1. Recall that a software *fault* is a static defect in the software, and a *failure* is external, incorrect behavior with respect to a specification.

Consider the following code snippet:

```java
static boolean positive(int[] arr) {
        // precondition: arr != null
        // postcondition: returns true iff all elements of arr are positive, i.e., > 0

        boolean result = true;
        for (int x: arr) {
                if (x < 0) {
                        result = false;
                        break;
                }
        }
        return result;
}
```

(a) Identify a fault in the implementation of `positive` and fix it.

(b) Implement the following JUnit test to show an invocation of `positive` that does not execute the fault:

```java
@Test public void noFaultExec() {

        assertTrue(                                                    );
}
```

(c) Implement the following JUnit test to show an invocation of `positive` that executes the fault but does not terminate in a failure:

```java
@Test public void faultExecAndErrorButNoFailure() {

        assertTrue(                                                    );
}
```

(d) Implement a JUnit test to show an invocation of `positive` that executes the fault and terminates in a failure, i.e., the execution of the test using JUnit should report 1 failure:

```java
@Test public void failure() {



}
```

2. Recall a *du-path* with respect to a variable v is a *simple* path that is *def-clear* with respect to v from a node n_i for which v is in $def(n_i)$ to a node n_j for which v is in $use(n_j)$.

Consider the following code fragment and answer the questions that follow using the node numbers given as comments:

```
w = x; // node 1
if (m > 0) {
       w++; // node 2
}
else {
       w = 2 * w; // node 3
}
// node 4: dummy node to model merging of control flow for "if-else"
if (y <= 10) {
       x = 5 * y; // node 5
}
else {
       x = 3 * y + 5; // node 6
}
z =  w + x; // node 7
```

(a) Draw a control flow graph for this fragment using the given node numbers. Label the edges with the conditions they represent.

(b) Which nodes have defs for variable `w`?

(c) Which nodes have uses for variable `w`?

(d) Are there any du-paths with respect to variable `w` from node 1 to node 7? If not, explain why not. If any exist, show one.

3. Recall that a path is *simple* if no node appears on it more than once, with the exception of the first and the last nodes, which may be the same.

The following code snippet gives a partial implementation of a class to represent paths:

```
import java.util.HashSet;
import java.util.Set;

public class Path {
    Node first; // first == null iff path is empty

    static class Node {
        Node next;
        int id;

        Node(int n) {
            id = n;
        }

        public boolean equals(Object o) {
            if (o.getClass() != Node.class) return false;
            return id == ((Node)o).id;
        }

        public int hashCode() {
            return id;
        }
    }
```

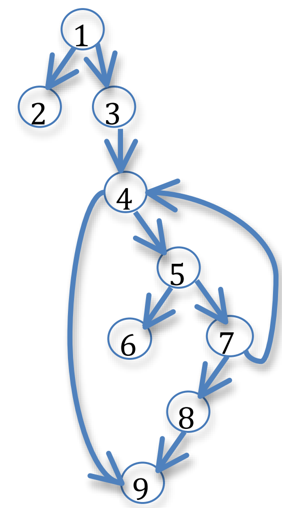
Implement the following method *isSimple* (in class *Path*) as specified by its postcondition:

```
    public boolean isSimple() {
        // postcondition: returns true iff "this" is a simple path

    }
```

4. Recall that a path is *prime* if it is simple and it does not appear as a proper subpath of any other simple path.

Compute all the prime paths for the control-flow graph below. You must show the steps of your algorithm. Make sure to clearly label the set of prime paths.



5. Recall that a *major* clause c_i in predicate *p determines p* if the *minor* clauses c_j in *p* (for $j \neq i$) have values so that changing the truth value of c_i changes the truth value of *p*.

Recall also the definition of *restricted active clause coverage* (*RACC*): For each *p* in *P* and each major clause c_i in *Cp*, choose minor clauses c_j, j != i, so that c_i determines *p*. TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses *cj* must be the same when c_i is true as when c_i is false, that is, it is required that $c_j(c_i = true) = c_j(c_i = false)$ for all c_j.

Recall next the definition of *correlated active clause coverage* (CACC): For each *p* in *P* and each major clause c_i in *Cp*, choose minor clauses c_j, j != i, so that c_i determines *p*. TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause *p* to be true for one value of the major clause c_i and false for the other, that is, it is required that $p(c_i = true)$!= $p(c_i = false)$.

Consider the predicate *p* = *a* && (!*b* || !*c*) and as its truth table:

	a	*b*	*c*	*a* && (!*b* || !*c*)
1	T	T	T	F
2	T	T	F	T
3	T	F	T	T
4	T	F	F	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(a) Find values for minor clauses *b* and *c* such that the major clause *a* determines the predicate *p*.

(b) How many ways can RACC be satisfied for the major clause *a*? For each way, identify the corresponding rows (using row numbers) from the truth table given.

(c) How many ways can CACC be satisfied for the major clause *a*? For each way, identify the corresponding rows (using row numbers) from the truth table given.