Name: Md. Habibur Rahman Rony
Student ID: 984582
Weekday: Week 3- Day 13

Answer to the Q. No.1:

```
Algorithm initResult(G)
S<-new Sequence

Algorithm preComponentVisit(G,v)
S.insertLast(v)

Algorithm result(G)
return S
```

Answer to the Q. No.2(a):

```
Algorithm BFS(G)
Input graph G
Output labeling of the edges and partition
of the vertices of G

    initResult( G )
     for all u in G.vertices( )
       preInitVertex(u)
       setLabel(u, UNEXPLORED)
     for all e in G.edges()
       preInitEdge(e)
       setLabel(e, UNEXPLORED)
     for all v in G.vertices()
          if getLabel(v) = UNEXPLORED
             preComponentVisit(G, v)
             BFS(G, v)
             postComponentVisit(G,v)

     result(G)
```

```
Algorithm BFS(G, s)

L <- new empty List
L.insertLast(s)
setLabel(s, VISITED)

while !L.isEmpty()
    v <- L.remove (L.first())
    vertexVisit(v)
    for all e in G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED then
            w <- opposite(v,e)
            if getLabel(w) =  UNEXPLORED then
                preDiscoveryTraversal(G, v, e, w)
                setLabel(e, DISCOVERY)
                setLabel(w, VISITED)
                L.insertLast(w)
                  postDiscoveryTraversal(G, v, e, w)
          else
            setLabel(e, CROSS)
             crossTraversal(G,v,e,w)

finishVertexVisit(G,s)
```

Answer to the Q. No.2(b):

```
Algorithm findPathBFS(G,u,v)
S<-new Sequence
path<-null
pathFound<-false
z<-v
for all n in G.vertices() do
        setLabel(p,UNEXPLORED)
for all l in G.edges() do
```

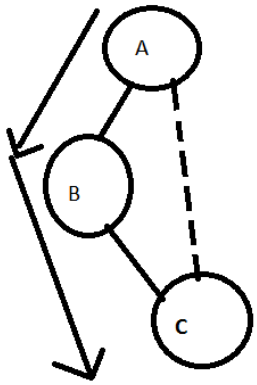
```
        setLabel(l,UNEXPLORED)
BFS(G,u)

if !pathFound = false then
  return NO_SUCH_PATH
else
  return path

Algorithm vertexVisit(v)
  if v=s then

     v.setPath(v)  {path is a property of node}


Algorithm preDiscoveryTraversal(G, v, e, w)
if !pathFound then
  w.setPath(v.getPath()+e+w)

Algorithm postDiscoveryTraversal(G, v, e, w)
if z=w then
  pathFound=true
  path<-w.getPath()
```

Answer to the Q. No.2(c):

```
Algorithm vertexVisit(s)
if v=s then
setParent(s,null,null)

Algorithm preDiscoveryTraversal(G, v, e, w)
if !cyleFound=false
setParent(w,v,e) {set parent and related edge}

Algorithm crossTraversal(G,v,e,w)
cyleFound=true
S<-new Stack()
Q<-new Queue()
while getParent(v)!=getParent(w) then
        S.push(v)
        S.push(getParentConnectedEdge(v))
        Q.enqueue(w)
        Q.enqueue(getParentConnectedEdge(w))

        v<- getParent(v)
        w<-getParent(w)
cyclePath<-new Sequence()
cyclePath.insertLast(getParent(v))

while !S.isEmpty() then
```

```
cyclePath.insertLast(s.pop())
cyclePath.insertLast(e)
while !Q.isEmpty() then
cyclePath.insertLast(Q.dequeue())
```

Answer to the Q. No.2(d):



If we see above example, where DFS is used to traverse a graph, To find path from node A to C,

It will traverse from node A to B, then from B to C. Even though shortest path is from A to C. So it's not guaranteed that template version or non-template version of DFS algorithm will find minimum edges path between two vertices.

Answer to the Q. No.3:

```
Algorithm findShortestPath(G,s,d)
z<-d  {z is  a subclass variable}
initResult( G )
DijkstraDistances(G, s)
result(G)

Algorithm DijkstraDistances(G, s)
Q <- new heap-based priority queue
for all v <- G.vertices()
        preInitVertex(u)
        if v = s
        setDistance(v, 0)
        else
        setDistance(v, INFINITY)
        Q.insertItem(getDistance(v), v)
while !Q.isEmpty()
        u <- Q.removeMin()
        vertexVisit(v)
```

```
for all e in G.incidentEdges(u)
{ relax edge e }
        z <- G.opposite(u,e)
        preDiscoveryTraversal(G, u, e, z)
        r <- getDistance(u) + weight(e)
        if r < getDistance(z)
                setDistance(z,r)
                beforeDistanceChange(G,u,e,z)
                Q.replaceKey(z,r)
                afterDistanceChange(G,u,e,z)
        postDiscoveryTraversal(G, u, e, z)
```

Algorithm *afterDistanceChange(G,u,e,z)*
setParent(z,u,e) {set parent and related edge}

Algorithm result(G)
S<-new Sequence()
while getparent(d)!=s do
        S.insertLast(d)
        e<-getParentConnectedEdge(d)
        S.insertLast(e)
        d<- getparent(d)
S.insertLast(d)
S.insertLast(getParentConnectedEdge(d))
S.insertLast(s)

Answer to the Q. No.4:

Algorithm initResult(G)
connectedComponent<-1 { connectedComponent is a subclass variable}

Algorithm postComponentVisit(G,v)
connectedComponent<- connectedComponent+1

Algorithm startVertexVisit(s)
setLabel(s, connectedComponent)