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

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

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

\[
\begin{align*}
\{ & \{ w > 0 \} \} \\
& x = 17; \\
& \{ \{ w > 0 \text{ and } x = 17 \} \} \\
& y = 42; \\
& \{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \} \} \\
& z = w + x + y; \\
& \{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + 59 \} \}
\end{align*}
\]

Backward:

\[
\begin{align*}
\{ & \{ w + 17 + 42 < 0 \} \} \\
& x = 17; \\
& \{ \{ w + x + 42 < 0 \} \} \\
& y = 42; \\
& \{ \{ w + x + y < 0 \} \} \\
& z = w + x + y; \\
& \{ \{ z < 0 \} \}
\end{align*}
\]
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 \text{ and } \text{cond} \}\}
S1
\[
\{\{ \text{P1} \}\}
\]
else
\[
\{\{ P \text{ and not } \text{cond} \}\}
S2
\[
\{\{ \text{P2} \}\}
\]
\[
\{\{ \text{P1 or P2} \}\}
\]

Backward reasoning

\[
\{\{ \text{cond and Q1 or not cond and Q2} \}\}
\]
if (cond)
\[
\{\{ \text{Q1} \}\}
S1
\[
\{\{ \text{Q} \}\}
\]
else
\[
\{\{ \text{Q2} \}\}
S2
\[
\{\{ \text{Q} \}\}
\]
\[
\{\{ \text{Q} \}\}
\]
Practice: Forward Reasoning

```java
{{i + j = 10}}
if (i > j) {
    {{ __________________ }}
    i = i - 1
    j = j + 1
    {{ __________________ }}
} else {
    {{ __________________ }}
    i = i + 1
    j = j - 1
    {{ __________________ }}
}
{{ __________________ }}
```
Practice: Backward Reasoning

```java
{{ ____________ }}
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
}
```
A **loop invariant** is an assertion that holds whenever the loop condition is evaluated:

\[
\{ \text{Inv: } \_\_\_\_\_\_ \} \\
\text{while (cond) } \{ \\
    S \\
\} \\
\]

- 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 “\text{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...

\[
\{ P \} \quad S1 \\
\{ \text{Inv: } I \} \\
\text{while (cond)} \\
\quad S2 \\
\quad S3 \\
\{ Q \} 
\]
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...

$$
\begin{array}{l}
\{ P \} \\
S1 \\
\{ P1 \} \\
\{ \text{Inv: } I \} \\
\quad \text{while } (\text{cond}) \\
\quad S2 \\
S3 \\
\{ Q \} \\
\end{array}
$$

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...

$$
\begin{align*}
\{ P \} & \quad S1 \\
\{ \text{Inv: } I \} & \quad \text{while (cond)} \\
& \quad \{ I \text{ and cond} \} \\
& \quad S2 \\
& \quad \{ P2 \} \\
& \quad S3 \\
& \quad \{ Q \}
\end{align*}
$$

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...

\[
\begin{align*}
&\{ P \} \\
&S1 \\
&\{ \text{Inv: } I \} \\
&\text{while (cond)} \\
&S2 \\
&\{ I \text{ and not cond} \} \\
&S3 \\
&\{ P3 \} \\
&\{ Q \}
\end{align*}
\]

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 $I$.

\[
\begin{align*}
\{\{ P \}\} & \quad S_1 \\
\{\text{Inv: } I}\} & \quad \text{while (cond)} \\
\quad S_2 \\
\quad S_3 \\
\{\{ Q \}\} \end{align*}
\]

Informally, we need:
- $I$ holds initially
- $I$ holds each time around
- $Q$ holds after we exit

Formally, we need validity of:
- $\{\{ P \}\} S_1 \{\{ I \}\}$
- $\{\{ I \text{ and cond} \}\} S_2 \{\{ I \}\}$
- $\{\{ I \text{ and not cond} \}\} S_3 \{\{ Q \}\}$

(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 $b[0] + ... + b[n-1]$:

```c
{{
    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 $b[0] + \ldots + b[n-1]$:

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

```c
{{ }}
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 $b[0] + ... + b[n-1]$:

```java
{{ }}
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

$s = \text{sum of first i numbers in b}$
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$:

```c
{{
 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 $b[0] + \ldots + b[n-1]$:

```plaintext
{s = 0 and 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 $s = b[0] + \ldots + b[i-1]$?

More formal

$s = \text{sum of all } b[k] \text{ with } 0 \leq k \leq i-1$
Example: sum of array

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

\[
\begin{align*}
\text{\{\{ s = 0; } \\
\text{\{ i = 0; } \\
\text{\{\{ s = 0 \text{ and } i = 0 } \}
\text{\{ Inv: s = b[0] + \ldots + b[i-1] } \}
\text{ while (i != n) } \{ \\
\text{ s = s + b[i]; } \\
\text{ i = i + 1; } \\
\} \\
\text{\{ s = b[0] + \ldots + b[n-1] } \}
\end{align*}
\]

- \((s = 0 \text{ and } i = 0) \text{ implies } s = b[0] + \ldots + b[i-1] ?\)

More formal:

\[
\begin{align*}
\text{s = sum of all } b[k] \text{ with } 0 \leq k \leq i-1 \\
i = 3 (0 \leq k \leq 2): s = b[0] + b[1] + b[2] \\
i = 2 (0 \leq k \leq 1): s = b[0] + b[1] \\
i = 1 (0 \leq k \leq 0): s = b[0] \\
i = 0 (0 \leq k \leq -1) s = 0
\end{align*}
\]
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$:

```plaintext
{{ }}
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 $0 \leq k \leq i-1$

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

There are no such indexes, so we need $s = 0$
Example: sum of array

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

\[
\begin{array}{l}
\{ \} \\
s = 0; \\
i = 0; \\
\{ \text{s = 0 and i = 0 } \} \\
\{ \text{Inv: } s = b[0] + \ldots + b[i-1] \} \\
\text{while (i != n) } \\
\quad s = s + b[i]; \\
\quad i = i + 1; \\
\} \\
\{ \text{s = b[0] + \ldots + b[n-1] } \}
\end{array}
\]

- \((s = 0 \text{ and } i = 0) \text{ implies } s = b[0] + \ldots + b[i-1] ?\)
  
  Yes. (An empty sum is zero.)
Example: sum of array

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

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

- \((s = 0 \text{ and } i = 0)\) implies \( I \)
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$:

```java
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 $\mathcal{I}$
- $\{\{ \mathcal{I} \text{ and } i != n \}\} \implies \{\{ \mathcal{I} \}\}$

CSE 331 Summer 2022
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

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

- $(s = 0$ and $i = 0)$ implies $I$
- $\{ I \text{ and } i ! = n \} \implies \{ s \}$
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$

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

- $(s = 0 \text{ and } i = 0)$ implies $\mathcal{I}$
- $\{[\mathcal{I} \text{ and } i \neq n] \}$ implies $s = b[0] + \ldots + b[n-1]$?
- $\{[\mathcal{I} \text{ and not } (i \neq n)] \}$ implies $s = b[0] + \ldots + b[n-1]$?
```
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$:

```java
{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 $i != n \} \implies \{I\}$
- $\{I$ 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 $O(\infty)$
  – 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.
The following code to compute $b[0] + \ldots + b[n-1]$:

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

s = b[0] + \ldots + b[n-1]`
```
Example HW problem

The following code to compute \( b[0] + ... + b[n-1] \):

```plaintext
{{
  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;
{i s = 0}
i = 0;
{i s = 0 and i = 0}
{i 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 invariant hold initially?
Warning: not just filling in blanks

The following code to compute $b[0] + \ldots + b[n-1]$:

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

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

Does loop body preserve invariant?
Warning: not just filling in blanks

The following code to compute $b[0] + ... + b[n-1]$:

```c
{ {} }  
  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 \( b[0] + \ldots + b[n-1] \):

\[
\begin{align*}
\text{\{ s = 0; } \\
\text{\{ s = 0 \}} \\
\text{i = 0; } \\
\text{\{ s = 0 and i = 0 \}} \\
\text{\{ Inv: s = b[0] + \ldots + b[i-1] \}} \\
\text{while (i != n) \{ } \\
\text{\{ s = b[0] + \ldots + b[i-1] \}} \\
\text{s = s + b[i]; } \\
\text{\{ s = b[0] + \ldots + b[i] \}} \\
\text{i = i + 1; } \\
\text{\{ s = b[0] + \ldots + b[i-1] \}} \\
\text{\} } \\
\text{\{ s = b[0] + \ldots + b[i-1] and not (i != n) \}} \\
\text{\{ s = b[0] + \ldots + b[n-1] \}}
\end{align*}
\]

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

HW has “?”s at these three places to indicate a triple that requires explanation.
Consider the following code to compute $b[0] + ... + b[n-1]$:

```c
{{ }}
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 \( b[0] + \ldots + b[n-1] \):

\[
\begin{align*}
\{&\} \\
\text{s} &= 0; \\
\text{i} &= -1; \\
\{&\text{ Inv: } s = b[0] + \ldots + b[i] \} \\
\text{while (i != n-1) \{} \\
&\quad \text{i} = i + 1; \\
&\quad s = s + b[i]; \\
\text{\}} \\
\{&\text{ s = b[0] + \ldots + b[n-1] }\}
\end{align*}
\]

\[\text{Changed from } i = 0\]
\[\text{Changed from } n\]
\[\text{Reordered}\]
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:

- $(s = 0 \text{ and } i = -1)$ implies $I$
  - $I$ holds initially

- $(I \text{ and } i = n-1)$ implies $Q$
  - $I$ implies $Q$ at exit
Example: sum of array (attempt 2)

Consider the following code to compute $b[0] + ... + b[n-1]$:

```c
{{ }}
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] }}
```
Example: sum of array (attempt 2)

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

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

- $(s = 0$ and $i = -1)$ implies $\mathsf{I}$
  - as before
- $(\mathsf{I}$ and $i != n-1$) $\implies$ $(\mathsf{I})$
  - reason backward
- $(\mathsf{I}$ and $i = n-1)$ implies $\mathsf{Q}$
  - as before
Example: sum of array (attempt 3)

Consider the following code to compute \( \sum_{i=0}^{n-1} b[i] \):

```c
{{ }}
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 $b[0] + \ldots + b[n-1]$:

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

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

First assertion is not Inv.
Consider the following code to compute $b[0] + ... + b[n-1]$:

```c
{{ }}
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:

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

For example, if $i = 2$, then

- $s = b[0] + b[1] + b[3]$
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
What should be the loop invariant in the following code for exponentiation:

```java
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;
}
```
What should be the loop invariant in the following code for exponentiation:

```java
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 $b$:

```java
{{ 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.)