

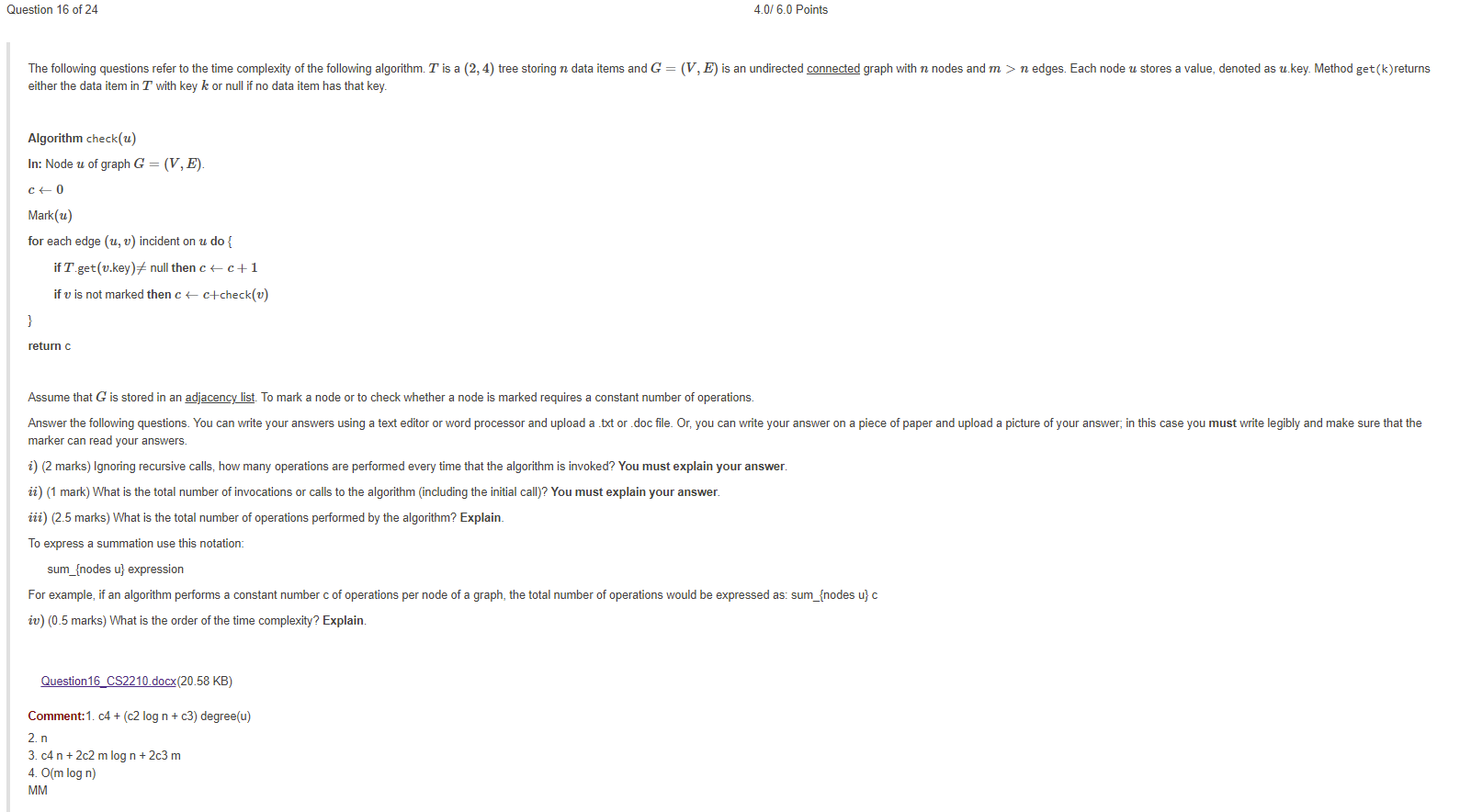
Question 15:

1. Worst case is if there is no similar key in T-tree and S-tree, then the code has to run through both trees n times.
2. Constant 1 is the nested S.get statement and above it is the first for loop “**for** j ← 0 **to** n-1 **do**” which is C2 and runs n times if no specific key is found. C3 is the second for loop “**for** i ← 0 **to** n-1 **do** ”

So we have **f(n) = (c1+c2\*n)\* c3\*n**

1. f(n) = (c1+c2\*n)\* c3\*n -->(get rid of constants)

f(n)=n(n) = **O(n^2)**



Question 16:

**Algorithm** check(*u*)

**In:** Node *u* of graph *G*=(*V*,*E*).

*c*←0

Mark(*u*)

**for** each edge (*u*,*v*) incident on *u* **do** {

**if** *T*.get(*v*.key)≠ null **then** *c*←*c*+1

**if** *v* is not marked **then** *c*←*c*+check(*v*)

}

**return** c

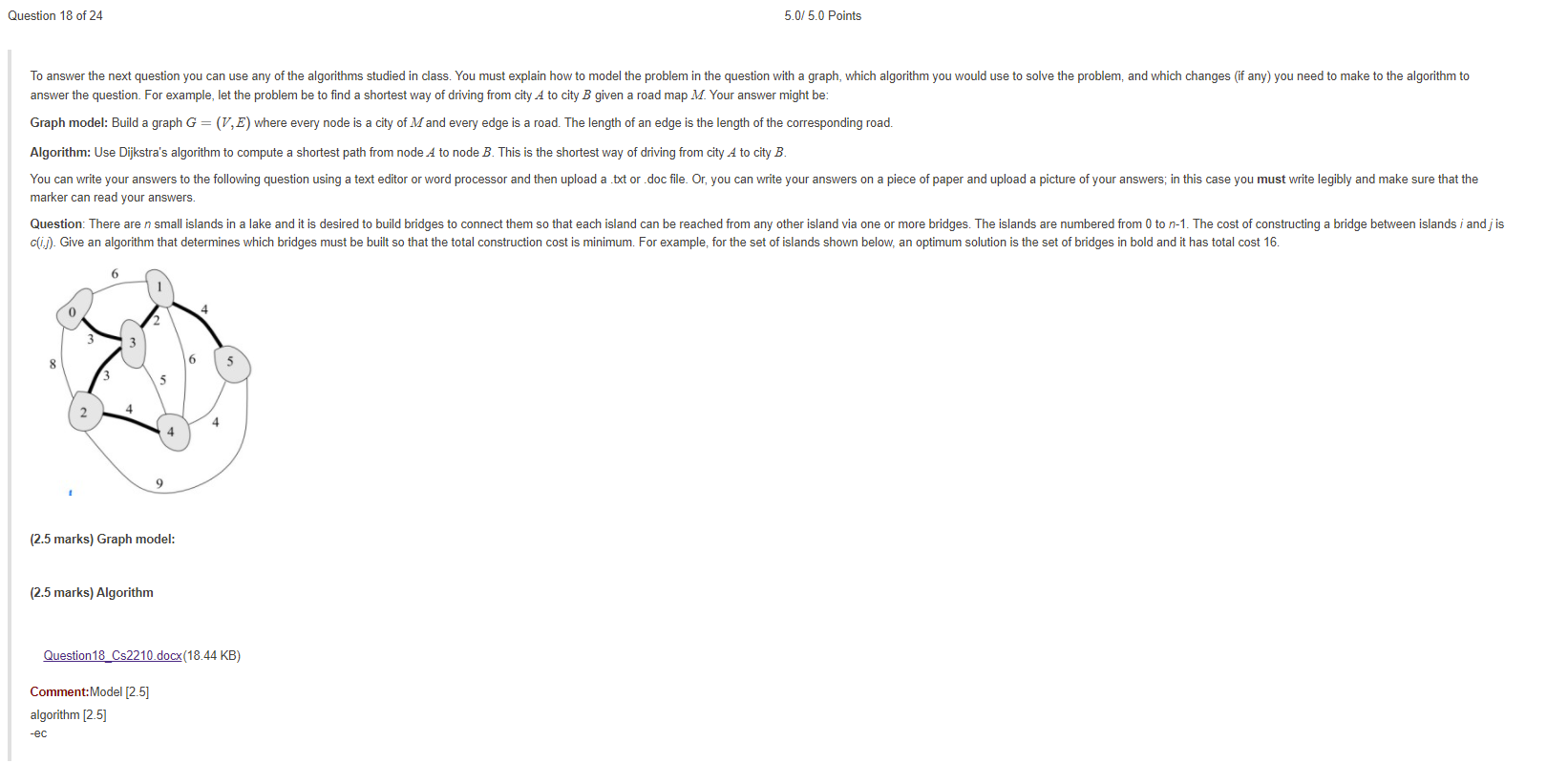
1.

i) roughly 4 operations are performed, since the first is c=0, and mark(u) is the next operation outside the for loops, the get operation and c=c+1 are the main ops inside the for loop, so while the first 2 **aren’t** repeated for every time the for loop is called.

ii)total calls are n

iii)f(n)= c1+(c2)\*n

iv)O(n)



Question 18:

Graph Model: Build graph G=(G,s) where every island “u” and every optimal edge is a bridge v.

Algo(G,s){

IN: weighted graph

For each vertex u of G dp{

u.dist<- null;

u.predesscor<-null

u.mark=false;

}

s.dist=0;

for i=0 to n-1 do{

min=null;

for each vertex v do{

if(mark==false) and (v.dist<min) then{

min = v.dist;

u=v;

}

u.mark =true;

for each edge incident on u do{

if u.dist+length(u,v)<v.d then{

v.dist=u.dist+length(u,v);

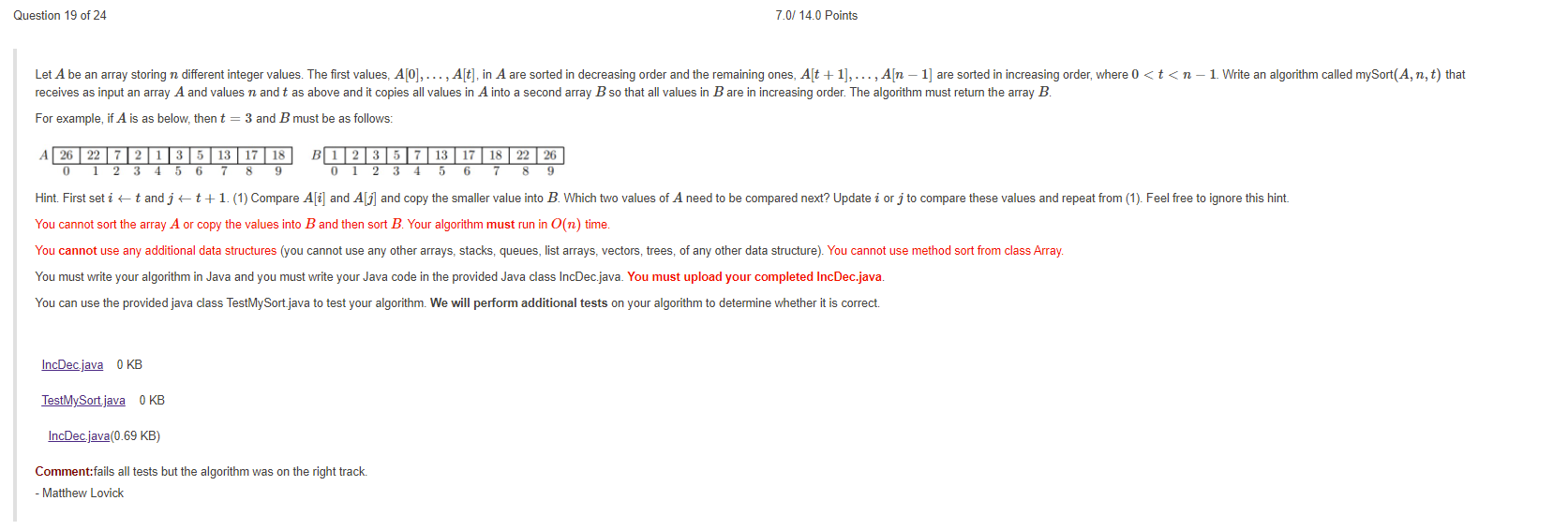
v.predesscor=u;

}

}

}

}



public class TestMySort {

public static void main(String[] args) {

int[] A = {26,22,7,2,1,3,5,13,17,18};

int[] result = {1,2,3,5,7,13,17,18,22,26};

int n = 10;

IncDec program = new IncDec();

int[] B = program.mySort(A,n,3);

int i = 0;

while ((i < n) && (B[i] == result[i])) ++i;

if (i == 10) System.out.println("Test passed");

else System.out.println("Test failed");

}

}

public class IncDec {

public int[] mySort(int[] A, int n, int t) {

/\* Input: Array A storing n different integers and values n, t, where 0 < t < n-1. Values A[0] ... A[t]

are sorted in decreasing order. The remaining values are sorted in increasing order.

Output: Array B storing all values of A in increasing order

\*/

int[] B = new int[n];

int i = n;

int j = t+1;

int c=0;

while (i<=n && j<=t) {

if(A[i] <= A[j]) {

B[c] = A[i];

c++; i++;

}

else {

B[c] = A[j];

c++; j++;

}

while(i<=t) {

B[c] = A[i];

c++; i++;

}

while(j<=n) {

B[c] = A[j];

c++; j++;

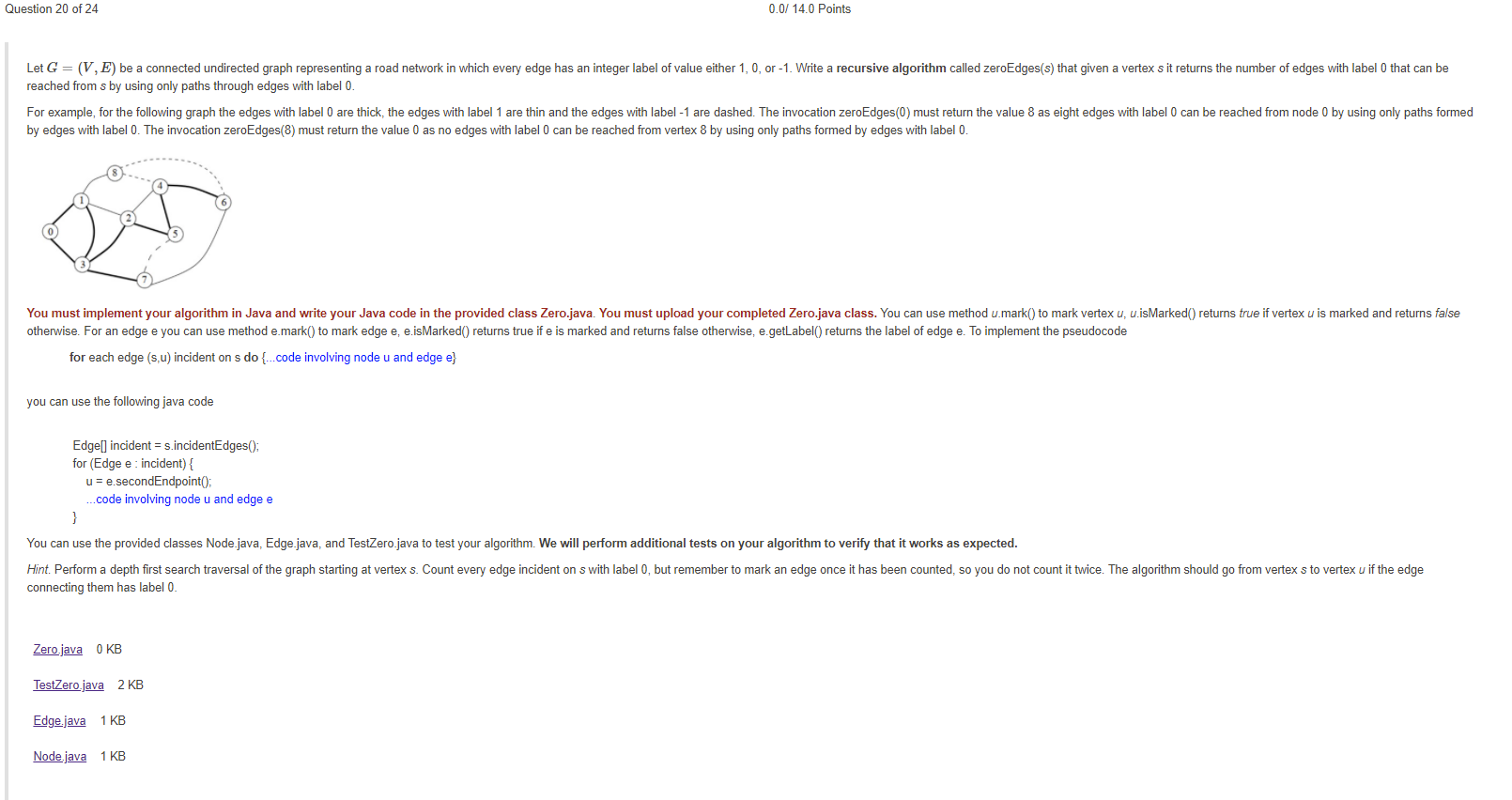
}

}

return B;

}

}



public class Zero {

/\* This method must return the number of edges with label 0 that

can be reached from node s by using only paths through edges

with label 0.

You can use the following methods from the class Node:

- u.mark(): marks node u

- u.isMarked(): returns true if u is marked; returns false otherwise

You can use these methods from the class Edge:

- e.mark(): mark edge e

- e.isMarked(): returns true if edge e is marked; returns false otherwise

- e.getLabel(): returns the label of edge e

To implement the pseudocode

for each edge e=(s,u) incident on s do {code involving node u and edge e}

use the following Java code

Edge[] incident = s.incidentEdges();

for (Edge e : incident) {

u = e.secondEndpoint();

code involving node u and edge e

}

\*/

public int zeroEdges(Node s) {

}

}

public class TestZero {

public static void main(String[] args) {

int[][]graph = {{1,3},{0,2,3,8},{1,3,4,5},{0,1,2,7},{2,5,6,8},{2,4,7},{4,7,8},{3,5,6},{1,4,6}};

int[][]edgeLabels = {{0,0},{0,1,0,1},{1,0,1,0},{0,0,0,0},{1,0,0,-1},{0,0,-1},{0,1,-1},{0,-1,1},{1,-1,-1}};

Node[] V = buildGraph(graph, edgeLabels, null);

Zero program = new Zero();

int result = program.zeroEdges(V[0]);

if (result == 8) System.out.println("Test 1 passed");

else System.out.println("Test 1 failed. Incorrect value returned is "+result+". Correct value is 8.");

unmark(V);

result = program.zeroEdges(V[8]);

if (result == 0) System.out.println("Test 2 passed");

else System.out.println("Test 2 failed. Incorrect value returned is "+result+". Correct value is 0.");

}

/\* Store the graph in an edge list \*/

public static Node[] buildGraph(int[][] graph, int[][] edgeLabels, int[] nodeLabels) {

Node[] V = new Node[graph.length];

Node[] listNodes;

Edge[] listEdges;

/\* Create the nodes of the graph \*/

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

if (nodeLabels != null) V[i] = new Node(nodeLabels[i],i);

else V[i] = new Node(i);

/\* Create the edges of the graph \*/

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

listNodes = new Node[graph[i].length];

listEdges = new Edge[graph[i].length];

for (int j = 0; j < graph[i].length; ++j) {

listNodes[j] = V[graph[i][j]];

listEdges[j] = addEdge(i,graph[i][j],edgeLabels[i][j],V);

}

V[i].addNodes(listNodes);

V[i].addEdges(listEdges);

}

return V;

}

/\* Creates an edge between nodes u and v, if that edge does not yet exist;

otherwise create a new edge (u,v) andlink it to edge (v,u). \*/

private static Edge addEdge(int u, int v, int label, Node[] V) {

Edge[] listEdges;

Edge e;

if (v < u) {

listEdges = V[v].incidentEdges();

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

if (listEdges[i].secondEndpoint() == V[u]) {

e = new Edge(label,V[u],V[v]);

e.setTwin(listEdges[i]);

listEdges[i].setTwin(e);

return e;

}

}

return new Edge(label,V[u],V[v]);

}

/\* Unmark all the nodes and edges of the graph \*/

private static void unmark(Node[] V) {

Edge[] listEdges;

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

V[i].unmark();

listEdges = V[i].incidentEdges();

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

listEdges[j].unmark();

}

}

}

public class Edge {

private int label;

private boolean marked;

private Node first, second; // Nodes connected by this edge

private Edge twin; // Copy (v,u) of edge (u,v)

public Edge(Node firstNode, Node secondNode) {

label = -1;

marked = false;

first = firstNode;

second = secondNode;

twin = null;

}

public Edge (int newLabel, Node firstNode, Node secondNode) {

label = newLabel;

marked = false;

first = firstNode;

second = secondNode;

twin = null;

}

// Returns first endpoint of this edge

public Node firstEndpoint() {

return first;

}

// Returns second endpoint of this edge

public Node secondEndpoint() {

return second;

}

public void mark() {

marked = true;

if (twin != null) twin.marked = true;

}

public boolean isMarked() {

return (marked == true);

}

public void unmark() {

marked = false;

}

public int getLabel() {

return label;

}

public void setTwin(Edge otherEdge) {

twin = otherEdge;

}

public Edge getTwin() {

return twin;

}

}

public class Node {

private int name;

private int label;

boolean marked;

private Node[] neighbours; // Neightbours of this node

private Edge[] incident; // Edges incident on this node

public Node(int newName) {

name = newName;

marked = false;

}

public Node(int newLabel, int newName) {

label = newLabel;

marked = false;

name = newName;

}

public void mark() {

marked = true;

}

public void unmark() {

marked = false;

}

public boolean isMarked() {

return (marked == true);

}

public int getLabel() {

return label;

}

public Node[] getNeighbours() {

return neighbours;

}

public Edge[] incidentEdges() {

return incident;

}

public void addNodes(Node[] newNeighbours) {

neighbours = newNeighbours;

}

public void addEdges(Edge[] newIncident) {

incident = newIncident;

}

public int getName() {

return name;

}

public int degree() {

return incident.length;

}

}

