# Homework 2

Due: Wednesday, April 22nd, 11:59pm

**Before you start**: This homework requires some formatting. Please bold or highlight your assertions. When proving implications, a brief English explanation is sufficient (you do not need a formal proof by calculation).

## Task 1 – Reason Your Delivery [10 pts]

In this problem, you will practice proving correctness of code with conditionals using forward and backward reasoning. Recall from lecture that we can use **assertions** (true/false claims at specific program points) to reason about what is true before and after each line of code.

Remember to **bold or highlight your assertions**.

**a)** A local restaurant charges a delivery fee that depends on how far away the customer is. Consider the following method that computes this fee:

```
/**
 * Computes the delivery fee for an order.
 * @param miles the distance to the customer in miles
 * @requires miles >= 0
 * @return the delivery fee, which must be at least 5
 */
public static int deliveryFee(int miles) {
    int fee;
    if (miles >= 8) {
        fee = 3 * miles - 10;
    } else {
        fee = 20 - miles;
    }
    return fee;
}
```

The precondition and postcondition for this method come from its specification: the `@requires` clause tells us what we can assume about the input, and the `@return` clause tells us what the method must guarantee about the output.

Use **forward** reasoning to fill in the assertions below. Then, show that the code is correct by explaining briefly (in a few sentences) why $P$ implies $Q$.

$\{\{\, miles \geqslant 0 \,\}\}$

```
if (miles >= 8) {
  {{ ______________________________ }}
  fee = 3 * miles - 10;
  {{ ______________________________ }}
} else {
  {{ ______________________________ }}
  fee = 20 - miles;
  {{ ______________________________ }}
}
```

$\{\{\, P:$ ____________________ or ____________________ $\}\}$

$\{\{\, Q: \ fee \geqslant 5 \,\}\}$

**b)** A game awards bonus points to players based on their current level. Consider the following method:

```
/**
 * Computes the bonus points awarded to a player.
 * @param level the player's current level
 * @requires level >= 0
 * @return the bonus points, which is at least 3
 */
public static int bonusPoints(int level) {
    int bonus;
    if (level >= 6) {
        bonus = 2 * level - 5;
    } else {
        bonus = 15 - level;
    }
    return bonus;
}
```

Use **forward** reasoning to fill in $P_1$ and $P_2$ (what is known upon entering each branch). Use **backward** reasoning to fill in $Q_1$ and $Q_2$ (what must be true before each assignment in order for the postcondition to hold) and the other assertions.

Fill in each blank by applying the reasoning rules as taught in lecture. You may simplify the resulting assertion, but do not weaken it. If you simplify an assertion, use "$\leftrightarrow$" to separate the simplified assertion from the original.

Then, show that the code is correct by explaining briefly why $P_1$ implies $Q_1$ and $P_2$ implies $Q_2$.

$\{\{\, level \geqslant 0 \,\}\}$
```
if (level >= 6) {
```
$\{\{\, P_1 :$ ________________________________________ $\}\}$

$\{\{\, Q_1 :$ ________________________________________ $\}\}$
```
    bonus = 2 * level - 5;
```
$\{\{$ ________________________________________ $\}\}$
```
} else {
```
$\{\{\, P_2 :$ ________________________________________ $\}\}$

$\{\{\, Q_2 :$ ________________________________________ $\}\}$
```
    bonus = 15 - level;
```
$\{\{$ ________________________________________ $\}\}$
```
}
```
$\{\{\, bonus \geqslant 3 \,\}\}$

## Task 2 – Hoare Lie in April **[12 pts]**

Code snippets A and B below both claim to compute the sum of a list of integers whose elements were multiplied by a factor of 4:

A.

$\{\{\, L = L_0 \,\}\}$

```
int z = 0;
int x = 4;
```

$\{\{$ Inv: The sum of $L_0$'s elements each multiplied by $x =$
$z +$ the sum of $L$'s elements each multiplied by $x \}\}$

```
while (L != null) {
```

$\{\{\, P_0$: ______________________________ $\}\}$

$\{\{\, Q_2$: ______________________________ $\}\}$

```
    z = z + L.hd * x;
```

$\{\{\, Q_1$: ______________________________ $\}\}$

```
    L = L.tl;
```

$\{\{\, Q_0$: ______________________________ $\}\}$

```
}
```

$\{\{\, z =$ the sum of $L_0$'s elements each multiplied by $x \,\}\}$

B.

$\{\{\, L = L_0 \,\}\}$

```
int d = 0;
int v = 4;
```

$\{\{$ Inv: The sum of $L_0$'s elements $= d +$ the sum of $L$'s elements $\}\}$

```
while (L != null) {
```

$\{\{\, P_0$: ______________________________ $\}\}$

$\{\{\, Q_2$: ______________________________ $\}\}$

```
    d = d + L.hd;
```

$\{\{\, Q_1$: ______________________________ $\}\}$

```
    L = L.tl;
```

$\{\{\, Q_0$: ______________________________ $\}\}$

```
}
d = d * v
```

$\{\{\, d = v \cdot$ the sum of $L_0$'s elements $\}\}$

**a)** Fill in the remaining assertions for A. Fill in $P_0$ and $Q_0$ using the Floyd logic rule for while loops. Use **backward reasoning** to derive $Q_1$ and $Q_2$. Remember to **bold or highlight your assertions**!

**b)** Does $P_0$ imply $Q_2$ in snippet A? Informally explain why or why not. (No proof required.)

**c)** Fill in the remaining assertions for B, in the same manner as part (a).

**d)** Does $P_0$ imply $Q_2$ in snippet B? Informally explain why or why not. (No proof required.)

**e)** Is the value in $z$ at the end of snippet A equal to the value in $d$ at the end of snippet B? Informally explain your answer in 1 or 2 sentences. (No proof required.)

## Task 3 – Little Endian, Big Idea [12 pts]

In this problem, you will practice reasoning about loops. This code accepts a list of bits (0 or 1), which collectively represent a binary number. These bits are stored in **little-endian** order: the head stores the least significant bit, and each successive node moves you toward a more significant bit, with the last node storing the most significant bit.

The Digits class represents this linked list. The Java class is defined below:

```
public class Digits {
    public int digit;   // always 0 or 1
    public Digits next;
}
```

We will define toInt($L$) to make the loop invariant more concise. L is an instance of the Digits class.

- If `L` is `null`, then toInt$(L) = 0$.
- If `L` is not `null`, then toInt$(L) = \texttt{L.digit} + 2 \cdot$ toInt$(\texttt{L.next})$.

For example, the number $4$ is represented as $001_2$ in **little endian** notation.

$$0 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 = 4.$$

As a list of Digits, the number $4$ is represented as 0::0::1::null. If expanded out, toInt(0::0::1::null) returns 4 (Write it out and convince yourself!).

In the code below, $L_0$ is the value of `L` before the loop (it may be `null` if the number is $0$). At the start of each loop iteration, `L` points at the current node which has not yet been processed.

We have provided the loop invariant in math. In English, the invariant states that the integer representation of $L_0$ is equal to $v$ plus $b$ times the integer representation of $L$.

$\{\{\ L = L_0\ \}\}$

```
int b = 1;
int v = 0;
```

$\{\{$ Inv: toInt$(L_0) = v + b \cdot$ toInt$(L)\ \}\}$

`while (L != null) {`

$\{\{\ P_1$: `L != null` and ____________________________ $\}\}$

$\{\{$ ______________________________________________ $\}\}$

`v = v + b * L.digit;`

$\{\{$ ______________________________________________ $\}\}$

`b = 2 * b;`

$\{\{$ ______________________________________________ $\}\}$

`L = L.next;`

$\{\{$ ______________________________________________ $\}\}$

$\{\{$ Inv: toInt$(L_0) = v + b \cdot$ toInt$(L)\ \}\}$

`}`

$\{\{\ P_2$: `L == null` and toInt$(L_0) = v + b \cdot$ toInt$(L)\ \}\}$

$\{\{\ v =$ toInt$(L_0)\ \}\}$

**a)** Briefly explain why the invariant is true immediately before the first iteration of the while loop (directly after `b` and `v` are initialized).

**b)** Briefly explain why the postcondition $v =$ toInt$(L_0)$ holds immediately after the loop exits.

**c)** Fill in the assertion $P_1$.

**d)** Use either **forward or backward reasoning** to reason through the loop body (do not mix the two styles in one answer). State your choice, fill in the assertions, then briefly explain why the invariant is maintained.

## Task 4 – Hoare's Code [12 pts]

Unfortunately, while scheming up his next functions to write, Benoit Le also gets his share of cosmic rays (Thanks sun!) corrupting his files and deleting parts of his valuable code. Luckily, Benoit Le is a good code commenter and has reasoned through his code, writing assertions using Floyd Logic. In **parts a, b, and c**, fill in the missing code to satisfy the Hoare triple. Then, explain briefly why the resulting assertion is satisfied.

### a)

```
public static int attemptSteal(int leadDistance, int currentBase) {
    int result;
   {{ 0 ⩽ currentBase ⩽ 2 }}
    // TODO: Write a conditional to satisfy the resulting assertion

   {{ (leadDistance > 12 and result = currentBase + 1) or
              (leadDistance ⩽ 12 and result = 0) }}
    return result;
}
```

### b)

```
public static int max(List L) {
    int max = L.hd;
    List curr = L.tl;
    {{ Inv: max stores the max of all elements seen so far }}
    while (curr != null) {
        // TODO: Fill in the loop to satisfy the invariant

    }
    {{ max stores the max of all elements in L }}
    return max;
}
```

**c)**

```
public static int longestStreak(List games) {
   {{ games is a List of 0s and 1s }}
   int maxStreak = 0;
   int currentStreak = 0;
   List curr = games;
   {{ Inv:
        1. maxStreak is the length of the longest sequence of 1s so far
        2. currentStreak is the length of the sequence of 1s ending exactly at but
           not including curr
        3. currentStreak ⩽ maxStreak }}
   while (curr != null) {
      // TODO: Fill in the loop to satisfy the invariant

   }
   {{ maxStreak is the length of the longest sequence of 1s in games }}
   return maxStreak;
}
```

**d)** Former 331 student Daniel finishes part (a), but decides to leave the loop body for part (b) and (c) empty, claiming that the loop invariant is still satisfied.

**i.** Is Daniel technically correct that the Hoare triple is valid when the loop body is empty? Explain why this is insufficient to prove that the function will eventually return the maximum value or longest streak.

**ii.** A program is **partially correct** if, *whenever it terminates*, it is guaranteed to satisfy the post-condition. Explain why Daniel's "empty" loop implementation is formally partially correct.

**iii.** To ensure a program is correct and finishes, we must prove **total correctness**, which requires both partial correctness and **termination**. What specific line of code in your part (b) and (c) implementation ensures progress toward termination?

## Task 5 – We Are Reason. [12 pts]

Faraz has a brilliant new billion dollar startup idea he wants to secure funding for: a cybersecurity B2B SaaS AI company! His presentation to VCs is in two days. However, in a great twist of cartoonish irony, his files have been compromised! Faraz doesn't know what might have gotten affected, so he needs your help to understand what the methods below are doing.

Below are two implementations of methods Faraz wrote.

**a)** The first method and its specification are below.

```
/**
 * Performs a highly secure operation on x and returns the result.
 * @param x a positive integer
 * @return a positive integer
 */
public static int topSecretEncryption(int x) {
  {{ ________________________________________ }}
   int y = x;
  {{ ________________________________________ }}
   x = x + 10;
  {{ ________________________________________ }}
   y = y - 5;
  {{ ________________________________________ }}
   x = x + y;
  {{ ________________________________________ }}
   x = x - 5;
  {{ ________________________________________ }}
   x = x / 2;
  {{ ________________________________________ }}
   return x;
}
```

You must use forward reasoning to fill in the assertions.

**b)** Faraz later consults Staff Engineer Lawrence, who informs him that his method is not very secure. He reflects on Lawrence's advice and thinks long and hard to rewrite the method. Below is the method that he ended up with:

```
/**
 * Performs a highly secure operation on x and returns the result.
 * @param x a positive integer
 * @return a positive integer
 */
public static int topSecretEncryption(int x) {
  {{ ________________________________________ }}
   int a = x * x;
  {{ ________________________________________ }}
   x = x + 1;
  {{ ________________________________________ }}
   int b = x * (x + 1);
  {{ ________________________________________ }}
   int c = b - a - 7;
  {{ ________________________________________ }}
   int d = (c + 5) / 3;
  {{ ________________________________________ }}
   return d;
}
```

You must use backwards reasoning to fill in the assertions.

**c)** Writing a spec with the weakest possible precondition and the strongest possible postcondition is often the most intuitive thing to do. However, this may not always be ideal for us as programmers. Give an example scenario where it may be more beneficial to have a stronger precondition or a weaker postcondition.