About A2 and feedback. Recursion

S2 has been graded. If you got 30/30, you will probably have no feedback.

If you got less than full credit, there should be feedback showing you which function(s) is incorrect.

If you don’t see feedback, ask for a regrade on the CMS. Please don’t email anyone asking for a regrade.

We will put on the course website some recursive functions for you to write, to get practice with recursion. This will not be an assignment. But if you know you need practice, practice!

Preconditions and Postconditions

Write it like this: \{Q\} S \{R\}


\{Q\} S \{R\} is a true-false statement. Read it as follows:
Execution of S begun in a state in which Q is true is guaranteed to terminate, and in a state in which R is true.

Annotating more completely with assertions

The blue things are assertions —pre- and post-conditions. They help prove that if 0 <= c at the beginning, ans = b^c before the return.

Axiomatic definition of a language

Hoare gave rules for deciding whether a Hoare triple \{Q\} S \{R\} was correct. Defined the language in terms of correctness instead of execution.

See that in later courses. We concentrate on one aspect: how to “prove” loops correct.
Axiomatic definition of a language

```java
/** Return b^c. Precondition 0 <= c */
public static int exp(int b, int c) {
    int ans;
    if (c == 0)  {
        ans= 1;
    } else {
        ans= b * exp(b, c-1);
    }
    return ans;
}
```

Hoare gave rules for deciding whether a Hoare triple \{Q\} S \{R\} was correct. Defined the language in terms of correctness instead of execution. See that in later courses. We concentrate on one aspect: how to “prove” loops correct.

Reason for introducing loop invariants

Algorithm to compute b^c.

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

Invariant: is true before and after each iteration

```
initialization;
// invariant P
while (B) {S}
```

Upon termination, we know P true; B 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 >= 0
  k= 1; s= 0;
  while (k <= n) {
    s= s + k;
    k= k + 1;
  }
{ s = sum of 0..n }
```

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

Second loopy question. Does it stop right? Upon termination, is postcondition true?

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 Hoare triple true?
{ inv && k <= n} repetend {inv}
Yes!
{s = sum of 0..k-1} s = s + k;
{k = k+1;}
{s = sum of 0..k-1}

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?

Note on ranges m..n

Range m..n contains n+1-m ints: m, m+1, ..., n
(Talk 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

array segment b[m..n]:

m n

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
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; \\
x &= b; \\
y &= c;
\end{align*}
// invariant \( y \geq 0 \) AND \\
// \( z \times x^y = b^c \)

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

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

We understand invariance without looking at initialization

Fourth loopy question. Does repetend keep invariant true?

Yes! Because of properties:

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

Designing while-loops or for-loops

Many loops process elements of an array \( b \) (or a String, or any list) in order: \( b[0], b[1], b[2], \ldots \)

If the postcondition is

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

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:

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

\( \text{inv P: } b \)

\begin{array}{c|c|c}
0 & k & b.length \\
\end{array}

or draw it as a picture

\( k = 0; \}

\{ \text{inv P} \}

while ( \( k \neq b.length \) ) {
    k = k + 1; \\
    \text{Process } b[k]; \\
}

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

Developing while-loops (or for loops)

Task: Process \( b[0..b.length-1] \)

Replace \( b.length \) in postcondition by fresh variable \( k \) to get invariant \( b[0..k-1] \text{ has been processed} \)

\( \text{inv P: } b \)

\begin{array}{c|c|c}
0 & k & b.length \\
\end{array}

or draw it as a picture

\( k = 0; \}

\{ \text{inv P} \}

while ( \( k \neq b.length \) ) {
    k = k + 1; \\
    \text{Process } b[k]; \\
}

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

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

\( \text{inv P: } b \)

\begin{array}{c|c|c}
0 & k & b.length \\
\end{array}

Counting 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; \}

\{ \text{inv P} \}

while ( \( k \neq b.length \) ) {
    k = k + 1; \\
    \text{Process } b[k]; \\
}

\( \{ \text{R: } s = \text{number of 0’s in } b[0..b.length-1] \} \)

\( \text{inv P: } b \)

\begin{array}{c|c|c}
0 & k & b.length \\
\end{array}

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

This invariant forces processing from beginning to end

\( \text{inv P: } b \)

\begin{array}{c|c|c}
0 & k & b.length \\
\end{array}

This invariant forces processing from end to beginning

\( \text{inv P: } b \)

\begin{array}{c|c|c}
0 & k & b.length \\
\end{array}
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:
<table>
<thead>
<tr>
<th>b</th>
<th>not processed</th>
<th>processed</th>
</tr>
</thead>
</table>

```
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!

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

```
pre: b 0 \( 4 \) 5 6 7 b.length
post: b 0 \( \leq v \) \( > v \) b.length
```

Example:
```
| 2 | 4 | 4 | 4 | 7 | 9 | 9 | 9 |
```

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

If \( v \) in \( b \), \( h \) is index of rightmost occurrence of \( v \).
If \( v \) not in \( b \), \( h \) is index before where it belongs.

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

```
pre: b 0 4 5 6 7 b.length
inv: b \( \leq v \) 4 5 6 7 b.length
```

Make first and last partitions empty:
```
h = -1; t = b.length;
```

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

```
pre: b 0 4 5 6 7 b.length
post: b \( \leq v \) \( > v \) b.length
```

Store a value in \( h \) to make this true:
```
0 4 5 6 7 b.length
```

Get loop invariant by combining pre- and post-conditions, adding variable \( t \) to mark the other boundary
```
inv: b \( \leq v \) \( ? \) \( > v \) b.length
```

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

```
pre: b 0 4 5 6 7 b.length
post: b \( \leq v \) \( > v \) b.length
inv: b \( \leq v \) \( ? \) \( > v \) b.length
```

Stop when \( ? \) 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?

<table>
<thead>
<tr>
<th>inv:</th>
<th>0</th>
<th>h</th>
<th>t</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>&lt;= v</td>
<td>e</td>
<td>t</td>
<td>b.length</td>
</tr>
<tr>
<td>b</td>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
<td></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;
}

If b[e] <= v, then so is every value to its left, since the array is sorted. Therefore, h = e; keeps the invariant true.

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

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

<table>
<thead>
<tr>
<th>inv:</th>
<th>0</th>
<th>h</th>
<th>t</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>&lt;= v</td>
<td>e</td>
<td>t</td>
<td>b.length</td>
</tr>
<tr>
<td>b</td>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
<td></td>
</tr>
</tbody>
</table>

h = -1; t = b.length;
while (h != t-1) {
    int e = (h+t)/2;
    if (b[e] <= v) h = e;
    else t = e;
}

If b[e] <= v, then so is every value to its left, since the array is sorted. Therefore, h = e; keeps the invariant true.