Q1

a. False
b. True
c. False
d. True
e. True
f. False
g. True
h. False

Q2

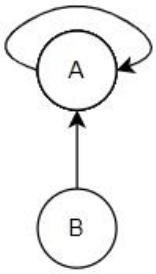
1) Initialization: Direct all nodes on the path to the root. For each vertex $u_i \in [A, Q]$, each vertex direct to themselves.

2) Find and Union: For each edge{u,v}, find the root of u and v. If roots of u and v are different, make smaller tree point to bigger one's root. If u and v are already in a same tree, and they all point to the root, then nothing will be changed.

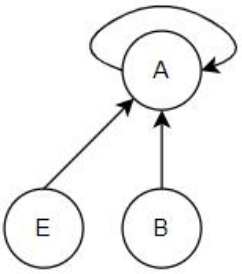
pseudo-code:

```
Initialize(int N)
    setsize = new int[N+1];
    parent = new int [N+1];
    for (int parent[e] = 0; e=1; e <= N; e++)
        parent[e] = 0;
        setsize[e] = 1;

int Find(int e)
    if (parent[e] == 0)
        return e
    else
        parent[e] = Find(parent[e])
        return parent[e]

Union(int i, int j)
    i = find(i);
     j = find(j);
    if setsize[i] < setsize[j]
    then
        setsize[j] += setsize[i];
        parent[i] = j;
    else
        setsize[i] += setsize[j];
        parent[j] = i ;
```

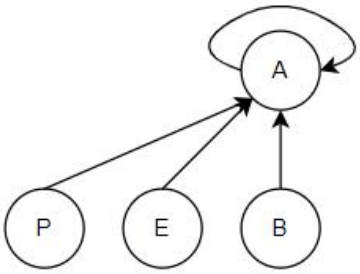
union(A, B)



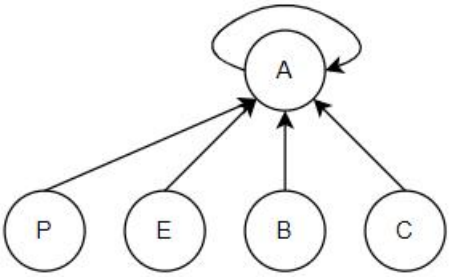
union(A, E)



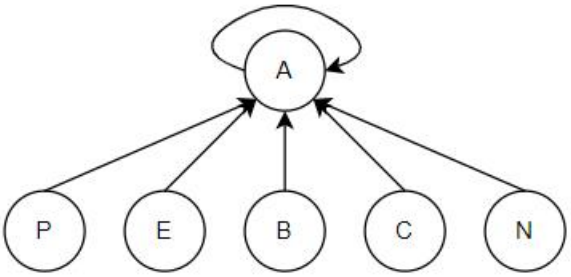
union(A, P)



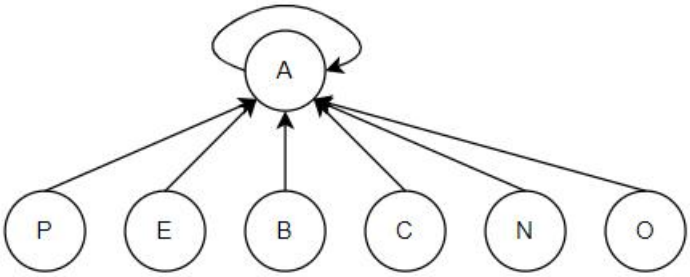
union(B, C)



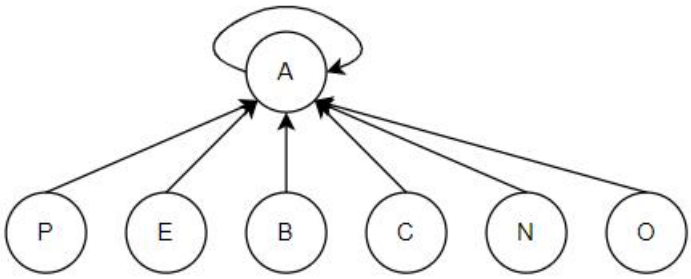
union(C, N)



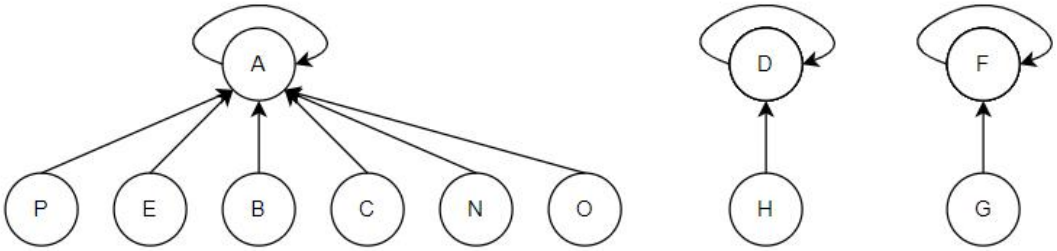
union(C, O)



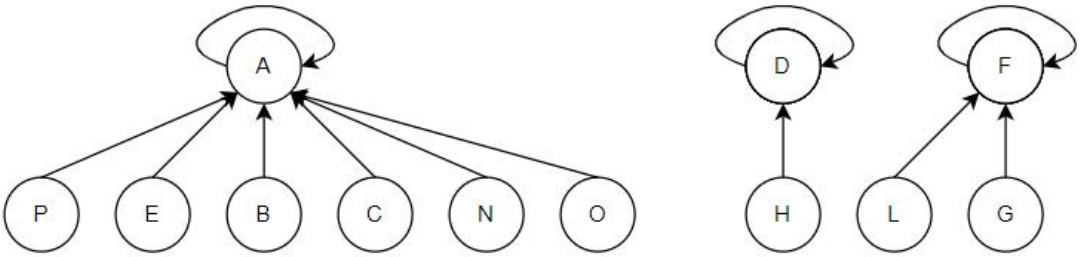
union(D, H)



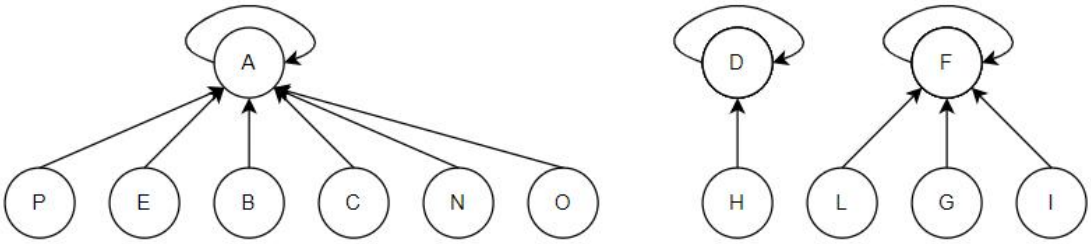
union(F, G)



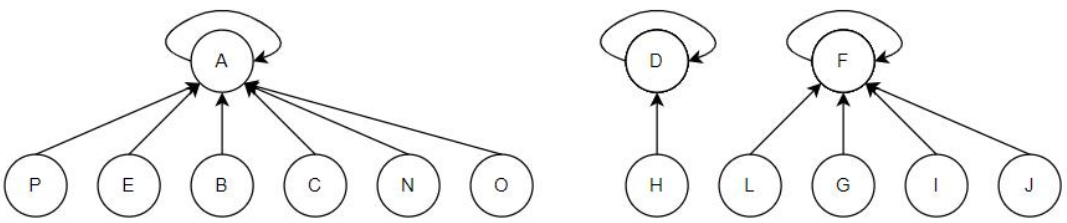
union(F, L)



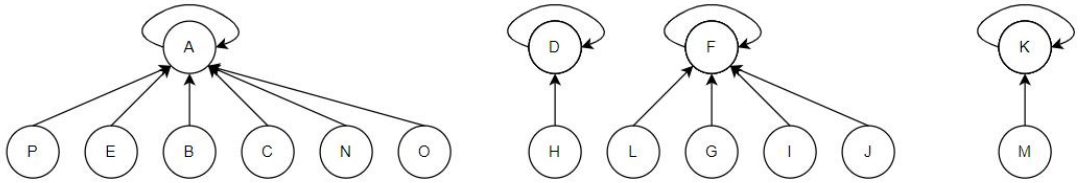
union(G, I)



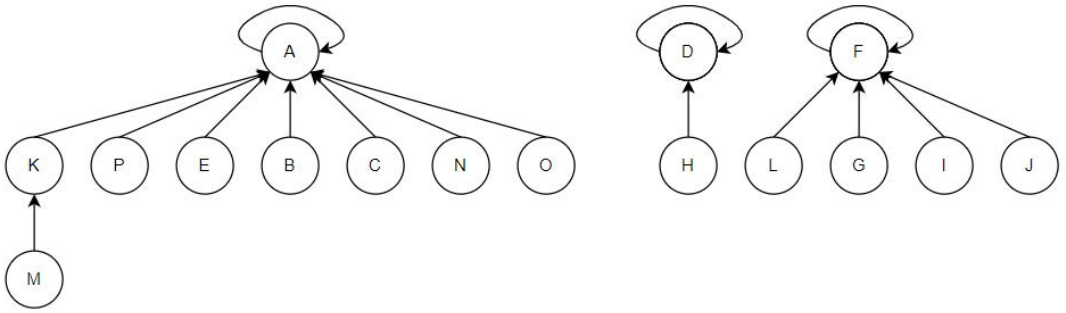
union(I, J)



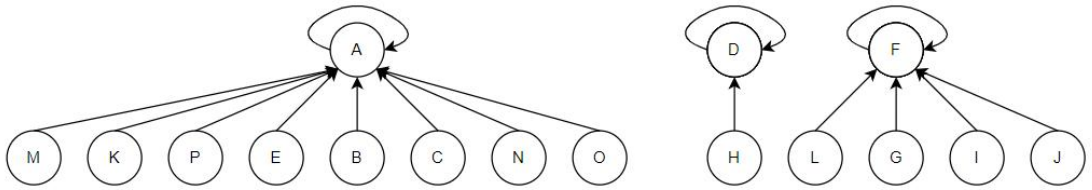
union(K, M)



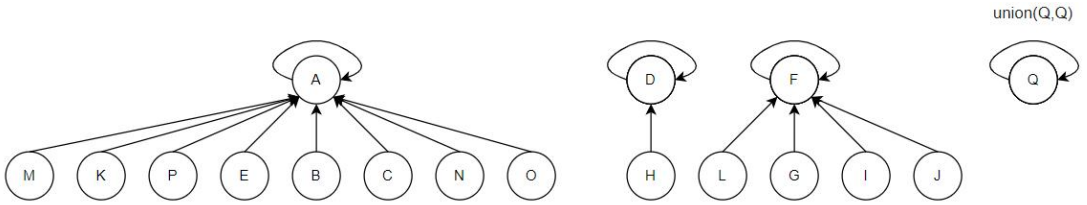
union(K, P)



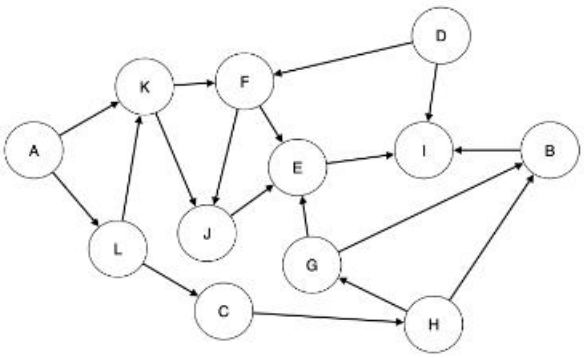
union(M, P)



union(Q,Q)



Q3



vertex	in-degree
A	0
B	2
C	1
D	0
E	3
F	2
G	1
H	1
I	3
J	2
K	2
L	1

Push vertices which do not have in-degree
queue: A, D

vertex	in-degree
A	0
B	2
C	1
D	0
E	3
F	2
G	1
H	1
I	3
J	2
K	2-1=1
L	1-1=0

Pop the front of the queue
A has 2 neighbors: K, L
Decrement their in-degree
Since L has no in-degree, push L into the queue
queue: A, D, L

vertex	in-degree
A	0
B	2
C	1
D	0
E	3
F	2-1=1
G	1
H	1
I	3-1=2
J	2
K	1
L	0

Pop the front of the queue
D has 2 neighbors: F, I
Decrement their in-degree
F and I still have in-degrees
queue: A, D, L

vertex	in-degree
A	0
B	2
C	1-1=0
D	0
E	3
F	1
G	1
H	1
I	2
J	2
K	1-1=0
L	0

Pop the front of the queue
L has 2 neighbors: K, C
Decrement their in-degree
Since K, C have no in-degree, push C, K into the queue
queue: A, D, L, C, K

vertex	in-degree
A	0
B	2
C	0
D	0
E	3
F	1
G	1
H	1-1=0
I	2
J	2
K	0
L	0

Pop the front of the queue
C has 1 neighbors: H
Decrement its in-degree
Since H has no in-degree, push H into the queue
queue: A, D, L, C, K, H

vertex	in-degree
A	0
B	2
C	0
D	0
E	3
F	1-1=0
G	1
H	0
I	2
J	2-1=1
K	0
L	0

Pop the front of the queue
K has 2 neighbors: F, J
Decrement their in-degree
Since F has no in-degree, push F into the queue
queue: A, D, L, C, K, H, F

vertex	in-degree
A	0
B	2-1=1
C	0
D	0
E	3
F	0
G	1-1=0
H	0
I	2
J	1
K	0
L	0

Pop the front of the queue
H has 2 neighbors: B, G
Decrement their in-degree
Since G has no in-degree, push G into the queue
queue: A, D, L, C, K, H, F, G

vertex	in-degree
A	0
B	1
C	0
D	0
E	3-1=2
F	0
G	0
H	0
I	2
J	1-1=0
K	0
L	0

Pop the front of the queue
F has 2 neighbors: J, E
Decrement their in-degree
Since J has no in-degree, push J into the queue
queue: A, D, L, C, K, H, F, G, J

vertex	in-degree
A	0
B	1-1=0
C	0
D	0
E	2-1=1
F	0
G	0
H	0
I	2
J	0
K	0
L	0

Pop the front of the queue
G has 2 neighbors: B, E
Decrement their in-degree
Since B has no in-degree, push B into the queue
queue: A, D, L, C, K, H, F, G, J, B

vertex	in-degree
A	0
B	0
C	0
D	0
E	1-1=0
F	0
G	0
H	0
I	2
J	0
K	0
L	0

Pop the front of the queue
J has 1 neighbor: E
Decrement its in-degree
Since E has no in-degree, push E into the queue
queue: A, D, L, C, K, H, F, G, J, B, E

vertex	in-degree
A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	2-1=1
J	0
K	0
L	0

Pop the front of the queue
B has 1 neighbor: I
Decrement its in-degree
I still has 1 in-degree
queue: A, D, L, C, K, H, F, G, J, B, E

vertex	in-degree
A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	1-1=0
J	0
K	0
L	0

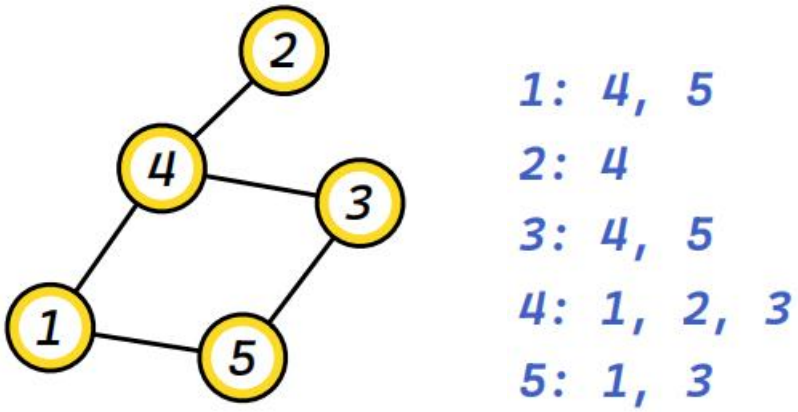
Pop the front of the queue
E has 1 neighbor: I
Decrement its in-degree
Since I has no in-degree, push I into the queue
queue: A, D, L, C, K, H, F, G, J, B, E, I

vertex	in-degree
A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Pop the front of the queue
I has no neighbor
queue: A, D, L, C, K, H, F, G, J, B, E, I
Then the queue is empty

Therefore, the topological order of vertices in the graph is A, D, L, C, K, H, F, G, J, B, E, I.

Q4
we can store the graph into an adjacency list. Store all neighbours of a vertex into the list of this vertex.
e.g.:



1) Scanning all neighbours of a given vertex
We can search each vertex in the list of the given vertex in the adjacency list.
The time complexity of this operation is O(deg(v)), if the given vertex is v.

2) Inserting a new edge that does not exist in the original graph
For inserting a new edge, we can know 2 vertices which linked by the new edge. For each of 2 vertices, add itself to the list of the other.

e.g.: add edge(2,3), add vertex 2 to the list of vertex 3, and add vertex 3 to the list of vertex 2

The time complexity of this operation is O(1)

3) Deleting a vertex from the graph, including all edges related to it

To delete a vertex v from the graph, including all edges related to it, firstly, we should scan each vertex in the list of the vertex v in the adjacency list, and delete all edges which is in the list of v in the adjacency list. Its time complexity is O(deg(v)). Then scan the edge of each vertex in the list of the vertex v, and delete the edge related to vertex v. In the worst case, it will traverse all edges except the edges deleting at first in the list of the vertex v in the adjacency list. Since it is an undirected, unweighted graph with n vertices and m edges, each edge records twice in the adjacency list, so its worst-case time complexity is O(2m-deg(v)).

The time complexity of this operation is O(deg(v)) + O(2m-deg(v)) = O(m)

We store the graph into an adjacency list, since there is a list for each vertex, it takes O(n) space. Since the list of each vertex includes all neighbour of this vertex, that we know the sum of all in-degree of all vertices is twice to the number of edges, it takes O(2m) space. Hence, its space complexity is O(n)+O(2m) = O(n+m).

Q5

pseudocode:

```
blood DFS(dict graph, int vertex_id, int parent, list visited)
    visited[vertex_id] = True;
    for neighbor in graph[vertex_id]
        if not visited[neighbor]
        then
            if DFS(graph, neighbor, vertex_id, visited)
            then
                return True;
        elif neighbor != parent
        then
            return True;
    return False

blood HasCycle(dict graph, int given_vertex)
    visited = [False] * len(graph);
    parent = -1;
    result = DFS(graph, given_vertex, parent, visited);
    return result;
```

algorithm:
In the HasCycle function, initialize a list to record visited vertices. All False in visited list means all vertices were not visited. And initialize a parent as -1, that means there is not parent for the given query vertex. Then query if there is a cycle, and return the result.
Then use DFS to traverse the graph, recording the vertices on the current path, and if a vertex that is already on the current path is found during DFS, there is a cycle. In the DFS function, set the current vertex as visited, and scan all neighbours of the current vertex. If the neighbour is not visited, then traverse on the current path; otherwise, if the neighbour is visited, and the neighbour is not parent of the current vertex, then return True, that means there is a cycle. After traversing all vertex that can be connected with the given query vertex, if there is not a cycle, then return False.

time complexity:
-In the HasCycle function, since graph is with n vertices and m edges, len(graph) is n. Hence, initializing visited list takes O(n).
-For the DFS function, set the current vertex be visited, visited[vertex_id] = True, takes O(1). And for scanning all neighbours of the current vertex, it takes O(deg(v)). Since we need to traverse all vertices in the worse case, and there is n vertices, so all vertices will be visited, takes nO(1) = O(n), and all neighbours of each vertex will be scanned, takes nO(deg(v)). Since deg(v) means the in-degree of the current vertex, and we know the sum of all in-degree of all vertices is twice to the number of edges, so nO(deg(v)) = O(2m) = O(m). Hence, the DFS function takes O(n)+O(m) = O(n+m).

For total time complexity is O(n)+O(n+m) = O(2n+m) = O(n+m).

Q6

pseudocode:

```
int ShortestDistance(dict graph, int start, int target)
    if (start == target)
        return 0;

    visited = [False] * len(graph);
    distances = [-1] * len(graph);
    pointer= [start];

    visited[start] = True;
    distances[start] = 0;

    while (len(pointer) != 0)
        current = pointer.pop(0);
        for neighbor in graph[current ]
            if not visited[neighbor]
            then
                visited[neighbor] = True;
                distances[neighbor] = distances[current] + 1;
                pointer.append(neighbor);
                if (neighbor == target)
                then
                    return distances[neighbor];
    return -1;
```

algorithm:

For the ShortestDistance function, we need to check if the target is same as the start at first. If their are same, return the distance is 0. Then initialize a list to record visited vertices, all False in visited list means all vertices were not visited. And initialize a list of distances, all -1in distances list means there is no distance from start vertices to all vertices. Also initialize pointer list to record vertices that be processed to simulate FIFO, and at beginning, there is only start vertex needs to be processed. Then traverse all vertices by using BFS to find the shortest distance. While the pointer list is not empty, set the first vertex in pointer list as the current vertex, and scan all neighbours of the current vertex. For each unvisited neighbour, set it as visited, since this neighbour is visited; set the distance of this neighbour as current vertex's distance + 1, since there is a unvisited edge between the current vertex and the unvisited neighbour; and add this neighbour to pointer list, that means set the unvisited neighbour to be processed, then now the pointer list can be FIFO. If this neighbour is target vertex, then return the distance between this neighbour to start target, that means this is the shortest distance; otherwise, continue traverse all vertices by using BFS to find the shortest distance. Finally, if it can not find the target vertex after traversing all vertices, then return -1, that mean there is no distance.

time complexity:

-To check if the target is same as the start at first, it takes O(1).

-Assume the graph with n vertices and m edges, initialize the visited list and distances list both take O(n).

-For the while loop, it can loop n times in the worse case, finding the target vertex after processing each vertex.

For the pointer.pop(0), to return and delete the first element in pointer list, pop() need to move all elements in the pointer. In the worse case, it takes O(n).

For scanning all neighbours of the current vertex, it takes O(deg(v)).

Since it can loop n times in the worse case, looping n times pop(0) takes nO(n) = $O(n^2)$, and all neighbours of each vertex will be scanned, takes nO(deg(v)). Since deg(v) means the in-degree of the current vertex, and we know the sum of all in-degree of all vertices is twice to the number of edges, so nO(deg(v)) = O(2m) = O(m). So the BFS takes $O(n^2 + m)$

For the total time complexity is $O(1)+O(n)+O(n^2 + m) = O(n^2 + m)$