CORRECTNESS
ISSUES AND
LOOP
INVARIANTS

Lecture 8
CS2110 – Fall 2015

The next several lectures

Study algorithms for searching and sorting arrays. Investigate their complexity – how much time and space they take “Formalize” the notions of average-case and worst-case complexity

We want you to know these algorithms
• Not by memorizing code but by
• Being able to develop the algorithms from their specifications and, when necessary, a small idea

We give you some guidelines and instructions on how to develop an algorithm from its specification.
Deal mainly with developing loops and loop invariants

How (not) to write an expression

/** Return value of "this person and p are intellectual siblings." *
 * Note: if p is null, they are not siblings. */
public boolean isPhDSibling(PhD p) {
    return p != null && this.equals(p) == false // p & this are not the same object
    && ((this.advisor1 != null && p.getFirstAdvisor() != null
        && this.advisor1.equals(p.getFirstAdvisor()))
        || (this.advisor2 != null && p.getSecondAdvisor() != null
            && this.advisor1.equals(p.getSecondAdvisor()))
        || this.advisor2.equals(p.getSecondAdvisor()))
    }

How to write an expression

• Avoid useless clutter, e.g.
  • unnecessary “this.”
  • unnecessary parentheses
  • redundant operations
• Put spaces around operators – use spaces to reflect relative precedences
• Be consistent, e.g.
  don’t use field for one object and getter for another
• Make the presentation on several lines reflect the structure of the expression

Simplify, don’t “complify” (complicate)
Show development of isPalindromes

```java
/** Return true iff s is a palindrome */
public static boolean isPalindrome(String s)

Our instructions said to visit each char of s only once!
```

isPalindromes: Set ispal to “s is a palindrome” (forget about returns for now. Store value in ispal.

```
Generalize to a picture that is true before/after each iteration

0                                                          s.length()

bac                                     …                                    cab

0                                      h                                      k

These sections are each others’ reverse

Do it with one variable?

```

Using only h makes it more difficult to figure out when to stop and doesn’t save any computation.

```
int h=  0;  int k=  s.length() – 1;
// s[0..h-1] is the reverse of s[k+1..]
while (h < k) {
    if (s.charAt(h) != s.charAt(k))
        return false;
    h=  h+1;  k=  k-1;
}
return true;
```

isPalindromes: written as a function.
Return when answer known

```
/** Return true iff s is a palindrome */
public static boolean isPal(String s) {
int h=  0;  int k=  s.length() – 1;
// invariant: s[0..h-1] is reverse of s[k+1..]
while (h < k) {
    if (s.charAt(h) != s.charAt(k))
        return false;
    h=  h+1;  k=  k-1;
}
return true;
```
Engineering principle

Break a project up into parts, making them as independent as possible. When the parts are constructed, put them together.

Each part can be understood by itself, without mentioning the others.

Reason for introducing loop invariants

Given \( c \geq 0 \), store \( b^c \) in \( x \):

\[
z = 1; \quad x = b; \quad y = c;
\]

while \( y \neq 0 \) {
  if \( y \) is even {
    x = x \times x; \quad y = y / 2;
  } else {
    z = z \times x; \quad y = y - 1;
  }
}

\( \{ z = b^c \} \)

Algorithm to compute \( b^c \).

Can’t understand any piece of it without understanding all. In fact, only way to get a handle on it is to execute it on some test case.

In fact, only way to get a handle on it is to execute it on some test case.

Need to understand initialization without looking at any other code.

Need to understand condition \( y \neq 0 \) without looking at loop body

Etc.

Invariant: is true before and after each iteration

\[
\text{initialization; } \quad \text{// invariant } P \\
\text{while (B) } \{ S \}
\]

Upon termination, we know \( P \text{ true; } B \text{ false} \)

“invariant” means unchanging. Loop invariant: an assertion —a true-false statement— that is true before and after each iteration of the loop —every time \( B \) is to be evaluated.

Help us understand each part of loop without looking at all other parts.

Simple example to illustrate methodology

Store sum of 0..n in \( s \)

Precondition: \( n \geq 0 \)

\[
\text{// } \{ n \geq 0 \}
\]

\( k = 1; \quad s = 0; \)

\[
\text{// } \text{inv: } s = \text{sum of } 0..k-1 \text{ \&\& } 0 \leq k \leq n+1
\]

while \( (k \leq n) \) {
  \( s = s + k; \)
  \( k = k + 1; \)
}

\( \{ s = \text{sum of } 0..n \} \)

First loopy question.

Does it start right?

Yes!

\( s = \text{sum of } 0..k-1 \)

\( \quad = \langle \text{substitute initialization} \rangle \)

\( \quad = \text{sum of } 0..1-1 \)

\( \quad = \langle \text{arithmetic} \rangle \)

\( \quad = \text{sum of } 0..0 \)

We understand initialization without looking at any other code

We understand initialization without looking at any other code

Second loopy question.

Does it stop right?

Upon termination, is postcondition true?

Yes!

\( \text{inv } \&\& \quad k = n \)

\( \quad \Rightarrow \langle \text{look at inv} \rangle \)

\( \quad \Rightarrow \text{inv } \&\& \quad k = n+1 \)

\( \quad \Rightarrow \text{inv} \)

\( \quad s = \text{sum of } 0..n+1-1 \)

We understand that postcondition is true without looking at init or repetend

We understand that postcondition is true without looking at init or repetend

Third loopy question.

Progress?

Does the repetend make progress toward termination?

Yes! Each iteration increases \( k \), and when it gets larger than \( n \), the loop terminates

We understand that there is no infinite looping without looking at init and focusing on ONE part of the repetend.
Simple example to illustrate methodology

Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
k= 1; s= 0;
// inv: s = sum of 0..k-1 &&
// 0 <= k <= n+1
while (k <= n) {
    s= s + k;
k= k + 1;
}
{s = sum of 0..n}

Fourth loopy question.
Invariant maintained by each iteration?
Is this property true?
{inv && k <= n} repetend {inv}
Yes!
{inv}

Note on ranges m..n

Range m..n contains n+1-m ints: m, m+1, ..., n
(Think about this as "Follower (n+1) minus First (m)")
2..4 contains 2, 3, 4: that is 4 + 1 - 2 = 3 values
2..3 contains 2, 3: that is 3 + 1 - 2 = 2 values
2..2 contains 2: that is 1 + 1 - 2 = 0 values
2..1 contains: that is 1 + 1 - 2 = 0 values
Convention: notation m..n implies that m <= n+1
Assume convention even if it is not mentioned!
If m is 1 larger than n, the range has 0 values

For loopy questions to reason about invariant

Given c >= 0, store b^c in x
z= 1; x= b; y= c;
// invariant y >= 0 AND
// z^x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x; y= y/2;
    } else {
        z= z*x; y= y - 1;
    }
}
{z = b^c}

Second loopy question.
Does it stop right?
When loop terminates, is z = b^c?
Yes! Take the invariant, which is true, and use fact that y = 0:

\[ z^x^y = b^c \]
\[ = y = 0 \]
\[ = z^x^0 = b^c \]
\[ = \text{arithmetic} \]
\[ = z = b^c \]

We understand loop condition without looking at any other code

Can’t understand this example without invariant!

Given c >= 0, store b^c in z
z= 1; x= b; y= c;
// invariant y >= 0 AND
// z^x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x; y= y/2;
    } else {
        z= z*x; y= y - 1;
    }
}
{z = b^c}

First loopy question.
Does it start right?
Is \{Q\} init \{P\} true?
Second loopy question:
Does it stop right?
Does P && ! B imply R?
Third loopy question:
Does repetend make progress?
Will B eventually become false?
Fourth loopy question:
Does repetend keep invariant true?
Is \{P && ! B\} S \{P\} true?

For loopy questions to reason about invariant

Given c >= 0, store b^c in x
z= 1; x= b; y= c;
// invariant y >= 0 AND
// z^x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x; y= y/2;
    } else {
        z= z*x; y= y - 1;
    }
}
{z = b^c}

Third loopy question.
Does repetend make progress toward termination?
Yes! We know that y > 0 when loop body is executed. The loop body decreases y.
We understand progress without looking at initialization

For loopy questions to reason about invariant

Given c >= 0, store b^c in x
z= 1; x= b; y= c;
// invariant y >= 0 AND
// z^x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x; y= y/2;
    } else {
        z= z*x; y= y - 1;
    }
}
{z = b^c}

Second loopy question.
Does it stop right?
When loop terminates, is z = b^c?
Yes! Take the invariant, which is true, and use fact that y = 0:

\[ z^x^y = b^c \]
\[ = y = 0 \]
\[ = z^x^0 = b^c \]
\[ = \text{arithmetic} \]
\[ = z = b^c \]

We understand loop condition without looking at any other code
For loopy questions to reason about invariant

Given \( c \geq 0 \), store \( b^c \) in \( x \)

\[
\begin{align*}
z & = 1; \quad x = b; \quad y = c; \\
// \text{invariant } y \geq 0 \text{ AND} \\
// & z \times x^y = b^c \\
\text{while } (y \neq 0) \{ \\
& \quad \text{if } (y \text{ is even}) \{ \\
& \qquad x = x \times x; \quad y = y/2; \\
& \quad \} \text{ else } \{ \\
& \qquad z = z \times x; \quad y = y - 1; \\
& \} \\
\} \quad x = b^c \\
\end{align*}
\]

Fourth loopy question. Does repetend keep invariant true?

Yes! Because of properties:
- For even, \( x^y = (x^2)^{y/2} \)
- \( z \times x^y = z \times x \times x^{y-1} \)

We understand invariance without looking at initialization.

Fourth loopy question.

Does repetend keep invariant true?

Yes! Because of properties:

1. For even, \( x^y = (x^2)^{y/2} \)
2. \( z \times x^y = z \times x \times x^{y-1} \)

Develop binary search for \( v \) in sorted array \( b \)

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>?</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt;= v</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

Store in \( h \) to make this true:

\[
\begin{align*}
0 & \quad h & \quad > v & \quad b.length \\
\end{align*}
\]

Example:

\[
\begin{align*}
0 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad b.length \\
\end{align*}
\]

If \( v \) is 4, 5, or 6, \( h \) is 5
If \( v \) is 7 or 8, \( h \) is 6

When does it start (what makes the invariant true)?

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>?</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv:</td>
<td>b</td>
<td>&lt;= v</td>
<td>?</td>
</tr>
</tbody>
</table>

Make first and last partitions empty:

\[
\begin{align*}
h & = -1; \quad t = b.length; \\
\end{align*}
\]

How does body make progress toward termination (cut ? in half) and keep invariant true?

<table>
<thead>
<tr>
<th>inv:</th>
<th>b</th>
<th>&lt;= v</th>
<th>?</th>
<th>&gt; v</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

Let \( e \) be index of middle value of ? Section. Maybe we can set \( h \) or \( t \) to \( e \), cutting ? section in half.

h = -1; t = b.length;
while (h != t - 1) {
    int e = (h + t) / 2;
}

Stop when ? section is empty. That is when \( h = t - 1 \). Therefore, continue as long as \( h \neq t - 1 \).

Stop when ? section is empty. That is when \( h = t - 1 \). Therefore, continue as long as \( h \neq t - 1 \).
How does body make progress toward termination (cut $t$ in half) and keep invariant true?

inv: $b[0] \leq v \iff t \geq b.length$

$h = -1; t = b.length;$

while $h != t-1$

if $b[e] \leq v$ then $h = e;$

else $t = e;$

If $b[e] > v$, then so is every value to its right, since the array is sorted. Therefore, $t = e$; keeps the invariant true.

Develop binary search in sorted array $b$ for $v$

pre: $b[0..b.length-1]$ has been processed

post: $b[0..k-1]$ has been processed

DON'T TRY TO MEMORIZE CODE!
Instead, learn to derive the loop invariant from the pre- and post-condition and then to develop the loop using the pre- and post-condition and the loop invariant.

Processing arrays from beg to end (or end to beg)

Task: Process $b[0..b.length-1]$

$k = 0;$

while $k != b.length$

$k = k + 1;$

Process $b[k]$;

[R: $b[0..b.length-1]$ has been processed]
Count the number of zeros in `b`.
Start with last program and refine it for this task

Task: Set `s` to the number of 0's in `b[0..b.length-1]`

- `k = 0;`  
- `s = 0;`  

```java
{inv P}
while ( k != b.length ) {
    if ( b[k] == 0 ) s = s + 1;
    k = k + 1;  // progress toward termination
}
{R: s = number of 0's in b[0..b.length-1]}

inv P:  
0                           k                             b.length
```

Be careful. Invariant may require processing elements in reverse order!

This invariant forces processing from beginning to end

0  k                                     b.length

inv P: 

```
0  k
```

Process elements from end to beginning

```java
k = b.length–1;  // how does it start?
while (k >= 0) {  // how does it end?
    Process b[k];  // how does it maintain invariant?
    k = k – 1;  // how does it make progress?
}
{R: b[0..b.length-1] is processed}

inv P:  
0                           k                             b.length
```

Heads up! It is important that you can look at an invariant and decide whether elements are processed from beginning to end or end to beginning!

For some reason, some students have difficulty with this. A question like this could be on the prelim!

Process elements from end to beginning

```java
k = b.length–1;  // how does it start?
while (k >= 0) {  // how does it end?
    Process b[k];  // how does it maintain invariant?
    k = k – 1;  // how does it make progress?
}
{R: b[0..b.length-1] is processed}

inv P:  
0                           k                             b.length
```