Shiyu Wang, Chen Bai, Anna Sun

1.

(a) let i = innocent person, g = gang member ? = not sure.

1) If we bring any pair of people to office, there will be 3 possible outcomes: i&i, i&g(g&i) and g&g.
2) pair : i & i g & i g & g
   outcomes: i & i g & ? ? & ?

(b)

1)Randomly pick 1 person and bring him with all other (n-1) people to the office

2)If equal or more than (n-1)/2 people reply that he is innocent, then we can confirm he's one of innocent people. With his reply, we can confirm all other (n-1) people's identity. Then the time is O(n).

3)If less than (n - 1)/2 people reply he is innocent, we can confirm he must be a gang member. So, in this case, we need to find an innocent person to go back to step 2 to finish the interrogation.

2.

True. Since the spanning tree always contains exactly n edges, the sum of weights of edges for spanning trees of G and G-squared would be related by G-squared$_{sum}$ = G$_{sum}^2$. Hence for two different spanning trees T1 and T2 if GSum(T1) ≥ GSum(T2), then G-squared$_{Sum}$(T1) ≥ G-squared$_{Sum}$(T2). Also if G-squared$_{Sum}$(T1) ≥ G-squared$_{Sum}$(T2), GSum(T1) ≥ GSum(T2). T is MST of G ⇔ T is MST of G-squared. proved.

3.

(a)

Building a heap by repeated insertion, we put the inserted number at bottom. (treat it as the biggest). then starting comparing the inserted number with its parent. if inserted number > parent, we don't need to do anything.

If inserted number < parent, we swap inserted number with its parent and compare the inserted number with its new parent.

Therefore, the worst case will be compare all the node along to the root.

Because there are n total numbers, the height of the heap is log(n), so each worst case of insertion will be O(log(n). therefore, n total insertions, it can take O(nlog(n)) time.

(b)

1) Randomly build a heap from a list of numbers, like a binary complete tree.
2) Consider all leafs are at their perfect positions.
3) Compare upper level node with its children.

if node < both children, we don't need to do anything.
if node > any child, we swap them, then compare node with its new children.(if they exist)

4) repeat step 3 for upper level, until the root.

Therefore, n/2 numbers (leafs) can be skipped to start, n/4 numbers (parents of leafs) can move down 1 level and n/8 numbers (grandparents of leafs) at most will be move down 2 levels. …… n/$2^{\log(n)}$ numbers can be move log(n -1) levels. Consider the worst case node move down total $\sum_{h=1}^{logn}(h * n / 2^{h+1})$. Then the worst case is O(n).

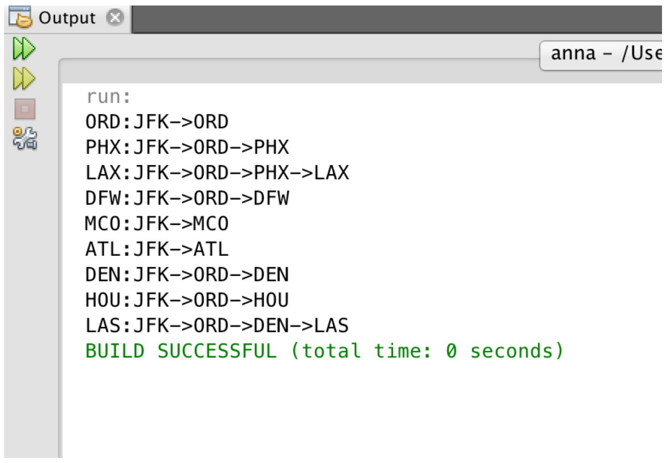
4.

a) the code is here, use Dijkstra algorithm:

```
public class Q4_1 {
    public static void main(String[] args){
          try {
           String pathname = "/Users/anna/NetBeansProjects/algo_2/src/Q4/input.txt";
File filename = new File(pathname);
   InputStreamReader reader = new InputStreamReader(
                   new FileInputStream(filename));
           BufferedReader br = new BufferedReader(reader);
           String line = "";
           line = br.readLine();
           int index = 0;
           while (line != null) {
               line = br.readLine();

                              String[] split = line.split(" ");
                              panth p = new panth(split[0], split[1], Integer.parseInt(split[2]));
                              nodes.add(p);
                     System.out.println(nodes.get(index).from + "  "
                            + nodes.get(index).to + "   " + nodes.get(index++).distance);
           }
       } catch (Exception e) {
           e.printStackTrace();
       }




          Dijkstra test=new Dijkstra();
       Node start=test.init();
       test.computePath(start);
       test.printPathInfo();
```

```
}

public static class Dijkstra {
    Set<Node> open=new HashSet<Node>();
    Set<Node> close=new HashSet<Node>();
    Map<String,Integer> path=new HashMap<String,Integer>();//the path distance
    Map<String,String> pathInfo=new HashMap<String,String>();//the path info
    public Node init(){
        //init path, use Integer.MAX_VALUE for the non-path
        path.put("MCO", 5);
        pathInfo.put("MCO", "JFK->MCO");

        path.put("ATL", 10);
        pathInfo.put("ATL", "JFK->ATL");
        path.put("ORD", 5);
        pathInfo.put("ORD", "JFK->ORD");

        path.put("DEN", Integer.MAX_VALUE);
        pathInfo.put("DEN", "A");
        path.put("HOU", Integer.MAX_VALUE);
        pathInfo.put("HOU", "A");
        path.put("DFW", Integer.MAX_VALUE);
        pathInfo.put("DFW", "A");
        path.put("PHX", Integer.MAX_VALUE);
        pathInfo.put("PHX", "A");
        path.put("LAX", Integer.MAX_VALUE);
        pathInfo.put("LAX", "A");
        path.put("LAS", Integer.MAX_VALUE);
        pathInfo.put("LAS", "A");

        //put start node into close, others into open
        Node start=new MapBuilder().build(open,close);
        return start;
    }
    public void computePath(Node start){
        Node nearest=getShortestPath(start);//get nearest node to start node, put it into close
        if(nearest==null){
            return;
        }
        close.add(nearest);
        open.remove(nearest);
        Map<Node,Integer> childs=nearest.getChild();
        for(Node child:childs.keySet()){
                if(open.contains(child)){//if the child node is in open
                    Integer newCompute=path.get(nearest.getName())+childs.get(child);
                    if(path.get(child.getName())>newCompute){//old distance > new distance
                        path.put(child.getName(), newCompute);
                        pathInfo.put(child.getName(),
pathInfo.get(nearest.getName())+"->"+child.getName());
                    }
                }
            }
            computePath(start);//repeat run itself, ensure all child nodes visited
            computePath(nearest);//recursive to the outter layer, until all nodes been visited
        }
        public void printPathInfo(){
            Set<Map.Entry<String, String>> pathInfos=pathInfo.entrySet();
            for(Map.Entry<String, String> pathInfo:pathInfos){
                System.out.println(pathInfo.getKey()+":"+pathInfo.getValue());
            }
        }
        /**
         * get the nearest child node
         */
        private Node getShortestPath(Node node){
            Node res=null;
            int minDis=Integer.MAX_VALUE;
            Map<Node,Integer> childs=node.getChild();
            for(Node child:childs.keySet()){
                if(open.contains(child)){
                    int distance=childs.get(child);
                    if(distance<minDis){
                        minDis=distance;
                        res=child;
                    }
                }
            }
            return res;
        }
}

    public static class  Node {
        private String name;
        private Map<Node,Integer> child=new HashMap<Node,Integer>();
        public Node(String name){
            this.name=name;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public Map<Node, Integer> getChild() {
            return child;
        }
        public void setChild(Map<Node, Integer> child) {
            this.child = child;
        }

}

    public static class MapBuilder {
        public Node build(Set<Node> open, Set<Node> close){
        Node nodeA=new Node("JFK");
        Node nodeB=new Node("MCO");
        Node nodeC=new Node("ORD");
        Node nodeD=new Node("DEN");
        Node nodeE=new Node("HOU");
        Node nodeF=new Node("DFW");
        Node nodeG=new Node("PHX");
        Node nodeH=new Node("ATL");
        Node nodeI=new Node("LAX");
        Node nodeJ=new Node("LAS");
        nodeA.getChild().put(nodeB, 5);
        nodeA.getChild().put(nodeH, 10);
        nodeA.getChild().put(nodeC, 5);

        nodeC.getChild().put(nodeD, 7);
        nodeC.getChild().put(nodeE, 2);
        nodeC.getChild().put(nodeF, 3);
        nodeC.getChild().put(nodeG, 8);
        nodeC.getChild().put(nodeH, 7);

        nodeD.getChild().put(nodeG, 1);
        nodeD.getChild().put(nodeJ, 13);

        nodeE.getChild().put(nodeB, 9);
        nodeF.getChild().put(nodeG, 11);
        nodeF.getChild().put(nodeE, 9);

        nodeG.getChild().put(nodeI, 6);

        nodeH.getChild().put(nodeE, 4);
        nodeH.getChild().put(nodeB, 9);

        nodeJ.getChild().put(nodeI, 2);
        nodeJ.getChild().put(nodeG, 4);

        open.add(nodeB);
        open.add(nodeC);
        open.add(nodeD);
        open.add(nodeE);
        open.add(nodeF);
        open.add(nodeG);
        open.add(nodeH);
        open.add(nodeI);
        open.add(nodeJ);
        close.add(nodeA);
        return nodeA;
    }
}



}
```

and the result wtih sample input:



```
run:
ORD:JFK->ORD
PHX:JFK->ORD->PHX
LAX:JFK->ORD->PHX->LAX
DFW:JFK->ORD->DFW
MCO:JFK->MCO
ATL:JFK->ATL
DEN:JFK->ORD->DEN
HOU:JFK->ORD->HOU
LAS:JFK->ORD->DEN->LAS
BUILD SUCCESSFUL (total time: 0 seconds)
```

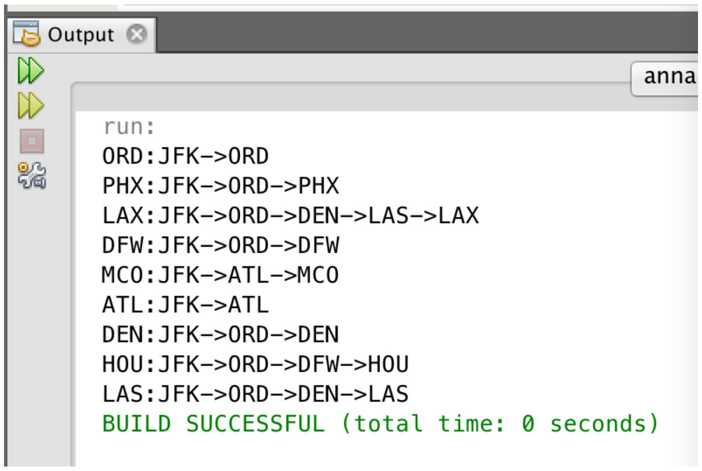
b) the code as here:

```
public class Q4_2 {

      public static void main(String[] args) throws IOException{
        ArrayList<panth> nodes = new ArrayList<panth>();
          Dijkstra test=new Dijkstra();
        Node start=test.init();
        test.computePath(start);
        test.printPathInfo();

          try {

            String pathname = "/Users/anna/NetBeansProjects/algo_2/src/Q4/input.txt"; /
File filename = new File(pathname);
            InputStreamReader reader = new InputStreamReader(
                    new FileInputStream(filename));
            BufferedReader br = new BufferedReader(reader);
            String line = "";
            line = br.readLine();
            int index = 0;
            while (line != null) {
                line = br.readLine();

                              String[] split = line.split(" ");
                              panth p = new panth(split[0], split[1], Integer.parseInt(split[2]));
                              nodes.add(p);
                        System.out.println(nodes.get(index).from + "  "
                                + nodes.get(index).to + "   " + nodes.get(index++).distance);

            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

public static class panth {
            public String from;
            public String to;
            public int distance;
            //constructor
            public panth(String from, String to, int distance) {
                        from = this.from;
                    to = this.from;
                    distance = this.distance;
          }
}

public static ArrayList<panth> readFile (String fileName) throws IOException {
          //build an arraylist to save panth
          ArrayList<panth> nodes = new ArrayList<panth>();
       try {
                    Scanner input = new Scanner(new File(fileName));
             //check the first lines
          String title = input.nextLine();
          if ( !"% node1 node2 edge-length".equals(title)) {
                    throw new RuntimeException("input-file is wrong");
                    }
          else {
              System.out.println(title);
                    InputStreamReader reader = new InputStreamReader(
                 new FileInputStream(fileName));
              BufferedReader br = new BufferedReader(reader);
              String line = "";
              line = br.readLine();
//            while (line != null) {
//               line = br.readLine();
//            }
              int index = 0;
          while (line!= null) {
                              String s = input.nextLine();
                              String[] split = s.split(" ");
                              panth p = new panth(split[0], split[1], Integer.parseInt(split[2]));
                              nodes.add(p);
                     System.out.println(nodes.get(index).from + "  "
                            + nodes.get(index).to + "   " + nodes.get(index++).distance);

                     line = br.readLine();
                              }
              }



   public static class Dijkstra {
       Set<Node> open=new HashSet<Node>();
       Set<Node> close=new HashSet<Node>();
```

```
Map<String,Integer> path=new HashMap<String,Integer>();//the path distance
Map<String,String> pathInfo=new HashMap<String,String>();//the path info

public Node init(){
   //init path, use Integer.MAX_VALUE for the non-path
   path.put("MCO", -5);
   pathInfo.put("MCO", "JFK->MCO");

   path.put("ATL", -10);
   pathInfo.put("ATL", "JFK->ATL");
   path.put("ORD", -5);
   pathInfo.put("ORD", "JFK->ORD");

   path.put("DEN", Integer.MAX_VALUE);
   pathInfo.put("DEN", "A");
   path.put("HOU", Integer.MAX_VALUE);
   pathInfo.put("HOU", "A");
   path.put("DFW", Integer.MAX_VALUE);
   pathInfo.put("DFW", "A");
   path.put("PHX", Integer.MAX_VALUE);
   pathInfo.put("PHX", "A");
   path.put("LAX", Integer.MAX_VALUE);
   pathInfo.put("LAX", "A");
   path.put("LAS", Integer.MAX_VALUE);
   pathInfo.put("LAS", "A");

   //put start node into close, others into open
   Node start=new MapBuilder().build(open,close);
   return start;
}
public void computePath(Node start){
   Node nearest=getShortestPath(start);//get nearest node to start node, put it into close
   if(nearest==null){
      return;
   }
   close.add(nearest);
   open.remove(nearest);
   Map<Node,Integer> childs=nearest.getChild();
   for(Node child:childs.keySet()){
      if(open.contains(child)){//if the child node is in open
         Integer newCompute=path.get(nearest.getName())+childs.get(child);
         if(path.get(child.getName())>newCompute){// old distance > new distance
            path.put(child.getName(), newCompute);
```

```
                        pathInfo.put(child.getName(),
pathInfo.get(nearest.getName())+"->"+child.getName());
                    }
                }
            }
            computePath(start);//repeat run itself, ensure all child nodes visited
            computePath(nearest);//recursive to the outter layer, until all nodes been visited
        }
        public void printPathInfo(){
            Set<Map.Entry<String, String>> pathInfos=pathInfo.entrySet();
            for(Map.Entry<String, String> pathInfo:pathInfos){
                System.out.println(pathInfo.getKey()+":"+pathInfo.getValue());

            }
        }
        /**
         * get the nearest child node
         */
        private Node getShortestPath(Node node){
            Node res=null;
            int minDis=Integer.MAX_VALUE;
            Map<Node,Integer> childs=node.getChild();
            for(Node child:childs.keySet()){
                if(open.contains(child)){
                    int distance=childs.get(child);
                    if(distance<minDis){
                        minDis=distance;
                        res=child;
                    }
                }
            }
            return res;
        }
}

    public static class  Node {
        private String name;
        private Map<Node,Integer> child=new HashMap<Node,Integer>();
        public Node(String name){
            this.name=name;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public Map<Node, Integer> getChild() {
            return child;
        }
        public void setChild(Map<Node, Integer> child) {
            this.child = child;
        }

}

    public static class MapBuilder {
        public Node build(Set<Node> open, Set<Node> close){
            Node nodeA=new Node("JFK");
            Node nodeB=new Node("MCO");
            Node nodeC=new Node("ORD");
            Node nodeD=new Node("DEN");
            Node nodeE=new Node("HOU");
            Node nodeF=new Node("DFW");
            Node nodeG=new Node("PHX");
            Node nodeH=new Node("ATL");
            Node nodeI=new Node("LAX");
            Node nodeJ=new Node("LAS");
            nodeA.getChild().put(nodeB, -5);
            nodeA.getChild().put(nodeH, -10);
            nodeA.getChild().put(nodeC, -5);
           nodeC.getChild().put(nodeD, -7);
            nodeC.getChild().put(nodeE, -2);
            nodeC.getChild().put(nodeF, -3);
            nodeC.getChild().put(nodeG, -8);
            nodeC.getChild().put(nodeH, -7);
            nodeD.getChild().put(nodeG, -1);
            nodeD.getChild().put(nodeJ, -13);
            nodeE.getChild().put(nodeB, -9);
            nodeF.getChild().put(nodeG, -11);
            nodeF.getChild().put(nodeE, -9);
            nodeG.getChild().put(nodeI, -6);
            nodeH.getChild().put(nodeE, -4);
            nodeH.getChild().put(nodeB, -9);
            nodeJ.getChild().put(nodeI, -2);
            nodeJ.getChild().put(nodeG, -4);
            open.add(nodeB);
            open.add(nodeC);
            open.add(nodeD);
            open.add(nodeE);
            open.add(nodeF);
            open.add(nodeG);
            open.add(nodeH);
            open.add(nodeI);
            open.add(nodeJ);
            close.add(nodeA);
            return nodeA;
    }
}
}
```

the running result shown as :



```
run:
ORD:JFK->ORD
PHX:JFK->ORD->PHX
LAX:JFK->ORD->DEN->LAS->LAX
DFW:JFK->ORD->DFW
MCO:JFK->ATL->MCO
ATL:JFK->ATL
DEN:JFK->ORD->DEN
HOU:JFK->ORD->DFW->HOU
LAS:JFK->ORD->DEN->LAS
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.

a)

1. We can seperate 27 coins into 3 parts, 9 coins each. number them p1, p2 & p3.
2. We first compare p1 and p2. there will be 3 possibilities:
    1) p1 = p2, then we know the lighter one is in p3.
    2) p1 > p2, then we know the lighter one is in p2.
    3) p1 < p2, then we know the lighter one is in p1.
3. What we need to do is separate the lighter part (either p1, p2 or p3) into 2 parts with 3 coins each. then comparing two parts to narrow the possibility into 1 of 3 coins.
4. take 2 out of 3 coins to compare their weight, there will again have 3 possibilities:
    1) same, then the rest one is the lighter one
    2) left < right, then the left coin is the lighter one
    3) left > right, the the right coin is the lighter one.

so we can use 3 weighings to find one out of 27 coins is lighter. By building the comparing tree that one node with 3 child, we can get the equation $3^h = n$, then $h = \log_3 n$, so we need $\Omega(\log n)$ weighings to find one out of n coins is lighter.

b)
Each measurement has three possible results: equal (=), left is lighter than right (<), or left is heavier than right (>). The three measurements together can have a total of 3x3x3 = 27 different outcomes, which are <<<, <<=, <<>, ……..

For coins, there are 24 possible answers, such as 1st coin is defective and heavier, 1st coin is defective and lighter. 2nd…….

Therefore, it is logically possible to map 24 answers on 27 outcomes, and we need to map 24 answers on 24 outcomes. From the mapping, we need to know which coin is defective and it is heavier or lighter.

First, we need to think which three outcomes need to be removed. obviously === should be removed, because it means the defective coin doesn't join the three measurements. In this case, we can know which one is defective but we never know it is lighter or heavier. so the left possibilities are:

1. < < <
2. < < =
3. < < >
4. < = <
5. < = =
6. < = >
7. < > <
8. < > =
9. < > >
10. = < <
11. = < =
12. = < >
13. = = <
14. = = >
15. = > <
16. = > =
17. = > >
18. > < <
19. > < =
20. > < >
21. > = <
22. > = =

and possible answers are:

a. 1st is lighter
b. 1st is heavier
c. 2nd is lighter
d. 2nd is heavier
e. 3rd is lighter
f. 3rd is heavier
g. 4th is lighter
h. 4th is heavier
i. 5th is lighter
j. 5th is heavier
k. 6th is lighter
l. 6th is heavier
m. 7th is lighter
n. 7th is heavier
o. 8th is lighter
p. 8th is heavier
q. 9th is lighter
r. 9th is heavier
s. 10th is lighter
t. 10th is heavier
u. 11th is lighter
v. 11th is heavier

23.	> = >	w.12th is lighter
24.	> > <	x. 12th is heavier
25.	> > =	
26.	> > >	

Then, we have to know answers are symmetric, such that b should be the opposite answer for a. Therefore if we match 1 to a, b has to be 26. Therefore there will be one pair of outcomes need to be removed.

Secondly, if we count the time that “<” appear at first place, it will be 9 times. It is the same at second place and third place and for “>”. because we won’t put 9 coins in one measurement, the pair of outcome that we need to remove should make “<, >” appear only 9 times. Obviously, one possible answer can be 1(< < <) and 26 (> > >). Then we will have 24 outcomes and 24 possible answers left.

Now, we can start mapping. remember we need to make at most 4 coins on each side in one measure, so each “symbol” can only appear 4 times in each row.

= = <	1st heavier	= = >	1st lighter
= < =	2nd heavier	= > =	2nd lighter
< = =	3rd heavier	> = =	3rd lighter
> > =	4th heavier	< < =	4th lighter
> = >	5th heavier	< = <	5th lighter
= > >	6th heavier	= < <	6th lighter
< < >	7th heavier	> > <	7th lighter
< > <	8th heavier	> < >	8th lighter
> < <	9th heavier	< > >	9th lighter
= < >	10th heavier	= > <	10th lighter
> = <	11th heavier	< = >	11th lighter
< > =	12th heavier	> < =	12th lighter

this mapping can tell us exactly where each coin participates in each measurement.
So, according to the mapping, one possible way can be

1st measure: 4 5 9 10 : 3 7 8 12
2nd measure: 4 6 8 12 : 2 7 9 10
3rd measure: 5 6 7 10 : 1 8 9 11

After 3 measure, we can find the answer in the mapping result.

6.

a) counter-example :

If n = 6, then 6 is not a power of 2.

Wrong proof explanation :

First of all, the inductive assumption for s(n) should only consider S(j) given j = n; Then given S(n), n is odd, n+1 should be even in S(n+1) and never be odd, so we should consider S(n+2) instead of S(n);

b) counter-example:

Assume the graph is a disconnected graph, there could be more than one cycle in the graph.

Wrong proof explanation:

First of all, base case should be S(1) and the only degree-2 1-node graph is a loop. Secondly, assume the truth of S(n) which is a cycle on n nodes. Attach a new degree-2 vertex with an edge to the graph, the edge connects to the vertex at both ends. Then, there are two cycles in the new graph S(n+1).

7.

(a).

To find the diameter of n-sided polygon, we can use “Rotating Calipers Algorithm”.

1) Find the two points which have biggest y, y_{max} and smallest y, y_{min} of all points of polygon. Then we got the first antipodal pair and $(y_{max} - y_{min})$ is the first possible diameter.

2) draw a pair of parallel tangent lines on the antipodal pairs. calculate the distance between them.

3) rotate the tangent lines to the same direction until one of them overlaps with a side of polygon. repeat step 2.

4) record all the possible diameters we get and compare their distance. the biggest one is the diameter of polygon. Note for a n-sided polygon, there are at most 3n/2 antipodal pairs.(one side matches 3 angel), so it will take O(n) time to find the diameter.

an example code can be:

```
ch[m+1]:=ch[1]; j:=2;
 for i:=1 to m do
   begin
   while cross(ch[i],ch[j],ch[i+1])<cross(ch[i],ch[j+1],ch[i+1]) do
       begin inc(j); if j>m then j:=1; end;
```

```
    writeln(ch[i].x,' ',ch[i].y,' ',ch[j].x,' ',ch[j].y);
    end;
```

P.S. “cross” means cross product. (which is used to compare possible diameters)