Announcements

- Lab 14 (there is no Lab 13) goes out next week and is the last lab.
- A5 out by early next week. This is the last assignment.
- Prelim 2 grading will happen over the weekend.
- Do the Loop Invariant Reading before the Lab.
Recall: Important Terminology

- **assertion**: true-false statement placed in a program to *assert* that it is true at that point
  - Can either be a *comment*, or an *assert* command

- **invariant**: assertion supposed to *always* be true
  - If temporarily invalidated, must make it true again
  - **Example**: class invariants and class methods

- **loop invariant**: assertion supposed to be true before and after each iteration of the loop

- **iteration of a loop**: one execution of its body
Recall: The **while-loop**

```
while <condition>:
    statement 1
    ...
    statement n
```

**precondition**

**postcondition**

- **Precondition**: assertion placed before a segment
- **Postcondition**: assertion placed after a segment
4 Tasks in this Lecture

1. Setting the table for more people
   - Building intuitions about invariants

2. Summing the Squares
   - Designing your invariants

3. Count num adjacent equal pairs
   - How invariants help you solve a problem!

4. Find largest element in a list
   - How you need to be careful during initialization
Task 1: Setting the table for more people

precondition: n_forks are needed @ table

k = 0

while k < n_more_guests:
    # body goes here
...
    k = k + 1

postcondition: n_forks are needed @ table

• **Precondition:** before we start, we should have 2 forks for each guest (dinner fork & salad fork)

• **Postcondition:** after we finish, we should still have 2 forks for each guest

Relationship Between Two
If precondition is true, then postcondition will be true
Q: Completing the Loop Body

precondition: n_forks are needed @ table

k = 0

while k < n_more_guests:
    k = k + 1

postcondition: n_forks are needed @ table

What statement do you put here to make the postcondition true?

A: n_forks += 2
B: n_forks += 1
C: n_forks = k
D: None of the above
E: I don’t know
A: Completing the Loop Body

precondition: \( n_{\text{forks}} \) are needed @ table

\[ k = 0 \]

\textbf{while} \( k < n_{\text{more\_guests}} \):

\[ k = k + 1 \]

\textbf{postcondition:} \( n_{\text{forks}} \) are needed @ table

What statement do you put here to make the postcondition true?

A: \( n_{\text{forks}} += 2 \) \textbf{CORRECT}
B: \( n_{\text{forks}} += 1 \)
C: \( n_{\text{forks}} = k \)
D: None of the above
E: I don’t know
Invariants: Assertions That Do Not Change

Loop Invariant: an assertion that is true before and after each iteration (execution of body)

precondition: n_forks are needed @ table

k = 0

#INV: n_forks = num forks needed with k more guests

while k < n_more_guests:
    n_forks += 2
    k += 1

invariant holds before loop

invariant still holds here

postcondition: n_forks are needed @ table
What’s a Helpful Invariant?

Loop Invariant: an assertion that is true before and after each iteration (execution of body)

• Documents the semantic meaning of your variables and their relationship (if any)
• Should help you understand the loop

Bad:

\[ n_{forks} \geq 0 \]

True, but doesn’t help you understand the loop

Good:

\[ n_{forks} == \text{num forks needed with } k \text{ more guests} \]

Useful in order to conclude that you’re adding guests to the table correctly
Task 2: Summing the Squares

**Task:** sum the squares of $k$ from $k = 2..5$

```
total = 0
k = 2
while k <= 5:
    total = total + k*k
    k = k + 1

POST: total is sum of 2...5
```

Loop processes range 2..5
What is the invariant?

**Task:** sum the squares of $k$ from $k = 2..5$

What is true at the end of each loop iteration?

```plaintext
total = 0;
k = 2
while k <= 5:
    total = total + k*k
    k = k + 1
```

POST: total is sum of 2...5

Total should have added in the square of $(k-1)$

$$\text{total} = \text{sum of squares of } 2..k-1$$
Summing Squares: Invariant Check #1

total = 0
k = 2

# INV: total = sum of squares of 2..k-1

while k <= 5:
    total = total + k*k
    k = k + 1

# POST: total = sum of squares of 2..5

Integers that have been processed: none
Range 2..k-1: 2..1 (empty)
Summing Squares: Invariant Check #2

total = 0

# INV: total = sum of squares of 2..k-1
while k <= 5:
    total = total + k*k
    k = k + 1
# POST: total = sum of squares of 2..5

Integers that have been processed: 2
Range 2..k-1: 2..2

after 1 iteration:

<table>
<thead>
<tr>
<th>total</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Integers that have been processed: 2

# invariant goes here

k = 2

while k <= 5:
    total = total + k*k
    k = k + 1

# invariant goes here

k = 2

# POST: total = sum of squares of 2..5

14
Summing Squares: Invariant Check #3

total = 0
k = 2

# INV: total = sum of squares of 2..k-1

while k <= 5:
    total = total + k*k
    k = k +1

# POST: total = sum of squares of 2..5

Integers that have been processed: 2, 3
Range 2..k-1: 2..3

After 2 iterations:

<table>
<thead>
<tr>
<th>total</th>
<th>0</th>
<th>4</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>xx</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

k = 2

# invariant goes here

k <= 5

true

total = total + k*k

k = k +1

false
Summing Squares: Invariant Check #4

total = 0

$k = 2$

# INV: total = sum of squares of 2..k-1

while $k \leq 5$:
    total = total + $k \times k$
    $k = k + 1$

# POST: total = sum of squares of 2..5

Integers that have been processed: 2, 3, 4

Range 2..k-1: 2..4

<table>
<thead>
<tr>
<th>total</th>
<th>0</th>
<th>4</th>
<th>13</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

Integers that have been processed: 2, 3, 4
Summing Squares: Invariant Check #5

\[
\text{total} = 0
\]

\[
k = 2
\]

**INV:** total = sum of squares of 2..k-1

**while** k <= 5:

\[
\text{total} = \text{total} + k^2
\]

\[
k = k + 1
\]

**POST:** total = sum of squares of 2..5

Integers that have been processed: 2, 3, 4, 5

Range 2..k-1: 2..5

<table>
<thead>
<tr>
<th>total</th>
<th>0</th>
<th>4</th>
<th>13</th>
<th>29</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

after 4 iterations:

\[
\text{total} \quad \text{4 iterations:}
\]

\[
k \quad \text{4 iterations:}
\]

Integers that have been processed: **2, 3, 4, 5**

Range 2..k-1: **2..5**
Invariant was always true just before test of loop condition. So it’s true when loop terminates.
Designing Integer \texttt{while}-loops

1. Recognize that a range of integers \( b..c \) has to be processed
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process \( k \))

# Process \( b..c \)

\texttt{Initialize variables (if necessary) to make invariant true}

\texttt{while} \quad k \leq c:
  # Process \( k \)
  k = k + 1

# Postcondition: range \( b..c \) has been processed
1. Recognize that a range of integers b..c has to be processed

Approach:

Will need to look at characters 0...len(s)-1

Will need to compare 2 adjacent characters in s.
Beyond that… not sure yet!
Task 3: count num adjacent equal pairs

2. Write the command and equivalent postcondition

3. Write the basic part of the while-loop (see postcondition)

```python
# set n_pair to number of adjacent equal pairs in s

while k < len(s):  # we’re deciding k is the second in the current pair
    # otherwise, we’d set the condition to k < len(s) - 1
    k = k + 1

# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]
```
Q: What range of s has been processed?

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop

```python
# set n_pair to number of adjacent equal pairs in s

while k < len(s):
    # POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]
    k = k + 1
```

What range of s has been processed?

A: 0..k
B: 1..k
C: 0..k–1
D: 1..k–1
E: I don’t know
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop

```python
# set n_pair to number of adjacent equal pairs in s
while k < len(s):
    k = k + 1
# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]
```

A: What range of s has been processed?

A: 0..k
B: 1..k
C: 0..k–1 CORRECT
D: 1..k–1
E: I don’t know
Q: What is the loop invariant?

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant

# set n_pair to number of adjacent equal pairs in s

# INVARIANT: while k < len(s):

    k = k + 1

# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]

A: n_pair = num adj. equal pairs in s[1..k]
B: n_pair = num adj. equal pairs in s[0..k]
C: n_pair = num adj. equal pairs in s[1..k-1]
D: n_pair = num adj. equal pairs in s[0..k-1]
E: I don’t know
A: What is the loop invariant?

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant

# set n_pair to number of adjacent equal pairs in s

# INVARINT: while k < len(s):
      k = k + 1
# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]

A: n_pair = num adj. equal pairs in s[1..k]
B: n_pair = num adj. equal pairs in s[0..k]
C: n_pair = num adj. equal pairs in s[1..k-1] CORRECT
D: n_pair = num adj. equal pairs in s[0..k-1]
E: I don’t know
Q: how to initialize k?

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization

# set n_pair to # adjacent equal pairs in s
\[ n\_pair = 0; \ k = \_ \]

# INV: n_pair = # adjacent equal pairs in s[0..k-1]
\textbf{while} \ k < \text{len}(s): \]

\[ k = k + 1 \]

# POST: n_pair = # adjacent equal pairs in s[0..\text{len}(s) - 1]

A: k = 0
B: k = 1
C: k = -1
D: I don’t know
A: how to initialize k?

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization

# set n_pair to # adjacent equal pairs in s
n_pair = 0; k = ?

# INV: n_pair = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    k = k + 1

# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]

A: k = 0
B: k = 1
C: k = -1
D: I don’t know
Q: What do we compare to “process k”?

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process k)

```python
# set n_pair to # adjacent equal pairs in s
n_pair = 0; k = 1

# INV: n_pair = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    k = k + 1

# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]
```

A: s[k] and s[k+1]  
B: s[k-1] and s[k]  
C: s[k-1] and s[k+1]  
D: s[k] and s[n]  
E: I don’t know
A: What do we compare to “process k”? 

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process k)

# set n_pair to # adjacent equal pairs in s
n_pair = 0; k = 1

# INV: n_pair = # adjacent equal pairs in s[0..k-1]

while k < len(s):
    k = k + 1

# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]

A: s[k] and s[k+1]  \textbf{CORRECT}
B: s[k-1] and s[k]
C: s[k-1] and s[k+1]
D: s[k] and s[n]
E: I don’t know
Task 3: count num adjacent equal pairs

2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process k)

# set n_pair to # adjacent equal pairs in s
n_pair = 0; k = 1

# INV: n_pair = # adjacent equal pairs in s[0..k-1]

while k < len(s):
    if (s[k-1] == s[k]):
        n_pair += 1
    k = k + 1

# POST: n_pair = # adjacent equal pairs in s[0..len(s)-1]
count num adjacent equal pairs: v1

Approach #1: compare s[k] to the character in front of it (s[k-1])

# set n_pair to # adjacent equal pairs in s
n_pair = 0
k = 1

# INV: n_pair = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    if (s[k-1] == s[k]):
        n_pair += 1
    k = k + 1

postcondition: n_pair = # adjacent equal pairs in s[0..len(s)-1]
count num adjacent equal pairs: v2

Approach #2: compare $s[k]$ to the character in after it ($s[k+1]$)

# set n_pair to # adjacent equal pairs in s

precondition: s is a string

$n\_pair = 0$

$k = 0$

# INV: $n\_pair = # adjacent equal pairs in s[0..k]$

while $k < \text{len}(s) - 1$:

  if ($s[k] == s[k+1]$):
    $n\_pair += 1$

  $k = k + 1$

postcondition: $n\_pair = # adjacent equal pairs in s[0..\text{len}(s)-1]$
Task 4: find largest element in list

1. Recognize that a range of integers b..c has to be processed
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process k)

# set big to largest element in int_list, a list of int, len(int_list) >= 1
Initialize variables (if necessary) to make invariant true

# Invariant: big is largest int in int_list[0...(k-1)]

while k < len(int_list):
    # Process k
    k = k + 1

# Postcondition: big = largest int in int_list[0..len(int_list)-1]
Q: What is the initialization? (careful!)

1. Recognize that a range of integers b..c has to be processed
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization

# set big to largest element in int_list, a list

# Invariant: big is largest int in int_list[0...k-1]

while  k < len(int_list):
    k = k + 1

# Postcondition: big = largest int in int_list[0..len(int_list)-1]
A: What is the initialization? (careful!)

1. Recognize that a range of integers b..c has to be processed
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization

# set big to largest element in int_list, a list

# Invariant: big is largest int in int_list[0...k-1]

An empty set of characters or integers has no maximum.

Be sure that 0..k–1 is not empty. You must start with k = 1.

# Postcondition: big = largest int in int_list[0..len(int_list)–1]
Task 4: find largest element in list

1. Recognize that a range of integers b..c has to be processed
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process k)

# set big to largest element in int_list, a list of int, len(int_list) >= 1
k = 1; big = int_list[0]

# Invariant: big is largest int in int_list[0...k-1]
while k < len(int_list):
    big = max(big, int_list[k])
    k = k + 1

# Postcondition: big = largest int in int_list[0..len(int_list)-1]