Lecture 23

Loop Invariants

Announcements for This Lecture

Assignments

- A6 due on **Wednesday**
  - Dataset should be done
  - Get on track this weekend
  - **Next Week**: ClusterGroup
- A7 will be last assignment
  - Will talk about next week
  - Posted on Tuesday
- There is lab next week
  - **No lab** week of Turkey Day

Prelim 2

- Thursday, 7:30-9pm
  - **A – K** (Uris G01)
  - **L – O** (Phillips 101)
  - **P – W** (Ives 305)
  - **X – Z** (Ives 105)
  - Conflicts received e-mail
- Graded by the weekend
  - Returned early next week
  - Regrade policy as before

Recall Lab 9

```
Welcome to CS 1110 Blackjack.
Rules: Face cards are 10 points. Aces are 11 points.
      All other cards are at face value.

Your hand:
2 of Spades
10 of Clubs

Dealer's hand:
5 of Clubs

Type h for new card, s to stop:
```

Play until player **stops** or **busts**

Recall Lab 9

Welcome to CS 1110 Blackjack.

Rules: Face cards are 10 points. Aces are 11 points.

All other cards are at face value.

Your hand:

2 of Spades

10 of Clubs

How do we design a complex while-loop like this one?

Dealer's hand:

5 of Clubs

Play until player **stops** or **busts**

Type h for new card, s to stop:

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

Assertions versus Asserts

- Assertions **prevent bugs**
  - Help you keep track of what you are doing
- Also **track down bugs**
  - Make it easier to check belief/code mismatches
- The assert statement is a (type of) assertion
  - One you are **enforcing**
  - Cannot always convert a comment to an assert

 # x is the sum of 1..n

Comment form of the assertion.

The root of all bugs!

x	?	n	1
x	?	n	3
x	?	n	0

Preconditions & Postconditions

precondition

```
# x  = sum of 1..n-1
x = x + n
n = n + 1
# x =  sum of 1..n-1
```

postcondition

n

1 2 3 4 5 6 7 8

x contains the sum of these (6)

n

1 2 3 4 5 6 7 8

x contains the sum of these (10)

- **Precondition:** assertion placed before a segment
- **Postcondition:** assertion placed after a segment

Relationship Between Two

If precondition is true, then postcondition will be true

Solving a Problem

precondition

```
# x  = sum of 1..n

n = n + 1
# x =  sum of 1..n
```

postcondition

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

A: x = x + 1
B: x = x + n
C: x = x + n+1
D: None of the above
E: I don't know

Solving a Problem

precondition

```
# x  = sum of 1..n

n = n + 1
# x =  sum of 1..n
```

postcondition

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

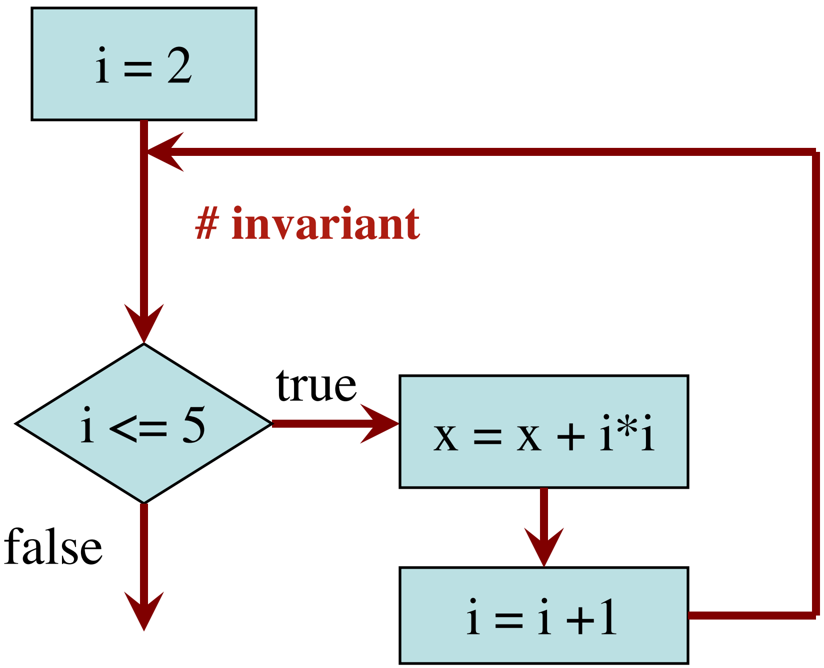
A: x = x + 1
B: x = x + n
C: x = x + n+1
D: None of the above
E: I don't know

Remember the new value of n

Invariants: Assertions That Do Not Change

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

```
x = 0; i = 2
while i <= 5:
    x = x + i*i
    i = i +1
# x = sum of squares of 2..5
```



Invariant:

x = sum of squares of 2..i-1

in terms of the range of integers that have been processed so far

The loop processes the range 2..5

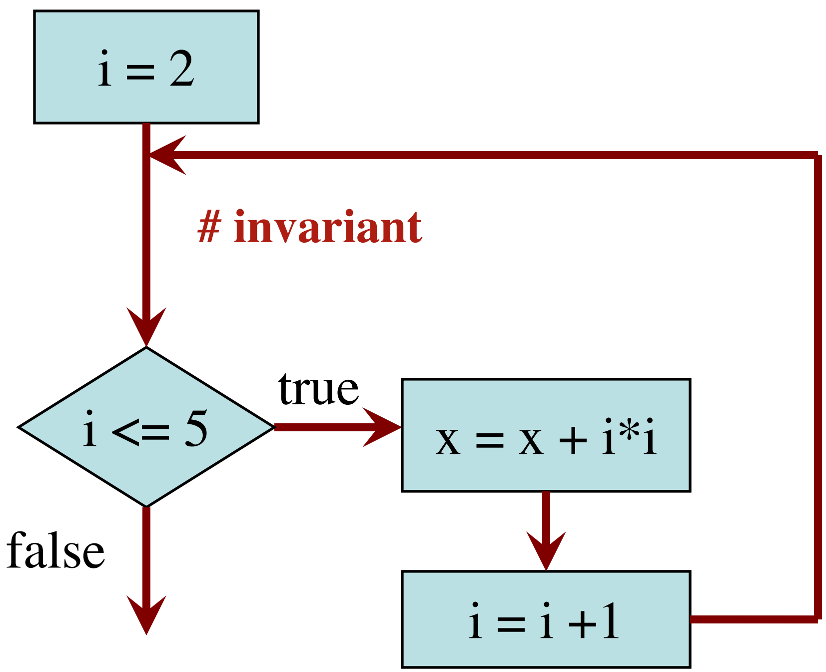
Invariants: Assertions That Do Not Change

```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

Integers that have been processed:

Range 2..i-1:

x	0
i	?



The loop processes the range 2..5

Invariants: Assertions That Do Not Change

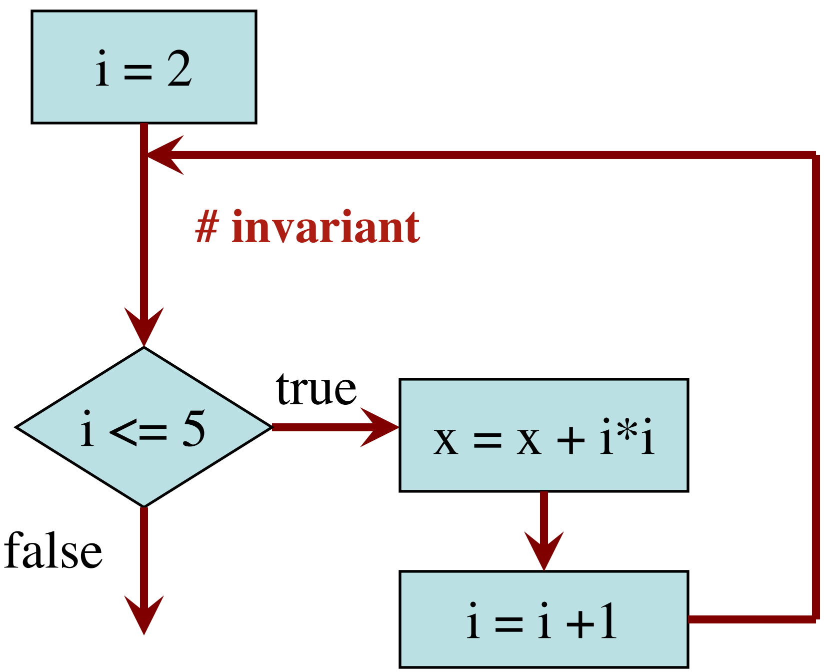
```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

Integers that have been processed:

Range 2..i-1: 2..1 (empty)

x 0

i 2



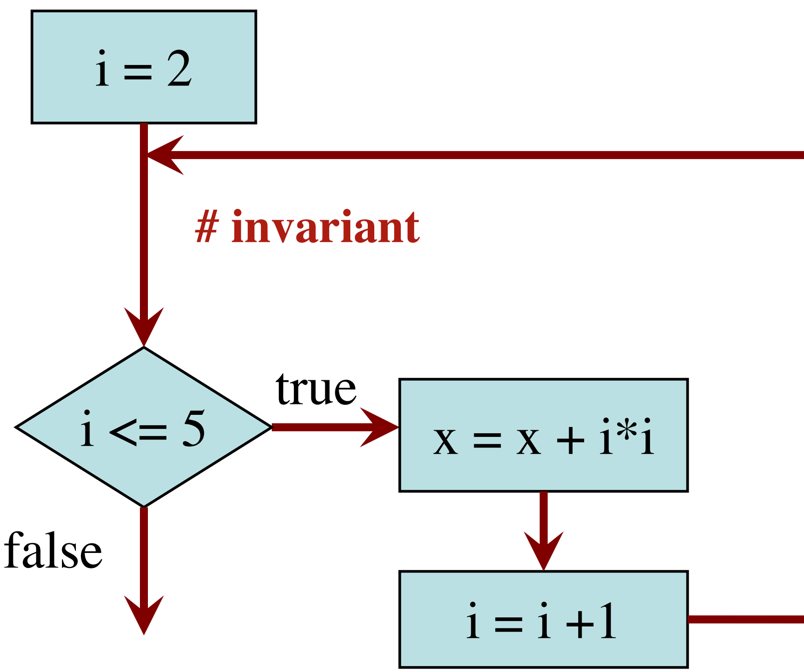
The loop processes the range 2..5

Invariants: Assertions That Do Not Change

```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

x	~~0~~ 4
i	~~1~~ ~~2~~ 3

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



The loop processes the range 2..5

Invariants: Assertions That Do Not Change

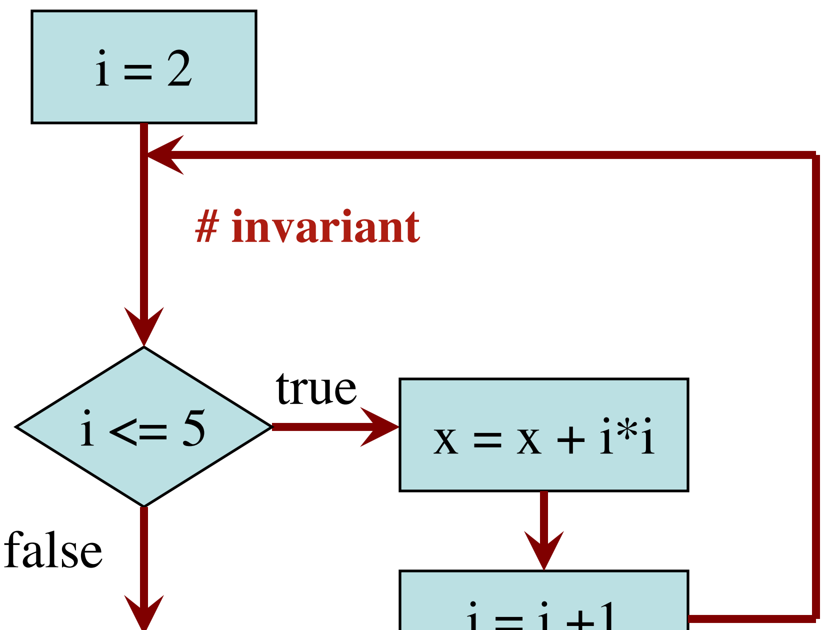
```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

Integers that have been processed: 2 , 3

Range 2..i-1: 2..3

x 13

i 4



The loop processes the range 2..5

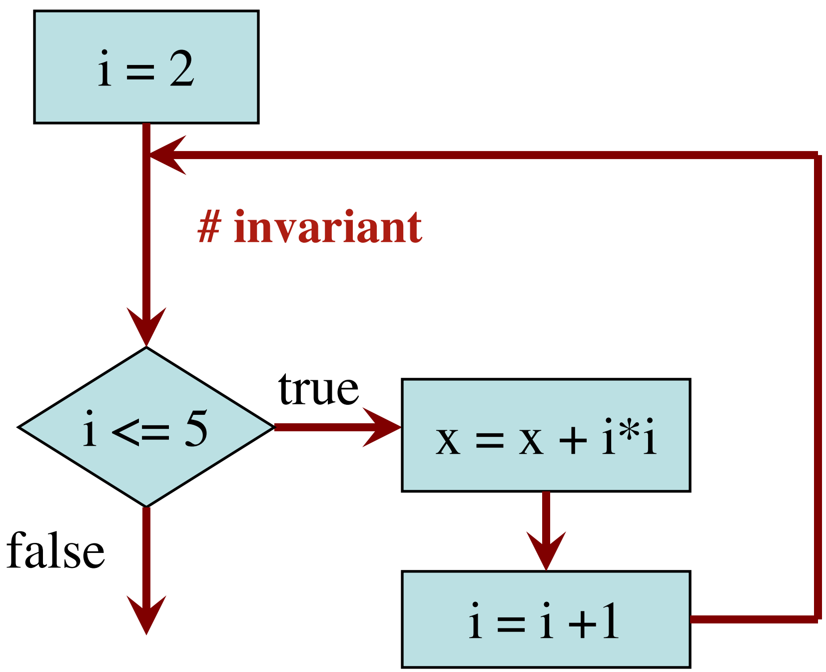
Invariants: Assertions That Do Not Change

```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

Integers that have been processed: 2, 3, 4

Range 2..i-1: 2..4

x	~~0~~ ~~4~~ ~~13~~ 29
i	~~?~~ ~~2~~ ~~3~~ ~~4~~ 5



The loop processes the range 2..5

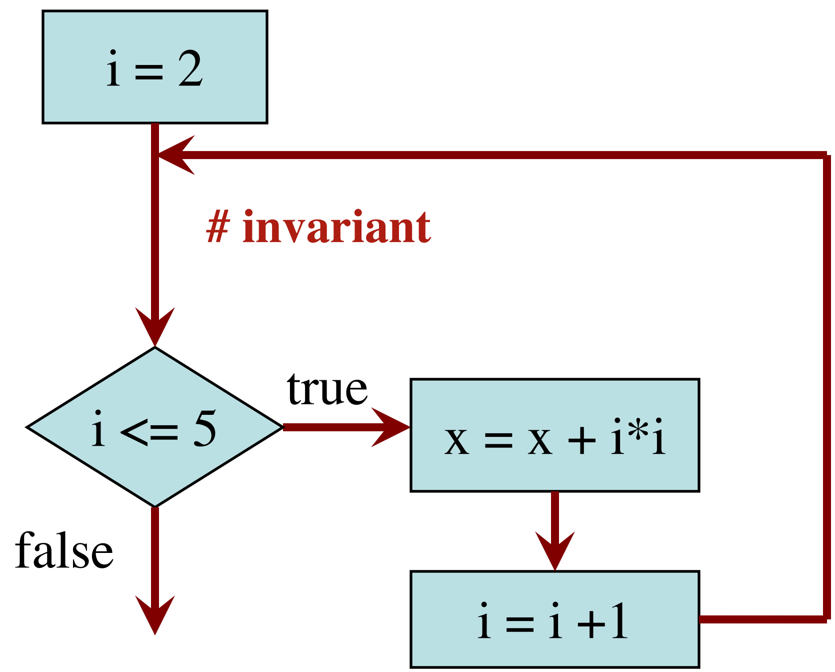
Invariants: Assertions That Do Not Change

```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

x	~~0~~ ~~4~~ ~~13~~ ~~29~~ 54
i	~~?~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6

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

Range 2..i-1: 2..5



The loop processes the range 2..5

Invariants: Assertions That Do Not Change

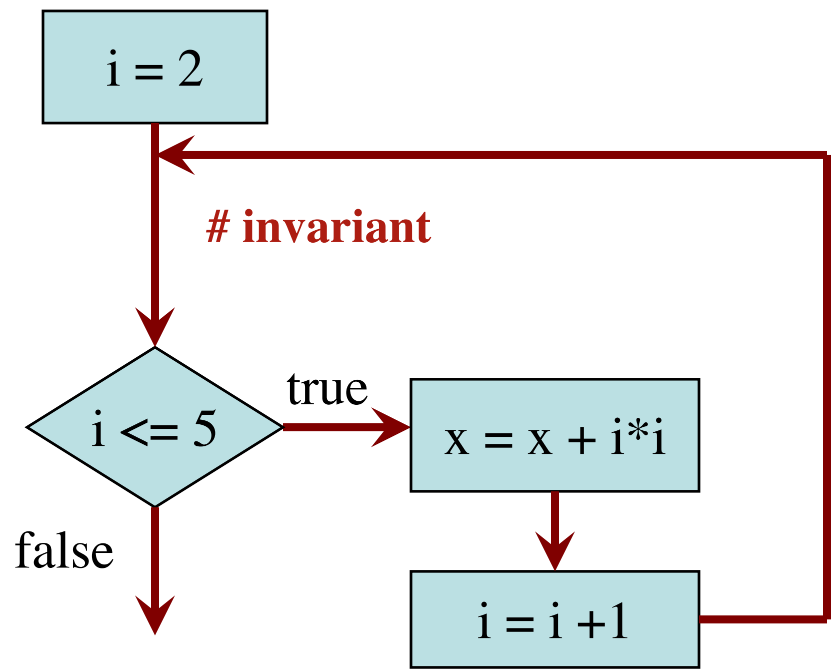
```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

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

Range 2..i-1: 2..5

Invariant was always true just before test of loop condition. So it's true when loop terminates

x	~~0~~ ~~4~~ ~~13~~ ~~29~~ 54
i	~~?~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6



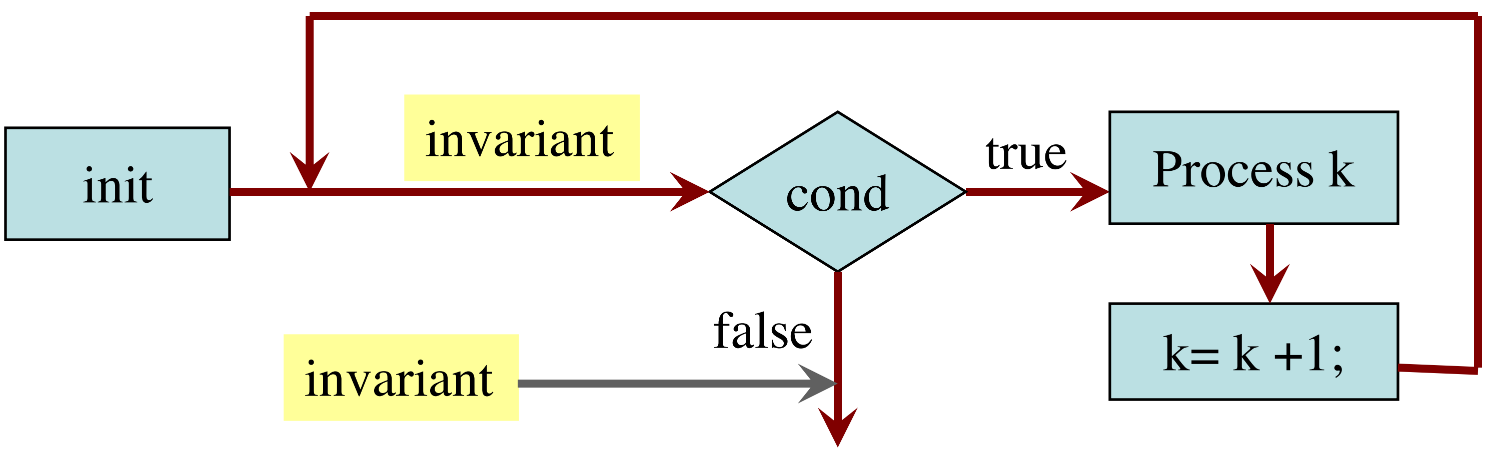
The loop processes the range 2..5

Designing Integer **while**-loops

```
# Process integers in a..b
# inv: integers in a..k-1 have been processed
k = a
while k <= b:
    process integer k
    k = k + 1
# post: integers in a..b have been processed
```

Command to do something

Equivalent postcondition



Designing Integer `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 repetend (process k)

Designing Integer 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 repetend (process k)

```
# Process b..c




# Postcondition: range b..c has been processed
```

Designing Integer 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 repetend (process k)

```
# Process b..c



while k <= c:

    k = k + 1
# Postcondition: range b..c has been processed
```

Designing Integer **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 repetend (process k)

```
# Process b..c


# Invariant: range b..k-1 has been processed
while k <= c:

    k = k + 1
# Postcondition: range b..c has been processed
```

Designing Integer 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 repetend (process k)

```
# Process b..c
Initialize variables (if necessary) to make invariant true
# Invariant: range b..k-1 has been processed
while k <= c:
    # Process k
    k = k + 1
# Postcondition: range b..c has been processed
```

Finding an Invariant

Command to do something

```
# Make b True if n is prime, False otherwise
```

```
# b is True if no int in 2..n-1 divides n, False otherwise
```

Equivalent postcondition

What is the invariant?

Finding an Invariant

Command to do something

```
# Make b True if n is prime, False otherwise



while k < n:
    # Process k;


    k = k +1
# b is True if no int in 2..n-1 divides n, False otherwise
```

Equivalent postcondition

What is the invariant?

Finding an Invariant

Command to do something

```
# Make b True if n is prime, False otherwise



# invariant: b is True if no int in 2..k-1 divides n, False otherwise
while k < n:
    # Process k;



    k = k +1
# b is True if no int in 2..n-1 divides n, False otherwise
```

Equivalent postcondition

What is the invariant?

1 2 3 … k-1 k k+1 … n

Finding an Invariant

Command to do something

```
# Make b True if n is prime, False otherwise
b = True
k = 2
# invariant: b is True if no int in 2..k-1 divides n, False otherwise
while k < n:
    # Process k;



    k = k +1
# b is True if no int in 2..n-1 divides n, False otherwise
```

Equivalent postcondition

What is the invariant?

1 2 3 … k-1 k k+1 … n

Finding an Invariant

Command to do something

```
# Make b True if n is prime, False otherwise
b = True
k = 2
# invariant: b is True if no int in 2..k-1 divides n, False otherwise
while k < n:
    # Process k;
    if n % k == 0:
        b = False
    k = k +1
# b is True if no int in 2..n-1 divides n, False otherwise
```

Equivalent postcondition

What is the invariant?

1 2 3 … k-1 k k+1 … n

Finding an Invariant

```
# set x to # adjacent equal pairs in s
```

Command to do something

for s = 'ebeee', x = 2

```
while k < len(s):
    # Process k

    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Equivalent postcondition

k: next integer to process.

Which have been processed?

A: 0..k

B: 1..k

C: 0..k–1

D: 1..k–1

E: I don't know

Finding an Invariant

```
# set x to # adjacent equal pairs in s
```

Command to do something

for s = 'ebeee', x = 2

```
while k < len(s):
    # Process k

    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Equivalent postcondition

k: next integer to process.
Which have been processed?

A: 0..k
B: 1..k
C: $\boxed{0..k–1}$
D: 1..k–1
E: I don't know

What is the invariant?

A: x = no. adj. equal pairs in s[1..k]
B: x = no. adj. equal pairs in s[0..k]
C: x = no. adj. equal pairs in s[1..k–1]
D: x = no. adj. equal pairs in s[0..k–1]
E: I don't know

Finding an Invariant

```
# set x to # adjacent equal pairs in s



# inv: x = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    # Process k

    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

for s = 'ebeee', x = 2

Equivalent postcondition

k: next integer to process.
Which have been processed?

A: 0..k
B: 1..k
C: 0..k–1
D: 1..k–1
E: I don't know

What is the invariant?

A: x = no. adj. equal pairs in s[1..k]
B: x = no. adj. equal pairs in s[0..k]
C: x = no. adj. equal pairs in s[1..k–1]
D: x = no. adj. equal pairs in s[0..k–1]
E: I don't know

Finding an Invariant

```
# set x to # adjacent equal pairs in s
x = 0

# inv: x = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    # Process k

    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

for s = 'ebeee', x = 2

Equivalent postcondition

k: next integer to process.
What is initialization for k?

A: k = 0
B: k = 1
C: k = –1
D: I don't know

Finding an Invariant

```
# set x to # adjacent equal pairs in s
x = 0
k = 1
# inv: x = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    # Process k

    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

for s = 'ebeee', x = 2

Equivalent postcondition

k: next integer to process.
What is initialization for k?

A: k = 0
B: $\boxed{k = 1}$
C: k = –1
D: I don't know

Which do we compare to "process" k?

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

Finding an Invariant

```
# set x to # adjacent equal pairs in s
x = 0
k = 1
# inv: x = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    # Process k
    x = x + 1 if (s[k-1] == s[k]) else 0
    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

for s = 'ebeee', x = 2

Equivalent postcondition

k: next integer to process.
What is initialization for k?

A: k = 0
B: $\boxed{\text{k = 1}}$
C: k = –1
D: I don't know

Which do we compare to "process" k?

A: s[k] and s[k+1]
B: $\boxed{\text{s[k-1] and s[k]}}$
C: s[k-1] and s[k+1]
D: s[k] and s[n]
E: I don't know

Reason carefully about initialization

```
# s is a string; len(s) >= 1
# Set c to largest element in s
c = ??
k = ??
# inv:
while k < len(s):
    # Process k
    k = k+1
#  c = largest char in s[0..len(s)–1]
```

Command to do something

Equivalent postcondition

1. What is the invariant?

Reason carefully about initialization

```
# s is a string; len(s) >= 1
# Set c to largest element in s
c = ??
k = ??
# inv: c is largest element in s[0..k–1]
while k < len(s):
    # Process k
    k = k+1
#  c = largest char in s[0..len(s)–1]
```

Command to do something

Equivalent postcondition

1. What is the invariant?

Reason carefully about initialization

```
# s is a string; len(s) >= 1
# Set c to largest element in s
c = ??
k = ??
# inv: c is largest element in s[0..k–1]
while k < len(s):
    # Process k
    k = k+1
#  c = largest char in s[0..len(s)–1]
```

Command to do something

Equivalent postcondition

1. What is the invariant?
2. How do we initialize c and k?

A: k = 0; c = s[0]

B: k = 1; c = s[0]

C: k = 1; c = s[1]

D: k = 0; c = s[1]

E: None of the above

Reason carefully about initialization

```
# s is a string; len(s) >= 1
# Set c to largest element in s
c = ??
k = ??
# inv:  c is largest element in s[0..k–1]
while k < len(s):
    # Process k
    k = k+1
#  c = largest char in s[0..len(s)–1]
```

Command to do something

Equivalent postcondition

1. What is the invariant?
2. How do we initialize c and k?

A: `k = 0;  c = s[0]`

B: `k = 1;  c = s[0]`

C: `k = 1;  c = s[1]`

D: `k = 0;  c = s[1]`

E: None of the above

An empty set of characters or integers has no maximum. Therefore, be sure that 0..k–1 is not empty. You must start with k = 1.