CSE 331: Software Design & Implementation
Section 2 – Code Reasoning

For these problems, assume that all numbers are integers, that integer overflow will never occur, and that division is truncating integer division (like in Java).

1. Fill in the blanks using **forward** reasoning.

```
{ x >= 0 ∧ y >= 0 }
y = 16;

{ x >= 0 ∧ y = 16 }
x = x + y;

{ x >= 16 ∧ y = 16 }
x = sqrt(x);

{ x >= 4 ∧ y = 16 }
y = y - x;

{ x >= 4 ∧ y = 16 - x }

⇒ { x >= 4 ∧ y <= 12 }
```

2. Fill in the blanks using **forward** reasoning.

```
{ true }
if (x>0) {

     { x > 0 }
     y = 2x;

     { x > 0 ∧ y = 2x }
} else {

     { x <= 0 }
     y = -2*x;

     { x <= 0 ∧ y = -2x }
}

{ (x > 0 ∧ y = 2x) ∨ (x <= 0 ∧ y = -2x) }

⇒ { y = 2|x| }
```

3. Fill in the blanks using **backward** reasoning.

```
{ x + 3 * b - 4 > 0                                  }
a = x + b;

{ a + 2 * b - 4 > 0                                  }
c = 2 * b - 4;

{ a + c > 0.                                         }
x = a + c;

{ x > 0 }
```

4. Fill in the blanks using **backward** reasoning.

```
{ y > 15 V (y <= 5 Λ y + z > 17)                     }
if (y > 5) {

     { y > 15                                        }
     x = y + 2

     { x > 17                                        }
} else {

     { y + z > 17                                    }
     x = y + z;

     { x > 17                                        }
}
{ x > 17 }
```

5. Fill in the blanks using **backward** reasoning (extra practice problems).

a.

```
{ x - 1 != 0                                         }
x = x - 2;

{ x + 1 != 0                                         }
z = x + 1;

{ z != 0 }
```

b.

```
{ 3 * y > 0                                          }
x = 2 * y;

{ x + y > 0                                          }
z = x + y;

{ z > 0 }
```

c.

```
{ v + 1 < 0 V -2 * w < 0                             }
w = 2 * w;

{ v + 1 < 0 V -w < 0                                 }
z = -w;

{ v + 1 < 0 V z < 0                                  }
y = v + 1;

{ y < 0 V z < 0                                      }
x = min(y, z);

{ x < 0 }
```

6. Prove that the following code computes the minimum.

This can be done using either forward or backward reasoning. Or a combination!
Forward reasoning:

```
{ true }
if (x < y) {

  { true /\ x < y                                    }
  m = x;

  { x < y /\ m = x                                   }
} else {

  { true /\ x >= y                                   }
  m = y;

  { x >= y /\ m = y                                  }
}
{ (x < y /\ m = x) \/ (x >= y /\ m = y)              }

{ m = min(x, y) }
```

Backward reasoning:

```
{ (x <= y /\ x < y) \/ (y <= x /\ x >= y) } ⇒ { true }
if (x < y) {

  { x <= y                                          }
  m = x;

  { m = min(x, y)                                   }
} else {

  { y <= x                                          }
  m = y;

  { m = min(x, y)                                   }
}
{ same                                              }

{ m = min(x, y) }
```

7. For each pair of logical assertions, circle the **stronger** logical assertion (or indicate that the pair is **incomparable**).

a.	`{ y > 23 }`	`{ y >= 23 }`
b.	`{ y = 23 }`	`{ y >= 23 }`
c.	`{ y < 0.23 }`	`{ y < 0.00023 }`
d.	`{ x = y * z }`	`{ y = x / z }` **(incomparable)**
e.	`{ is_prime(y) }`	`{ is_odd(y) }` **(incomparable)**

7d is fairly non-trivial, requiring close examination of possible program states. Also, remember that variables are always integers, and a forward slash is always truncating integer division unless otherwise stated.

To see that the assertions in 7d are incomparable, consider the following program states: x is 5, z is 2, and y is 2; and x is 0, z is 0, and y is 0.

8. Verify that the following code correctly determines if x is power of 2.

```
{{ Precondition: x >= 1 }}

int k = 0;
int y = 1;

{{ Inv: y = 2^k and y/2 < x }}
while (y < x) {
  y = y * 2;
  k = k + 1;
}

{{ y = 2^k and y/2 < x <= y }}

if (y == x) {
  {{ Postcondition: x is a power of 2 }}
  return true;
} else {
  {{ Postcondition: x is not a power of 2 }}
  return false;
}
```

- **Inv is true initially since, when k = 0, we have $y = 2^k = 2^0 = 1$, which is true.**
- **Inv remains true around the loop body:**
  - **backward reasoning turns $y = 2^k$ into $y = 2^{k+1}$ and then $2y = 2^{k+1}$, which is Inv again once we divide both sides by 2.**
  - **Backward reasoning turns $y/2 < x$ into $2y/2 < x$, or equivalently, $y < x$. This is true at the top of the loop by the loop condition.**
- **Thus, we exit the loop with $y = 2^k$ and $y/2 < x$ and not $(y < x)$, i.e., $x <= y$.**
- **In the "if" case of the if statement, we have $x = y = 2^k$, so x is indeed a power of 2.**
- **In the "else" case of the if statement, we have $2^{k-1} = y/2 < x < y = 2^k$, so x cannot be a power of 2 because it falls between subsequent powers of 2.**
  - **Note: the "y/2 < x" part of the invariant was not needed for the loop itself, but we needed to keep track of it so that we could prove this "else" is correct!**