CSE 331
Software Design & Implementation
Topic: Reasoning about Loops

💬 Discussion: What would be your ideal vacation spot?
Reminders

• Slides available on course calendar
• Check that you have a Gitlab repository

Upcoming Deadlines

• Prep. Quiz: HW2 due Monday (6/26)
• HW2 due Thursday (6/29)
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 linked in Ed discussion post
    - But we use Java 17, not Java 11
Recap: Reasoning
Floyd Logic

- A Hoare triple is two assertions and one piece of code:
  - $P$ 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
Reasoning: Assignments

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 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 }}
```
{\{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 \textit{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”
A **loop invariant** is an assertion that holds whenever the loop condition is evaluated:

\[
\{\text{Inv: } \quad \}\ \\
\text{while } (\text{cond}) \ {\} \\
\text{  } S \\
\text{  } \}
\]
Unrolling a Loop

Code:

```java
i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
j = j * 2;
}
```

Unrolled Statements:
Unrolling a Loop

Code:

Unrolled Statements:

i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
    j = j * 2;
}
Unrolling a Loop

Code:

\[
\begin{align*}
i &= 0; \\
j &= 1; \\
\text{while } (i < 2) \{} \\
\quad &i = i + 1; \\
\quad &j = j \times 2; \\
\text{\}}
\end{align*}
\]

Unrolled Statements:

\[
\begin{align*}
i &= 0 \\
j &= 1
\end{align*}
\]
Unrolling a Loop

Code:

```
i = 0;
j = 1;
while (i < 2) {
i = i + 1;
j = j * 2;
}
```

Unrolled Statements:

```
i = 0
j = 1
check i < 2
```
Unrolling a Loop

Code:

```c
i = 0;
j = 1;
while (i < 2) {
i = i + 1;
j = j * 2;
}
```

Unrolled Statements:

```c
i = 0
j = 1
check i < 2
i = i + 1 = 1
```
Unrolling a Loop

Code:

```c
i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
    j = j * 2;
}
```

Unrolled Statements:

```
i = 0
j = 1
check i < 2
  i = i + 1 = 1
  j = j * 2 = 2
```
Unrolling a Loop

Code:

```java
i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
j = j * 2;
}
```

Unrolled Statements:

```java
i = 0
j = 1
check i < 2
i = i + 1 = 1
j = j * 2 = 2
check i < 2
```
Unrolling a Loop

Code:

```
i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
j = j * 2;
}
```

Unrolled Statements:

```
i = 0
j = 1
check i < 2
i = i + 1 = 1
j = j * 2 = 2
check i < 2
i = i + 1 = 2
```
Unrolling a Loop

Code:

\[
\begin{align*}
  i &= 0; \\
  j &= 1; \\
  \text{while } (i < 2) \{ \\
      &\quad i = i + 1; \\
      &\quad j = j \times 2; \\
  \} 
\end{align*}
\]

Unrolled Statements:

\[
\begin{align*}
  i &= 0 \\
  j &= 1 \\
  \text{check } i < 2 \\
  i &= i + 1 = 1 \\
  j &= j \times 2 = 2 \\
  \text{check } i < 2 \\
  i &= i + 1 = 2 \\
  j &= j \times 2 = 4
\end{align*}
\]
Unrolling a Loop

Code:

```c
i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
    j = j * 2;
}
```

Unrolled Statements:

```c
i = 0
j = 1
check i < 2
i = i + 1 = 1
j = j * 2 = 2
check i < 2
i = i + 1 = 2
j = j * 2 = 4
check i < 2
```
Unrolling a Loop

Code:

```plaintext
i = 0;
j = 1;
while (i < 2) {
    i = i + 1;
    j = j * 2;
}
```

Unrolled Statements:

```plaintext
i = 0
j = 1
check i < 2
i = i + 1 = 1
j = j * 2 = 2
check i < 2
i = i + 1 = 2
j = j * 2 = 4
check i < 2
```
Loop Invariant

Suppose we know that a loop has a loop invariant $\text{Inv}$. Where does $\text{Inv}$ hold?

```latex
{{ \text{Inv} }}
\text{while } (\text{cond}) \{ \\
\quad \{ \text{...} \} \\
\quad \text{S1} \\
\quad \{ \text{...} \} \\
\quad \text{S2} \\
\quad \{ \text{Inv and ...} \} \\
\}
\{ \text{...} \}
```

Lupin variants
Suppose we know that a loop has a loop invariant \texttt{Inv}. Where does \texttt{Inv} hold?

\[
\begin{align*}
\{\texttt{Inv}\} \\
\textbf{while} \ (\texttt{cond}) \ \{ \\
\quad \{\texttt{Inv and ... }\} \\
\quad \texttt{S1} \\
\quad \{\ ... \} \\
\quad \texttt{S2} \\
\quad \{\texttt{Inv and ... }\} \\
\} \\
\{\ ... \} 
\end{align*}
\]
Loop Invariant

Suppose we know that a loop has a loop invariant $\text{Inv}$. Where does $\text{Inv}$ hold?

\[
\begin{aligned}
\{\{ \text{Inv} \}\} \\
\text{while} \ (\text{cond}) \ \{ \\
\quad \{\{ \text{Inv and ... } \}\} \\
\quad \text{S1} \\
\quad \{\{ ... \}\} \\
\quad \text{S2} \\
\quad \{\{ \text{Inv and ... } \}\} \\
\} \\
\{\{ \text{Inv and not cond} \}\}
\end{aligned}
\]
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...

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

$$\{\text{ P }\} \xrightarrow{S1} \{\text{ P1 }\} \xrightarrow{\{\text{ Inv: } I\} \quad \text{ while (cond)} \quad S2} \{\text{ Q }\}$$

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...

$$\{\{ P \} \} \quad S1$$
$$\{\{ \text{Inv: } I \} \}$$
$$\quad \text{while (cond)}$$
$$\quad \quad S2$$
$$\{\{ I \text{ and not cond} \} \}$$
$$\quad S3$$
$$\{\{ P3 \} \}$$
$$\{\{ Q \} \}$$

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 \( I \).

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

Informally, we need:
- \( I \) holds initially
- \( I \) holds each time around
- \( Q \) holds after we exit

Formally, we need validity of:
- \( \{\{ P \}\}\ S1 \{\{ I \}\} \)
- \( \{\{ I \text{ and } \text{cond} \}\}\ S2 \{\{ I \}\} \)
- \( \{\{ I \text{ and not } \text{cond} \}\}\ S3 \{\{ Q \}\} \)

(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 $b[0] + \ldots + b[n-1]$:  

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

Equivalent to:

```c
s = 0;
for (int i = 0; i != n; i++)
    s = s + b[i];
```
Example: sum of array

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

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

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

```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] }}
}}
```
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] }}
```

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

Less formal:

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

Consider 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;
}
```

• $(s = 0$ and $i = 0)$ implies $s = b[0] + \ldots + b[i-1]$?

Less formal

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

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;
  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] }}
```

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

More formal:

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

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

- $i = 3 \ (0 \leq k \leq 2): s = b[0] + b[1] + b[2]$
- $i = 2 \ (0 \leq k \leq 1): s = b[0] + b[1]$
- $i = 1 \ (0 \leq k \leq 0): s = b[0]$
- $i = 0 \ (0 \leq k \leq -1) s = 0$
Example: sum of array

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

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

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

More formal

\[
\begin{align*}
\text{s = sum of all } b[k] \text{ with } 0 \leq k \leq i-1 \\
\text{when } i = 0, \text{ we want to sum over all indexes } k \text{ satisfying } 0 \leq k \leq -1 \\
\text{There are no such indexes, so we need } s = 0
\end{align*}
\]
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] }}
```

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

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

```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] }}
```

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

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

```c
{{
    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 $I$
- ${{ I and i != n }} \land {{ I }}$ ?
Example: sum of array

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

```plaintext

s = 0;
i = 0;

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 $i != n}\} \rightarrow I$

$\{s = b[0] + ... + b[i]\}$
Example: sum of array

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

```plaintext
{{ }}
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 $\mathcal{I}$
- $\{\{ \mathcal{I} \text{ and } i != n \}\} \Rightarrow \{\{ \mathcal{I} \}\}$
- $\{\{ \mathcal{I} \text{ and not } (i != n) \}\}$ implies $s = b[0] + ... + b[n-1]$?
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[n-1] }}
```

- $(s = 0$ and $i = 0)$ implies $I$
- $\{I \text{ and } i \neq n\} \implies \{I\}$
- $\{I \text{ and } i = n\}$ 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 $O(\text{infinity})$
  – 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 $b[0] + \ldots + b[n-1]$:

```c
{{ }}
 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 HW problem

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

\[
\begin{align*}
\{ & \} \\
& s = 0; \\
& \{ s = 0 \} \\
& i = 0; \\
& \{ s = 0 \text{ and } i = 0 \} \\
& \{ \text{Inv: } s = b[0] + \ldots + b[i-1] \} \\
& \text{while (i != n) } \\
& \{ s + b[i] = b[0] + \ldots + b[i] \} \text{ or equiv } \{ s = b[0] + \ldots + b[i-1] \} \\
& \quad s = s + b[i]; \\
& \{ s = b[0] + \ldots + b[i] \} \\
& \quad i = i + 1; \\
& \{ s = b[0] + \ldots + b[i-1] \} \\
& \} \\
& \{ s = b[0] + \ldots + b[i-1] \text{ and not (i != n) } \} \\
& \{ s = b[0] + \ldots + b[n-1] \}
\end{align*}
\]
Warning: not just filling in blanks

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

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

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

```java
s = 0;

i = 0;

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

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 \( b[0] + \ldots + b[n-1] \):

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

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

Does postcondition hold on termination?
Warning: not just filling in blanks

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

```
{{ }}
s = 0;
{{ s = 0 }}
i = 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...

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 \( b[0] + \ldots + b[n-1] \):

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

Changed from $i = 0$

Changed from $n$

Reordered
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] }}
```

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

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

- $(I$ and $i != n-1)$ implies $I$
  - reason backward

- $(I$ and $i = n-1)$ implies $Q$
  - as before
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 \( b[0] + \ldots + b[n-1] \):

\[
\begin{aligned}
    \text{\{\{ \}\}} \quad &
    s = 0; \\
    i = -1; \\
    \text{\{\{ Inv: } s = b[0] + \ldots + b[i] \text{\}}}
    \end{aligned}
\]

while \((i \neq n-1)\) {
    \[
    \begin{aligned}
    s &= s + b[i]; \\
    i &= i + 1;
    \end{aligned}
    \]
\}

\[
\begin{aligned}
\text{\{\{ s = b[0] + \ldots + b[n-1] \}}
\end{aligned}
\]

Suppose we miss-order the assignments to \( i \) and \( s \)...

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

\[
\begin{aligned}
\text{\{\{ s + b[i] = b[0] + \ldots + b[i+1] \}} \\
\text{\{\{ s = b[0] + \ldots + b[i+1] \}} \\
\text{\{\{ Inv: } s = b[0] + \ldots + b[i] \text{\}}}
\end{aligned}
\]

First assertion is not Inv.
Example: sum of array (attempt 3)

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

```plaintext
{{ }}
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:

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

For example, if $i = 2$, then

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

\[
\begin{align*}
\text{\{0} & \leq n \leq b.\text{length}\} \\
i & = k = 0; \\
\text{while (i} & \neq n) \{ \\
\quad \text{if (b[i]} & < 0) \{ \\
\quad\quad \text{swap b[i], b[k];} \\
\quad\quad k & = k + 1; \\
\quad\} \\
\quad i & = i + 1; \\
\} \\
\text{\{b[0], ..., b[k-1]} & < 0 \leq b[k], ..., b[n-1]\}}
\end{align*}
\]

(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.)