CSE 331

Software Design & Implementation

Topic: Reasoning about Loops

Discussion: What would be your ideal vacation spot?

Reminders

- Check that you have a Gitlab repository!

Upcoming Deadlines

• Prep. Quiz: HW2	due Monday (6/27)
• HW2	due Thursday (6/30)

Last Time…

- Motivation for CSE 331
- Assignment statements
- Conditional statements

Today's Agenda

- Upcoming Assignments
- Quick Recap: Reasoning
- Loop invariants

Upcoming Assignments

Prep. Quiz: HW2

- Due on Monday night
  - designed to be a litmus test – ask for help early in the week
  - probably should do this earlier than Monday
  - focuses on forward and backward reasoning

HW2

- Due on Thursday night
  - Part 1 is a reasoning worksheet
  - Parts 2-3 involve setting up your programming environment
  - Parts 4-8 involve some basic programming
  - Part 9 involves applying reasoning to code
- Follow setup instructions carefully!
  - If you skip a step, it will take *much* longer to find and fix
  - Demo is available on Canvas

Recap: Reasoning

Floyd Logic

- A Hoare triple is two assertions and one piece of code:

  {*P*} *S* {*Q*}

  - *P* the precondition
  - *S* the code
  - *Q* the postcondition

- A Hoare triple **{*P*} *S* {*Q*}** is called valid if:
  - in any state where P holds,
    executing S produces a state where Q holds
  - i.e., if *P* is true before *S*, then *Q* must be true after it
  - otherwise, the triple is called invalid
  - code is correct iff triple is valid

Reasoning Forward & Backward

- Forward:
  - start with the **given** precondition
  - fill in the **strongest** postcondition

$\{P\}\ S\ \{?\}$

- Backward
  - start with the **required** postcondition
  - fill in the **weakest** precondition

$\{?\}\ S\ \{Q\}$

- Finds the “best” assertion that makes the triple valid

Reasoning: Assignments

Forward:

{{ w > 0 }}

```
x = 17;
```

{{ w > 0 and x = 17 }}

```
y = 42;
```

{{ w > 0 and x = 17 and y = 42 }}

```
z = w + x + y;
```

{{ w > 0 and x = 17 and y = 42 and z = w + 59 }}

Backward:

{{ w + 17 + 42 < 0 }}

```
x = 17;
```

{{ w + x + 42 < 0 }}

```
y = 42;
```

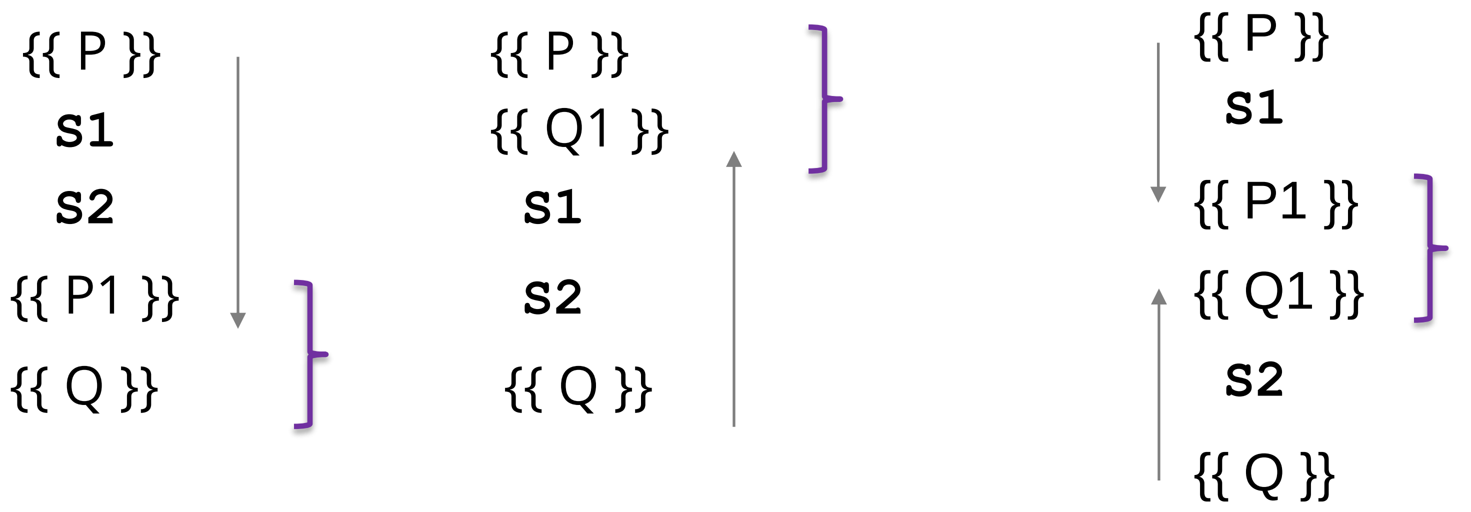
{{ w + x + y < 0 }}

```
z = w + x + y;
```

{{ z < 0 }}

Validity with Fwd & Back Reasoning

Reasoning in either direction gives valid assertions. Just need to check adjacent assertions (i.e. top assertion must imply bottom one)



Reasoning: If Statements

Forward reasoning

```
{{ P }}
if (cond)
   {{ P and cond }}
   S1
   {{ P1 }}
else
   {{ P and not cond }}
   S2
   {{ P2 }}
{{ P1 or P2 }}
```

Backward reasoning

```
{{ cond and Q1 or not cond and Q2 }}
if (cond)
   {{ Q1 }}
   S1
   {{ Q }}
else
   {{ Q2 }}
   S2
   {{ Q }}
{{ Q }}
```

Practice: Forward Reasoning

```
{{ i + j = 10 }}
if (i > j) {
  {{ ________________________ }}
  i = i – 1
  j = j + 1
  {{ ________________________ }}
} else {
  {{ ________________________ }}
  i = i + 1
  j = j - 1
  {{ ________________________ }}
}
{{ ________________________ }}
```

Practice: Backward Reasoning

```
{{ ________________ }}
if (x != 0) {
  {{ ________________________ }}
  z = x
  {{ ________________________ }}
} else {
  {{ ________________________ }}
  z = x + 1
  {{ ________________________ }}
}
{{ z > 0 }}
```

Loop Invariants

Reasoning So Far

- Mechanical reasoning about assignment and conditionals

- All code can be rewritten using only:
  - assignments
  - if statements
  - while loops

- Only part we are missing is **loops**

- (We will also cover function calls later.)

Reasoning About Loops

- Loop reasoning is not as easy as with “=“ and “if”
  - Because of Rice’s Theorem (mentioned in 311): checking any non-trivial semantic property about programs is **undecidable**

- We need help (i.e., more information) before the reasoning again becomes a mechanical process

- That help comes in the form of a “loop invariant”

Loop Invariant

A **loop invariant** is an assertion that holds whenever the loop condition is evaluated:

{{ Inv: ___________ }}

```
while (cond) {
  S
}
```

Lupin variants

Loop Invariant

A **loop invariant** is an assertion that holds whenever the loop condition is evaluated:

{{ Inv: __________ }}

```
while (cond) {
  S
}
```

Lupin variants

- It holds when we **first get to** the loop.
- It holds each time we execute `S` and **come back to** the top.

Notation: I'll use "**Inv**:" to indicate a loop invariant.

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning...

```
{{ P }}
 S1

{{ Inv: I }}
 while (cond)
   S2

 S3
{{ Q }}
```

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning…

```
{{ P }}
  S1
{{ P1 }}

{{ Inv: I }}
  while (cond)
    S2

  S3
{{ Q }}
```

Need to check that P1 implies **`I`**
(i.e., that **`I`** is true the first time)

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning...

```
{{ P }}
  S1

{{ Inv: I }}
  while (cond)
    {{ I and cond }}
     S2
    {{ P2 }}

  S3
{{ Q }}
```

Need to check that P2 implies **`I`** again (i.e., that **`I`** is true each time around)

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different)
with a loop invariant **`I`**.

Let's try forward reasoning...

{{ P }}
```
S1
```
{{ Inv: **`I`** }}
```
while (cond)
  S2
```
{{ **`I`** and not cond }}
```
S3
```
{{ P3 }}
{{ Q }}

Need to check that P3 implies Q
(i.e., Q holds after the loop)

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different)
with a loop invariant **`I`**.

```
{{ P }}
 S1

{{ Inv: I }}
 while (cond)
   S2

 S3
{{ Q }}
```

Informally, we need:

- **`I`** holds initially
- **`I`** holds each time around
- **`Q`** holds after we exit

Formally, we need validity of:

- {{ P }} `S1` {{ **`I`** }}
- {{ **`I`** and cond }} `S2` {{ **`I`** }}
- {{ **`I`** and not cond }} `S3` {{ Q }}

(can check these with backward reasoning instead)

More on Loop Invariants

- Loop invariants are crucial information
  - needs to be provided before reasoning is mechanical

- Pro Tip: always document your invariants for *non-trivial* loops
  - don't make code reviewers guess the invariant

- Pro Tip: with a good loop invariant, the code is easy to write
  - all the creativity can be saved for finding the invariant
  - more on this in later lectures…

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Equivalent to:

```
s = 0;
for (int i = 0; i != n; i++)
  s = s + b[i];
```

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

Less formal

s = sum of first i numbers in b

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

Less formal

s = sum of first i numbers in b

When i = 0, s needs to be the sum of the first 0 numbers, so we need s = 0.

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

More formal

s = sum of all b[k] with 0 ≤ k ≤ i-1

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

More formal

s = sum of all b[k] with 0 ≤ k ≤ i-1
i = 3 (0 ≤ k ≤ 2): s = b[0] + b[1] + b[2]
i = 2 (0 ≤ k ≤ 1): s = b[0] + b[1]
i = 1 (0 ≤ k ≤ 0): s = b[0]
i = 0 (0 ≤ k ≤ -1) s = 0

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

More formal

s = sum of all b[k] with $0 \le k \le i-1$

when i = 0, we want to sum over all indexes k satisfying $0 \le k \le -1$

There are no such indexes, so we need s = 0

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

Yes. (An empty sum is zero.)

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **`I`**

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = 0;
```
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
  {{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
  i = i + 1;
```
  {{ s = b[0] + ... + b[i-1] }}
```
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **I**
- {{ **I** and i != n }} S {{ **I** }} ?

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
   s = s + b[i];
   i = i + 1;
  {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[n-1] }}
```

- (s = 0 and i = 0) implies **I**
- {{ **I** and i != n }} S {{ **I** }} ?

{{ s + b[i] = b[0] + … + b[i] }}
{{ s = b[0] + … + b[i] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[i-1] and not (i != n) }}

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **`I`**
- {{ **`I`** and i != n }} `S` {{ **`I`** }} ?
- {{ **`I`** and not (i != n) }} implies
  s = b[0] + … + b[n-1] ?

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **`I`**
- {{ **`I`** and i != n }} `S` {{ **`I`** }}
- {{ **`I`** and i = n }} implies `Q`

These three checks verify that the outermost triple is valid (i.e., that the code is correct).

Termination

- Technically, this analysis does not check that the code **terminates**
  - it shows that the postcondition holds if the loop exits
  - but we never showed that the loop actually exits

- However, that follows from an analysis of the running time
  - e.g., if the code runs in $O(n^2)$ time, then it terminates
  - an infinite loop would be O(infinity)
  - any finite bound on the running time proves it terminates

- It is normal to also analyze the running time of code we write, so we get termination already from that analysis.

Example HW problem

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ ______________ }}
i = 0;
{{ ______________ }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ ______________ }}
  s = s + b[i];
  {{ ______________ }}
  i = i + 1;
  {{ ______________ }}
}
{{ ______________ }}
{{ s = b[0] + ... + b[n-1] }}
```

Example HW problem

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0  }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s + b[i] = b[0] + ... + b[i] }} or equiv {{ s = b[0] + ... + b[i-1] }}
  s = s + b[i];
  {{  s = b[0] + ... + b[i] }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i] }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Does invariant hold initially?

Are we done?
No, need to also check...

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i] }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Does loop body preserve invariant?

Are we done?
No, need to also check...

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i] }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?
No, need to also check...

Does postcondition hold on termination?

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i] }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?
No, need to also check...

HW has “?”s at these three places to indicate a triple that requires explanation

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}] Changed

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = -1;
```
{{ Inv: s = b[0] + ... + b[i] }}
```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```
{{ s = b[0] + ... + b[n-1] }}

Changed from i = 0

Changed from n

Reordered

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Work as before:

- (s = 0 and i = -1) implies **`I`**
  - **`I`** holds initially
- (**`I`** and i = n-1) implies **`Q`**
  - **`I`** implies **`Q`** at exit

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

{{ s + b[i+1] = b[0] + ... + b[i+1] }}

{{ s + b[i] = b[0] + … + b[i] }}

{{ Inv: s = b[0] + ... + b[i] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = -1;
{{ Inv: s = b[0] + ... + b[i] }}
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
{{ s = b[0] + ... + b[n-1] }}
```

- (s = 0 and i = -1) implies **I**
  - as before
- {{ **I** and i != n-1 }} S {{ **I** }}
  - reason backward
- (**I** and i = n-1) implies **Q**
  - as before

Example: sum of array (attempt 3)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Suppose we miss-order the assignments to `i` and `s`…

Where does the correctness check fail?

Example: sum of array (attempt 3)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Suppose we miss-order the assignments to `i` and `s`…

We can spot this bug because the invariant does not hold:

{{ s + b[i] = b[0] + ... + b[i+1] }}
{{ s = b[0] + ... + b[i+1] }}
{{ Inv: s = b[0] + ... + b[i] }}

First assertion is not Inv.

Example: sum of array (attempt 3)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Suppose we miss-order the assignments to `i` and `s`…

We can spot this bug because the invariant does not hold:

{{ s = b[0] + ... + b[i-1] + b[i+1] }}

For example, if i = 2, then

s = b[0] + b[1] + b[2] vs
s = b[0] + b[1] + b[3]

Before next class…

1. Try to do Prep. Quiz: HW2 before Monday!
   - Reasoning questions
   - Designed to take no more than 15 minutes

2. Read the HW2 spec early!
   - Reasoning worksheet
   - Environment setup
   - Applying reasoning to code

Extras

Extra: x^y (attempt 1)

What should be the loop invariant in the following code for exponentiation:

```
public int pow(int x, int y) {
    {{ y >= 0 }}
    int z = 0;
    int i = 0;

    {{ Inv: ____________________ }}
    while (i != y) {
        z = z * x;
        i = i + 1;
    }

    {{ z = x ^ y }}
    return z;
}
```

Extra: x^y (attempt 2)

What should be the loop invariant in the following code for exponentiation:

```
public int pow(int x, int y) {
    {{ y >= 0 }}
    int z = 0;

    {{ Inv: ____________________ }}
    while (y != 0) {
        z = z * x;
        y = y - 1;
    }

    {{ z = x ^ y }}
    return z;
}
```

Extra: partition array

Consider the following code to put the negative values at the beginning of array `b`:

{{ 0 <= n <= b.length }}

```
i = k = 0;
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}

(Also: precondition is true throughout the code. I'll skip writing that to save space...)

(Also: b contains the same numbers since we use swaps.)