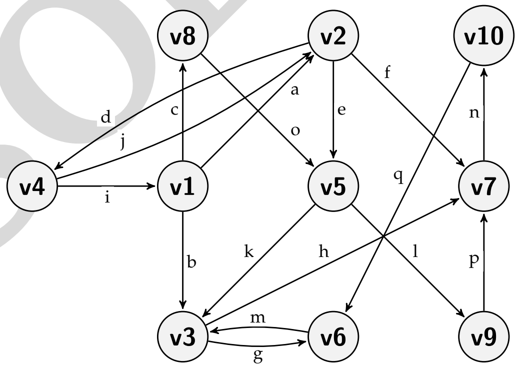
problem 1 *DFS*

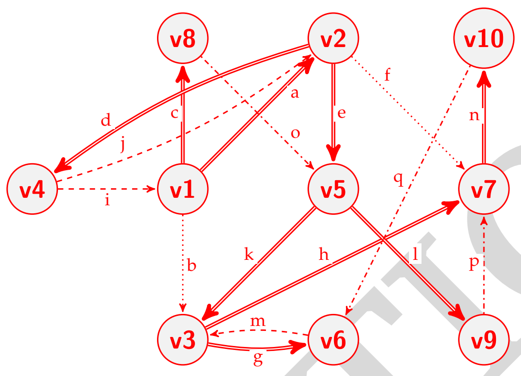
Consider the following graph:



Graph G

Perform DFS on the graph, using the recursive DFS algorithm. Start at vertex v_1. When processing a given node, and there are multiple potential outgoing edges (i.e., adjacent nodes), always choose the one that is *earlier* alphabetically. Below you will need to fill in the tables to indicate the start and finish time for each node, as well as classify each edge as one of the four types: tree, back, descendant, and cross.

Solution:
In the solution graph below, double edges are tree edges, dashed are back edges, dotted are forward/descendant edges, and dash-dotted are cross edges.



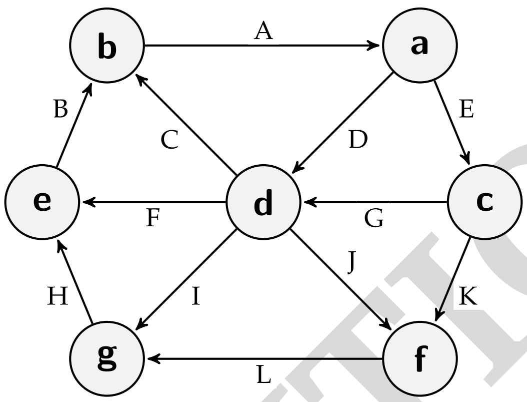
Graph $G_{solution}$

Node	Start time	End time
v1	1	20
v2	2	17
v3	6	13
v4	3	4
v5	5	16
v6	7	8
v7	9	12
v8	18	19
v9	14	15
v10	10	11

Edge	Start node	End node	Edge type
a	v1	v2	tree
b	v1	v3	descendant
c	v1	v8	tree
d	v2	v4	tree
e	v2	v5	tree
f	v2	v7	descendant
g	v3	v6	tree
h	v3	v7	tree
i	v4	v1	back
j	v4	v2	back
k	v5	v3	tree
l	v5	v9	tree
m	v6	v3	back
n	v7	v10	tree
o	v8	v5	cross
p	v9	v7	cross
q	v10	v6	cross

PROBLEM 2 *Dijkstra's Shortest Path*

Consider the following graph:



Graph G

You are to perform Dijkstra's shortest path algorithm on the graph. The graph above does not list weights – to get the weights for the graph, see the URL listed on the Canvas landing page. Note that you will have different edge weights and a different starting node than others, and you must answer the question using the weights assigned to you.

1. Edit the above graph, and put the weights in for each edge.

   Solution: (the solution is the edited graph above)

2. Perform Dijkstra's shortest path, filling in the table below.

   Solution:

Node	Found?	Distance	Path
a			(type here)
b			
c			
d			
e			
f			
g			

3. List which vertex had their path updated more than once – meaning that vertex's distance was updated again after being set the first time. For each of those vertices, list each value it had.

   Solution: This is specific to each student

problem 3 *Algorithm Creation*

Consider a graph $G = (V, E)$ that is a standard binary tree. Each node, other than the root node, has indegree of 1 (an edge coming in from the parent), and up two two outgoing edges to the children, which we'll call $left$ and $right$. As this is a tree, there are no cycles. In fact, it's a DAG (directed acyclic graph); it's is also (weakly) connected.

Consider the recursive DFS algorithm presented in class, and found in CLRS:

```
DFS($G$)
1   for each vertex $u \in G.V$
2       do
3           $u.color = $ WHITE
4           $u.\pi = $ NIL
5   $time = 0$
6   for each vertex $u \in G.V$
7       do
8           if $u.color == $ WHITE
9               then
10                  DFS-Visit($G, u$)
```

```
DFS-Visit($G, u$)
1   $time = time + 1$
2   $u.d = time$
3   $u.color = $ GRAY
4   for each $v \in G.Adj[u]$
5       do
6           if $v.color == $ WHITE
7               then
8                   $v.\pi = u$
9                   DFS-Visit($G, v$)
10  $u.color = $ BLACK
11  $time = time + 1$
12  $u.f = time$
```

Suppose we want to change the location of where we set $u.d = time$ so that the discovery times increase in a pre-order, in-order, or post-order traversal orders. How would you modify the DFS(G) and/or the DFS-Visit(G, v) algorithms above to have these discovery times increase in a pre-order traversal of the tree? Similarly, how would you modify the code for an in-order traversal? Also, post-order? Your explanation can be in English prose – you do not have to use the formal notation shown above.

Solution: Since it's connected, we only need DFS-Visit. The **for** loop would need to consider the outgoing edges in *left*, *right* order. The time set for the current node (the $u.d$ time) should be set as it is (line 2) for pre-order, at the end (line 12) for post-order, and between the calls to DFS-Visit($G, v.left$) and DFS-Visit($G, v.right$) for in-order. The times stored in $u.d$ are then the traversal orders.