Checking Correctness of a Loop

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

\[
\begin{align*}
\{ \text{P} \} & \quad \{ \text{Inv: } I \} & \quad \{ \text{Q} \} \\
S1 & \quad \text{while (cond)} & \quad S2 \\
S3 & \quad \text{(can check these with backward reasoning instead)}
\end{align*}
\]

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

Formally, we need validity of:
- $\{ \text{P} \} S1 \{ I \}$
- $\{ I \text{ and cond } \} S2 \{ I \}$
- $\{ I \text{ and not cond } \} S3 \{ Q \}$
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 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 \( b[0] + \ldots + b[n-1] \):

```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] }}
```

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

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

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

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

More formal:

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

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

```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 $I$
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 = 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 }} s {{ I }} ?
```
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 = 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 ${{ I }}$

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

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

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

- $(s = 0 \text{ and } i = 0)$ implies $I$
- $\{ \{ I \text{ and } i \neq n \} \} \land \{ I \}$
- $\{ \{ I \text{ and not } (i \neq n) \} \}$ 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 $\mathbb{I}$
- $\{\{ \mathbb{I} \text{ and } i != n \}\} \subseteq \{\{ \mathbb{I} \}\}$
- $\{\{ \mathbb{I} \text{ and } i = n \}\}$ implies $\mathbb{Q}$

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: 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] }}  \[\text{Changed}\]
while (i != n-1) {
    i = i + 1;
    s = s + b[i];
}

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

Changed from $i = 0$

Changed from $n$

Reordered
Example: sum of array (attempt 2)

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

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

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

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

```java
{{ }}
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] }}
```

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

- $\{\{ I \text{ and } i \neq n-1 \}\} \implies \{\{ I \}\}$
  - reason backward

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

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

```java
{{ }}
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 $s$...

Where does the correctness check fail?
Example: sum of array (attempt 3)

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

Suppose we miss-order the assignments to $i$ and $s$:

```c
{{ }}
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] }}
```

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

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

First assertion is not Inv.
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:

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

For example, if $i = 2$, then

$$s = b[0] + b[1] + b[2] \quad \text{vs} \quad s = b[0] + b[1] + b[3]$$
Thinking About Loop Invariants

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

This triple is valid iff

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

• I holds initially
• I holds each time we execute S
• Q holds when I holds and cond is false
Thinking About Loop Invariants

• Loop invariant comes out of the algorithm idea
  – describes partial progress toward the goal
  – how you will get from start to end

• Essence of the algorithm idea is:
  – invariant
  – how you make progress on each step (e.g., \( i = i + 1 \))

• Code is *ideally* just details...
Loop Invariant $\rightarrow$ Code

In fact, can usually deduce the code from the invariant:

- When does loop invariant satisfy the postcondition?
  - gives you the termination condition

- What is the easiest way to satisfy the loop invariant?
  - gives you the initialization code

- How does the invariant change as you make progress?
  - gives you the rest of the loop body
Example: max of array

Write code to compute max(b[0], ..., b[n-1]):

\[
\begin{align*}
\{ \text{ b.length } \geq n \text{ and } n > 0 \} \\
\end{align*}
\]

??

\[
\begin{align*}
\{ \text{ Inv: } m = \text{ max}(b[0], ..., b[i-1]) \} \\
\text{ while } (?) \{ \\
\end{align*}
\]

??

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

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

\[
\begin{cases}
\text{Pre: } b.length \geq n \text{ and } n > 0 \\
\text{Inv: } m = \max(b[0], \ldots, b[i-1]) \\
\text{while (??) } \\
\quad ?? \\
\text{Post: } m = \max(b[0], \ldots, b[n-1])
\end{cases}
\]

When does Inv imply postcondition?
Example: max of array

Write code to compute \( \text{max}(b[0], \ldots, b[n-1]) \):

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

When does Inv imply postcondition? Happens when \( i = n \)
Example: max of array

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

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

Write code to compute max(b[0], ..., b[n-1]):

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

Easiest way to make this hold?
Example: max of array

Write code to compute max(b[0], ..., b[n-1]):

\[
\begin{align*}
\text{INV: } & m = \text{max}(b[0], \ldots, b[i-1]) \\
\text{while (}i\neq n\text{) } & \\
\text{INV: } & m = \text{max}(b[0], \ldots, b[n-1])
\end{align*}
\]

Easiest way to make this hold?
Take \( i = 1 \) and \( m = \text{max}(b[0]) \)
Example: max of array

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

\[
\begin{align*}
\{ & \text{b.length} \geq n \text{ and } n > 0 \} \\
\text{int } i &= 1; \\
\text{int } m &= b[0]; \\
\{ & \text{Inv: m = max(b[0], \ldots, b[i-1])} \\
\text{while } (i \neq n) & \{ \\
\text{??} \\
& \} \\
\{ & \text{m = max(b[0], \ldots, b[n-1])} \\
\end{align*}
\]
Write code to compute $\max(b[0], \ldots, b[n-1])$:

\[
\begin{align*}
&\begin{array}{l}
\text{inv: } m = \max(b[0], \ldots, b[i-1]) \\
\end{array} \\
\text{int } i = 1; \\
\text{int } m = b[0]; \\
\text{while } (i \neq n) \\
\quad \text{??} \\
\text{}} \end{align*}
\]

How do we progress toward termination? (comes from the algorithm idea)
Example: max of array

Write code to compute \( \max(b[0], ..., b[n-1]) \):

\[
\begin{align*}
\text{{ \{ b.length >= n and n > 0 \}}} \\
\text{int i = 1;} \\
\text{int m = b[0];}
\end{align*}
\]

\[
\begin{align*}
\text{{ Inv: m = max(b[0], ..., b[i-1]) }} \\
\text{while (i != n) \{ }
\end{align*}
\]

??

\[
\begin{align*}
\text{i = i + 1;} \\
\text{\}}
\end{align*}
\]

\[
\begin{align*}
\text{{ m = max(b[0], ..., b[n-1]) }}
\end{align*}
\]

How do we progress toward termination? We start at \( i = 1 \) and end at \( i = n \), so try this.
Example: max of array

Write code to compute max(b[0], ..., b[n-1]):

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

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

??
i = i + 1; {{ m = max(b[0], ..., b[i]) }}
{{ m = max(b[0], ..., b[i-1]) }}
}
{{ m = max(b[0], ..., b[n-1]) }}
Example: max of array

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

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

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

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

{{ 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]) }}
```
Example: max of array

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

\[
\begin{aligned}
\{ & \{ b.length \geq n \text{ and } n > 0 \} \\
& \text{int } i = 1; \\
& \text{int } m = b[0]; \\
\{ & \{ \text{Inv: } m = \max(b[0], \ldots, b[i-1]) \} \\
& \text{while } (i != n) \{ \\
& \quad \text{if } (b[i] > m) \\
& \quad \quad m = b[i]; \\
& \quad \quad i = i + 1; \\
& \} \\
\{ & \{ m = \max(b[0], \ldots, b[n-1]) \} \\
\end{aligned}
\]
Invariants are Essential

Invariant + progress step is the essence of the algorithm idea
• rest is hopefully just details that follow from the invariant

Work toward thinking at the level of invariants not code
• gain confidence that you can do the rest without difficulty

\[ m = \max(b[0], ..., b[i-1]) \]

\[ m = \max(b[i], ..., b[n-1]) \]
Loop Invariant Design Pattern

Loop invariant is often a weakening of the postcondition
– partial progress with completion a special case
– small enough weakening that Inv + one condition gives Q

1. sum of array
   – postcondition: s = b[0] + b[1] + … + b[n-1]
   – loop invariant: s = b[0] + b[1] + … + b[i-1]
     • gives postcondition when i = n

2. max of array
   – postcondition: m = max(b[0], b[1], …, b[n-1])
   – loop invariant: m = max(b[0], b[1], …, b[i-1])
     • gives postcondition when i = n
Loop Invariant Design Patterns

Algorithm Idea formalized in: Invariant + progress step

- how do you make progress toward termination?
  - if condition is $i \neq n$ (and $i \leq n$)
    try $i = i + 1$
  - if condition is $i \neq j$ (and $i \leq j$)
    try $i = i + 1$ or $j = j - 1$
Finding the loop invariant

Not every loop invariant is simple weakening of postcondition, but…
• that is the easiest case
• it happens a lot

In this class (e.g., homework):
• if I ask you to find the invariant, it will very likely be of this type
  – I will ask you to write more complex code when the invariant given
  – I will you to check correctness of even more complex code
  – HW2-4 will practice these
• to learn about more ways of finding invariants: CSE 421