Homework 7

Due: Wednesday, March 4th, 11:59pm

Task 1 – Assert Your Dominance [6 pts]

In this task, you will consider the following assertions about an array, `L`, of integers.

```
// Assertion A
L[i] > 0 for any 0 <= i < len(L)

// Assertion B
L[i] > 0 for any 0 <= i < len(L) - 1

// Assertion C
L[i] >= 0 for any 0 <= i < len(L)

// Assertion D
L[i] is even for 0 <= i < len(L) / 2

// Assertion E
L[i] is even for any 0 <= i < len(L) where i is even

// Assertion F
L[i] is even for any 0 <= i < len(L)
```

Fill in the blanks below to state whether the assertions are weaker, stronger, or incomparable.

Note that this question asks about *assertion* strength, not specification strength. Recall that an assertion P is *stronger* than an assertion Q if P implies Q.

Assertion A is ______________ than Assertion B.

Assertion A is ______________ than Assertion C.

Assertion B is ______________ than Assertion C.

Assertion D is ______________ than Assertion E.

Assertion D is ______________ than Assertion F.

Assertion E is ______________ than Assertion F.

Task 2 – College Kids and Their Celsius Addiction [16 pts]

In this problem, you will implement the following function, which takes a list containing temperatures in Fahrenheit and converts it into a list of the same temperatures in Celsius.

If a temperature t is in Fahrenheit, then the converted temperature in Celsius is $\frac{5}{9}(t-32)$, rounded down to the nearest integer.

```
/**
 * Converts a list of temperatures in Fahrenheit to Celsius
 * @param A a list of temperatures in Fahrenheit, which must not be null
 * @modifies A
 * @effects A = celsius(A_0), where celsius is defined by
 *     celsius(nil) := nil
 *     celsius(x :: L) := floor(5 * (x - 32) / 9) :: celsius(L)
 */
public void fahrToCels(int[] A) {
}
```

With each loop invariant below, fill in the missing parts of the code to make it correct with the **given invariant**. (If the code works correctly with some other invariant, it is not correct.)

a)
```
int i = ________________;

// Inv: A[.. i] = celsius(A_0[.. i]) and A[i+1 ..] = A_0[i+1 ..]
while (________________) {

   // fill in loop body

}
```

b)
```
int i = ________________;

// Inv: A[i ..] = celsius(A_0[i ..]) and A[.. i-1] = A_0[.. i-1]
while (________________) {

   // fill in loop body

}
```

c)
```
int i = ________________;

// Inv: A[.. i-1] = celsius(A_0[.. i-1]) and A[i ..] = A_0[i ..]
while (________________) {

   // fill in loop body

}
```

d)
```
int i = ________________;

// Inv: A[i+1 ..] = celsius(A_0[i+1 ..]) and A[.. i] = A_0[.. i]
while (________________) {

   // fill in loop body

}
```

Task 3 – Loop or Salad? [21 pts]

The function accumulate : (List) → List converts a list of numbers into a list of the sums of those numbers from each point to the end of the list. It is defined recursively as follows:

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

where sum : (List) → $\mathbb{Z}$ is defined by

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

For example, accumulate$(3 :: 2 :: 1 :: \mathsf{nil}) = 6 :: 3 :: 1 :: \mathsf{nil}$.

A direct translation of this into Java code would give an $O(n^2)$ algorithm. The following code, which runs in $O(n)$ time, claims to calculate the same result. Below, you will prove that it is correct.

```
{{ $P_0$: len$(A) > 0$ and $A = A_0$ }}
int i = A.length;
{{ $P_1$: ________________________________________ }}
{{ Inv: $A[..\ i-1] = A_0[..\ i-1]$ and $A[i\ ..] = \mathsf{accumulate}(A_0[i\ ..])$ }}
while (i != 0) {
    if (i == A.length) {
      {{ $P_2$: ________________________________ }}
      {{ $Q_2$: ________________________________ }}
      // do nothing
    } else {
      {{ $P_3$: ________________________________ }}
      {{ $Q_3$: ________________________________ }}
      A[i-1] = A[i-1] + A[i];
    }
    i = i - 1;
    {{ $A[i] :: A[i+1\ ..] = \mathsf{accumulate}(A_0[i] :: A_0[i+1\ ..])$ }}
}
{{ $P_4$: ________________________________________ }}
{{ $Q$: $A = \mathsf{accumulate}(A_0)$ }}
```

a) Fill in P_1, P_2, P_3, and P_4 above by **forward** reasoning.

b) Fill in Q_2 and Q_3 by **backward** reasoning from the assertion at the bottom of the loop.

c) Prove that P_1 implies that Inv holds initially.

d) Prove that P_4 implies the postcondition Q.

e) Prove that P_3 implies Q_3. Feel free to use the following two facts in your proof:

• $A[i] = \mathsf{sum}(A_0[i\ ..])$	This holds since $A[i\ ..] = \mathsf{accumulate}(A_0[i\ ..])$.[1]
• $A[i-1] = A_0[i-1]$	This holds since $A[..\ i-1] = A_0[..\ i-1]$.[2]

f) Prove that P_2 implies Q_2. Feel free to use the following two facts in your proof:

• $\mathsf{len}(A) = \mathsf{len}(A_0)$	This holds since we have not changed the length of the array
• $A[i-1] = A_0[i-1]$	This holds for the same reason as in part (e)

[1]The fact that the latter two lists are equal means all their elements are equal. $A[i]$ is the first element of the left list and $\mathsf{sum}(A_0[i\ ..])$ is the first element of the right list, so those two elements must be equal.

[2]Again, list equality means all the elements are equal. In this case, we want their last elements.

Task 4 – Up, Up, and Array! [12 pts]

We can define a list of lists inductively as follows:

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

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

The following function merge : (LList) → List merges a list of lists into a single list, where R is of LList type:

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

For example, we can see that

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

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

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

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

The code below, once completed, will calculate `merge`. It takes its input and output as `ArrayList`s. You may use Java list methods to fill in the loop body.

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> merge(ArrayList<ArrayList<Integer>> L) {
    ArrayList<Integer> R = _______________________;
    int j = __________;

    // Inv: R = merge(L[.. j - 1]) and j <= len(L)
    while (________________) {
        ArrayList<Integer> T = L.get(j);
        int k = 0;

        // Inv: R = merge(L[.. j - 1] ++ [T[.. k - 1]]) and
        //      j <= len(L) and k <= len(T)
        while (________________) {
            // fill in loop body to maintain inner invariant


        }

        j = __________;
    }

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

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

b) How should we initialize `R` and `j` for this function?

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

Task 5 – Extra Credit: Imply and Demand [0 pts]

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

Note: While this may look familiar from lecture, there is a difference becaus of the if statement that precedes the equality (it's an implication!). This is part of the claim that you must prove.

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

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

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

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

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

Introduction

Define $P(n)$ to be the claim that **if n $\leqslant$ len(L),**

____

Base Case

Inductive Hypothesis

Suppose that ____

Inductive Step

Conclusion