the iterator will revisit the nodes that it has been visited for at most once in this algorithm, the complexity of this algorithm should be $C(n) \in \Theta(|V| + 2|E|)$, which runs linear in the size of the graph.

(b)

function HUB($\langle V, E\rangle$)
  //Use adjacency list.
  //**Input:** Graph $G = \langle V, E\rangle$
  //**Output:** An array *hub*
  create an array *hub*
  $minRemoteness \leftarrow \infty$
  for each vertex v **in** V **do**
    $dist \leftarrow$ DISTANCE($\langle V, E\rangle, v$) ▷ Call function DISTANCE and store the returned value to array *dist*
    $maxDistance \leftarrow 0$
    for $j \leftarrow 1$ **to** n **do**
      if $dist[j] \geq maxDistance$ **then**
        $maxDistance \leftarrow dist[j]$ ▷ Locate the max distance
    if $maxDistance \leq minRemotness$ **then**
      $minRemoteness \leftarrow maxDistance$ ▷ Locate the minimal remoteness
      $hub \leftarrow v$
  return *hub*

Complexity Analysis: Every time we call this function, it calls function DISTANCE, which runs in linear, that is $\Theta(n)$ times. The basic operation of the algorithm is the comparison: whether $dist[j] \geq maxDistance$ or not, which conduct n times, plus one comparison between $maxDistanceminRemotness$, altogether $n+1$ times. These comparisons conduct n times, in which n is the number of nodes. Therefore, in the worst case the complexity of this algorithm is $C(n) \in n(n-1) = O(n^2)$.

4. (a)

```
function FindAttractor(A[·,·], n)
    //Use adjacency matrix to present the Graph in the matrix,
    //A[k,j] = 1 iff (k,j) ∈ E, A[k,j] = 0 iff (k,j) ∉ E
    //Input: matrix A[·,·], each node is labelled 1 to n
    //Output: i if i is an attracter, −1 if the Graph has no attractor
    for k ← 1 to n do
        for j ← 1 to n do
            if k ≠ j then
            if A[k,j] = 0 and A[j,k] = 1 then ▷ Check if ∀j, (k,j) ∉ E and (j,k) ∈ E
                i ← k
            else
                i ← −1
                break                ▷ Break the inner loop and switch to the next node
            if i ≠ −1 then
                return i
    return −1
```

Complexity Analysis: There are all together n nodes in the adjacency matrix. Thus, the size of the adjacency matrix is n^2. The basic operations are the two comparisons whether $A[k,j] = 0$ and $A[j,k] = 1$ or not. In the worst case, one can't find a sink in a graph. This algorithm will iterate every cell in the adjacency matrix and in each iteration compare twice. Thus, the worst case complexity $C(n) \in O(2n^2) = O(n^2)$.

(b)

```
function FindAttractor(A[·,·], n)
    //Use adjacency matrix to present the Graph in the matrix,
    //A[k,j] = 1 iff (k,j) ∈ E, A[k,j] = 0 iff (k,j) ∉ E
    //Input: matrix A[·,·], each node is labelled 1 to n
    //Output: i if i is an attracter, −1 if the Graph has no attracter
    i ← 1
    j ← 1
    while i ≤ n and j ≤ n do
        if A[i, j] = 1 then                  ▷ Check if ∃j ∈ V with (i,j) ∈ E
            i ← i + 1                        ▷ If true move to the next row
        else
            j ← j + 1                        ▷ Else keep on iterating in the same row
    return i
    if IsAttractor(A[·,·], i) = true then ▷ Call function IsAttractor to check
  if the returned node is an attractor
        return i
    else
        return −1
    return −1
```

```
function IsAttractor(A[·,·], i)
    //This function is used to check if a given node i is an attractor
```

```
//Input: The adjacency matrix A, a given node i
//Output: true if the node i is an attractor, false if it isn't an attractor
for k ← 1 to n do
    if A[i, k] ≠ 0 then                         ▷ Check if ∀k, (i, k) ∉ E
        return false
for j ← 1 to n do
    if j ≠ i then
        if A[j, i] ≠ 1 then                     ▷ Check if ∀j, (j, i) ∈ E
            return false
return true
```

Complexity Analysis: There are all together n nodes in the adjacency matrix. The complexity of function IsAttractor is $c(n) \in \Theta(2n-1)$, because we only iterate one row and one column(except the node itself when iterating the column) of the matrix with the size n. The worst case complexity of the function FindAttractor is also $c(n) \in O(2n)$, because in the worst case, the iterator may have to skip every row until the last row and iterate all items in the last row, which is $2n$ times of iterations. In the worst case($i = |V|$), everytime we call the function FindAttractor, we must also call the function IsAttractor. Hence, the worst case complexity for function IsAttractor is $C(n) \in O(4n-1) = O(n)$, which runs in linear time.