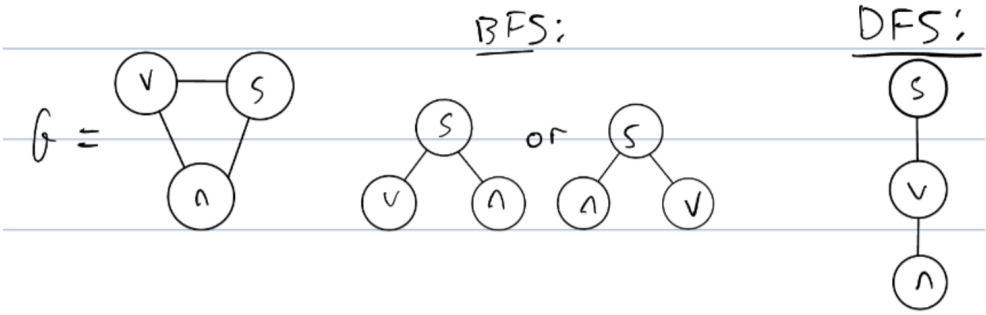
[redacted]

Q1

$G = (V, E)$ is a Directed Graph that is Connected. Let there be an arbitrary Node $s \in V$. (applicable for or parts of the question)

$A)$ When running $BFS(G, s)$ for any permutation we will get that v is a leaf.

The Claim is incorrect, if we run $DFS(G)$ then v won't necessarily be a leaf.
Counter Example:
for $s \neq c \in V$, v is a leaf for any execution of $DFS(G, s)$. In the following example when running $DFS(G)$ we get a permutation where v isn't a leaf.



$B)$ The claim is correct, for a node v, when running $DFS(G)$ we get that v is a leaf for any result that $DFS(G)$ gives us.
Let's assume by contradiction that there exists an arrangement of the nodes that when running $BFS(G, s)$ such that in the resulting tree- v will not be a leaf.

Since the Graph is directed and connected, The BFS will return a shortest path tree from the rooted node s as shown in the Lecture.
From the assumption we get there exists $l \in V : (v, l) \in E'$ and $l.\pi = v$. Therefore, their exists an execution of $DFS(G)$ where v won't be a leaf and will be the parent of l.
This is in contradiction to the fact that v is a leaf for any execution of $DFS(G)$. Therefore, for every execution of $BFS(G, s)$ we will get that v is a leaf ■

$C)$ The claim is True, From the definition of $BFS(G, s)$ which we will mark the tree returned as T_1 it will return a shortest paths tree from the rooted node s (single tree as It's given that the tree is directed and connected). Therefore, the Height of the tree which we will mark as l_1 is the distance between s and a leaf on the deepest level of the tree which we will mark this node, or one of the nodes as $v \in V$ (since it's not necessarily unique).
Thus, if we run $DFS(G, s)$ we will get that the minimal distance between s to v is at least l_1 since by definition earlier, l_1 is the shortest distance between s and v, Therefore, the tree rooted at s when running $DFS(G, s)$ (which tree we will mark as $T_2)mustbeatleast\$l_1$.
$\implies T_1 - Height \leq T_2 - Height$

Q3

1. According to the given algorithm, in each iteration of the algorithm the Nodes that we add to the list are the ones with an $in\ degree$ of 0. Therefore, in a directed graph, all the nodes $u_1, \ldots, u_k \in V$ (where $k > 0, k \in \mathbb{Z}$ while the graph is not empty) that have an $in\ degree$ of 0 will be added to the list. Each of these Nodes may have a edge with any other Node $v \in V/\{u_1, \ldots, u_k\}$ but there doesn't exist an edge- $\forall i \in [k] : (v, u_i)$ and u_i will be placed in the List before any other v, which satisfies that if G contains an edge (u, v) then u will be in the list before v. We also remove the edges from the nodes we added to the list so we don't search through them again.
   Then in the next iteration we are left with $V/\{u_1, \ldots, u_k\}$ Nodes.
   We repeat this process inductively until we run out of Nodes in the graph. As mentioned above this will satisfy the definition of a topological sort algorithm of a DAG since at any stage of the algorithm it's clear that for any Node where G contains an edge (u, v) such that u will appear before v in the ordering of the list.
2.

The loop runs on all the nodes which is $\mathcal{O}(V)$ time.
The second double loop on lines 6,7 runs on all the Nodes and their arcs and will run $\mathcal{O}(V + E)$
The while loop will also be $\mathcal{O}$ (V+E) since each node only get's added to the remove queue once, and each node we iterate through each arc on line 15 at most once. Therefore, overall the runtime of the function is $\mathcal{O}(V + E)$ and is a topological sort.
Code for part A which is equivalent to a topological sort.

```
1. TopologicalSort(G)
2.     L = {}
3.     remove = NewQueue
3.     for all v ∈ G.V Do :
4.         v.inDeg = 0
5.         v.list = {}
6.     for all u ∈ G.V Do :
7.         for all v ∈ G.Adj[u] :
8.             v.inDeg+ = 1
9.             u.list.append(v)
10.    for all v ∈ G.V Do :
11.        if v.inDeg == 0 :
12.            remove.add(v)
13.    while remove ≠ Nil Do :
14.        l = remove.pop()
15.        for all v ∈ l.list Do :
16.            v.inDeg− = 1
17.            if v.inDeg == 0 :
18.                remove.add(v)
19.        L.append(l)
20.        remove arcs from l
21.    return L
```

This algorithm is a topological sort that implements the steps in part 1.
A is line 2. where we initialize an empty list
B is lines 3, 11, 17 where we find the nodes with an in degree of 0
C is line the while loop on line 14 where we iterate the process until the graph is empty
D is line 21 where we return the list.

שאלה 4

1. נשתמש ב *dfs*_עם התאמות קטנות לבעיה שלנו.

ניצור רשימת שכנויות חדשה של הגרף $\vec{G}$ כך ש $\vec{G}.adj[v]$ הן כל הקשתות שיוצאות מ*v*. לכן נשנה את הפונקציה *DFS_VISIT* כך ש- אם *v* ברשימת השכנויות של *u* וכאשר סרקנו את רשימת השכנויות של *u* *v* היה לבן נוסיף את *v* ל - $\vec{G}.adj[u]$.

וכן אם *v* ברשימת השכנויות של *u* וכאשר סרקנו את רשימת השכנויות של *u* *v* היה אפור אז במידה ו*v* איננו הורה של *u* משמע ש *v* צאצא של *u* ולכן נוסיף את *v* בחינת ל ל - $\vec{G}.adj[u]$.

3. → נניח כי לא קיימת קשת אחורית מצאצא של *v* לאב קדמון של *u* ב-$\vec{G}$, לפי משפט מההרצאה בגרף לא מכוון כל הקשתות הן קשתות עץ או קשתות אחוריות,

לכן נקבל שכל הקשתות בתת העץ המושרה ב- *v* הן קשתות עץ ולכן הן לא מתחברות עם צמתים שלא בתת העץ , ולכן אם נסיר את הקשת (u,v) לא יהיה ניתן להגיע מהצומת *u* אל תת העץ כלומר הגרף לא יהיה קשיר ולכן הקשת (u,v) הינה קשת שוברת.

←נניח כי הקשת (u,v) היא קשת עץ וקשת שוברת ונניח בשלילה שקיימת קשת אחורית מצאצא של *v* לאב קדמון של *u* ב- $\vec{G}$.

נניח בשלילה כי *v* הוא אב קדמון של *u* ו *u* הוא צאצא של *v* אזי מתקיים כי לפי ההנחה כי הקשת (u,v) היא קשת אחורית וגם קשת עץ וזה כמובן לא אפשרי.

כלומר קיים מסלול $v \leftarrow v$ של צאצא $\leftarrow u$ של קדמון אב $\leftarrow u$ ולכן אם נסיר את הקשת (u,v) עדיין נוכל ללכת במסלול המוגדר ולכן קשת זו לא שוברת , סתירה.

Q4

$2)$
Prove that for every $Broken\ arc$ in G will be classified as a $tree\ arc$ when executing $DFS(G)$

Let's take an arbitrary $(v, u) \in E$ that's a $brokeB\ Edge$ and where $v, u \in V$.

let's run $DFS(G)$. We now have a graph $G_\pi = (V_\pi, E_\pi)$ where $V = V_\pi$ and $E_\pi = \{(v.\pi, v) | v \in V, v.\pi \neq NIL\}$ (with arcs going both ways)
We proved in the TA that every arc is either a $tree\ edge$ or a $back\ edge$ (DFS TA slide 14)

Since (u, v) is a $broken\ edge$ in $E \implies$ We get two subgraphs which are connected, and when (u, v) is removed then these two graphs where one is connected by all the Nodes which are connected to u and the second by all the nodes connected to v. These two graphs let's call them $G_1 = (V_1, E_1), G_2 = (V_2, E_2) : V_1 \cap V_2 = \emptyset$ ^ $V_1 \cup V_2 = V$ and since the the edge is broken we also get that $E_1 \cap E_2 = \emptyset$ ^ $E_1 \cup E_2 = E \implies$ Therefore, u and v where the arc between them is broken, is the only connection between these two Nodes (Otherwise the graph will still be connected . in contradiction to the definition of a $broken\ edge) \implies$
In an execution of $DFS(G)$, if v can is discovered by $u \implies v.color = white \implies$ by definition (u, v) is a $Tree\ Edge$
Otherwise, u will be discovered by v for the first time $\implies v.color = white \implies$ by definition that (v, u) is a $Tree\ Edge$

$\implies$ Any $Broken\ Edge\ (v, u)$ will be classified as a $Tree\ Edge$ when running $DFS(G)$

$4)$ Run SCC algorithm and if the amount of vertices returned is greater than 1 then the Graph is "Broken", since SCC checks how many strong components (is undirected which is the same as saying directed graph with arrows in both directions) the graph has, and if we disconnect the "Broken Edge" that defines how many connected components the graph will have which will become 2 if the Edge is removed otherwise will stay 1. SCC runs at runtime of $\Theta(V + E)$ as shown in the Lecture ■

שאלה 5

האלגוריתם שלנו מאוד דומה ל BFS, ההבדל הוא זה שאנו לא מתיחסים לשדה הפאי ושהוספנו תנאי למסלול לכן מבחינת סיבוכיות זמן בדומה לBFS_האתחול יהיה $o(n)$ זמן , בנוסף כל צומת תוכנס לתור לכל היותר פעמיים וכן תוצא מהתור לכל היותר פעמיים כלומר $o(n)$ לכל ענייני הוצאה והכנסה מהתור . הגודל הכולל של רשימות השכנויות הינו $o(m)$ ולכן בסה"כ זמן הריצה של האלגוריתם הינו $o(n+m)$.

מבחינת נכונות האלגוריתם מתקיים כי האלגוריתם המתואר הינו *BFS* עם תנאי על אופי המסלולים כלומר לפי הוכחת הנכונות בהרצאה על *BFS* מתקיים כי כל המסלולים שיתקבלו באלגוריתם שלנו יקיימו ש*v.d_inc = δ (s,v)* לכל *v* וכן שאם צומת שאינה *s* נגישה מ*s* אז שדה הפאי שלה הוא הצומת שקודם לו במסלול פשוט מ*s* ל*v*.

Q5

```
for all V ∈ G.V − S Do :
    v.color = white
    v.d_inc = infinity
s.color = gray
s.d_inc = 0
Q = {}
Enqueue(Q, S)
While Q.length ≠ 0 Do :
    u = Dequeue(Q)
    for all v ∈ G.adj[u] do :
        if v.color == white and f[v] = f[u] do :
            v.color = gray
            v.d_inc = u.d_inc = 1
            Enqueue(Q, v)
    u.color = black
```