CSE 331: Software Design & Implementation Fall 2025

Homework 8

Due: Friday, November 21st, 6pm

Task 1 – Jumping Through Loops [16 pts]

Recall the function from section Task 2:

```
/**
 * Writes over each copy of y in A with the value z.
 * @param A the array being modified, y the value to replace,
         z the value to replace y
 * @modifies A
 * @effects A = replace(A_0, y, z)
 */
public void replace(int[] A, int y, int z) { .. }
```

For each loop invariant below, fill in the missing parts of the code (the initial value of i and the loop condition and body) to make it correct with the **given invariant**. (If the code works correctly with some other invariant, it is not correct.)
Note about the notation: A[..i] means the subarray of A from index 0 up to and including index i; A[i+1 ..] means the subarray of A from index i+1 up to and including the last index.

a)

```
int i = __________

// Inv: A[.. i] = replace(A_0[.. i], y, z) and A[i+1 ..] = A_0[i+1 ..]
while (__________) {



}
```

b)

```
int i = __________

// Inv: A[.. i-1] = A_0[.. i-1] and A[i ..] = replace(A_0[i ..], y, z)
while (__________) {



}
```

Task 2 – The Join of the Realm [12 pts]

We can define a list of lists inductively as follows:

$$\textbf{type}\ \mathsf{LList} := \mathsf{nil} \mid \mathsf{lcons}(\mathsf{List}, \mathsf{LList})$$

As with other lists, we will use "::" as a shorthand for lcons as well.

The following function concatenates a list of lists into a single list:

$$\begin{aligned}
\mathsf{join}(\mathsf{nil}) &:= \mathsf{nil} \\
\mathsf{join}(\mathsf{nil} :: R) &:= \mathsf{join}(R) \\
\mathsf{join}((x :: L) :: R) &:= x :: \mathsf{join}(L :: R)
\end{aligned}$$

For example, we can see that

$$\begin{aligned}
&\mathsf{join}([[1,2],[3,4]]) && \\
&= \mathsf{join}((1 :: 2 :: \mathsf{nil}) :: (3 :: 4 :: \mathsf{nil}) :: \mathsf{nil}) && \\
&= 1 :: \mathsf{join}((2 :: \mathsf{nil}) :: (3 :: 4 :: \mathsf{nil}) :: \mathsf{nil}) && \text{def of join} \\
&= 1 :: 2 :: \mathsf{join}(\mathsf{nil} :: (3 :: 4 :: \mathsf{nil}) :: \mathsf{nil}) && \text{def of join} \\
&= 1 :: 2 :: \mathsf{join}((3 :: 4 :: \mathsf{nil}) :: \mathsf{nil}) && \text{def of join} \\
&= 1 :: 2 :: 3 :: \mathsf{join}((4 :: \mathsf{nil}) :: \mathsf{nil}) && \text{def of join} \\
&= 1 :: 2 :: 3 :: 4 :: \mathsf{join}(\mathsf{nil} :: \mathsf{nil}) && \text{def of join} \\
&= 1 :: 2 :: 3 :: 4 :: \mathsf{join}(\mathsf{nil}) && \text{def of join} \\
&= 1 :: 2 :: 3 :: 4 :: \mathsf{nil} && \text{def of join}
\end{aligned}$$

It is possible to prove (in the same manner as we did in Task 1 of section) that the following facts hold:

$$\begin{aligned}
\mathsf{join}(R + \mathsf{nil}) &:= \mathsf{join}(R) \\
\mathsf{join}(R + [L + [x]]) &:= \mathsf{join}(R + [L]) + [x]
\end{aligned}$$

The second statement says that the last element in the last list becomes the last element of the result.

You can use these facts in reasoning about the code on the next page.

The code below, once completed, will calculate `join`. It takes its input and output as `ArrayLists`.

Note that an `ArrayList`, like an array, represents the mathematical List type. However, its concrete representation leaves extra space at the end of the array so that we can append in constant time. You will want to take advantage of that extra operation below.

```
public ArrayList<Integer> join(ArrayList<ArrayList<Integer>> L) {
    ArrayList<Integer> R = _______________________;
    int j = __________;

    // Inv: R = join(L[.. j])
    while (________________) {
        ArrayList<Integer> T = L.get(__________);
        int k = _______;

        // Inv: R = join(L[.. j] ++ [T[.. k]])
        while (________________) {
            // fill in loop body to maintain inner invariant


        }

        j = __________;
    }

    // Post: R = join(L)
    return R;
}
```

a) Explain, in your own words, what the inner loop invariant says.

b) What list does T need to hold for this inner loop to be making progress on the outer invariant?

c) Fill in the missing parts of the code to make it correct with the **given invariants**. (If the code works correctly with some other invariant, it is not correct.)

Task 3 – The Sort Jester **[15 pts]**

The following code implements insertion sort, putting elements in the array A into increasing order.

```
int j = 0;
{{ $P_1$: ____________________________________ }}
{{ Inv$_1$ : $(A[i-1] \leqslant A[i]$ for any $i$: $0 < i < j)$ and $(A[i] \leqslant A[k]$ for any $i, k$: $0 \leqslant i < j \leqslant k < \text{len}(A))$ }}
while (j != A.length) {
    int m = j;
    {{ $P_2$: ________________________________ }}
    {{ Inv$_2$: Inv$_1$ and $(A[j] \leqslant A[k]$ for any $k$: $j \leqslant k < m)$ }}
    while (m != A.length) {
        if (A[j] > A[m]) {
            {{ $P_3$: ______________________________ }}
            {{ $Q_3$: ______________________________ }}
            int t = A[m];
            A[m] = A[j];
            A[j] = t;
        }
        m = m + 1;
    }
    {{ $P_4$: __________________________________ }}
    {{ $Q_4$: __________________________________ }}
    j = j + 1;
}
{{ $P_5$: ____________________________________ }}
{{ Post: $A[i-1] \leqslant A[i]$ for any $0 < i < \text{len}(A)$ }}
```

a) Explain, in your own words, what the outer invariant, Inv_1, says.

b) Fill in P_1. Then, show that it implies the outer invariant.

c) Fill in P_5. Then, show that it implies the postcondition.

d) Fill in P_2. Then, show that it implies the inner invariant, Inv_2.

e) Fill in P_4 using forward reasoning and Q_4 using backward reasoning. Then prove that P_4 implies Q_4.

f) Fill in P_3 using forward reasoning and Q_3 using backward reasoning. Then, prove that P_3 implies Q_3.

Note that the code in the inner loop does not modify any indices below j. This means that the first part of Inv_1 must remain true. Likewise, swapping elements in that range cannot break the second part of Inv_1. Hence, it is only the second part of Inv_2 that we need to check further.

Task 4 – Extra Credit: Feelin' Proovy [10 pts]

Prove that, if $n \leqslant \mathsf{len}(L)$, then $\mathsf{prefix}(L, n) \mathbin{+\!\!+} \mathsf{suffix}(L, n) = L$ by induction <u>on n</u>.

As a reminder, we define the following functions as such:

prefix : $(\mathsf{List}, \mathbb{N}) \to \mathsf{List}$

$$\mathsf{prefix}(L, 0) := \mathsf{nil}$$
$$\mathsf{prefix}(a :: L, n+1) := a :: \mathsf{prefix}(L, n)$$

suffix : $(\mathsf{List}, \mathbb{N}) \to \mathsf{List}$

$$\mathsf{suffix}(L, 0) := L$$
$$\mathsf{suffix}(a :: L, n+1) := \mathsf{suffix}(L, n)$$