Loop invariants

Loop invariants: Section 5.5 in Rosen

The programmer's wife tells him “Go to the store and pick up a loaf of bread. If they have eggs, get a dozen.”

The programmer comes back with 12 loaves of bread.

Loop invariants as a way of reasoning about the state of your program

We want to prove: \( i = n \) right after the loop

Example: loop index value after loop

// precondition: n>=0
int i = 0;
// i<=n : loop invariant while (i < n){
    // i < n test passed
    // AND
    // i<=n loop invariant
    i++;
    // i <= n loop invariant
} // i=n AND i <= n \( \rightarrow \) i=n

So we can conclude the obvious:
\( i=n \) right after the loop

Loop invariants

- A way to reason about the correctness of a program
- A loop invariant is a predicate
  - that is true directly before the loop executes
  - that is true before and after each repetition of the loop body
  - and that is true directly after the loop has executed
  - i.e., it is kept invariant by the loop.
Loop invariants

- Combined with the loop condition, the loop invariant allows us to reason about the behavior of the loop:

  \( \text{loop invariant} \)
  \[ \text{while(test)} \]
  \[ \text{<test AND loop invariant> S;} \]
  \[ \text{<loop invariant>} \]
  \[ \text{<not test AND loop invariant>} \]

What does it mean...

- If we can prove that
  the loop invariant holds before the loop
  and that
  the loop body keeps the loop invariant true
  i.e. \( \text{<test AND loop invariant> S; <loop invariant>} \)
  then we can infer that
  \( \text{not test AND loop invariant} \)
  holds after the loop terminates

Example: sum of elements in an array

```java
int total (int[] elements){
    int sum = 0, i = 0, n = elements.length;
    // sum == sum of elements from 0 to i-1
    while (i < n){
        // sum == sum of elements 0...i-1
        sum += elements [i];
        // sum == sum of elements 0...i-1
    }
    // i==n (previous example) AND
    // sum == sum elements 0...i-1
    // \to sum == sum of elements 0...n-1
    return sum;
}
```

Loop Invariant for Selection Sort

```java
public void selectionSort (Comparable [] array){
    int min;
    for (int i = 0; i < array.length-1; i++) {
        min = i;
        for (int j = i+1; j < array.length; j++){
            if (array[j].compareTo(array[min]) < 0)
                min = j;
        }
        swap (array, min, i);
    }

    \text{Invariant?}
```
Loop Invariant for Selection Sort

```java
public void selectionSort (Comparable [] array) {
    int min;
    for (int i = 0; i < array.length-1; i++) {
        min = i;
        for (int j = i+1; j < array.length; j++){
            if (array[j].compareTo(array[min]) < 0) 
                min = j;
        }
        swap (array, min, i);
    }
}

Invariant: array[0]...array[i] are in sorted order
```

Loop Invariant for Insertion Sort

```java
public void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        Comparable temp = array[i];
        int position = i;
        while (position > 0 && array[position-1].compareTo(temp) > 0) {
            array[position] = array[position-1];
            position--;
        }
        array[position] = temp;
    }
}

Invariant: array[0]...array[i-1] consists of elements originally in this range of indexes, but in sorted order
```

Bubble Sort

```java
public void bubbleSort(Comparable [] array) {
    for (int position = array.length-1; position>=0; position--) {
        for (int i = 0 ; i < position; i++) {
            if (array[i].compareTo(array[i+1]) > 0) 
                swap(array, i, i+1);
        }
    }
}

Invariant?
```

```java
public void bubbleSort(Comparable [] array) {
    for (int position = array.length-1; position>=0; position--) {
        for (int i = 0 ; i < position; i++) {
            if (array[i].compareTo(array[i+1]) > 0) 
                swap(array, i, i+1);
        }
    }

Inner Invariant: array[i] is the largest element in the first i elements in the array
Outer Invariant: After i iterations the largest i elements are in their correct sorted position
```
Closed Curve Game

There are two players, Red and Blue. The game is played on a rectangular grid of points:

```
6 . . . . . .
5 . . . . . .
4 . . . . . .
3 . . . . . .
2 . . . . . .
1 . . . . . .
1 2 3 4 5 6 7
```

Red draws a red line segment, either horizontal or vertical, connecting any two adjacent points on the grid that are not yet connected by a line segment. Blue takes a turn by doing the same thing, except that the line segment drawn is blue. Red's goal is to form a closed curve of red line segments. Blue's goal is to prevent Red from doing so.

We can express this game as a computer program:

```java
while (more line segments can be drawn) {
    Red draws line segment;
    Blue draws line segment;
}
```

Question: Does either Red or Blue have a winning strategy?

Answer: Yes! Blue is guaranteed to win the game by responding to each turn by Red in the following manner:

If Red drew a horizontal line segment:

- `let i and j be such that Red’s line segment connects (i, j) with (i, j+1)`
- If `i>1`:
  - `draw a vertical line segment connecting (i-1, j+1) with (i, j+1)`
  - Else:
    - `draw a line segment anywhere`

If Red drew a vertical line segment:

- `let i and j be such that Red’s line segment connects (i, j) with (i+1, j)`
- If `j>1`:
  - `draw a horizontal line segment connecting (i+1, j-1) with (i+1, j)`
  - Else:
    - `draw a line segment anywhere`
```

Closed Curve Game

By following this strategy Blue guarantees that Red does not have an “upper right corner” at any step.

So, the invariant is:

There does not exist on the grid a pair of red line segments that form an upper right corner.

And in particular, Red has no closed curve!
### Example: Egyptian multiplication

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

19 x 5:

<table>
<thead>
<tr>
<th>/2</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>/2</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>/2</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>/2</td>
<td>1</td>
<td>80</td>
</tr>
</tbody>
</table>

Throw away all rows with even A:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Add B's

95

\[ \text{---> the product!!} \]

### Can we show it works? Loop invariants!!

```
// precondition: left > 0 AND right > 0
int a=left, b=right, p=0;  // p: the product
// p + (a*b) == left * right
while (a!=0){
    // a!=0 and p + (a*b) == left * right
    // loop condition and loop invariant
    if (odd(a))  p+=b;
    a/=2;
    b*=2;
    // p + (a*b) == left*right
}
// a==0 and p+a*b == left*right  \rightarrow  p == left*right
```

### Try it on 7 * 8

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```
3   16  \leftrightarrow b: 8
1   32  \leftrightarrow b: 24
0   64  \leftrightarrow b: 56
```

### Try it on 8*7

| left | right | a | b | p |
| 8    | 7     | 8 | 7 | 0 |

```
4   14  \leftrightarrow b: 8
2   28  \leftrightarrow b: 0
1   56  \leftrightarrow b: 0
0   118 \leftrightarrow b: 56
```
Relation to binary representation $19 \times 5$

\[
\begin{array}{c}
00101 \\
10011 \\
\hline
101 5 \\
1010 10 \\
00000 \\
00000 \\
1010000 80 \\
\hline
1011111 95
\end{array}
\]

Incorporating loop invariants into your code

- An assertion is a statement that says something about the state of your program
- Should be true if there are no mistakes in the program
- Can be used to check that a loop invariant holds true

Example:

```c
int p=..., d=...;
int q = p/d;
int r = p%d;
assert p == q*d + r;
```

Example

```
// precondition: left >0 AND right >0
int a=left, b=right, p=0; //p: the product
assert (p + (a*b) == left * right); //loop invariant
while (a!=0):
    assert (p + (a*b) == left * right);
    if (odd(a))  p+=b;
    a/=2;
    b*=2;
    assert (p + (a*b) == left*right);
}
assert (p + (a*b) == left * right);
// a==0 and p+a*b == left*right ➔ p == left*right
```

Using assertions

- Assertions may slow down execution. For example, if an assertion checks to see if the element to be returned is the smallest element in the list, then the assertion would have to do the same amount of work that the method would have to do
- Therefore assertions can be enabled and disabled
- Assertions are, by default, disabled at run-time
- Think of assertions as a debugging tool
- Don’t use assertions to flag user errors, because assertions can be turned off
Enabling assertions

- Need to set a compiler flag.
- Go to Run -> Run Configurations -> Arguments, and in the box labeled VM arguments, enter either -enableassertions or just -ea

Summary: Loop Invariant Reasoning

// loop invariant true before loop
while (b){
  // b AND loop invariant
  S;
  // loop invariant
}
// not b AND loop invariant

loop invariants as a way of proving correctness of loop-based algorithms