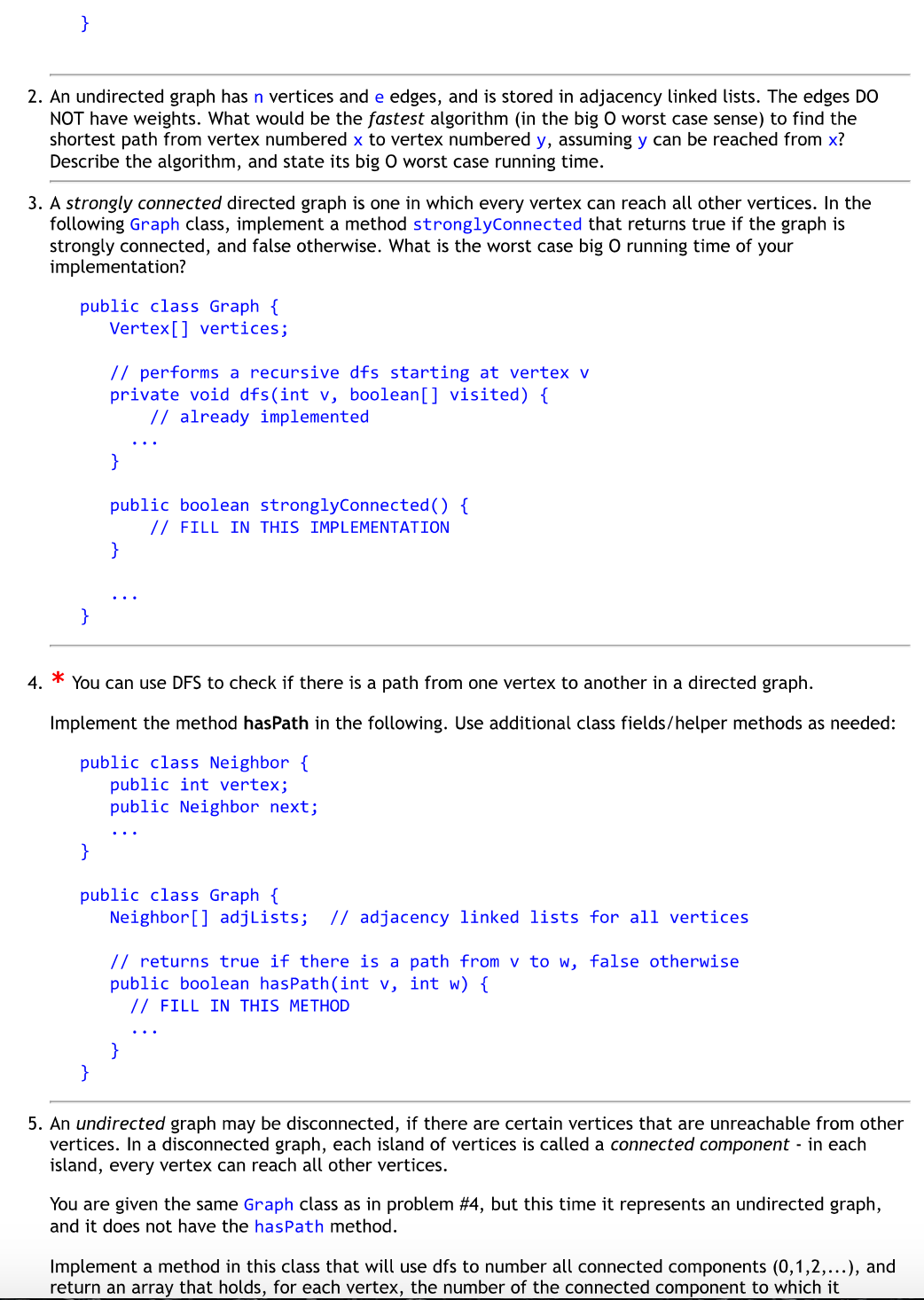
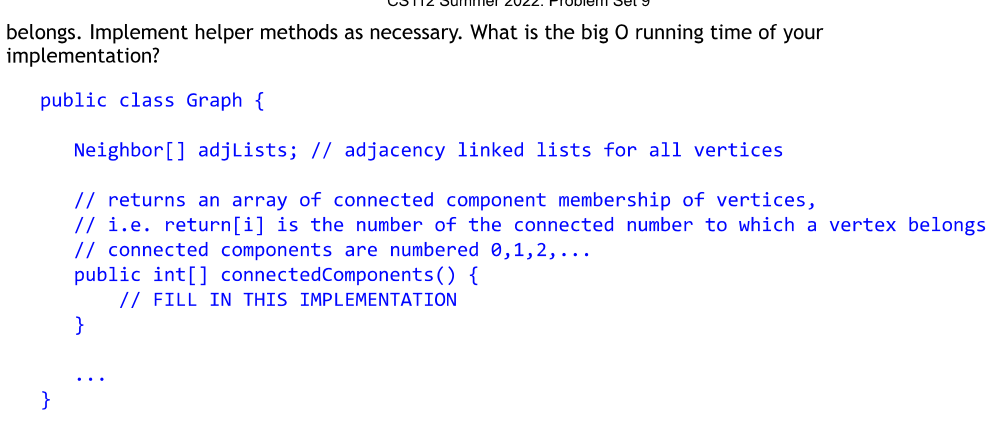


****

****

1)

public int[] indegrees() {

int[] indeg = new int[this.vertices.length];

for (int i = 0; i < this.vertices.length; i++) {

Neighbor current = this.vertices[i].neighbors;

while (current != null) {

indeg[current.vertex]++;

current = current.next;

}

}

return indeg;

}

public String[] topsort() {

int[] pathsRemaining = indegrees();

String[] sortedTable = new String[vertices.length];

int tableCounter = 0;

Queue myQueue = new Queue();

for (int i = 0; i < vertices.length; i++) {

if (pathsRemaining[i] == 0) {

myQueue.enqueue(i);

sortedTable[tableCounter] = vertices[i].name;

tableCounter++;

}

}

while (!myQueue.isEmpty) {

int v = myQueue.dequeue;

pathsRemaining[v]--;

if (pathsRemaining[v] == 0) {

sortedTable[tableCounter] = vertices[v].name;

tableCounter++;

}

Neighbor current = vertices[v].neighbors;

while (current != null) {

myQueue.enqueue(current.vertex);

current = current.next;

}

}

return sortedTable;

}

2)

We would use a breadth-first search algorithm to find the shortest path. We will start by creating two queues and enqueueing the initial element x in the first queue. Next, we will dequeue x and enqueue every vertex on its adjacency list, including the information about the previous vertex into the second queue. Essentially, every time we dequeue from queue 1, our dequeue from queue 2 will give us the information about the shortest path to get to queue 1’s dequeued element from x. We will continuously dequeue elements from both queues and enqueue elements and their path information into queues 1 and 2 until we reach y in queue 1. After that, we will dequeue from queue 2 to obtain the shortest path to get to y from x.

The Big O would be O(n+e) as you would have to navigate through all the vertices and edges to get from x to y.

3)

public boolean stronglyConnected() {

for (int i = 0; i < vertices.length; i++) {

boolean[] checkArray = new boolean[vertices.length];

dfs(vertices[i], checkArray);

for (int j = 0; j < vertices.length; j++) {

if (checkArray[j] == false)

return false;

}

return true;

}

}

Worst case Big O runtime is **O(n^2+ne)** as you would have to check that every vertex has an edge to every other vertex.

4)

public boolean hasPath(int v, int w) {

boolean[] myMarked=new boolean[adjLists.length];

return hasPath(v, w, myMarked);

}

public boolean hasPath(int v, int w, boolean[] marked) {

marked[v] = true;

Neighbor current = adjLists[v];

while (current != null) {

if (current.vertex == w) {

return true;

}

if (!marked[current.vertex]) {

hasPath(current.vertex, w, marked);

}

current = current.next;

}

return false;

}

5)

public void explore(int v,int compValue, int[] connectedArray) {

if(connectedArray[v]>-1) return;

connectedArray[v]=compValue;

Neighbor current=adjLists[v];

while (current!=null) {

if (connectedArray[current.vertex]==-1) {

explore(current.vertex, compValue, connectedArray);

}

current=current.next;

}

}

public int[] connectedComponents() {

int compCount=-1;

int[] connectedArray=new int[adjLists.length];

for (int i=0;v<adjLists.length;i++) {

connectedArray[i]=-1;

}

for (int v=0;v<adjLists.length;v++) {

if (connectedArray[v]==-1) {

compCount++;

explore(v,compCount,connectedArray);

}

}

}

Big O: **O(n+e)** as it traverses all vertices and edges.