Quiz Section 3: Floyd Logic

Task 1 – Found Guilty of Reason

In this problem, you will practice proving correctness of straight-line code using **forward reasoning**.

Fill in each blank by applying the rules *exactly* as taught in lecture. Then, if you want, you can simplify the resulting assertion, but do not weaken it (such as substituting/dropping facts). Separate any simplified statement from the original by "↔".

a) Use forward reasoning to fill in the missing assertions in the following code:

```
{{ y > 5 and z > 2 }}
x = 4 * y - 3;
{{ ____________________________________ }}
y = y - 5;
{{ ____________________________________ }}
z = z * y;
{{ P: _________________________________ }}
{{ Q: x < 2z + 20 }}
```

b) Show that the code is correct by explaining how P implies Q.

c) First, fill in Q using the postcondition (hint, how does `y` relate to the `@returns` clause?). Then, use forward reasoning to fill in the missing assertions in the following code:

```
// Computes the square of (x + 1)
// @param x The number to increment and square
// @return (x + 1)^2
public static int nextSquare(int x) {
  int y = x * x;
  {{ ____________________________________ }}
  y = y + (2 * x);
  {{ ____________________________________ }}
  y = y + 1;
  {{ P: _________________________________ }}
  {{ Q: _________________________________ }}
  return y;
}
```

d) Show that the code is correct by explaining how P implies the postcondition.

Task 2 – Does a Duck Say "Back"?

In this problem, you will practice proving correctness of straight-line code using **backward reasoning**.

a) Use backward reasoning to fill in the missing assertions in the following code:

```
{{ P: x < w + 1 and w > 0 }}
{{ Q: ____________________________________ }}
y = 4 * w;
{{ ____________________________________ }}
x = x * 2;
{{ ____________________________________ }}
z = x - 8;
{{ z < y }}
```

b) Show that the code is correct by explaining how P implies Q.

c) Use backward reasoning to fill in the missing assertions in the following method:

```
// Performs a simple calculation on c
// @param c The input integer
// @requires c >= 0
// @returns an integer >= c
public static int calculation(int c) {
 {{ P: c >= 0 }}
 {{ Q: ____________________________________ }}
 b = 2 * c;
 {{ ____________________________________ }}
 c = c - 1;
 {{ ____________________________________ }}
 a = b + 1;
 {{ a >= c }}
 return a;
}
```

d) Show that the code is correct by explaining how the precondition implies Q.

Task 3 – Nothing To Be If-ed At

Use forward reasoning to fill in the assertions. Explain how what we know at the end of the conditional implies the post condition.

```
// Calculates a safe substring length starting from a given index.
// @param start The starting index of the substring
// @param desired The desired substring length
// @param len The total length of the original string
// @requires 0 <= start < len and desired >= 0
// @return A window size such that start + window <= len
public static int calcWindow(int start, int desired, int len) {
  {{ ____________________________________ }}
  int window;
  if (start + desired <= len) {
    {{ ____________________________________________________________ }}
    window = desired;
    {{ P1: ________________________________________________________ }}
  } else {
    {{ ____________________________________________________________ }}
    window = len - start;
    {{ P2: ________________________________________________________ }}
  }
 {{ ____________________________________ }}
  return window;
}
```

Task 4 – Everybody Loops

In this problem, we will prove the correctness of a method containing a loop that finds the quotient of x divided by 10, i.e., the *largest* value y such that $10y \leqslant x$. To say that y is the largest such value means that any larger value would not satisfy the inequality, i.e., that $10(y+1) \nleqslant x$.

We denote the initial value of the parameter x at the top of the method by x_0. This is explicitly stated in the precondition as the fact "$x = x_0$" (note this is not strictly necessary - you always know $x = x_0$ until you modify it!). The first two facts of Q are from the spec postcondition, which say that y is the quotient of x_0 divided by 10. The third fact is specific to our implementation, and says that x is the remainder, i.e., the remaining amount not divisible by 10.

This method calculates the quotient without division. Instead, it just uses subtraction. It operates by increasing y and decreasing x each time around. The first part of the invariant says that the distance from x_0 down to $10y$ (i.e., $x_0 - 10y$) is the same as the distance from x down to 0 (i.e., $x - 0 = x$). The second part of the invariant says that x has not moved below 0 (i.e., $x \geqslant 0$).

```
// Computes the integer quotient of x divided by 10
// @param x The numerator
// @requires x >= 0
// @return The largest integer y such that 10 * y <= x_0
public static int divideByTen(int x) {
  {{ x = x_0 and x_0 >= 0 }}
  int y = 0;
  {{ P1: _____ }}
  {{ Inv: x_0 - 10y = x and x >= 0 }}
  while (x >= 10) {
    {{ _____ }}
    y = y + 1;
    {{ _____ }}
    x = x - 10;
    {{ P3: _____ }}
    {{ Q2: _____ }}
  }
  {{ P2: _____ }}
  {{ Q1: 10y <= x_0 and x_0 < 10(y+1) and x = x_0 - 10y }}
  return y;
}
```

a) Fill in P1, then show that the invariant is true when we get to the top of the loop the first time.

b) Fill in P2, then show that Q1 holds when we exit the loop.

c) Fill in Q2 (Hint: what do we know at the end of a loop?). Then, forward reason to P3. Show that P3 implies Q2, proving that the body of the loop is correct.