# Loop invariants

Loop invariants: Section 5.5 in Rosen

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

```
<precondition: n>0>
int i = 0;
 while (i < n){
    i = i+1;
}
<post condition: i==n>
```

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  → 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:

```
 <loop invariant>
while(test){
    <test AND loop invariant>
     S;
    <loop invariant>
}
<not test AND loop invariant>
```

# What does it mean...

```
<loop invariant>
while(test){
   <test AND
   loop invariant>
      S;
    <loop
   invariant>
}
<not test AND
 loop invariant>
```

**If we can prove** that
the loop invariant holds before the loop
and that
the loop body keeps the loop invariant true
i.e. <test AND loop invariant> S; <loop invariant>

**then we can infer** that

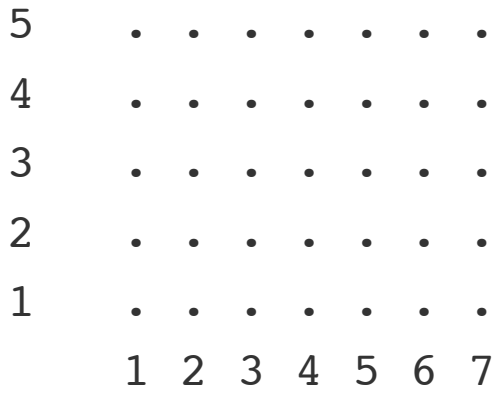
not test AND loop invariant
holds after the loop terminates

# Example: sum of elements in an array

```
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];
        i++;
        // sum == sum of elements 0...i-1
    }
    // i==n (previous example) AND
    // sum == sum elements 0...i-1
    // →  sum == sum of elements 0...n-1
    return sum;
}
```

# Closed Curve Game

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




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.

See http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html

# Closed Curve Game

- We can express this game as a computer program:

```
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?

# Closed Curve Game

- **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
    }
 } else  // 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

| | | A | B | |
|---|---|---|---|---|
| | | 19 | 5 | |
| 19 x 5: | /2 | 9 | 10 | *2 |
| | /2 | 4 | 20 | *2 |
| | /2 | 2 | 40 | *2 |
| | /2 | 1 | 80 | *2 |

throw away all rows with even A:

| A | B |
|---|---|
| 19 | 5 |
| 9 | 10 |
| 1 | 80 |
| | ——— |
| add B's | 95 |

--> 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  loop invariant
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  →  p == left*right
```

# Try it on 7 * 8

| left | right | a | b | p |
|---|---|---|---|---|
| 7 | 8 | 7 | 8 | 0 |
| | | 3 | 16 | +=b: 8 |
| | | 1 | 32 | +=b: 24 |
| | | 0 | 64 | +=b: 56 |

# Try it on 8*7

| left | right | a | b | p |
|---|---|---|---|---|
| 8 | 7 | 8 | 7 | 0 |
| | | 4 | 14 | 0 |
| | | 2 | 28 | 0 |
| | | 1 | 56 | 0 |
| | | 0 | 118 | +=b: 56 |

# Relation to binary representation 19*5

```
   00101
   10011
  ------
     101    5
    1010   10
   00000
  000000
 1010000   80
 -------
 1011111   95
```

# Summary: Loop Invariant Reasoning

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

not b helps you make a stronger observation than loop invariant alone.