1. [10 points total; 5 points each part]

Consider the following code snippet:

```
static int countNonNegative(int[] arr) {
        // precondition: arr != null

 1:     int count = 0;
 2:     for (int i = 1; i <= arr.length; i++) {
 3:             int element = arr[i - 1];
 4:             if (element > 0) {
 5:                     count++;
 6:             } else {
 7:                     count = count;
 8:             }
 9:     }
10:     return count;
}
```

Consider the following modification to the method countNonNegative:

```
static int mutantCountNonNegative(int[] arr) {
        int count = 0;
        for (int i = 1; i <= arr.length; i++) {
                int element = arr[i - 1];
                if (element > 0) {
                        count++;
                } else {
                        i++;
                }
        }
        return count;
}
```

(a) Provide an input array of length >= 3 such that the output of countNonNegative and mutantCountNonNegative differ for that input.

(b) Provide an input array such that the output of countNonNegative and mutantCountNonNegative do not differ for that input but during their executions the corresponding states at some control point differ.

Question 2. [5 points total]

Consider the following method max that returns the maximum value among the three input numbers:

```
static int max(int x, int y, int z) {
        int result = x;
        if (y > result) result = y;
        if (z > result) result = z;
        return result;
}
```

Consider the following mutant of max:

```
static int max1(int x, int y, int z) {
        int result = x;
        if (y < result) result = y;
        if (z > result) result = z;
        return result;
}
```

Give an assignment of values to *x*, *y*, and *z* such that the output of *max* and *max1* are different.

3. [10 points total] Recall that in input space partitioning, a *characteristic* provides a basis for a *partition*, which must satisfy *completeness* (i.e., the partition covers the entire input domain) and *disjointness* (i.e., the blocks in the partition must not overlap) properties.

Recall also the definition of *base choice coverage criterion*: a base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic; subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

Consider the following method:

```
static <T> java.util.List<T> concatenate(List<T> l, List<T> m) {
        // precondition: input lists are non-null
        // postcondition: returns the concatenation of l and m (i.e.,
        //              elements of l followed by elements of m)

        List<T> result = new LinkedList<T>(l);
        result.addAll(m);
        return result;
}
```

Consider the following two characteristics and the corresponding partition blocks:

Characteristic: size of list l
- l.isEmpty()
- l.size() >= 1

Characteristic: relation between l and m
- l == m
- l.containsAll(m) && l != m
- m.containsAll(l) && l != m

Let the input domains for l and m consist of all *non-null* lists of strings (i.e., `java.util.String` objects).

(a) [3 points] Does the partition for "size of list l" satisfy completeness and disjointness? If false, modify at most one of the two blocks such that the partition satisfies completeness and disjointness properties.

(b) [4 points] Does the partition for "relation between l and m" satisfy completeness and disjointness? If false, modify at most one of the three blocks such that the partition satisfies completeness and disjointness properties.

(c) [3 points] If the "base choice" criterion were applied to the two characteristics and their corresponding partitions (exactly as given in this question), how many test requirements would result?

4. [30 points total] Consider the following code snippet:

```
public class LinkedList<T> {
        Entry<T> header;

        static class Entry<T> {
                Entry<T> next, previous;
        }

        public boolean repOk() {
                /* Node: 1 */
                if (header == null) {
                        return false;                                   // Node: 2
                }
                Set<Entry<T>> visited = new HashSet<Entry<T>>();// Node: 3
                visited.add(header);                                    // Node: 4
                Entry<T> current = header;                              // Node: 5
                /* Node: 6 */
                while (true) {
                        Entry<T> next = current.next;  // Node: 7
                        /* Node: 8 */
                        if (next == null) {
                                return false;                           // Node: 9
                        }
                        /* Node: 10 */
                        if (next.previous != current) {
                                return false;                           // Node: 11
                        }
                        current = next;                         // Node: 12
                        /* Node: 13 */
                        if (!visited.add(next)) {
                                break;                                  // Node: 14
                        }
                }
                return true;                                            // Node: 15
        }
```

(a) [3 points] Does the repOk method have any infeasible branch? If yes, which one?

(b) [7 points] What is the minimum number of times we must execute repOk to get *full statement coverage*? Justify your answer.

(c) [8 points] Draw the control-flow graph for repOk as given in this question. (Do not refactor the code.) Use the given node id's. Clearly label any other nodes you introduce.

(d) [7 points] Implement the following JUnit test method, which provides full statement coverage of repOk by invoking it the minimum number of times required to achieve full statement coverage:

```
@Test public void fullStatementCoverage() {

}
```

(e) [5 points] Implement the following method removeCycles (in class LinkedList) that modifies its input "this" as described in the pre- and post-conditions:

```
public void removeCycles() {
        // precondition: repOk()
        // postcondition: (1) there are no cycles in the graph reachable from
        //     header along next or previous fields in the post-state &&
        //     (2) the set of nodes reachable from header in pre-state is the
        //         same in the post-state









}
```