Quiz Section 3: ADTs

The next few problems concern the following ADT:

```
/**
 * Represents an immutable collection of integers.
 *
 * Clients can think of a set as a list of integers. However, they can only ask
 * if an integer is present or not. The order of the integers and the number of
 * times an integer appears in the list are inaccessible and do not matter.
 */
public interface IntSet {
  /**
   * Determines whether n is in the list.
   * @param n the number to look for in the list
   * @returns contains(n, obj), where
   *     contains(n, nil)    := false
   *     contains(n, m :: L) := true           if m = n
   *     contains(n, m :: L) := contains(n, L) if m /= n
   */
  public boolean contains(int n);

  /**
   * Creates and returns a new list containing n as well as all of obj.
   * @param n the number to add to the new list.
   * @returns n :: obj
   */
  public IntSet add(int n);

  /** ... */
  public IntSet remove(int n);
}
```

Task 1 – Teacher's Set [12 pts]

Answer the following questions about the specification of `IntSet`.

a) Explain in your own words what `@return n :: obj` means. In particular, what is "`obj`" in this context? Why does this mathematical expression make sense?

b) Suppose that we have an `IntSet T` whose abstract state is the list $1 :: 2 :: 3 ::$ nil. What mathematical value is returned by the expression `T.add(4)` according to the `add` function specification.

c) Write a specification for the method `remove`. It should return a list that contains all of the numbers in the current list *except* for the number `n`, which should no longer be present.
(Hint: your spec should include a math definition similar to that of `contains`)

Task 2 – Jumping Through Dupes [12 pts]

In this problem, we will return to the original specification of `IntSet`, whose abstract state is a list of elements possibly containing duplicates. We will consider three different concrete representations for it:

```
    public class IntSetImpl implements IntSet {

(1)   // AF: obj = this.elems
      private int[] elems;

(2)   // AF: obj = this.elems
      // RI: this.elems contains no dups
      private int[] elems;

(3)   // AF: obj = this.elems
      // RI: this.elems is sorted
      private int[] elems;

      public IntSetImpl(int[] elems) {
        this.elems = elems;
      }
```

For each of the methods shown below, state the concrete representations (1–3) for which it would satisfy the specification of the method in `IntSet`. In each case, briefly explain why.

a)
```
public boolean contains(int n) {
  return Arrays.binarySearch(this.elems, n) >= 0;
}
```

Note: Binary Search is an algorithm that finds the position of a target value within a sorted array.

b)
```
public boolean contains(int n) {
  for (int i = 0; i < this.elems.length; i++) {
    if (this.elems[i] == n)
      return true;
  }
  return false;
}
```

c)

```
public IntSet add(int n) {
  if (this.contains(n)) {
    return this;
  } else {
    int[] newElems = new int[this.elems.length + 1];
    System.arraycopy(this.elems, 0, newElems, 1, this.elems.length);
    newElems[0] = n;
    return new IntSetImpl(newElems);
  }
}
```

For System.arraycopy() syntax, see: JavaDoc

d)

```
public IntSet remove(int n) {
  for (int i = 0; i < this.elems.length; i++) {
    if (this.elems[i] == n) {
      int[] newElems = new int[this.elems.length - 1];
      System.arraycopy(this.elems, 0, newElems, 0, i);
      System.arraycopy(this.elems, i+1, newElems, i,
          this.elems.length - i - 1);
      return new IntSetImpl(newElems);
    }
  }
  return this;
}
```

Task 3 – Hold Down the Sort [12 pts]

Consider the following implementation of `IntSetImpl`, which ensures that the representation invariant is satisfied by sorting the elements in the constructor:

```
public class IntSetImpl implements IntSet {
    // AF: obj = this.elems
    // RI: this.elems is sorted in ascending order
    private int[] elems;

    public IntSetImpl(int[] elems) {
        this.elems = elems;

        // Put the elements in sorted order.
        for (int i = 1; i < elems.length; i++) {
            int key = elems[i];
            int j = i - 1;
            while (j >= 0 && elems[j] > key) {
                elems[j + 1] = elems[j];
                j--;
            }
            elems[j + 1] = key;
        }
    }
```

a) How many test cases are required to get proper coverage of the constructor? Explain your answer and also give a specific set of test inputs that would give proper coverage.