CSE 331 Autumn 2022 HW1

General rules:

- For logical operators, you may use words (e.g., "or") or any standard symbols (e.g., "V").
- Assume that
  - all numbers are integers
  - integer overflow will never occur
  - integer division rounds toward zero (as in Java)
- Simplify but **do not weaken** (i.e., change the set of described states) in your assertions.
- Wherever possible, rewrite your assertions to only refer to the current state of variables rather than using subscripts.

1. **Forward reasoning with assignment statements.** Find the strongest postcondition for each sequence using forward reasoning, writing the appropriate assertion in each blank space. The first assertion in part (a) is supplied as an example.

   a. {{ }}
   x = 4;
   {{ x = 4 }}
   y = 5 * x;
   {{ _____ }}
   z = y / 4;
   {{ _____ }}
   y = x + z;
   {{ _____ }}

   b. {{ $0 \leq x < 100$ }}
   y = x + 1;
   {{ _____ }}
   z = y – 4;
   {{ _____ }}
   x = y – z;
   {{ _____ }} (be careful!)

   c. {{ $50 \leq x < 100$ }}
   x = x – 50;
   {{ _____ }}
   x = x / 10;
   {{ _____ }}
   x = 10 – x;
   {{ _____ }}

2. **Backward reasoning with assignment statements.** Find the weakest precondition for each sequence using backward reasoning, writing the appropriate assertion in each blank space.

   a. {{ _____ }}
   x = x * 2;
   {{ _____ }}
   y = 5 + x;
   {{ $1 \leq y \leq 15$ }}

   b. {{ _____ }}
   y = 6 – x;
   {{ _____ }}
   x = x * 3;
   {{ $x \leq y$ }}

   c. {{ _____ }} (be careful!)
   x = x / 3;
   {{ _____ }}
   y = x – 5;
   {{ $1 \leq y \leq 10$ }}

   d. {{ _____ }}
   z = v + 2;
   {{ _____ }}
   x = 2w + 4;
   {{ _____ }}
   y = 2*z;
   {{ $y \leq 0 \leq x$ }} (… or if you prefer {{ $y \leq 0$ and $0 \leq x$ }})

3. **Forward reasoning with if/else statements.** Find the strongest postcondition for the following conditional statement using forward reasoning, inserting the appropriate assertion in each blank.

a. {{ $0 \leq x \leq 15$ }}

if (x <= 10)

{{ _____ }}

y = 5;

{{ _____ }}

else

{{ _____ }}

y = x / 5;

{{ _____ }}

{{ _____ }}

b. {{ }}

if (x < 4)

{{ _____ }}

x = 3 * x;

{{ _____ }}

else

{{ _____ }}

x = x + 8;

{{ _____ }}

{{ _____ }}

4. **Backward reasoning with if/else statements.** Find the weakest precondition for the following conditional statement using backward reasoning, inserting the appropriate assertion in each blank.

a.
```
{{ _____ }}
if (x >= 10)
    {{ _____ }}
    y = x – 5;
    {{ _____ }}
else
    {{ _____ }}
    y = 2 * x;
    {{ _____ }}
{{ 0 ≤ y ≤ 10 }}
```

b.
```
{{ _____ }}
if (x >= 0)
    {{ _____ }}
    x = -x;
    {{ _____ }}
else
    {{ _____ }}
    x = x + 1;
    {{ _____ }}
{{ x < 10 }}
```

5. **Hoare triples.** State whether each Hoare triple is valid. If it is invalid, give a counterexample.

   a. {{ $0 \le x \le 99$ }}
   x = x + 1;
   {{ $1 < x < 100$ }}

   b. {{ $20 < x < 100$ }}
   y = x - 10;
   {{ $10 \le y \le 90$ }}

   c. {{ $0 \le x \le 200$ }}
   if (x > 100)
   x = x / 2;
   else
   x = x + 50;
   {{ $x \ge 50$ }}

   d. {{ -100 < x < 100 }}
   if (x > 0)
   x = x / 2;
   else
   x = (100 + x) / 2;
   {{ x < 50 }}

6. **Weakest conditions.** Circle the weakest condition in each list.

   a. {{ x < 0 }} {{ x < 5 }} {{ x < -5 }}

   b. {{ x > 1 or y > x }} {{ x > 1 and y > 1 }} {{ x > 1 or (y > x and x > 2) }}

   c. {{ if x > 1, then y > x }} {{ if x > 1, then y > 1 }} {{ x > 1 and y > 1 }}

   d. {{ b > 0 }} {{ |b| != 3 }} {{ b != -3 }}

7. **Verifying correctness.** For each block of code, fill in the intermediate assertions in the direction indicated by the arrows. Finally, state whether the code is correct (i.e., whether *all* triples are valid).

   a. {{ 1 ≤ x }}
   ↓ y = x + 5;
   {{ _____ }}
   ↓ x = 2 * x;
   {{ _____ }}
   z = x / 2 + 3;
   {{ 4 ≤ z }}

   b. {{ x ≤ 6 }}
   y = 2 * x;
   {{ _____ }}
   ↑ x = x * 3;
   {{ _____ }}
   ↑ z = x – 6;
   {{ z < y }}

   c. {{ 0 < x }}
   ↓ **if** (y >= 5*x)
   {{ _____ }}
   x = 25 / y;
   {{ _____ }}
   else
   {{ _____ }}
   x = y / x;
   {{ _____ }}
   ↑
   {{ x ≤ 5 }}

8. **Verifying correctness of loops.** Fill in the missing assertions by reasoning in the direction indicated by the arrows. Then, in the places where two assertions appear next to each other with no code between (see the "?"s), provide an explanation of why the top assertion implies the bottom one.

   Note: You may use "n" as a short hand for "A.length".

```
{{ Pre: }} (i.e., nothing is assumed other than A is not null, which is an implicit constraint)
int find(int[] A, int val) {
  {{ ____________________________________________________________ }}
↓ int i = -1;
  {{ ____________________________________________________________ }}

?

  {{ Inv: A[0] != val, A[1] != val, …, A[i] != val }}
  while (i+1 != A.length) {
↓
    {{ ____________________________________________________________ }}
↓   if (A[i+1] == val) {
      {{ ____________________________________________________________ }}

?

      {{ Post: A[i+1] = val }}
      return i+1;
    } else {
↓     {{ ____________________________________________________________ }}

?

      {{ ____________________________________________________________ }}
↑   }
    {{ ____________________________________________________________ }}
↑   i = i + 1;
    {{ ____________________________________________________________ }}
↑
  }
↓
  {{ A[0] != val, A[1] != val, …, A[i] != val and i+1 = A.length }}

?

  {{ Post: A[0] != val, A[1] != val, …, A[n-1] != val }}
  return -1;
}
```

9. **More loop correctness.** Fill in the missing assertions by reasoning in the direction indicated by the arrows. Then, in the places where two assertions appear next to each other with no code between (see the "?"s), provide an explanation of why the top assertion implies the bottom one.

   Notation: You may use "n" as a short-hand for "A.length".

```
{{ Pre: 0 < n }}
float evalPoly(float[] A, float v) {
↓ int i = A.length – 1;
  {{ ____________________________________________________________ }}
↓ int j = 0;
  {{ ____________________________________________________________ }}
↓ float val = A[i];
  {{ ____________________________________________________________ }}

?


  {{ Inv: val = A[i] + A[i+1] v + … + A[n-1] v^j and i + j = n – 1 }}
  while (j != A.length – 1) {
↓
    {{ ____________________________________________________________ }}
↓   j = j + 1;
    {{ ____________________________________________________________ }}
↓   i = i – 1;
    {{ ____________________________________________________________ }}

?


    {{ ____________________________________________________________ }}
↑   val = val * v + A[i];
    {{ ____________________________________________________________ }}
↑
  }
↓
  {{ ____________________________________________________________ }}

?


  {{ Post: val = A[0] + A[1] v + A[2] v^2 + … + A[n-1] v^(n-1) }}
  return val;
}
```