Quiz Section 6: Midterm Review – Solutions

Task 1 – I Couldn't Square Less [9 pts]

We plan to provide the following method:

```
/**
* Calculates the integer square root of n.
* ...
* @return The integer k such that (k-1)^2 < n <= k^2
*/
public static int isqrt(int n);
```

a) When n is a perfect square, the integer k from the spec will satisfy $k^2 = n$. This tells us that $k = \sqrt{n}$. But how does k relate to $\sqrt{n}$ *when n is positive but not a perfect square*?

k is $\sqrt{n}$ rounded up to the nearest integer.

b) Keeping in mind your answer to (a), what is an another, equally reasonable but incomparable, way to specify this function? Your answer should leave the precondition unchanged and not change the postcondition in any case where n is a perfect square.

Write the `@return` statement for this new specification below.

```
@return The integer k such that k*k <= n < (k+1)*(k+1)
```

c) This specification precisely defines the return value for valid inputs, but it does not make sense when `n` is not positive. Give **two** distinct ways of turning this into a specification that fully defines the behavior for all integer inputs, now including non-positive values.

Your two specifications must still return the value described in the original specification when `n` is positive but they should be incomparable specifications overall.

```
@throws IllegalArgumentException if n <= 0
@return The integer k such that (k-1)^2 < n <= k^2
```

```
@return The integer k such that (k-1)^2 < n <= k^2, OR -1 if n <= 0
```

d) Part (c) said that the spec does not make sense when `n` is not positive. Negative values of `n` would not work since their square roots are complex, but why does it **not** make sense when `n` is 0?

Every integer k has $0 \leqslant (k-1)^2$. Thus, we cannot have $(k-1)^2 < n = 0$.

e) Give a **third** distinct specification that is weaker than both of your specifications from part (c).

Again, it must still return the value described in the original specification when `n` is positive.

```
@requires n >= 1
@return The integer k such that (k-1)^2 < n <= k^2
```

Task 2 – I Plead The Flip [15 pts]

The function negate : (List) → List takes in a list and flips the sign of every value in the List. It is defined recursively as follows:

$$\begin{aligned}\text{negate}(\text{nil}) &:= \text{nil}\\ \text{negate}(x :: L) &:= (-x) :: \text{negate}(L)\end{aligned}$$

The function twice : (List) → List takes a list as input and returns a list in which element has been doubled. It is defined as follows:

$$\begin{aligned}\text{twice}(\text{nil}) &:= \text{nil}\\ \text{twice}(x :: L) &:= 2x :: \text{twice}(L)\end{aligned}$$

Using these definitions, prove the following claim **by induction** on S:

$$\text{negate}(\text{twice}(S)) = \text{twice}(\text{negate}(S))$$

Define $P(S)$ to be the claim that $\text{negate}(\text{twice}(S)) = \text{twice}(\text{negate}(S))$. We will prove this by structural induction.

Base Case (nil). We can see that

$$\begin{aligned}&\text{negate}(\text{twice}(\text{nil}))\\ &\quad= \text{negate}(\text{nil}) && \text{def of twice}\\ &\quad= \text{nil} && \text{def of negate}\\ &\quad= \text{twice}(\text{nil}) && \text{def of twice}\\ &\quad= \text{twice}(\text{negate}(\text{nil})) && \text{def of negate}\end{aligned}$$

Inductive Hypothesis. Suppose that $\text{negate}(\text{twice}(L)) = \text{twice}(\text{negate}(L))$ holds for some list L.

Inductive Step. We must prove that $P(x :: L)$ holds for any $x : \mathbb{Z}$.

$$\begin{aligned}&\text{negate}(\text{twice}(x :: L))\\ &\quad= \text{negate}(2x :: \text{twice}(L)) && \text{def of twice}\\ &\quad= -2x :: \text{negate}(\text{twice}(L)) && \text{def of negate}\\ &\quad= -2x :: \text{twice}(\text{negate}(L)) && \text{Inductive Hypothesis}\\ &\quad= \text{twice}(-x :: \text{negate}(L)) && \text{def of twice}\\ &\quad= \text{twice}(\text{negate}(x : L)) && \text{def of negate}\end{aligned}$$

Conclusion. $P(S)$ holds for any list S by structural induction.

Task 3 – Don't Set The Small Stuff [10 pts]

Answer the following questions about the specification of `BoundedIntSet` starting on the third-to-last page.

a) Fill in the following table showing the abstract state resulting **after** each operation on the left is performed, starting from the abstract state shown in the previous row. In the right column, show the result of calling `contains(9)` in the state listed in the middle column of that row.

	$(1, 10, 3 :: 9 :: \text{nil})$	`S.contains(9)`
`S.add(1)`	$(1, 10, 1 :: 3 :: 9 :: \text{nil})$	T
`S.add(2)`	$(1, 10, 2 :: 1 :: 3 :: 9 :: \text{nil})$	T
`S.remove(3)`	$(1, 10, 2 :: 1 :: 9 :: \text{nil})$	T
`S.remove(9)`	$(1, 10, 2 :: 1 :: \text{nil})$	F
`S.add(9)`	$(1, 10, 9 :: 2 :: 1 :: \text{nil})$	T

b) Write a **complete** JavaDoc specification for the method `remove`, starting with the English description provided on the second-to-last page.

You can assume that the mathematical function remove : $(\mathbb{Z}, \text{List}) \to \text{List}$ is already defined as follows and is well-known to the client:

$$\begin{aligned} \text{remove}(n, \text{nil}) &:= \text{nil} \\ \text{remove}(n, m :: L) &:= \text{remove}(n, L) && \text{if } m = n \\ \text{remove}(n, m :: L) &:= m :: \text{remove}(n, L) && \text{if } m \neq n \end{aligned}$$

```
/**
 * Returns a triple with the same bounds but a new list that no longer
 * contains n.
 * @param n The integer to remove from the set
 * @returns (min, max, remove(n, elems))
 */
```

Task 4 – Does a Bear Loop In the Woods? [15 pts]

The function add-const : ($\mathbb{Z}$, List) $\to$ List returns a list with the given constant value added to every element in the list. It is defined as follows:

$$\begin{aligned} \text{add-const}(x, \text{nil}) &:= \text{nil} \\ \text{add-const}(x, m :: L) &:= (x + m) :: \text{add-const}(x, L) \end{aligned}$$

Also, recall the function sum : (List) $\to \mathbb{Z}$, which is defined by:

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

The following loop claims to compute the sum of a list where five has been added to every element without actually constructing a new list:

$\{\!\{ L = L_0 \}\!\}$
```
int total = 0;
```
$\{\!\{ P_1 \text{: } L = L_0 \text{ and total} = 0 \}\!\}$
$\{\!\{ \text{Inv: sum(add-const}(5, L_0)) = \text{total} + \text{sum(add-const}(5, L)) \}\!\}$
```
while (L != null) {
```
$\{\!\{ P_2 \text{: sum(add-const}(5, L_0)) = \text{total} + \text{sum(add-const}(5, L)) \text{ and } L = L.\text{hd} :: L.\text{tl} \}\!\}$
$\{\!\{ Q_2 \text{: sum(add-const}(5, L_0)) = L.\text{hd} + 5 + \text{total} + \text{sum(add-const}(5, L.\text{tl})) \}\!\}$
```
    total = L.hd + 5 + total;
```
$\{\!\{ \text{sum(add-const}(5, L_0)) = \text{total} + \text{sum(add-const}(5, L.\text{tl})) \}\!\}$
```
    L = L.tl;
```
$\{\!\{ \text{sum(add-const}(5, L_0)) = \text{total} + \text{sum(add-const}(5, L)) \}\!\}$
```
}
```
$\{\!\{ P_3 \text{: sum(add-const}(5, L_0)) = \text{total} + \text{sum(add-const}(5, L)) \text{ and } L = \text{nil} \}\!\}$
$\{\!\{ Q \text{: sum(add-const}(5, L_0)) = \text{total} \}\!\}$

This code assumes that lists are represented with the same Java class as used in lecture:

```
public class List {
    public int hd;
    public List tl;
}
```

a) Use **forward** reasoning to fill in assertion P_1 above.

b) Use **forward** reasoning to fill in assertion P_3 above.

c) Use **backward** reasoning to fill in assertion Q_2 above.

Continued on the next page...

d) Prove that the invariant is true initially by showing that P_1 implies the invariant.

$$\begin{aligned}
\text{sum}(\text{add-const}(5, L_0)) &= 0 + \text{sum}(\text{add-const}(5, L_0)) \\
&= \text{total} + \text{sum}(\text{add-const}(5, L_0)) && \text{since total} = 0 \\
&= \text{total} + \text{sum}(\text{add-const}(5, L)) && \text{since } L = L_0
\end{aligned}$$

e) Prove that the postcondition Q holds by showing that P_3 implies Q.

$$\begin{aligned}
\text{sum}(\text{add-const}(5, L_0)) &= \text{total} + \text{sum}(\text{add-const}(5, L)) \\
&= \text{total} + \text{sum}(\text{add-const}(5, \text{nil})) && \text{since } L = \text{nil} \\
&= \text{total} + \text{sum}(\text{nil}) && \text{def of add-const} \\
&= \text{total} && \text{def of sum}
\end{aligned}$$

f) Prove that the body of the loop preserves the invariant by showing that P_2 implies Q_2.

$$\begin{aligned}
\text{sum}(\text{add-const}(5, L_0)) &= \text{total} + \text{sum}(\text{add-const}(5, L)) \\
&= \text{total} + \text{sum}(\text{add-const}(5, L.\text{hd} :: L.\text{tl}) && \text{since } L = L.\text{hd} :: L.\text{tl} \\
&= \text{total} + \text{sum}((L.\text{hd} + 5) :: \text{add-const}(5, L.\text{tl}) && \text{def of add-const} \\
&= \text{total} + L.\text{hd} + 5 + \text{sum}(\text{add-const}(5, L.\text{tl}) && \text{def of sum}
\end{aligned}$$

Task 5 – A Positive FeedStack Loop [10 pts]

Answer the following questions about the specification of `MutableIntStack` class on the last page:

a) Explain why the specifications of `push` and `pop` tell us that this ADT satisfies the LIFO principle (the **L**ast element **I**n is the **F**irst element **O**ut).
I.e., how does calling `pop` and `push` satisfy the principle?

A push followed by a pop will remove the element that was pushed, so the last element in (pushed) is the first element out (popped).

b) In what ways are the specifications of `pop` and `peek` the same?
Hint: Compare their tags :)

They both require the stack to be non-empty and return the same value.

c) In what ways are the specifications of `pop` and `peek` different?
Hint: Think about how the object will be affected by a call to `pop` vs. a call to `peek`.

They differ in the resulting abstract state (shorter vs unchanged).

d) Suppose that `S` is an instance of this class whose abstract state is $7 :: 3 ::$ nil. What is the abstract state of S after the following code:

```
T.pop();
T.push(5);
T.push(5);
T.push(1);
T.pop();
```

The resulting state would be $5 :: 5 :: 3 ::$ nil.

e) Consider the following implementation of a helper function, `pushIfPositive`:

```
public static void pushIfPositive(MutableIntStack stack, int n) {
  if (n > 0) {
      stack.push(n);
  }
}
```

Which of the following specifications is the **weakest** valid specification that it satisfies?

`@modifies n` and `@effects` $\texttt{stack}_0$ `= n :: stack`
`@modifies stack` and `@effects` $\texttt{stack}_0$ `= n :: stack`
`@modifies stack` and `@effects` $\texttt{stack}_0$ `= stack`
`@modifies stack` Correct

The last one is the weakest as it is the only valid specification (note: adding to `@modifies` weakens specification strength).

Interface for task 3:
Note that this interface is similar to `MutableBoundedIntSet` from HW5 task 4 but this version is **not** mutable and some specifications differ slightly.

```
/**
 * Represents a triple (min, max, elems), where min and max are integers,
 * elems is a list of integers, and every integer x in the list elems
 * satisfies min <= x <= max.
 */
public interface BoundedIntSet {

  /**
   * Determines whether n is in the list
   * @param n the integer to look for
   * @returns contains(n, elems), 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);

  /**
   * Return a triple with the same bounds but a new list that contains n
   * as well (if it is within the bounds).
   * @param n the integer to add to the list
   * @returns (min, max, n :: elems)  if min <= n <= max
   *               (min, max, elems)  if n < min or max < n
   */
  public BoundedIntSet add(int n);

  /**
   * Returns a triple with the same bounds but a new list that no longer
   * contains n.
   * ... (To be written in Task 3 part b) ...
   */
  public BoundedIntSet remove(int n);
}
```

Class for task 3: The following class will implement `BoundedIntSet` using one the concrete representation shown.

```
public class BoundedIntSetImpl implements BoundedIntSet {
  // AF: obj = (this.min, this.max, this.vals)
  // RI: all integers in this.vals between this.min and this.max (inclusive)

  private final int min;
  private final int max;
  private final int[] vals;

  public BoundedIntSet(int min, int max, int[] vals) {
    this.min = min;
    this.max = max;
    this.vals = vals;
  }
}
```

Interface for task 5:

```
/**
 * Represents a mutable list of integers that can be only changed by adding to
 * or removing from the front.
 */
public interface MutableIntStack {

  /**
   * Returns the size of the list
   * @returns len(obj)
   */
  public int size();

  /**
   * Adds a new integer to the front of the list
   * @param n the integer to add
   * @modifies obj
   * @effects obj = n :: obj_0
   */
  public void push(int n);

  /**
   * Removes and returns the front element from the list
   * @requires len(obj) != 0
   * @modifies obj
   * @effects obj_0 = n :: obj
   * @returns n
   */
  public int pop();

  /**
   * Returns the front element from the list
   * @requires len(obj) != 0
   * @returns n where obj = n :: L
   */
  public int peek();
}
```