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

Assignment 1, Semester 2, 2017

Released: 9 August. Deadline: 31 August at 23:00

Tiancong Li

1. Consider the recurrence relation $T(n) = 2T(n/2) + 1$, with $T(1) = 1$.

   (a) Use the Master Theorem to find a correct Θ expression for $T(n)$.
   ANSWER:
   As the Master Theorem described, for a recurrence relation $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$,
   when $f(n) = O(n^c)$ and $c < c_{crit}$ where $c_{crit} = log_b a$,
   we have: $T(n) = \Theta(n^{c_{crit}})$.
   So, for the specific relation $T(n) = 2T(n/2) + 1$,
   we have: $c_{crit} = log_b a = 1$ and $f(n) = 1$.
   Let $c = 0$,
   then we have: $f(n) = O(n^c) = O(1)$ and $0 = c < c_{crit} = 1$.
   So, we have: $T(n) = \Theta(n^{c_{crit}}) = \Theta(n)$.

   (b) Use telescoping to give an exact closed-form definition of $T(n)$.
   ANSWER:
   Let $n = 2^k$ and $S(k) = T(2^k)$,
   so we have: $T(2^k) = 2T(2^{k-1}) + 1$,
   and so: $S(k) = 2S(k-1) + 1$.
   Multiply both sides by 2^{-k},
   so we have: $2^{-k}S(k) = 2^{-(k-1)} + 2^{-k}$.
   Let $U(k) = 2^{-k}S(k)$,
   so we have: $U(k) = U(k-1) + 2^{-k}$.
   This telesopes yielding: $U(k) = U(0) + 1 - 2^{-k}$,
   where $U(0) = S(0) = T(1) = 1$,
   so we have: $U(k) = 2 - 2^{-k}$,
   and so: $2^{-k}S(k) = 2 - 2{-}k$,
   and so: $T(2^k) = 2(k+1) - 1$.
   Since $k = logn$,
   so we have: $T(n) = 2^{logn} * 2 - 1 = 2n - 1$.

   (c) Suppose somebody presents you with the following argument, to show that $T(n) \notin \Theta(n)$.

   > If we had $T(n) \in \Theta(n)$ then $T(n) \leq k \cdot n$ for some constant k. But, substituting into the recurrence relation, we get $2 \cdot k \cdot n/2 + 1 \leq k \cdot n$. And that in turn is clearly false for all k. Hence we must have $T(n) \notin \Theta(n)$.

   Do you agree with this argument? Why or why not?
   ANSWER:
   Disagree.
   If $T(n) \in \Theta(n)$,
   Then we have: $T(n) \leq k \cdot n$,
   and so: $T(n/2) \leq k \cdot n/2$.
   So we have: $T(n) = 2T(n/2) + 1 \leq 2 \cdot k \cdot n/2 + 1$.
   So there is no way to get $2 \cdot k \cdot n/2 + 1 \leq k \cdot n$
   from $T(n) \leq k \cdot n$ and $T(n) \leq 2 \cdot k \cdot n/2 + 1$.

2. DNA nucleobases are `A` (adenine), `C` (cytosine), `G` (guanine) and `T` (thymine). Consider the problem of counting, in a DNA string s, the number of sequences (substrings) that start with an `A` and end with a `C`. For example, there are four such sequences in `TACAAGCGA`.

   (a) Using pseudo-code, give a brute-force algorithm to solve this problem.
   ANSWER:

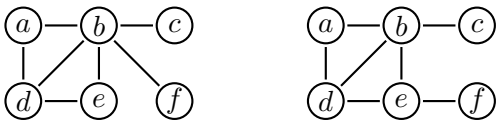
```
function COUNTING(s, n, start, end)                ▷ count substrings in s with size n
    count ← 0
    for i ← 0, n − 2 do
        if s[i] = start then
            for j ← i + 1, n − 1 do
                if s[j] = end then
                    count ← scount + 1
    return count
```

   (b) Design a method that solves the problem with a single left-to-right scan of the string s and present it using pseudo-code. Briefly explain why the complexity is linear in $|s|$.
   ANSWER:

```
function COUNTING(s, n, start, end)
    count ← 0
    countStart ← 0
    for i ← 0, n − 1 do
        if s[i] = start then
            countStart ← countStart + 1
        else if s[i] = end then
            count ← count + countStart
    return count
```

   By counting the number of the starting character, we can get the total count of substrings by adding current *countStart* to the total *count* everytime when an ending character is found, and this process only scans the string one time. So, the complexity is linear in $|s|$

3. Given two nodes v and u in a connected, undirected graph $G = \langle V, E\rangle$, their *distance* is the length of the shortest path between v and u (here each edge is of length 1). The *remoteness* of a node is its distance to the node furthest away, that is, $rem(v) = \max\{dist(v, u) \mid u \in V\}$. A *hub* in G is a node which has minimal remoteness, compared to all other nodes. That is, v is a hub iff $rem(v) = \min\{rem(u) \mid u \in V\}$. For example, b is a hub in the graph on the left (below), whereas b, d and e all are hubs in the graph on the right. Note that a connected graph always has at least one hub.



   (a) Using pseudo-code, design a function that, given a connected undirected graph $\langle V, E\rangle$ and a node $v \in V$, determines all distances from v, that is, for each $u \in V$, gives $dist(v, u)$. Assume that nodes are labelled 1 to n, so that your function can return an array *dist* with $dist[i]$ giving i's distance to v. The algorithm should run in time that is linear in the size of the graph.
   ANSWER:

```
function GETDISTANCES(⟨V, E⟩, v, n)
    initArray(dist, n)                              ▷ create an all zero array
    initQueue(queue)                                ▷ create an empty queue
    inject(queue, v)
```

```
    while queue is non-empty do
        u ← eject(queue)
        for each edge(u, w) do
            if dist[w] = 0 and w ≠ v then
                dist[w] ← dist[u] + 1
                inject(queue, w)
    return dist
```

Use BFS to scan the graph starting from v, never go back to any nodes already discovered. So, after one single scan, the shortest distance could be found.

(b) Design an algorithm that, given a connected undirected graph $\langle V, E\rangle$, identifies a hub (any hub) in the graph. More precisely, use pseudo-code to define a function HUB that takes $\langle V, E\rangle$ as input and returns a node which is a hub. Aim for an $O(n^2)$ algorithm, where n is the size of the graph.

ANSWER:

```
function getHub(⟨V, E⟩, n)
    initArray(remotes, n)                          ▷ create an all zero array
    for each v in V do
        dist = getDistances(⟨V, E⟩, v, n)
        remotes[v] = maxDistance(dist, n)
    return minRemoteNode(remotes, n)
function maxDistance(dist, n)
    max ← 0
    for i ← 1, n do
        if dist[i] > max then
            max = dist[i]
    return max
function minRemoteNode(remotes, n)
    min ← remotes[1]
    node = 1
    for i ← 2, n do
        if remotes[i] < min then
            min ← remotes[i]
            node ← i
    return node
```

Get the HUB by its definition. Since *getDistance* and *maxDistance* are both $O(n)$, after traverse each v, the complexity is $O(n^2)$.

4. Let $\langle V, E\rangle$ be a directed graph. A node $v \in V$ is an *attractor* iff v is a sink, and moreover, for every node $u \in V$ with $u \neq v$, $(u, v) \in E$. Note that a graph can have at most one attractor. We want an algorithm that will identify an attractor in an input graph $\langle V, E\rangle$ if there is one. Assume that a graph is represented as an adjacency matrix A, with the nodes in V labelled 1 to n. The function ATTRACTOR$(A[.,.], n)$ should return i if node i is an attractor, and 0 if the graph has no attractor.

(a) Give an $O(n^2)$ algorithm for the problem, using pseudo-code.

ANSWER:

```
function attractor(A, n)
    attractor ← 0
    initArray(mark, n)                             ▷ create an all zero array
    initArray(attract, n)                          ▷ create an all zero array
    initQueue(queue)                               ▷ create an empty queue
    for i ← 1, n do
```

```
        if mark[i] = 0 then
            inject(queue, i)
            mark[i] ← 1
            while queue is non-empty do
                u ← eject(queue)
                for j ← 1, n do
                    if A[u][j] = 1 then
                        attract[u] ← attract[u] − 1
                        attract[j] ← attract[j] + 1
                        if mark[j] = 0 then
                            inject(queue, j)
                            mark[j] ← 1
    for i ← 1, n do
        if attract[i] = n − 1 then
            attractor ← i
    if A[attractor][attractor] = 1 then          ▷ attractor cannot be self connected
        attractor ← 0
    return attractor
```

By using an *attract* array, for every node in the graph, when an outgoing edge is found, -1 is added to the corresponding bit, and when an incoming edge is found, 1 is added. So, the *attractor* is the one which hold the value of $n-1$ at the end. The algorithm uses BFS, and its complexity is $O(n^2)$

(b) (Harder.) Give an $O(n)$ algorithm for the problem, using pseudo-code. Maximum marks will (only) be given for a solution that is both readable, intelligible, carefully explained and analysed, and, importantly, runs in linear time.

ANSWER:

```
function ATTRACTOR(A, n)
    attractor ← 0
    i ← 1
    j ← 1
    while i ≤ n and j ≤ n do
        if A[i][j] = 1 then
            i ← i + 1
        else
            j ← j + 1
    if i ≤ n & ISATTRACTOR(A, i, n) then
        attractor ← i
    return attractor
function ISATTRACTOR(A, i, n)
    for j ← 1, n do
        if A[i][j] = 1 then return false
        if A[j][i] = 0 and i ≠ j then return false
    return true
```

This algorithm uses "decrease and conquer" principle. We change the problem of finding the attractor of graph with size of n to the problem that prune $n-1$ vertices which are not attractor and check whether the remaining one is an attractor.

As shown below, if the 2nd vertice is the attractor, the matrix should looks like this, according to the definition of attractor. And when we arrive the end along with the path described in the above algorithm, every vertice except the 2nd one could be pruned, since: if $A[i][2] = 1, i \neq 2$, then i could not be the attractor. Likewise, if $A[2][i] = 0, i \neq 2$, then i could not be the attractor. To be more detailed, an attractor i must satisfy that $A[i][j] = 0$, for every node $j \in V$ and $A[j][i] = 1$, for every $j \in V$ with $i \neq j$

So, after the check for the 2nd node, we could know whether there is an attractor, which is the 2nd vertice here, or not.

There are at most $2n$ operations for pruning, and n operations for checking, so the complexity is $O(n)$.

$$\begin{pmatrix} a_{11} & 1 & a_{13} & \dots & a_{1n} \\ 0 & 0 & 0 & \dots & 0 \\ a_{31} & 1 & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & 1 & a_{n3} & \dots & a_{nn} \end{pmatrix}$$

Tiancong Li
24 August 2017