Exercise Session 10

Exercise 1.
(CLRS 22.1-3) The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Describe efficient algorithms for computing G^T from G, for both adjacency-list and adjacency-matrix representations of G. Analyse the running times of your algorithms.

Solution 1.
An algorithm for computing the transpose of G represented with adjacency-lists is described below.

```
ADJACENCYLIST-TRANSPOSE($G$)
1  Initialise a new graph $G^T$ with $G^T.V = G.V$ and no edges.
2  // Populate adjacency-list for $G^T$
3  for each $u \in G.V$
4      for each $v \in G.Adj[u]$
5          LIST-INSERT($G^T.Adj[v], u$)
6  return $G^T$
```

Executing line 1 takes $\Theta(|V|)$ while executing the for loop in lines 3–5 takes $\Theta(|E|)$ because the LIST-INSERT operation takes $\Theta(1)$ time and the number of executions of line 5 is $|E| = \sum_{u \in V} |Adj[u]|$. Therefore the running time of the procedure ADJACENCYLIST-TRANSPOSE is $\Theta(|V| + |E|)$.

An algorithm for computing the transpose of G represented with adjacency-matrix is described below.

```
ADJACENCYMATRIX-TRANSPOSE($G$)
1  Initialise a new graph $G^T$ with $G^T.V = G.V$
2  // Populate adjacency-matrix for $G^T$
3  for $i = 1$ to $|G.V|$
4      for $j = 1$ to $|G.V|$
5          $G^T.A[j, i] = G.A[i, j]$
6  return $G^T$
```

Executing line 1 takes $\Theta(|V|)$ while executing the for loop in lines 3–5 takes $\Theta(|V|^2)$. Therefore the running time of the procedure ADJACENCYMATRIX-TRANSPOSE is $\Theta(|V|^2)$.

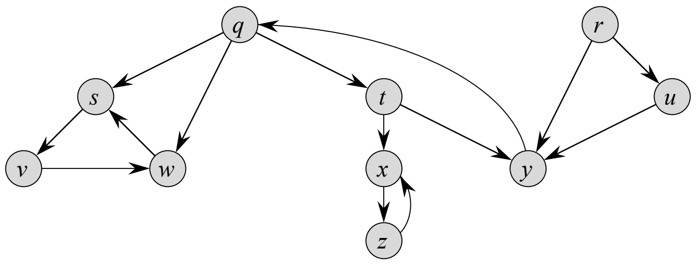
Exercise 2.
The diameter of a tree $T = (V, E)$ is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyse the running time of your algorithm.

Solution 2.
The following algorithm is obtained by suitably modifying the pseudocode of the breadth-first search procedure BFS (see CLRS pp. 595) and it is assumed to be run given the root of the tree. Specifically, DIAMETER(T, s) closely follows the implementation for BFS(T, s) which explores the graph G stating from a source vertex $s \in V$ visiting all vertices that are reachable from s. While doing so, it updates the variable $diam_s$ satisfying the following loop invariant: before each iteration of the loop in lines 11–20 the value of $diam_s$ is greater than or equal to the attribute $u.d$ of any black vertex u. Analogously to the correctness proof for BFS one can can show that DIAMETER(T, s) returns the distance value $\delta(s, u)$ of a vertex that is at maximal distance from s, which corresponds to the diameter of T.

```
DIAMETER-AUX(T, s)
for each $v \in T.V \setminus \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$diam_s = \infty$
$Q = \emptyset$
ENQUEUE(Q, s)
while $Q \neq \emptyset$
    $u$ = DEQUEUE(Q)
    for each $v \in T.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE(Q, v)
    $u.color$ = BLACK
    $diam_s = u.d$
return $diam_s$
```

The running time of DIAMETER(T, s) is $\Theta(|V| + |E|)$, which can be shown using the same argument used in Lecture 10 for BFS.

Exercise 3.
Consider the graph G depicted below.



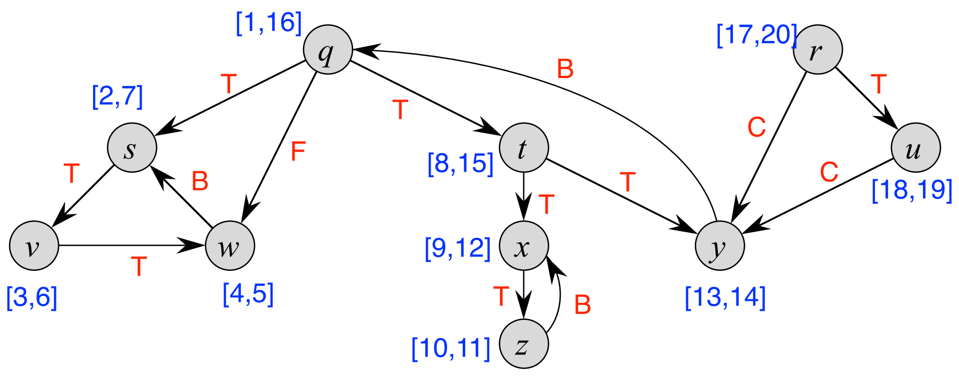
(a) Write the intervals for the discovery time and finishing time of each vertex in the graph obtained by performing a depth-first search visit of G.

(b) Write the corresponding "parenthesization" of the vertices in the sense of Theorem 22.7 in CLRS

(c) Assign with each edge a label T (tree edge), B (back edge), F (forward edge), C (cross edge) corresponding to the classification of edges induced by the DFS visit performed before.

(d) If G admits a topological sorting, then show the result of TOPOLOGICAL-SORT(G).

Solution 3.
(a) (c) The figure below shows the intervals for discovery time and finishing time for each vertex (in blue) and the classification of edges corresponding to that specific visit. Note that another choice for the order of visit of vertices may yield a different outcome.



(b) The “parenthesization” corresponding to the DFS visit described in the figure above is

$$(q\,(s\,(v\,(w\,w)\,v)\,s)\,(t\,(x\,(z\,z)\,x)\,(y\,y)\,t)\,q)\,(r\,(u\,u)\,r)\,.$$

(d) G is not acyclic, e.g., the path $x \to z \to x$ describes a cycle. The fact that G is not acyclic can also be evinced by the fact that some edges in the graph were classified as back edges (see Lemma 22.11 CLRS). Therefore G does not admit a topological sorting. Another

Exercise 4.
(CLRS 22.4-5) Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(|V| + |E|)$. What happens to this algorithm if G has cycles?

Solution 4.
The pseudocode implementing the algorithm described above is the following.

```
TOPOLOGICAL-SORT($G$)
 1  create the array $in\text{-}degree[1..|V|]$
 2  compute $G^T$
 3  for each $v \in V$
 4      $in\text{-}degree[v] = |G^T.Adj[v]|$
 5  $Q = \emptyset$
 6  for each $v \in V$ such that $in\text{-}degree[v] == 0$
 7      ENQUEUE($Q, v$)
 8  while $Q \neq \emptyset$
 9      $u$ = DEQUEUE(Q)
10      print $u$
11      for each $v \in G.Adj[u]$
12          $in\text{-}degree[v] = in\text{-}degree[v] - 1$
13          if $in\text{-}degree[v] == 0$
14              ENQUEUE($Q, v$)
```

To find vertices of in-degree 0 we compute the transpose of G then we store the length of each adjacency list in the array *in-degree*. From line 5 we start collecting in a queue Q all vertices having in-degree equal to 0. The while loop in lines 8–14 picks at every iteration a vertex which has in-degree 0 and “deletes” it form the graph by updating the array *in-degree* as if all its outgoing edges where removed. If during this “deletion” some vertex tuns out to have in-degree equal to 0, then the vertex is enqueued in Q (line 14). Notice that vertices which have already in-degree equal to 0 will never be updated again because no other vertex u has one vertex $v \in Adj[u]$ such that $v \in Q$. The use of a queue Q for storing vertices ensures that they will processed in the same order as they become vertices with 0 in-degree. This guarantees that the printing order of the vertices is correct.

Running time Analysis. Computing the transpose of G takes $\Theta(|V|+|E|)$. The for loop in lines 3–4 takes time $\Theta(\sum_{v\in V}|G^T.Adj[v]|) = \Theta(|E^T|) = \Theta(|E|))$. The initialisation of the queue (lines 5–7) takes $\Theta(|V|)$. As for the while loop in lines 8–14 notice that each vertex enters in the queue at most once and updating the in-degree array is performed on for each vertex in the adjacency list $Adj[u]$ of the vertex u which has been just picked out form Q. Hence, the execution of the while loop takes $O(|V|+|E|)$. Summing up we have that TOPOLOGICAL-SORT(G) runs in time $O(|V|+|E|)$.

If the are no cycles, all vertices will be eventually inserted in Q and printed out. If there are cycles, not all vertices will be printed, because some in-degrees never become 0.

Exercise 5.
Assume G is a directed acyclic graph. Give an efficient algorithm to compute the graph of strongly connected components of G, and analyse the running time of your algorithm.

Solution 5.
By definition of strongly connected component two distinct vertices $u, v \in V$ belong to the same component if there is a cycle containing both u and v. Since G is assumed to be a DAG each component will be a singleton containing exactly one vertex. Therefore, returning the graph G will suffice which takes $\Theta(1)$ time.

Exercise 6.
(CLRS 22.5-1) How can the number of strongly connected components of a graph change if a new edge is added?

Solution 6.
Consider the graph of $G = (V, E)$ and one pair of vertices $(u, v) \notin E$ which is added to G. There are tree cases

1. if both u and v belong to the same component then the SCCs after the insertion of (u, v) do not change
2. if u and v belong to two distinct components, say C_u and C_v and the component graph has already an edge (C_u, C_v), then the SCCs after the insertion of (u, v) in G do not change.
3. if u and v belongs to two distinct components, say C_u and C_v and (C_u, C_v) is not an edge of the component graph, then after the insertion of (u, v) the two components will merge forming the SCC $C_u \cup C_v$. All other components remain the same.