CSE 331: Backwards Reasoning and Loop Invariants

Backward reasoning: Given Q and c, find the *weakest* precondition so that `{{ P }}` c `{{ Q }}` is valid.

Backward reasoning for assignment statements:

```
{{ _____________________ }}
x = x + 1;
{{ x ≥ 4 }}
```

```
{{ _____________________ }}
y = x + 5;
{{ y < 0 }}
```

Checking validity with backwards reasoning: Given `{{ P }}` c `{{ Q }}`, forget about P for a second,
and compute the weakest precondition in `{{ ____ }}` c `{{ Q }}`, call it S.
Then check whether P implies S.

Rule of Sequence: For multiple lines of code, can reason forwards and backwards in multiple steps.

```
{{ x ≤ 4 }}
y = x + 1;
{{ _____________________ }}
z = x + y;
{{ _____________________ }}
```

```
{{ _____________________ }}
y = x + 1;
{{ _____________________ }}
z = x + y;
{{ z ≤ 10 }}
```

Can **check validity** for multiple lines of code forward or backward.
Forward: Compute strongest post after all lines, check it implies given post.
Backward: Compute weakest pre before all lines, check if it is *implied by* given pre.

```
{{ x ≤ 4 }}
y = x + 1;
{{ _____________________ }}
z = x + y;
{{ _____________________ }}
{{ z ≤ 15 }}
```

```
{{ x ≤ 0 }}
{{ _____________________ }}
y = x + 1;
{{ _____________________ }}
z = x + y;
{{ z ≤ 10 }}
```

Summary of assignment statements:
Forward: (Everything you knew before – anything invalidated) + any thing new
Backward: Plug and chug

Reasoning about Loops: What is strongest post after these loops

```
int i = 0;
while (i < n) {
    i++;
}
{{ _____________________ }}
```

```
int s = 0;
int i = 0;
while (i < n) {
    s += 2 * i + 1;
    i++;
}
{{ _____________________ }}
```

```
int s = 0;
int i = 0;
while (i < a.length) {
    s += a[i];
    i++;
}
{{ _____________________ }}
```

```
boolean b = false;
int i = 0;
while (i < a.length) {
    if (a[i] == x) {
        b = true;
    }
}
{{ _____________________ }}
```

```
int i = 0;
while (i < a.length) {
    a[i]++;
    i++;
}
{{ _____________________ }}
```

A **loop invariant** is an assertion that helps reason about a loop.
In a loop like `while (e) body` , the invariant should be true whenever the loop checks condition `e`.
In other words, it should be true (1) before the first iteration, (2) between every iteration, and (3) after the last iteration.

We can use loop invariants to check the validity of a Hoare triple involving a loop as follows:
Given `{{ P }} while (e) body {{ Q }}` and loop invariant I, we check:
I is true the first time the code reaches the loop: P implies I.
I is preserved by the body of the loop: `{{ I and e }} body {{ I }}` is valid
Q will be true after the loop exits: I and `!e` implies Q.