CORRECTNESS ISSUES
AND LOOP INVARIANTS

Lecture 8
CS2110 – Spring 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.

Many (most) of you could use instruction on
developing algorithms, keeping things simple

```java
String[] dummy = s.split("*"); // turns s into string array
int len = s.length()-1; // length of string s
String a = ""; // will be reverse of s
for (int b = len; b > -1; b--){
    a += dummy[b];
}
if (s.equals(a))  return true;
else  return false;
```

This submitted code for body of isPalindrome didn’t work because split
wasn’t used properly – and it wasn’t debugged

Why calculate the reverse of s?

Some principles and strategies for development

• Don’t introduce a variable without a good reason.
• Put local variables as close to their first use as possible.
• Structure expressions to make them readable.
• Make the structure of the program reflect the structure of the data.
• Never have lots of syntax errors.
• Intersperse coding and testing: code a little, test a little.
• Write the class invariant while putting in field declarations.
• Write a method spec before writing the method body.
• Use assert statements to check method preconditions – as along as
it doesn’t complicate program too much and doesn’t change the
time-complexity of the method.

Show development of isPalindrome

```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!
```

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

Think of checking equality of outer chars, then chars inside them, then chars inside them, etc.

```plaintext
bac … cab
```

Key idea:
Generalize this to a picture that is true before/after each iteration
**isPalindrome**: 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()
 sbac     …    cusch
```

These sections are each others’ reverse

```
0    h   ?     k
```

```
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;
}
ispal=    h >= k;
```

**isPalindrome**: Set ispal to “s is a palindrome”

```
0                                s.length()
 sbac     …    cusch
```

These sections are each others’ reverse

```
0    h   ?     k
```

```
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;
}
ispal=    h >= k;
```

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

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.

Need to understand initialization without looking at any other code.

Need to understand condition y != 0 without looking at loop body etc.

```
init  {P}
B      \true
\false
S
```

Upon termination, we know P true, B false {P and !B}

“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 >= 0
// { n >= 0}
\n\nFirst loopy question.
Does it start right?
Does initialization make invariant true?

\n\nYes! s = sum of 0..k-1
\n\nWe understand initialization without looking at any other code

Simple example to illustrate methodology

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;
}
\n\nWe understand that postcondition is true without looking at init or repetend

Simple example to illustrate methodology

Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
\n\nSecond loopy question.
Does it stop right?
Upon termination, is postcondition true?

Yes!

\n\ninv && ! k <= n
\n\n\nWe 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;
}
\n\nThird 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

\n\nWe 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}
\n\nFourth loopy question.
Invariant maintained by each iteration?
Is this property true?

Yes!

\n\n{inv && k <= n} repetend {inv}

\n\n4 loopy questions to ensure loop correctness

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 loop invariant true?
Is \{P && ! B\} \{P\} true?

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 2 + 1 – 2 = 1 value
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

\n\narray segment b[m..n]:
\n\n| m | n |
Can’t understand this example without invariant!

First loopy question. Does it start right? Does initialization make invariant true?

Given c \geq 0, store b^c in z

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

We understand initialization without looking at any other code

For loopy questions to reason about invariant

Second loopy question. Does it stop right? When loop terminates, is \( z = b^c \)!

Third loopy question. Does it stop making progress toward termination?

Yes! We know that \( y > 0 \) when loop body is executed. The loop body decreases \( y \).

For loopy questions to reason about invariant

Fourth loopy question. Does it keep invariant true?

Yes! Because of properties:

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

We understand invariance without looking at initialization

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

Get loop invariant by combining pre- and post-conditions, adding variable \( t \) to mark the other boundary

inv: \begin{array}{cccc}
0 & h & t & b.\text{length} \\
\leq v & ? & > v \\
\end{array}

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

post: \begin{array}{cccc}
0 & h & ? & b.\text{length} \\
\leq v & h & > v \\
\end{array}

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

Example:

\begin{array}{ccccccccc}
0 & 2 & 4 & 5 & 6 & 7 & 9 & 9 & 9 \\
\leq v & 4, 5, 6 & > v \\
\end{array}

If \( v \text{ is 4, 5, or 6, } h \text{ is 5} \)

If \( v \text{ in } b, h \text{ is index of rightmost occurrence of } v. \)

If \( v \text{ not in } b, h \text{ is index before where it belongs.} \)
### How does it start (what makes the invariant true)?

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

Make first and last partitions empty:

\[ h = -1; \ t = \text{b.length}; \]

### When does it end (when does invariant look like postcondition)?

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

\[ h = -1; \ t = \text{b.length}; \]

**while** (h != t-1) {
  \[
  \text{h} = \text{e};
  \]

**else** t = e;

Stop when \( \text{? section} \) is empty. That is when \( h = t-1 \).

Therefore, continue as long as \( h != t-1 \).

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

| inv: | b <= v | ? | > v |

**Let e be index of middle value of \( ? \) Section.**

**Maybe we can set h or t to e, cutting \( ? \) section in half**

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

<table>
<thead>
<tr>
<th>Pre:</th>
<th>b</th>
<th>?</th>
<th>b.length</th>
</tr>
</thead>
</table>

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

| Post: | b | <= v | > v |

**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.

**PRACTICE THIS ON KNOWN ALGORITHMS!**
Many loops process elements of an array `b` (or a String, or any list) in order: `b[0]`, `b[1]`, `b[2]`, …
If the postcondition is

\[ R: b[0..b.length-1] \text{ has been processed} \]

Then in the beginning, nothing has been processed, i.e.

\[ b[0..-1] \text{ has been processed} \]

After \( k \) iterations, \( k \) elements have been processed:

\[ P: b[0..k-1] \text{ has been processed} \]

Invariant:

\[ \begin{array}{cccc}
0 & k & b.length \\
\hline
\text{processed} & \text{not processed} \\
\end{array} \]

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

\[ k= 0; \]  
\[ \{ \text{inv } P \} \]
\[ \text{while (} \quad \text{) } \} { \]
\[ \text{Process } b[k]; \quad \text{// maintain invariant} \]
\[ k= k + 1; \quad \text{// progress toward termination} \}
\[ } \]
\[ \{ R: b[0..b.length-1] \text{ has been processed} \} \]

Most loops that process the elements of an array in order will have this loop invariant and will look like this.

\[ \begin{array}{cccc}
0 & k & b.length \\
\hline
\text{processed} & \text{not processed} \\
\end{array} \]

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

\[ k= 0; \]  
\[ s= 0; \]  
\[ \{ \text{inv } P \} \]
\[ \text{while (} \quad \text{) } \} { \]
\[ \text{if (} \quad \text{) } \} { \]
\[ \quad s= s + 1; \quad \text{// progress toward termination} \]
\[ k= k + 1; \quad \text{// progress toward termination} \}
\[ } \]
\[ \{ R: s = \text{number of 0’s in } b[0..b.length-1] \} \]

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

This invariant forces processing from beginning to end

\[ \begin{array}{cccc}
0 & k & b.length \\
\hline
\text{processed} & \text{not processed} \\
\end{array} \]

This invariant forces processing from end to beginning

\[ \begin{array}{cccc}
0 & k & b.length \\
\hline
\text{not processed} & \text{processed} \\
\end{array} \]

Process elements from end to beginning

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

```
k = b.length–1;
while (k >= 0) {
    Process b[k];
    k = k – 1;
}
{R: b[0..b.length-1] is processed}  

inv P:  
      0   k  b.length
      not processed  processed
```