Reminders

• HW1 is due tonight

• HW2 will be posted on Saturday (due Wednesday)
  – another worksheet
  – covers loops, so a little harder

• Quiz 1 is due Monday
Review: Straight-line Code
Forward & Backward Reasoning

Forward reasoning

\[
\{\{ P \}\}\rightarrow S \rightarrow \{\{ ? \}\}
\]

- P is what we know initially
- Work downward
- Determine what holds after S executes

Backward reasoning

\[
\{\{ ? \}\}\leftarrow S \leftarrow \{\{ Q \}\}
\]

- Q is what we want at the end
- Work upward
- Determine what must hold initially before S executes
Assignment Rule

Forward reasoning

\[
\begin{align*}
\{ \{ P \} \} \\
\text{x = expr;} \\
\{ \{ ? \} \}
\end{align*}
\]
Assignment Rule

Forward reasoning

\[
\{ P \}\}
\]
\[
x = expr;
\]
\[
\{ P \text{ and } x = expr \}\}
\]

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes \( x \) not used in \( P \))
Assignment Rule

Forward reasoning

\[
\begin{align*}
\{ \{ P \} \} \\
x &= \text{expr} \\
\{ \{ P \land x = \text{expr} \} \}
\end{align*}
\]

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes \( x \) not used in \( P \))

Backward reasoning

\[
\begin{align*}
\{ \{ ? \} \} \\
x &= \text{expr} \\
\{ \{ Q \} \}
\end{align*}
\]
Assignment Rule

Forward reasoning

\[
\begin{align*}
\{\ P \} \\
&\ x = \text{expr} \\
\{\ P \ \text{and} \ x = \text{expr} \}
\end{align*}
\]

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes \( x \) not used in \( P \))

Backward reasoning

\[
\begin{align*}
\{\ Q[x=\text{expr}] \} \\
&\ x = \text{expr} \\
\{\ Q \}
\end{align*}
\]

- just substitution
- most general conditions for getting \( Q \) after \( x = \text{expr} \);
Assignment Example

Forward reasoning

\[
\begin{align*}
\{\ w = 3 \} \\
x &= y - 5; \\
\{\ ? \} \\
\end{align*}
\]
Assignment Example

Forward reasoning

\[
\begin{align*}
\{ w = 3 \} \\
x &= y - 5; \\
\{ w = 3 \text{ and } x = y - 5 \}
\end{align*}
\]
Assignment Example

Forward reasoning

```
{{ w = 3 }}
x = y - 5;
{{ w = 3 and x = y - 5 }}
```

Backward reasoning

```
{{ ? }}
x = y - 5;
{{ w = x + 5 }}
```
Assignment Example

Forward reasoning

\{ w = 3 \}
\text{x = y - 5;}
\text{\{ w = 3 and x = y - 5 \}}

Backward reasoning

\text{\{ w = y \}}
\text{x = y - 5;}
\text{\{ w = x + 5 \}}
Sequence Rule

Forward reasoning

\[
\{ \{ P \} \}
\]

S1
S2
\[
\{ \{ ? \} \}
\]
Sequence Rule

Forward reasoning

\[
\{ \{ P \} \} \\
S1 \\
\{ \{ ? \} \} \\
S2 \\
\{ \{ ? \} \}
\]
Sequence Rule

Forward reasoning

\[
\{ \text{P} \}\]
S1
\[
\{ \text{P1} \}\]
S2
\[
\{ ? \}\]
Sequence Rule

Forward reasoning

\[
\begin{align*}
\{ \{ P \} \} \\
S1 \\
\{ \{ P1 \} \} \\
S2 \\
\{ \{ P2 \} \}
\end{align*}
\]
Sequence Rule

Forward reasoning

\{\{ P \}\}\}
S1
\{\{ P1 \}\}\}
S2
\{\{ P2 \}\}\}

Backward reasoning

\{\{ ? \}\}\}
S1
\{\{ Q \}\}\}
S2
Sequence Rule

Forward reasoning

\[
\{ \{ P \} \} \\
S_1 \\
\{ \{ P1 \} \} \\
S_2 \\
\{ \{ P2 \} \}
\]

Backward reasoning

\[
\{ \{ ? \} \} \\
S_1 \\
\{ \{ ? \} \} \\
S_2 \\
\{ \{ Q \} \}
\]
**Sequence Rule**

**Forward reasoning**

\[
\begin{align*}
&\{ \{ P \} \} \\
&S1 \\
&\{ \{ P1 \} \} \\
&S2 \\
&\{ \{ P2 \} \}
\end{align*}
\]

**Backward reasoning**

\[
\begin{align*}
&\{ \{ ? \} \} \\
&S1 \\
&\{ \{ Q2 \} \} \\
&S2 \\
&\{ \{ Q \} \}
\end{align*}
\]
Sequence Rule

Forward reasoning

\{\{ P \} \}
S1
\{\{ P1 \} \}
S2
\{\{ P2 \} \}

Backward reasoning

\{\{ Q1 \} \}
S1
\{\{ Q2 \} \}
S2
\{\{ Q \} \}
If-Statement Rule

Forward reasoning

{{ P }}
if (cond)
    S1
else
    S2
{{ ? }}
If-Statement Rule

Forward reasoning

```plaintext
{P}
if (cond)
  {P and cond}
  S1
else
  {P and not cond}
  S2
{?}
```
If-Statement Rule

Forward reasoning

```plaintext
{{ P }}
if (cond)
  {{ P and cond }}
  S1
  {{ P1 }}
else
  {{ P and not cond }}
  S2
  {{ P2 }}
{{ ? }}
```
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
\text{if } (\text{cond}) \\
\{\{ P \text{ and } \text{cond} \}\} \\
\text{S1} \\
\{\{ P1 \}\} \\
\text{else} \\
\{\{ P \text{ and not } \text{cond} \}\} \\
\text{S2} \\
\{\{ P2 \}\} \\
\{\{ P1 \text{ or } P2 \}\}
\end{align*}
\]
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{ \{ P \} \} \\
\text{if (cond)} \\
\{ \{ P \text{ and } \text{cond} \} \} \\
S1 \\
\{ \{ \text{P1} \} \} \\
\text{else} \\
\{ \{ P \text{ and not } \text{cond} \} \} \\
S2 \\
\{ \{ \text{P2} \} \} \\
\{ \{ \text{P1 or P2} \} \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ \{ ? \} \} \\
\text{if (cond)} \\
S1 \\
\text{else} \\
S2 \\
\{ \{ Q \} \}
\end{align*}
\]
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{ \{ P \} \} \\
\text{if (cond)} \\
\{ \{ P \text{ and } \text{cond} \} \} \\
S1 \\
\{ \{ P1 \} \} \\
\text{else} \\
\{ \{ P \text{ and not } \text{cond} \} \} \\
S2 \\
\{ \{ P2 \} \} \\
\{ \{ P1 \text{ or } P2 \} \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ \{ ? \} \} \\
\text{if (cond)} \\
S1 \\
\{ \{ Q \} \} \\
\text{else} \\
S2 \\
\{ \{ Q \} \} \\
\{ \{ Q \} \}
\end{align*}
\]
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
\text{if (cond)} \\
\{\{ P \text{ and} \ cond \}\} \\
S1 \\
\{\{ P1 \}\} \\
\text{else} \\
\{\{ P \text{ and not} \ cond \}\} \\
S2 \\
\{\{ P2 \}\} \\
\{\{ P1 \text{ or} \ P2 \}\}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{\{ ? \}\} \\
\text{if (cond)} \\
\{\{ Q1 \}\} \\
S1 \\
\{\{ Q \}\} \\
\text{else} \\
\{\{ Q2 \}\} \\
S2 \\
\{\{ Q \}\} \\
\{\{ Q \}\}
\end{align*}
\]
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{\{P\}\} \\
\text{if (cond)} \\
\{\{P \text{ and } \text{cond}\}\} \\
S1 \\
\{\{P1\}\} \\
\text{else} \\
\{\{P \text{ and } \neg \text{cond}\}\} \\
S2 \\
\{\{P2\}\} \\
\{\{P1 \text{ or } P2\}\}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{\{\text{cond and Q1 or } \neg \text{cond and Q2}\}\} \\
\text{if (cond)} \\
\{\{Q1\}\} \\
S1 \\
\{\{Q\}\} \\
\text{else} \\
\{\{Q2\}\} \\
S2 \\
\{\{Q\}\}
\end{align*}
\]
If-Statement Example

Forward reasoning

```{ }
if (x >= 0)
    y = x;
else
    y = -x;
{?}
If-Statement Example

Forward reasoning

```plaintext
if (x >= 0)
  {x >= 0}
  y = x;
else
  {x < 0}
  y = -x;
{?}
```
If-Statement Example

Forward reasoning

```plaintext
{{ }}
if (x >= 0)
  {{ x >= 0 }}
y = x;
  {{ x >= 0 and y = x }}
else
  {{ x < 0 }}
y = -x;
  {{ x < 0 and y = -x }}
{{ ? }}
```
If-Statement Example

Forward reasoning

```plaintext
{{}}
if (x >= 0)
  {{ x >= 0 }}
y = x;
  {{ x >= 0 and y = x }}
else
  {{ x < 0 }}
y = -x;
  {{ x < 0 and y = -x }}
{{ ? }}
```

**Warning**: many write `{{ y >= 0 }}` here. That is true but it is strictly weaker. (It includes cases where y != x)
If-Statement Example

Forward reasoning

```plaintext
{}{}
if (x >= 0)
  {{ x >= 0 }}
y = x;
  {{ x >= 0 and y = x }}
else
  {{ x < 0 }}
y = -x;
  {{ x < 0 and y = -x }}
{{ (x >= 0 and y = x) or (x < 0 and y = -x) }}
```
If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
  {{ x >= 0 }}
  y = x;
  {{ x >= 0 and y = x }}
else
  {{ x < 0 }}
  y = -x;
  {{ x < 0 and y = -x }}
{{ y = |x| }}
```
If-Statement Example

Forward reasoning

\[
\begin{align*}
&\{\} \\
&\text{if } (x \geq 0) \\
&\quad \{\ x \geq 0 \} \\
&\quad y = x; \\
&\quad \{\ x \geq 0 \text{ and } y = x \} \\
&\text{else} \\
&\quad \{\ x < 0 \} \\
&\quad y = -x; \\
&\quad \{\ x < 0 \text{ and } y = -x \} \\
&\{\ y = |x| \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
&\{\ ? \} \\
&\text{if } (x \geq 0) \\
&\quad y = x; \\
&\text{else} \\
&\quad y = -x; \\
&\{\ y = |x| \}
\end{align*}
\]
If-Statement Example

Forward reasoning

{ }
if (x >= 0)
{{ x >= 0 }}
y = x;
{{ x >= 0 and y = x }}
else
{{ x < 0 }}
y = -x;
{{ x < 0 and y = -x }}
{{ y = |x| }}

Backward reasoning

{ ? }
if (x >= 0)
y = x;
{{ y = |x| }}
else
{{ y = |x| }}
y = -x;
{{ y = |x| }}
{{ y = |x| }}
If-Statement Example

Forward reasoning

```plaintext
{{ }}
if (x >= 0)
    {{ x >= 0 }}
y = x;
    {{ x >= 0 and y = x }}
else
    {{ x < 0 }}
y = -x;
    {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```plaintext
{{ ? }}
if (x >= 0)
    {{ x = |x| }}
y = x;
    {{ y = |x| }}
else
    {{ -x = |x| }}
y = -x;
    {{ y = |x| }}
{{ y = |x| }}
```
If-Statement Example

Forward reasoning

\[
\begin{align*}
\{ \{ \} \} \\
\text{if } (x \geq 0) \\
\quad \{ \{ x \geq 0 \} \} \\
\quad y = x; \\
\quad \{ \{ x \geq 0 \text{ and } y = x \} \} \\
\text{else} \\
\quad \{ \{ x < 0 \} \} \\
\quad y = -x; \\
\quad \{ \{ x < 0 \text{ and } y = -x \} \} \\
\{ \{ y = |x| \} \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ \{ ? \} \} \\
\text{if } (x \geq 0) \\
\quad \{ \{ x \geq 0 \} \} \\
\quad y = x; \\
\quad \{ \{ y = |x| \} \} \\
\text{else} \\
\quad \{ \{ x \leq 0 \} \} \\
\quad y = -x; \\
\quad \{ \{ y = |x| \} \} \\
\{ \{ y = |x| \} \}
\end{align*}
\]
If-Statement Example

Forward reasoning

\[
\begin{align*}
\{ \} \\
\text{if } (x \geq 0) \\
\{ x \geq 0 \} \\
y = x; \\
\{ x \geq 0 \text{ and } y = x \} \\
\text{else} \\
\{ x < 0 \} \\
y = -x; \\
\{ x < 0 \text{ and } y = -x \} \\
\{ y = |x| \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ \} \\
\text{if } (x \geq 0) \\
\{ x \geq 0 \} \\
y = x; \\
\{ y = |x| \} \\
\text{else} \\
\{ x \leq 0 \} \\
y = -x; \\
\{ y = |x| \} \\
\{ y = |x| \}
\end{align*}
\]
If-Statement Example

Forward reasoning

\{
\}
if \ (x \geq 0) 
\{
\ x \geq 0 \}
y = x;
\{ \ x \geq 0 \ \text{and} \ y = x \}
else
\{ \ x < 0 \}
y = -x;
\{ \ x < 0 \ \text{and} \ y = -x \}
\{ \ y = |x| \}

Backward reasoning

\{ \ x \geq 0 \ \text{or} \ x < 0 \}
if \ (x \geq 0) 
\{ \ x \geq 0 \}
y = x;
\{ \ y = |x| \}
else
\{ \ x \leq 0 \}
y = -x;
\{ \ y = |x| \}
\{ \ y = |x| \}
If-Statement Example

Forward reasoning

```plaintext
{{ }}
if (x >= 0)
  {{ x >= 0 }}
y = x;
  {{ x >= 0 and y = x }}
else
  {{ x < 0 }}
y = -x;
  {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```plaintext
{{ }}
if (x >= 0)
  {{ x >= 0 }}
y = x;
  {{ y = |x| }}
else
  {{ x <= 0 }}
y = -x;
  {{ y = |x| }}
{{ y = |x| }}
```
Verifying Correctness (Inspection)

Two different ways of checking \( \{\{ P \}\} S \{\{ Q \}\} \)

Use forward reasoning:

\[
\begin{align*}
\{\{ P \}\} & \quad S \\
\{\{ Q' \}\} & \\
\end{align*}
\]

- Find \( Q' \) assuming \( P \).
- Check that \( Q' \) implies \( Q \).
  - weaken postcondition

Use backward reasoning:

\[
\begin{align*}
\{\{ P' \}\} & \\
S & \\
\{\{ Q \}\} & \\
\end{align*}
\]

- Find \( P' \) that produces \( Q \).
- Check that \( P \) implies \( P' \).
  - strengthen precondition

You know how to verify correctness of straight-line code.
You will do this on HW1.
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```c
{{ }}
if (x >= 0)
  y = div(x,2);
else
  y = -div(-x+1,2);
{{ 2y = x or 2y = x - 1 }}
```

Assume that \( \text{div}(a,b) \) computes \( a/b \) rounded towards zero.
Code to compute \( x/2 \) rounded toward minus infinity (usual division).
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```plaintext
{{ }}
if (x >= 0)
{{ x >= 0 }}
y = div(x, 2);
else
{{ x < 0 }}
y = -div(-x+1, 2);
{{ 2y = x or 2y = x - 1 }}
```
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```c
if (x >= 0)
    {{ x >= 0 }}
y = div(x, 2);
    {{ 2y = x or 2y = x - 1 }}
else
    {{ x < 0 }}
y = -div(-x+1, 2);
    {{ 2y = x or 2y = x - 1 }}
{{ 2y = x or 2y = x - 1 }}
```
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```c
{{ }}</
if (x >= 0)
{{ x >= 0 }}
y = div(x, 2);
{{ 2y = x or 2y = x - 1 }}
else
{{ x < 0 }}
y = -div(-x+1, 2);
{{ 2y = x or 2y = x - 1 }}
{{ 2y = x or 2y = x - 1 }}
```
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```plaintext
if (x >= 0)
    y = div(x, 2);
    2y = x or 2y = x - 1
else
    y = -div(-x+1, 2);
    2y = x or 2y = x - 1
```
Loops
Loop Invariant

A **loop invariant** is one that always holds at the top of the loop:

{\{ Inv: I \}\}

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 “Inv:” to indicate a loop invariant.
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \}\} \text{ while } (\text{cond}) \ S \ \{\{ Q \}\}
\]

This triple is valid iff: there is a loop invariant I such that

\[
\{\{ P \}\}
\{\{ \text{Inv}: I \}\}
\text{while } (\text{cond})
\quad S
\quad \{\{ Q \}\}
\]

- I holds initially
- I holds each time we execute S
- Q holds when I holds and cond is false
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{ \{ P \} \} \text{ while (} \text{cond}) \ S \ \{ \{ Q \} \}
\]

This triple is valid iff: there is a loop invariant I such that

\[
\begin{align*}
\{ \{ P \} \} \\
\{ \text{Inv: } I \} \\
\text{while (} \text{cond}) \\
S \\
\{ \{ Q \} \}
\end{align*}
\]

• P implies I
• I holds each time we execute S
• Q holds when I holds and cond is false
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \} \} \text{ while (cond) } S \{\{ Q \} \}
\]

This triple is valid iff: there is a loop invariant I such that

- \{\{ P \} \}
- \{\{ \text{Inv: } I \} \}
- while (cond)
  \[ S \]
- \{\{ Q \} \}
  - P implies I
  - \{\{ I \text{ and cond} \} \} S \{\{ I \} \} \text{ is valid}
  - Q holds when I holds and cond is false
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
{\{ P \}} \text{ while } (\text{cond}) \ S \ {\{ Q \}}
\]

This triple is valid iff: there is a loop invariant \( I \) such that

\[
{\{ P \}} \quad \{\text{Inv: } I\} \\
\text{while } (\text{cond}) \\
\quad S \\
{\{ Q \}}
\]

\[
\begin{align*}
\text{• } & P \text{ implies } I \\
\text{• } & \{ I \text{ and } \text{cond} \} \ S \ {\{ I \}} \text{ is valid} \\
\text{• } & (I \text{ and not } \text{cond}) \text{ implies } Q
\end{align*}
\]
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \}\} \text{ while } (\text{cond}) \ S \ \{\{ Q \}\}
\]

This triple is valid iff: there is a loop invariant \( I \) such that

\[
\begin{align*}
\{\{ P \}\} \\
\{\{ \text{Inv: } I \}\} \\
\text{while } (\text{cond}) \\
S \\
\{\{ Q \}\}
\end{align*}
\]

- \( P \) implies \( I \)
- \( \{\{ I \ \text{and} \ \text{cond} \}\} \ S \ \{\{ I \}\} \) is valid
- \( (I \ \text{and not} \ \text{cond}) \) implies \( Q \)
More on Loop Invariants

- We need a loop invariant to check validity of a while loop.
- There is no automatic way to generate these.
  - (A theory course will explain why…)

- For this lecture, all loop invariants will be given.
- Next lecture will discuss how to choose a loop invariant.

- Pro Tip: always document your invariants for non-trivial loops
  - as we just saw, much easier for others to check your code
  - possible exception for loops that are “obvious”
- Pro Tip: with a good loop invariant, the code is easy to write
Example: sum of array

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

```
{{ b.length >= n }}
s = 0;
i = 0;
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 $b[0] + \ldots + b[n-1]$:

{{ b.length >= n }}
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 \( b[0] + \ldots + b[n-1] \):

```java
{{ b.length >= n }}
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 $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
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 $b[0] + \ldots + b[n-1]$: 

```java
{{ b.length >= n }}
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 
  $s = b[0] + \ldots + b[i-1]$ ?

  Yes. (An empty sum is zero.)
Example: sum of array

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

```javascript
{{ b.length >= n }}
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 $b[0] + \ldots + b[n-1]$:

$$\{ \text{b.length }\geq n \}$$

s = 0;
i = 0;

$$\{ \text{Inv: s }= b[0] + \ldots + b[i-1] \}$$

while (i != n) {

$$\{ \text{s }= b[0] + \ldots + b[i-1] \text{ and i }\neq n \}$$

s = s + b[i];
i = i + 1;

$$\{ s = b[0] + \ldots + b[i-1] \}$$

}

$$\{ s = b[0] + \ldots + b[n-1] \}$$

- $s = 0 \text{ and } i = 0$ implies I
- $\{ \text{I and i }\neq n \} \text{ S } \{ \text{I} \}$?
Example: sum of array

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

```java
{{ b.length >= n }}
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 }} ?

Yes (e.g., by backward reasoning)

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

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

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

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

Yes. (I is the postcondition when we have \(i == n\).)
Example: sum of array

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

```java
{{ b.length >= n }}
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] }}
```

- $(s = 0 \text{ and } i = 0)$ implies I
- $\{\{ I \text{ and } i != n \}\}$ S $\{\{ I \}\}$
- $\{\{ I \text{ and } i == n \}\}$ implies Q

These three checks verify that the postcondition holds (i.e., 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: sum of array (attempt 2)

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

```java
{{ b.length >= n }}
s = 0;
i = -1;
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 $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
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] }}
```
Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}

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 }} $\Rightarrow$ {{ I }}
  - reason backward:

  ```java
  {{ s + b[i+1] = b[0] + \ldots + b[i+1] }}
  {{ s + b[i] = b[0] + \ldots + b[i] }}
  ```

- (I and i = n-1) implies Q as before
Example: sum of array (attempt 3)

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

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

Suppose we use $i != n$ instead of $i != n-1$...

We can spot this bug because the postcondition no longer follows.

When $i = n$, we get:

$$s = b[0] + ... + b[n]$$

which is wrong
Example: sum of array (attempt 4)

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

```java
{{ b.length >= n }}
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 misorder the assignments to $i$ and $s$...

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

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

First assertion is not I.
Example: max of array

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

\[
\begin{align*}
&\{ \text{b.length }\geq n \text{ and } n > 0 \} \\
m &= b[0] \\
i &= 1 \\
\text{while } (i \neq n) \{ \\
&\quad \text{if } (b[i] > m) \\
&\quad \quad m = b[i]; \\
&\quad \quad i = i + 1;
&\}
\end{align*}
\]

\[
\{ m = \max(b[0], \ldots, b[n-1]) \}
\]
Example: max of array

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

{{ b.length $\geq$ n  and n $>$ 0 }}
m = b[0];
i = 1;
{{ Inv: m = max(b[0], \ldots, b[i-1]) }}
while (i != n) {
    if (b[i] $>$ m)
        m = b[i];
i = i + 1;
}
{{ m = max(b[0], \ldots, b[n-1]) }}
Example: max of array

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

```java
{{ b.length >= n and n > 0 }}
m = b[0];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) }}
while (i != n) {
    if (b[i] > m)
        m = b[i];
    i = i + 1;
}
{{ m = max(b[0], ..., b[n-1]) }}
```

- I holds initially: $m = \max(b[0])$
- Postcondition follows from invariant and $i = n$.
- Remains to check loop body...
Example: max of array

Consider the following code to compute $\max(b[0], ..., b[n-1])$:

\[
\begin{align*}
\{ & \text{ Inv: } m = \max(b[0], ..., b[i-1]) \} \\
\text{ while } (i != n) \{ \\
& \text{ if } (b[i] > m) \\
& \quad m = b[i] ; \\
& \{ \text{ m = max(b[0], ..., b[i]) } \} \\
& \quad i = i + 1 ; \\
& \{ \text{ m = max(b[0], ..., b[i-1]) } \} \\
\} 
\end{align*}
\]
Example: max of array

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

\[
\begin{aligned}
\{ \text{Inv: } m &= \max(b[0], \ldots, b[i-1]) \} \\
\text{while } (i \neq n) \{ \\
\quad &\text{if } (b[i] > m) \\
\quad &\quad m = b[i]; \\
\quad &\text{else} \\
\quad &\quad ; \\
\quad &\{ \text{m = max(b[0], \ldots, b[i])} \} \\
\quad i &= i + 1; \\
\} 
\end{aligned}
\]
Example: max of array

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

```c
{{ Inv: m = max(b[0], ..., b[i-1]) }}
while (i != n) {
    if (b[i] > m)
        m = b[i];
    {{ m = max(b[0], ..., b[i]) }}
    else
        ;
    {{ m = max(b[0], ..., b[i]) }}
    i = i + 1;
}
```
Consider the following code to compute $\max(b[0], \ldots, b[n-1])$:

{ Inv: $m = \max(b[0], \ldots, b[i-1])$ }
while (i != n) {
  if (b[i] > m)
    { b[i] = \max(b[0], \ldots, b[i]) }
    m = b[i];
    { m = \max(b[0], \ldots, b[i]) }
  else
    { m = \max(b[0], \ldots, b[i]) }
  i = i + 1;
}
Example: max of array

Consider the following code to compute $\max(b[0], ..., b[n-1])$:

```java
{{ Inv: m = max(b[0], ..., b[i-1]) }}
while (i != n) {
    {{ (b[i] > m and b[i] = max(b[0], ..., b[i])) or (b[i] <= m and m = max(b[0], ..., b[i]) }}
    if (b[i] > m)
        {{ b[i] = max(b[0], ..., b[i]) }}
    m = b[i];
    else
        {{ m = max(b[0], ..., b[i]) }}
    ;
    i = i + 1;
}
```

check that I implies this…
(requires some thought)
Consider the following code to compute $\max(b[0], \ldots, b[n-1])$:

$$
\begin{array}{l}
\{ \text{Inv: } m = \max(b[0], \ldots, b[i-1]) \}
\end{array}
$$

while ($i \neq n$) {

$$
\begin{array}{l}
\{ \text{m = } \max(b[0], \ldots, b[i-1]) \}
\end{array}
$$

if ($b[i] > m$)

$$
\begin{array}{l}
m = b[i];
\end{array}
$$

else

$$
;\end{array}
$$

$i = i + 1;$

}
Example: max of array

Consider code to compute $\text{indexOfMax}(b[0], \ldots, b[n-1])$:

$$
\begin{align*}
&\text{b.length} \geq n \text{ and } n > 0 \\
k = 0; \\
m = b[k]; \\
i = 1; \\
\text{while } (i \neq n) \{ \\
\quad \text{if } (b[i] > m) \{ \\
\quad\quad k = i; \\
\quad\quad m = b[k]; \\
\quad\} \\
\quad i = i + 1; \\
\}
\end{align*}
$$

$$
\begin{align*}
&\text{m = max}(b[0], \ldots, b[n-1]) \text{ and } m = b[k]
\end{align*}
$$
Example: max of array

Consider code to compute `indexOfMax(b[0], ..., b[n-1])`:

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    if (b[i] > m) {
        k = i;
        m = b[k];
    }
    i = i + 1;
}
{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}
```
Example: max of array

Consider code to compute $\text{indexOfMax}(b[0], \ldots, b[n-1])$:

\[
\begin{align*}
\{\{ \text{b.length} \geq n \text{ and } n > 0 \}\} \\
k &= 0; \\
m &= b[k]; \\
i &= 1; \\
\{\{ \text{Inv: } m = \max(b[0], \ldots, b[i-1]) \text{ and } m = b[k] \}\} \\
\text{while } (i \neq n) \{ \\
\quad \text{if } (b[i] > m) \{ \\
\quad \quad k = i; \\
\quad \quad m = b[k]; \\
\quad \}\n\quad i = i + 1; \\
\}\end{align*}
\]

\[
\{\{ \text{m = b[0] and } k = 0 \text{ and } i = 1 \}\} \\
\{\{ \text{m = max(b[0], \ldots, b[0]) and } m = b[k] \}\}
\]
Example: max of array

Consider code to compute $\text{indexOfMax}(b[0], ..., b[n-1])$:

\[
\begin{align*}
\{ & \text{ b.length } \geq n \text{ and } n > 0 \} \\
  & \begin{array}{l}
    k = 0; \\
    m = b[k]; \\
    i = 1; \\
    \{ & \text{ Inv: } m = \max(b[0], ..., b[i-1]) \text{ and } m = b[k] \} \\
    & \text{while (i } \neq n) \{ \\
    & \quad \text{if (b}[i] > m) \{ \\
    & \quad \quad k = i; \\
    & \quad \quad m = b[k]; \\
    & \quad \} \\
    & \quad i = i + 1; \\
    & \} \\
\{ & \text{ m } = \max(b[0], ..., b[i-1]) \text{ and } m = b[k] \} \text{ implies } \\
\{ & \text{ m } = \max(b[0], ..., b[n-1]) \text{ and } m = b[k] \}
\end{array}
\end{align*}
\]
Example: max of array

Consider code to compute $\text{indexOfMax}(b[0], \ldots, b[n-1])$:

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;

{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    if (b[i] > m) {
        k = i;
        m = b[k];
    }
    i = i + 1;
}

{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}
```

- I holds initially
- I and $i = n$ implies postcondition
Consider code to compute \texttt{indexOfMax(b[0], ..., b[n-1])}:

\begin{verbatim}
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    if (b[i] > m) {
        k = i;
        m = b[k];
    }
    i = i + 1;
    {{ m = max(b[0], ..., b[i-1]) and m = b[k] }}
}
{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}
\end{verbatim}

- I holds initially
- I and i = n implies postcondition
Example: max of array

Consider code to compute `indexOfMax(b[0], ..., b[n-1])`:

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;

{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    if (b[i] > m) {
        k = i;
        m = b[k];
    }
}

{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}
```

- I holds initially
- I and i = n implies postcondition
Example: max of array

Consider code to compute $\text{indexOfMax}(b[0], \ldots, b[n-1])$:

\[
\begin{align*}
\{ \text{b.length} &\geq n \text{ and } n > 0 \} & & \text{\textbullet I holds initially} \\
\begin{array}{l}
k = 0; \\
m = b[k]; \\
i = 1; \\
\{ \text{Inv: } m = \max(b[0], \ldots, b[i-1]) \text{ and } m = b[k] \} \\
\end{array} & & \text{\textbullet I and } i = n \text{ implies postcondition} \\
\text{while } (i \neq n) \{ & \\
\begin{array}{l}
\text{if } (b[i] > m) \{ \\
k = i; \\
m = b[k]; \\
\{ \text{ m = } \max(b[0], \ldots, b[i]) \text{ and } m = b[k] \} \\
\} \text{ else } { \\
\{ \text{ m = } \max(b[0], \ldots, b[i]) \text{ and } m = b[k] \} \\
\}
\end{array} & \\
i = i + 1;
\}
\end{align*}
\]
Example: max of array

Consider code to compute `indexOfMax(b[0], ..., b[n-1])`:

\[
\begin{align*}
\{ \text{b.length} & \geq n \text{ and } n > 0 \} \\
& \quad \text{I holds initially} \\
& \quad \text{I and } i = n \text{ implies postcondition} \\
& \quad \text{Inv: } m = \text{max(b[0], ..., b[i-1]) and } m = b[k] \\
& \quad \text{while (i != n) } \{ \\
& \quad \quad \text{if (b[i] > m) } \{ \\
& \quad \quad \quad k = i; \\
& \quad \quad \quad \{ \text{b[k] = max(b[0], ..., b[i]) and b[k] = b[k]} \} \\
& \quad \quad \quad m = b[k]; \\
& \quad \quad \} \quad \text{else } \{ \\
& \quad \quad \quad \{ \text{m = max(b[0], ..., b[i]) and m = b[k]} \} \\
& \quad \quad \} \\
& \quad \quad i = i + 1; \\
& \quad \} \\
\end{align*}
\]
Example: max of array

Consider code to compute `indexOfMax(b[0], ..., b[n-1])`:

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    if (b[i] > m) {
        {{ b[i] = max(b[0], ..., b[i]) }}
        k = i;
m = b[k];
    } else {
        {{ m = max(b[0], ..., b[i]) and m = b[k] }}
    }
i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition
Example: max of array

Consider code to compute `indexOfMax(b[0], ..., b[n-1])`:

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    {{ (b[i] > m) and b[i] = max(b[0], ..., b[i]) or (b[i] <= m) and m = max(b[0], ..., b[i]) and m = b[k] }}
    if (b[i] > m) {
        k = i;
        m = b[k];
    }
    i = i + 1;
}

- I holds initially
- I and i = n implies postcondition

Remains to show that I is stronger than this (i.e., I implies this):
- if b[i] > m = max(b[0], ..., b[i-1]), then b[i] = max(b[0], ..., b[i])
- if b[i] <= m = max(b[0], ..., b[i-1]), then m = max(b[0], ..., b[i])
```
Consider code to compute `indexOfMax(b[0], ..., b[n-1]):`

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    {{ m = max(b[0], ..., b[i-1]) and m = b[k] }}
    if (b[i] > m) {
        {{ b[i] = max(b[0], ..., b[i]) }}
        k = i;
        m = b[k];
    } else {
        {{ m = max(b[0], ..., b[i]) and m = b[k] }}
    }
    i = i + 1;
```

- I holds initially
- I and i = n implies postcondition

Alternatively: reason forward down to each of these assertions
Example: max of array

Consider code to compute \texttt{indexOfMax(b[0], ..., b[n-1])}:

\begin{verbatim}
{{ b.length \geq n \text{ and } n > 0 }}
    k = 0;
    m = b[k];
    i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
    if (b[i] > m) {
        {{ b[i] > m and m = max(b[0], ..., b[i-1]) and m = b[k] }}
        {{ b[i] = max(b[0], ..., b[i]) }}
        k = i;
        m = b[k];
    } else {
        {{ b[i] \leq m and m = max(b[0], ..., b[i-1]) and m = b[k] }}
        {{ m = max(b[0], ..., b[i]) and m = b[k] }}
    }
}\end{verbatim}

\begin{itemize}
    \item I holds initially
    \item I and i = n implies postcondition
\end{itemize}

\textbf{first implies second ?}

\textbf{?}
Example: max of array

Consider code to compute `indexOfMax(b[0], ..., b[n-1])`:

```java
{{ b.length >= n and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
  if (b[i] > m) {
    k = i;
    m = b[k];
  }
  i = i + 1;
}
{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}
```

- I holds initially
- I and i = n implies postcondition
- I holds after loop body
Example: partition array

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

```c
{{ } }
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: b contains the same numbers since we use swaps.)
```
Example: partition array

Consider the following code to put the negative values at the beginning of array \( b \):

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

\[
\{\text{b[0], ..., b[k-1] < 0 \leq b[k], ..., b[n-1]} \}
\]
Example: partition array

Consider the following code to put the negative values at the beginning of array \( b \):

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

- \( I \) holds initially:
  - \( b[0], ..., b[-1] \) is empty
- \( I \) and \( i = n \) implies postcondition
Example: partition array

Consider the following code to put the negative values at the beginning of array \( b \):

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

- I holds initially
- I and \( i = n \) implies postcondition
Example: partition array

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

```c
{
  i = k = 0;
  {{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
  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[i-1] }}
  }
  {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
}
```

- $I$ holds initially
- $I$ and $i = n$ implies postcondition
Example: partition array

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

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

• I holds initially
• I and i = n implies postcondition
Consider the following code to put the negative values at the beginning of array $b$:

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

- $I$ holds initially
- $I$ and $i = n$ implies postcondition
Example: partition array

Consider the following code to put the negative values at the beginning of array \( b \):

\[
\begin{align*}
&\{\} \\
i = k = 0; \\
&\{ \text{Inv: } b[0], \ldots, b[k-1] < 0 \leq b[k], \ldots, b[i-1] \} \\
&\text{while (i != n) } \{ \\
&\quad \text{if (b[i] < 0) } \{ \\
&\quad\quad \{ \text{b[0], ..., b[k-1] < 0 \leq b[k], ..., b[i-1] and b[i] < 0 } \} \\
&\quad\quad \text{swap b[i], b[k];} \\
&\quad\quad k = k + 1; \\
&\quad\quad \{ \text{b[0], ..., b[k-1] < 0 \leq b[k], ..., b[i] } \} \\
&\quad \} \text{ else } \{ \\
&\quad \quad \{ \text{b[0], ..., b[k-1] < 0 \leq b[k], ..., b[i-1] and b[i] >= 0 } \} \\
&\quad \quad \{ \text{b[0], ..., b[k-1] < 0 \leq b[k], ..., b[i] } \} \\
&\quad \} \\
i = i + 1; \\
&\} \\
\end{align*}
\]

- \( I \) holds initially
- \( I \) and \( i = n \) implies postcondition
Example: partition array

Consider the following code to put the negative values at the beginning of array \( b \):

```c
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  } else {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] >= 0 }}
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
• I holds initially
• I and i = n implies postcondition
```

CSE 331 Spring 2017
Example: partition array

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

```c
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
    if (b[i] < 0) {
        {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
        swap b[i], b[k];
        k = k + 1;
        {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
    }
    i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- $I$ holds initially
- $I$ and $i = n$ implies postcondition

Remain to check this…
Example: partition array

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

```java
{{ }}
i = k = 0;
{{ Inv: $b[0]$, ..., $b[k-1] < 0 \leq b[k]$, ..., $b[i-1]$ }}
while (i != n) {
    if (b[i] < 0) {
        {{ b[0], ..., b[k-1] < 0 \leq b[k], ..., b[i-1] and b[i] < 0 }}
        swap b[i], b[k];
        k = k + 1;
        {{ b[0], ..., b[k-1] < 0 \leq b[k], ..., b[i] }}
    }
    i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 \leq b[k], ..., b[n-1] }}
```

- $I$ holds initially
- $I$ and $i = n$ implies postcondition
Example: partition array

Consider the following code to put the negative values at the beginning of array \( b \):

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

• \( I \) holds initially

• \( I \) and \( i = n \) implies postcondition

This is a valid triple.
(Takes some thought.)
Example: partition array

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

```c
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
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] }}
```

- I holds initially
- I and $i = n$ implies postcondition
- I holds after loop body