CS1110 Final Exam Review Session 1

Spring 2017

Loop Invariants & Sequence Algorithms

On the Exam (May 18th 9am, Barton):

- String, list, and dictionary processing (for loops)
- Testing and debugging
- Objects, classes (+ subclasses and inheritance)
- Name resolution
- Frames and the call stack
- Recursion
- While loops & invariants
- Sequence and sorting algorithms

On the Exam (May 18th 9am, Barton):

- String, list, and dictionary processing (for loops)
- Testing and debugging
- Objects, classes (+ subclasses and inheritance)
- Name resolution
- Frames and the call stack
- Recursion
- While loops & invariants
- Sequence and sorting algorithms

Notes on Range Notation

- Pay attention to range:

  a..b or a+1..b or a…b-1 or …

- This affects the loop condition!
  - Range a..b-1, has condition k < b
  - Range a..b, has condition k < b +1
- Note that a..a-1 denotes an empty range
  - There are no values in it
- Note: b[a:b] in python represents b[a..b-1]

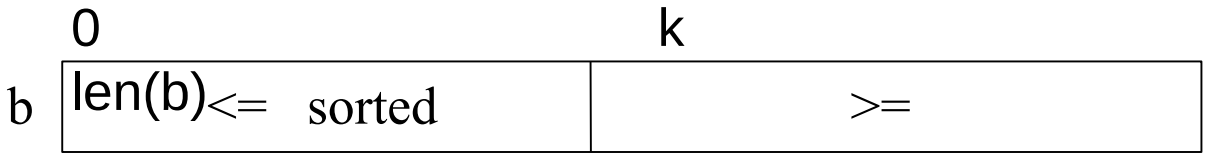
DOs and DON'Ts #1

- DO use variables given in the invariant.
- DON'T use other variables.

```
    # invariant: b[h..] contains the sum of c[h..] and
d[k..],
    # except that the carry into position k-1 is in 'carry'
    while _____ :
        # Okay to use b, c, d, h, k, and carry
        # Anything else should be 'local' to whi
```

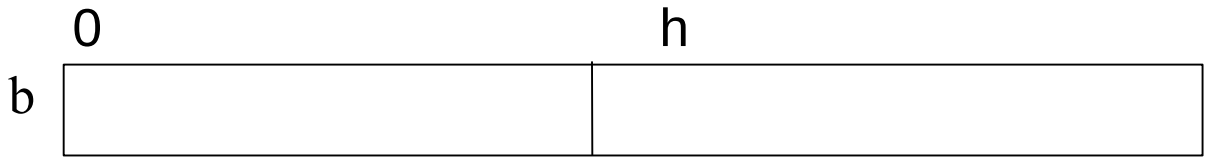
Will cost you points on the exam!

Horizontal Notation for Sequences



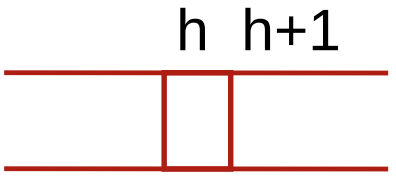
Example of an assertion about an sequence b. It asserts that:

1. b[0..k–1] is sorted (i.e. its values are in ascending order)
2. Everything in b[0..k–1] is ≤ everything in b[k..len(b)–1]



Given index h of the first element of a segment and index k of the element that follows that segment, the number of values in the segment is k – h.

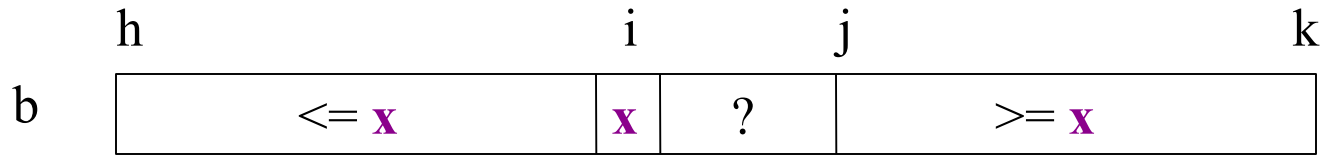
b[h .. k – 1] has k – h elements in it.



(h+1) – h = 1

DOs and DON'Ts #2

- DON'T put variables directly above vertical line.



  - Where is j?
  - Is it unknown or >= x?
  - Lines are BETWEEN elements and hence have no index associated with them

What we'll ask you to do on the exam

- Write body of a loop to satisfy a given invariant.
- Given an invariant with code, identify all errors.
- Given an example, rewrite it with new invariant.

- You will NOT be responsible for coming up with your own invariants during this timed exam (e.g. as on A5 in generate food grid).

Why Invariants??

- Suppose you were trying to sum up all of the elements of a list b:
  - Even if we were constrained to only use while loops there are many possible solutions…

Why Invariants??

```
i = 0
tot = 0
# inv #1: total is sum of b[0..i-1]
while i < len(b):
    tot += b[i]
    i += 1
return tot
```

Why Invariants??

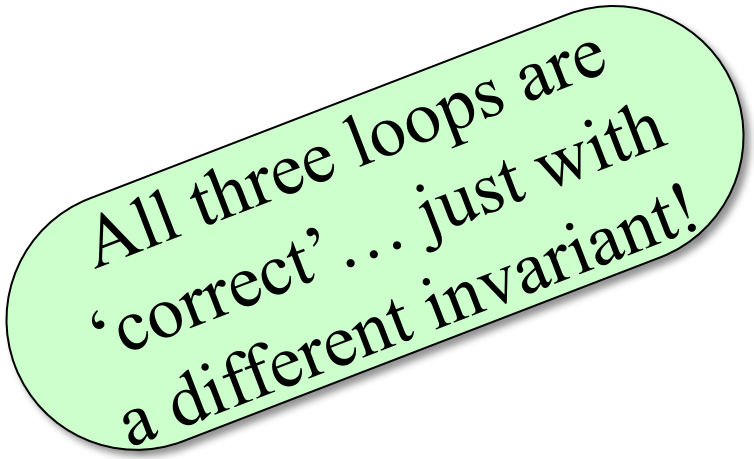
```
i = 0
tot = 0
# inv #1: total is sum of b[0..i-1]
while i < len(b):
    tot += b[i]
    i += 1
return tot

i = -1
tot = 0
# inv #2: total is sum of b[0..i]
while i < len(b) -1:
    tot += b[i+1]
    i += 1
return tot
```

Why Invariants??

```
i = 0
tot = 0
# inv #1: total is sum of b[0..i-1]
while i < len(b):
    tot += b[i]
    i += 1
return tot
```

```
i = len(b)
tot = 0
# inv #3: total is sum of b[i..len(b)-1]
while i > 0:
    tot += b[i-1]
    i -= 1
return tot
```

```
i = -1
tot = 0
# inv #2: total is sum of b[0..i]
while i < len(b) -1:
    tot += b[i+1]
    i += 1
return tot
```



Why Invariants??

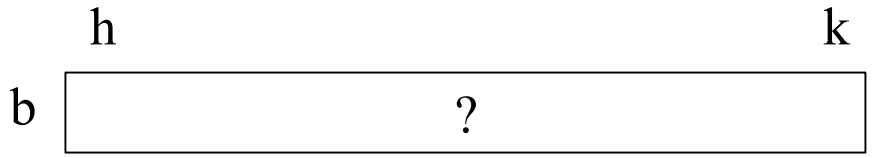
- Suppose you were trying to sum up all of the elements of a list b:
  - Even if we were constrained to only use while loops there are many possible solutions…
- An invariant is a theoretical tool to help you understand that a loop is working correctly
- Invariants also help you design complicated loops (Including Assignment 5!) by telling the programmer the state of what has been done.

Definitions

- What's an Invariant?
  - An assertion that is supposed to "always" be true
  - If temporarily invalidated, must make it true again
- Loop Invariant – An assertion that should be true before and after every iteration of the loop
  - E.g. tot is the sum of elements b[0..i-1].
  - References the loop variables (tot and i loop vars)
- Class Invariant – assertion on value of attribute
  - E.g. [int, 0…maxValue]

Algorithm Inputs

- We may specify that the list in the algorithm is
  - b[0..len(b)-1] or
  - a segment b[h..k] or
  - a segment b[m..n-1]
- **Work with whatever is given!**



- How many elements are in this array?
  - b[h..k] has k+1–h elements

Steps to Tackle many invariant problems

1. Identify Range: [0..len(b) -1], [h..k], other?
2. Identify Loop variables and direction(s) of processing
3. Draw Box Diagram for the Invariant
4. "Push" invariant boundary lines to Precondition state to find the initialization conditions
5. "Push" invariant boundary lines to Postcondition state to find the termination condition
6. Flip the termination condition to its opposite to get the while loop condition (Use strict inequalities: <, >, or !=; not <= or >=)
7. Identify the next element to process (i? i+1? i-1?)
8. Write inside of loop to process next element and make progress

A Simple Example

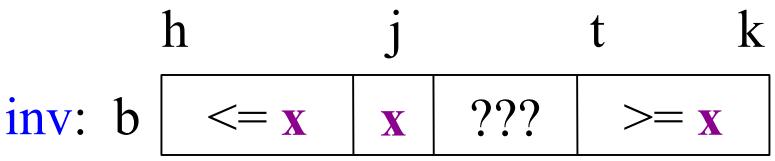
```
def sum_htok(b, h, k):
    """Sum all the elements in a list from position h to (and including)
    position k """
    i =
    tot =
    # inv: tot is the sum of b[i+1..k]
    while        :


    # post: tot is the sum of b[h..k]
    return tot
```

A Simple Example

```
def sum_htok(b, h, k):
    """Sum all the elements in a list from position h to (and including)
    position k """
    i = k
    tot = 0
    # inv: tot is the sum of b[i+1..k]
    while i > h -1 :
        tot += b[i]
        i = i - 1
    # post: tot is the sum of b[h..k]
    return tot
```

Partition Example

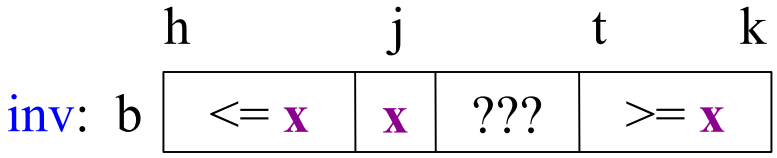
```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```



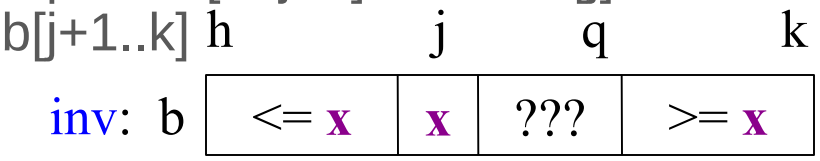

```
# Make invariant true at start
j =
q =
# inv: b[h..j–1] <= x = b[j] <= b[q+1..k]
while      :
    
    
    
    
    
    
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

Partition Example

```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

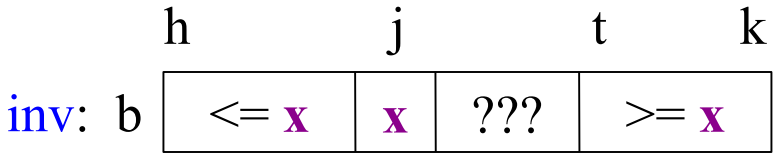



```
# Make invariant true at start
j =
q =
# inv: b[h..j–1] <= x = b[j] <= b[q+1..k]
while      :
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

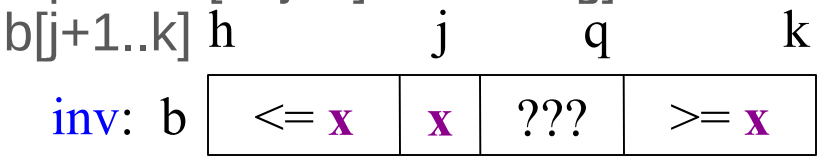


Partition Example

```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

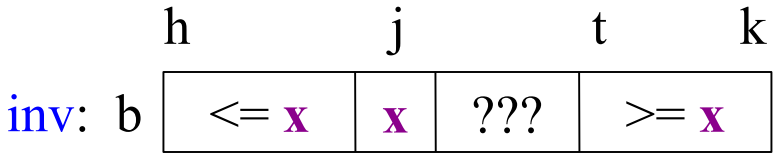



```
# Make invariant true at start
j = h
q = k
# inv: b[h..j–1] <= x = b[j] <= b[q+1..k]
while j < q:
    
    
    
    
    
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

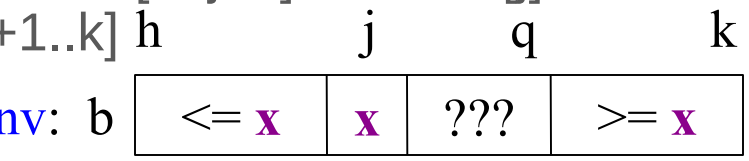


Partition Example

```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

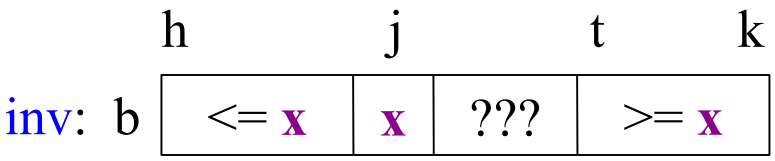



```
# Make invariant true at start
j = h
q = k
# inv: b[h..j–1] <= x = b[j] <= b[q+1..k]
while j < q:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[q]
        q=q–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```



Partition Example

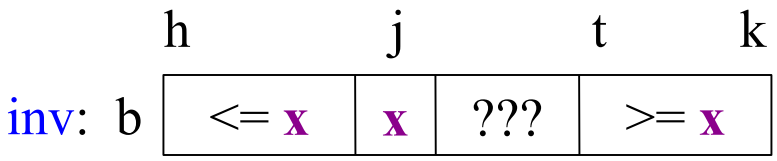
```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```



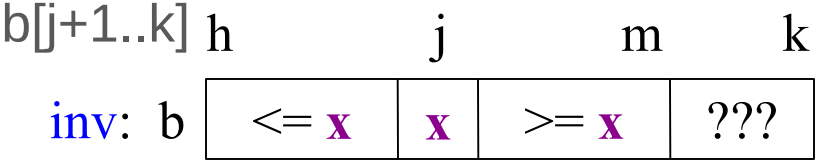

```
# Make invariant true at start
  j =
m =
# inv: b[h..j–1] <= x = b[j] <= b[j+1..m]
while      :
    
    
    
    
    
    
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

Partition Example

```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

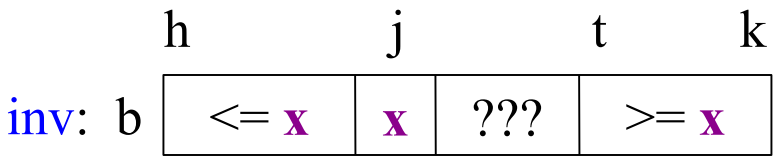



```
# Make invariant true at start
  j = h
m = h
# inv: b[h..j–1] <= x = b[j] <= b[j+1..m]
while      :
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

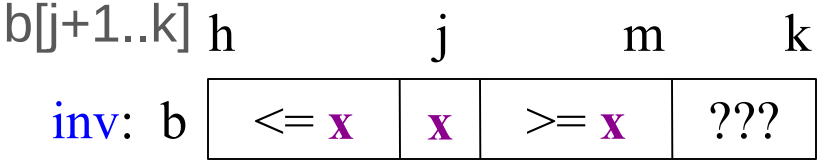


Partition Example

```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```

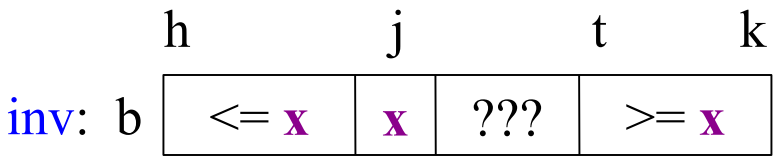



```
# Make invariant true at start
  j = h
m = h
# inv: b[h..j–1] <= x = b[j] <= b[j+1..m]
while m < k:
```

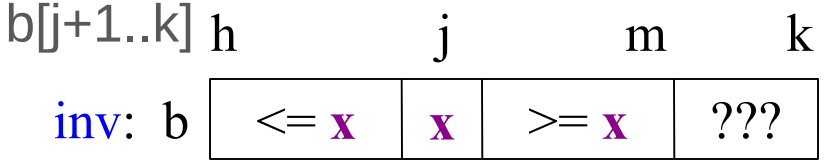


Partition Example

```
# Make invariant true at start
j = h
t = k+1
# inv: b[h..j–1] <= x = b[j] <= b[t..k]
while j < t–1:
    if b[j+1] <= b[j]:
        swap b[j] and b[j+1]
        j = j+1
    else:
        swap b[j+1] and b[t-1]
        t=t–1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```




```
# Make invariant true at start
  j = h
m = h
# inv: b[h..j–1] <= x = b[j] <= b[j+1..m]
while m < k:
    if b[m+1] <= b[j]:
        swap b[j] and b[m+1]
        swap b[j+1] and b[m+1]
        m = m+1; j=j+1
    else:
        m = m+1
# post: b[h..j–1] <= x = b[j] <= b[j+1..k]
```



DNF – ID broken invariants

```
def dnf (b, h, k):
    """Returns: partition points as a tuple ( i,j )"""
    t = h; i = k+1, j = k;
    # inv: b[h..t] < 0, b[t+1..i-1] ?, b[i..j] = 0,  b[j+1..k] > 0
    while t < i:
        if b[i-1] < 0:
            swap(b,i-1,t)
            t = t+1
        elif b[i -1] == 0:
            i = i-1
        else:
            swap(b,i-1,j)
            i = i-1; j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i , j)
```

DNF – ID broken invariants

```
def dnf (b, h, k):
    """Returns: partition points as a tuple ( i,j )"""
    t = h h-1; i = k+1, j = k;
    # inv: b[h..t] < 0, b[t+1..i-1] ?, b[i..j] = 0,  b[j+1..k] > 0
    while t < i t+1 < i:
        if b[i-1] < 0:
            swap(b,i-1,t t+1)
            t = t+1
        elif b[i -1] == 0:
            i = i-1
        else:
            swap(b,i-1,j)
            i = i-1; j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i , j)
```

Other Searching & Sorting

- Mergesort / Quicksort: Partition on each side of the list and then merge back together
- Selection sort: find minimum value in part of the list, swap it with next element to check
- Linear search: check each next element, if you found what you're looking for return.
- Binary search: on a sorted list, look at middle element, and then look at the side where the element might fall if middle not what you want

Good Luck!