http://www.cs.cornell.edu/courses/cs1110/2019sp

Lecture 23:
Loop Invariants

[Online Reading]

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

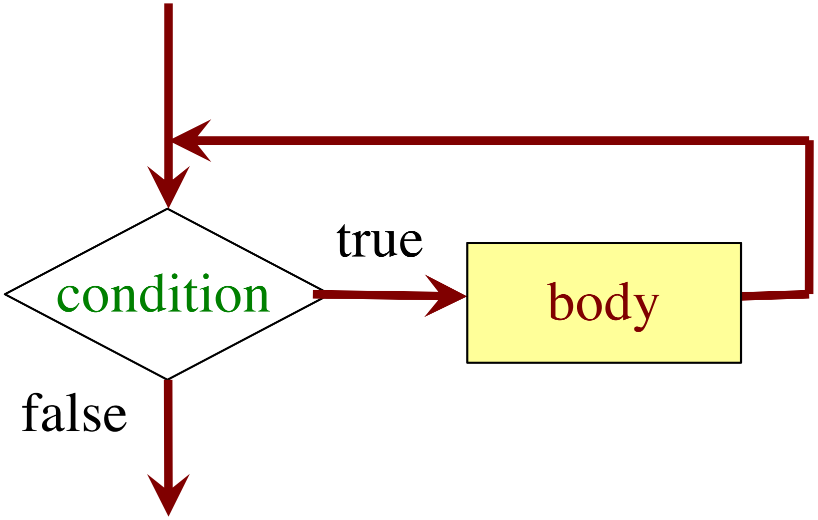
Recall: Important Terminology

- **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

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

body



true

condition

body

false

- **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

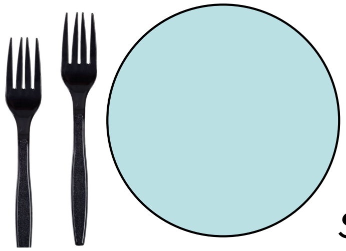
```
precondition: n_forks tells us how many forks are needed
k = 0

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

postcondition: n_forks tells us how many forks are needed
```

Relationship Between Two

If precondition is true, then postcondition will be true



one place setting

- **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***

Q1: Completing the Loop Body

precondition: n_forks tells us how many forks are needed

```
k = 0
while k < n_more_guests:

    k = k + 1
```

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

postcondition: n_forks tells us how many forks are needed

A: n_forks +=2
B: n_forks += 1
C: n_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 tells us how many forks are needed
k = 0
#INV: n_forks == num forks needed with k more guests
while k < n_more_guests:
    n_forks += 2
    k += 1
postcondition: n_forks tells us how many forks are needed
```

invariant holds before loop

invariant still holds here

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 >= 0`

True, but *doesn’t help you understand the loop*

Good:

`n_forks ==` num forks needed with `k` more guests

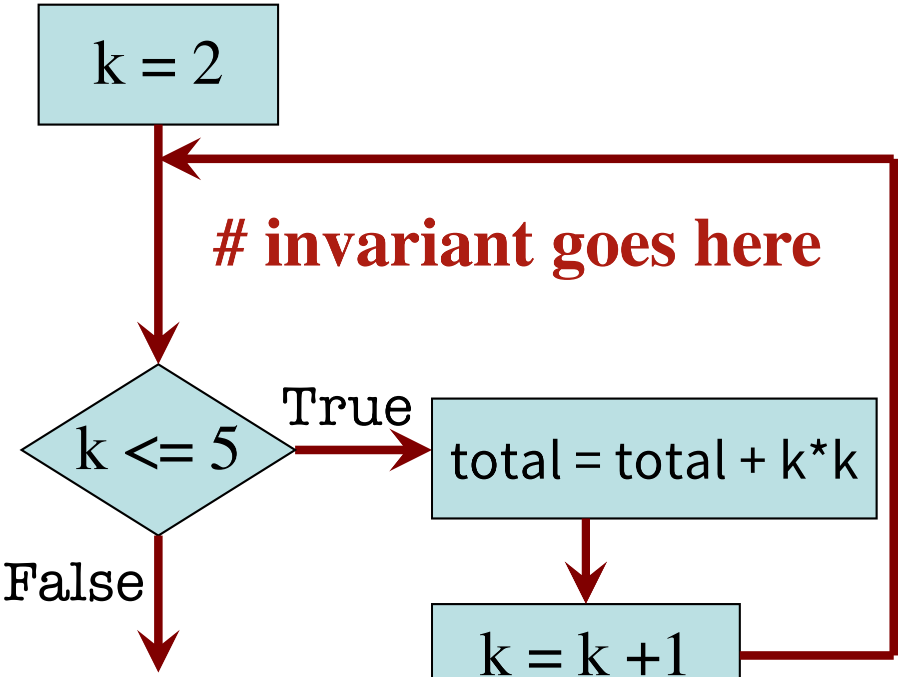
Useful in order to conclude that you’re adding guests to the table correctly

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?

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

What is true here?

POST: total is sum of 2...5

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

total = sum of squares of 2..k-1

Summing Squares: Invariant Check #1

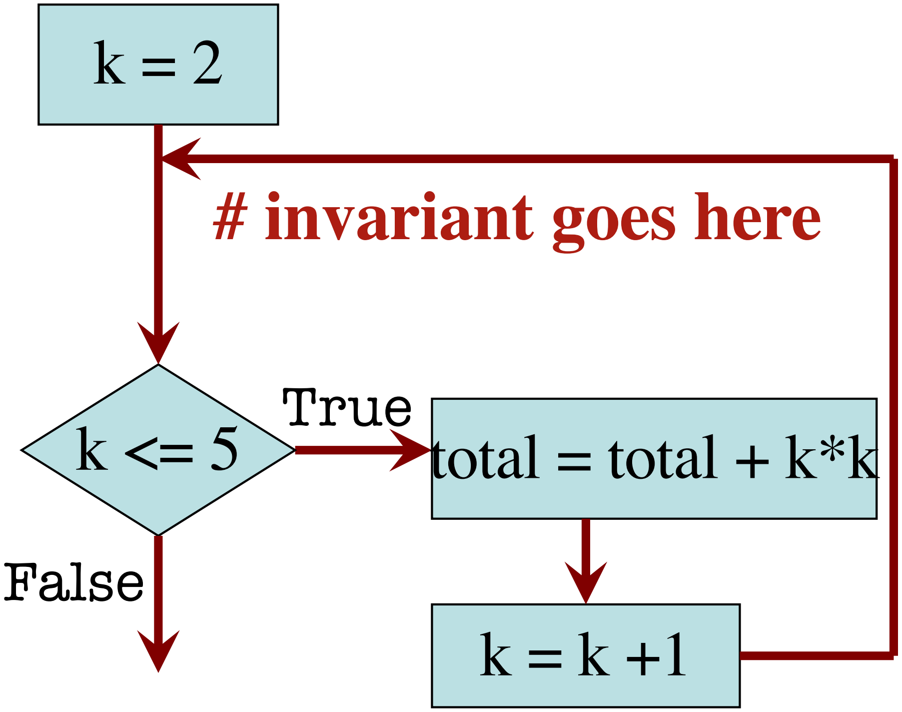
```
total = 0
 k = 2
# INV: total = sum of squares of 2..k-1
while k <= 5:
    total = total + k*k
    k = k +1
# POST: total = sum of squares of 2..5
```

0

before any iteration:

total 0

k 2



Integers that have been processed: **none**

Range 2..k-1: **2..1 (empty)**

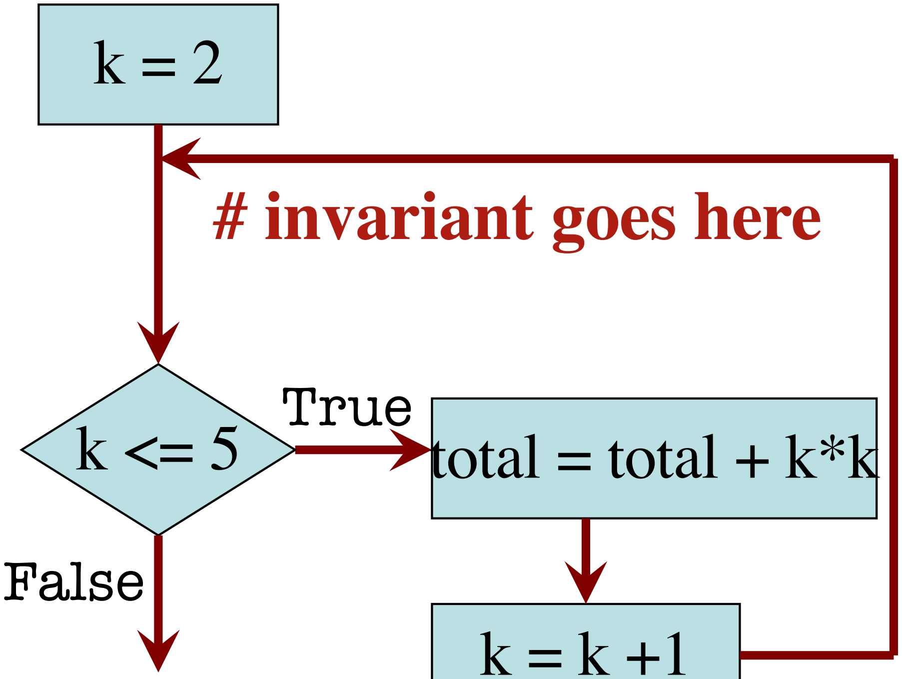
Summing Squares: Invariant Check #2

```
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
```

after 1 iteration:

total ~~0~~ 4

k ~~2~~ 3



Integers that have been processed: 2

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

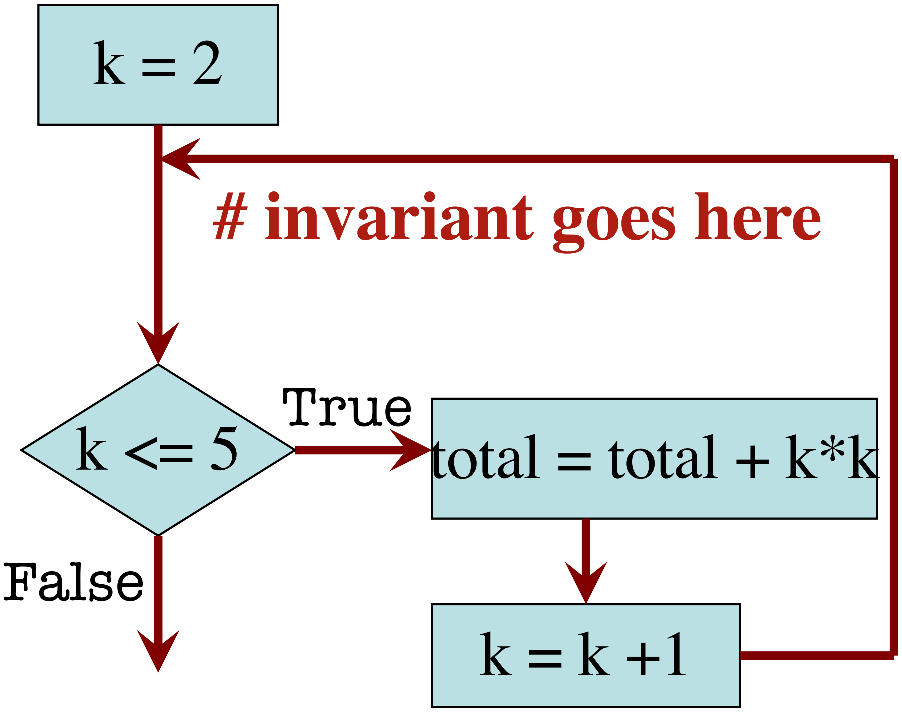
Summing Squares: Invariant Check #3

```
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
```

after 2 iterations:

total ~~0~~ ~~4~~ 13

k ~~2~~ ~~3~~ 4



k = 2

invariant goes here

k <= 5

True

total = total + k*k

k = k +1

False

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

Range 2..k-1: **2..3**

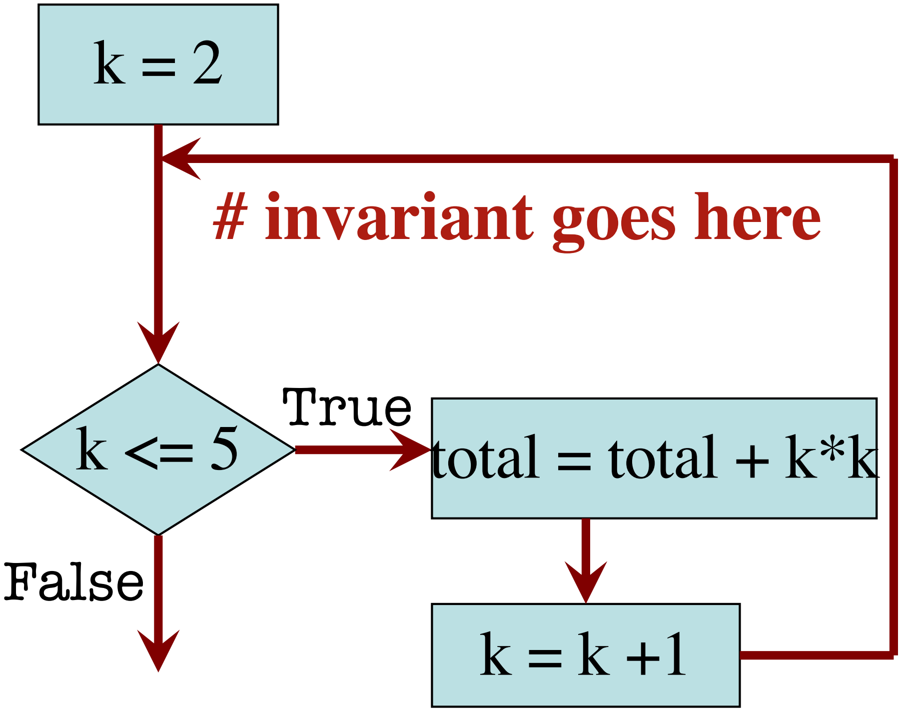
Summing Squares: Invariant Check #4

```
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
```

after 3 iterations:

total ~~0~~ ~~4~~ ~~13~~ 29

k ~~2~~ ~~3~~ ~~4~~ 5



Integers that have been processed: 2, 3, 4

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

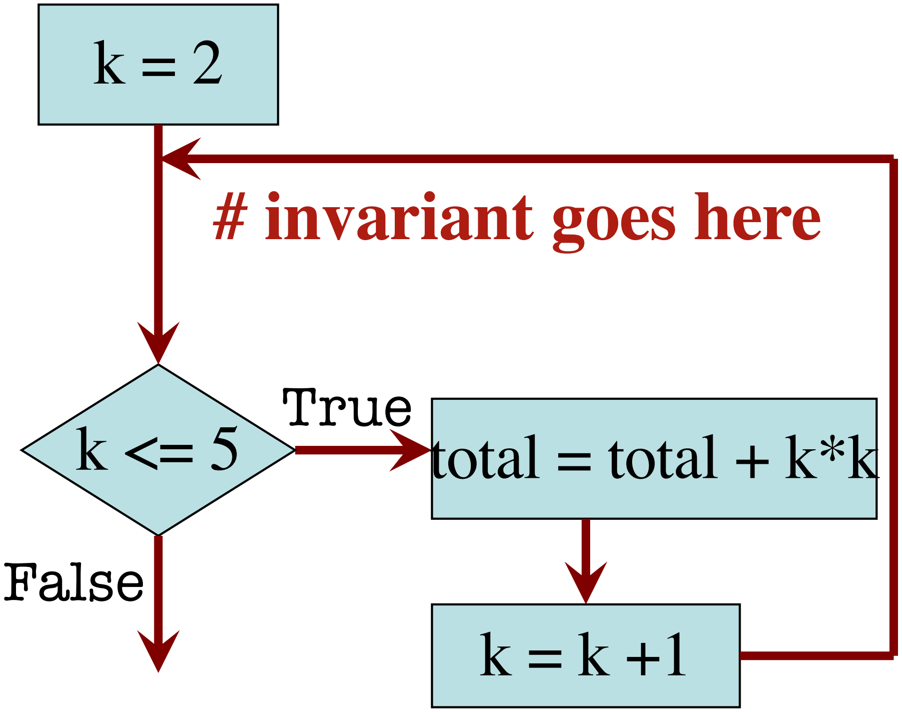
Summing Squares: Invariant Check #5

```
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
```

after 4 iterations:

total ~~0~~ ~~4~~ ~~13~~ ~~29~~ 54

k ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6



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

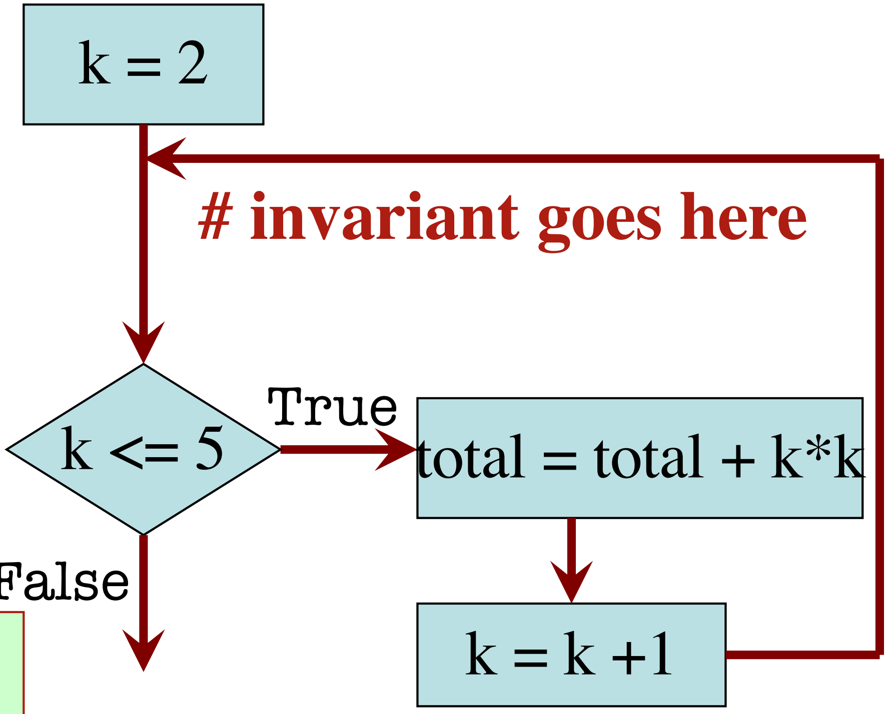
Range 2..k-1: 2..5

True Invariants → True Postcondition

```
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
```

total ~~0~~ ~~4~~ ~~13~~ ~~29~~ 54

k ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6



k = 2

 # invariant goes here

k <= 5

True

total = total + k*k

k = k +1

False

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

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 body (aka 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
```

Task 3: count num adjacent equal pairs

1. Recognize that a range of integers b..c has to be processed

`s = 'ebeee', n_pair = 2`

`s = ‘xxxxbee’, n_pair = 4`

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)

```
# 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]
```

Q2: What range of s has been processed?

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

```
# 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: 0..k

B: 1..k

C: 0..k–1

D: 1..k–1

E: I don’t know

k: next integer to process.

What range of s has been processed?

Q3: 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

Q4: 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

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

```
# 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]
```

Q5: 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]

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

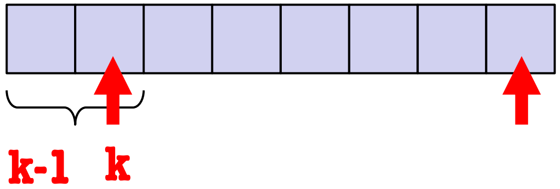
2. Write the command and equivalent postcondition
3. Write the basic part of the while-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the body (aka repetend) (# Process k)

```
# 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

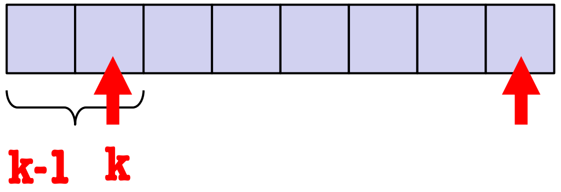
Approach #1: compare s[k] to the character in front of it (s[k-1])

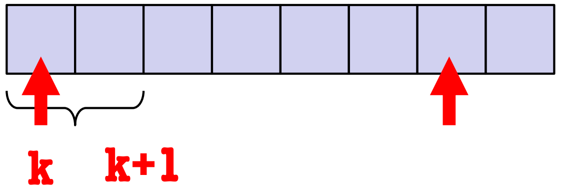


```
# set n_pair to # adjacent equal pairs in s
precondition: s is a string
n_pair = 0
k = 1
# INV: n_pair = # adjacent equal pairs in s[0..k-1]
while  k < len(s):
    if (s[k-1] == s[k]):
        n_pair += 1
    k = k + 1
postcondition: n_pair = # adjacent equal pairs in s[0..len(s)-1]
```

count num adjacent equal pairs: v1→ v2

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



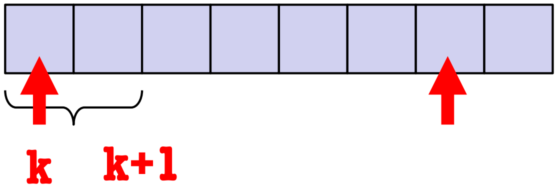


```
# set n_pair to # adjacent equal pairs in s
precondition: s is a string
n_pair = 0
~~k = 1~~  k = 0

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

Q6: 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

A: k = 0; big = int_list[0]

B: k = 1; big = int_list[0]

C: k = 1; big = int_list[1]

D: k = 0; big = int_list[1]

E: None of the above

```
# 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]
```

A6: 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

A: k = 0; big = int_list[0]

B: k = 1; big = int_list[0]

C: k = 1; big = int_list[1]

D: k = 0; big = int_list[1]

E: None of the above

```
# 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]
```

✓

Clicker Answers

Q1: A: `n_forks +=2`

Q2: C: 0..k–1

Q3: D: `n_pair` = num adj. equal pairs in s[0..k–1]

Q4: B: k = 1

Q5: B: s[k-1] and s[k]

Q6: B: `k = 1;  big = int_list[0]`