CORRECTNESS ISSUES AND LOOP INVARIANTS

Lecture 8
CS2110 – Fall 2015

Final exam

Taken from this webpage:

https://registrar.cornell.edu/Sched/EXFA.html

CS 2110 001 Sat, Dec 12 2:00 PM

It will be optional. We will tell you as soon as everything is graded what your letter grade will be if you don't take it. You tell us whether you will take the final or not. Taking it can lower (rarely) as well as raise your grade.

Usually, ~20% of the class takes the final

The next several lectures

Study algorithms for searching and sorting arrays.
Investigate their complexity –how much time and space they take
"Formalize" the notions of average-case and worst-case complexity

We want you to *know* these algorithms

- *Not* by memorizing code but by
- Being able to *develop* the algorithms from their specifications and, when necessary, a small idea

We give you some guidelines and instructions on how to develop an algorithm from its specification.
Deal mainly with developing **loops and loop invariants**

Relative precedence of && and ||

What is the value of

true || true && false

How (not) to write an expression


```
/** Return value of "this person and p are intellectual siblings. "
  * Note: if p is null, they are not siblings.  */
public boolean isPhDSibling(PhD p) {
  return p != null //p cannot be null
   && this.equals(p) == false //p & this are not the same object
  //have a non-null advisor in common
  && ((this.adv1!= null && p.getFirstAdvisor()!= null
          && this.adv1.equals(p.getFirstAdvisor()))
          || (this.adv1!= null && p.getSecondAdvisor()!=null
          && this.adv1.equals(p.getSecondAdvisor()))
          || (this.adv2!= null && p.getFirstAdvisor()!=null
          && this.adv1.equals(p.getSecondAdvisor()))
          | && this.adv2.equals(p.getSecondAdvisor())));
          }
```

How to write an expression

- Avoid useless clutter, e.g.
  - unnecessary "this.”
  - unnecessary parentheses
  - redundant operations
- Put spaces around operators –use spaces to reflect relative precedences
- Be consistent, e.g.

  don't use field for one object and getter for another
- Make the presentation on several lines reflect the structure of the expression

Simplify, don’t “complify” (complicate)

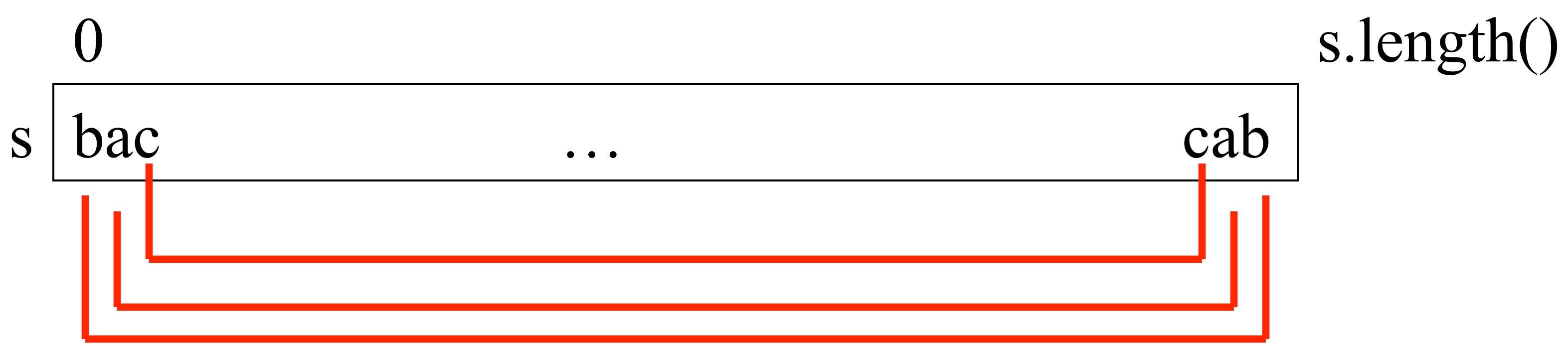
Show *development* of isPalindrome


```
/** Return true iff s is a palindrome */
public static boolean isPalindrome(String s)
```

Our instructions said to visit each char of s only once!

isPalindrome: Set ispal to “s is a palindrome” (forget about returns for now. Store value in ispal.

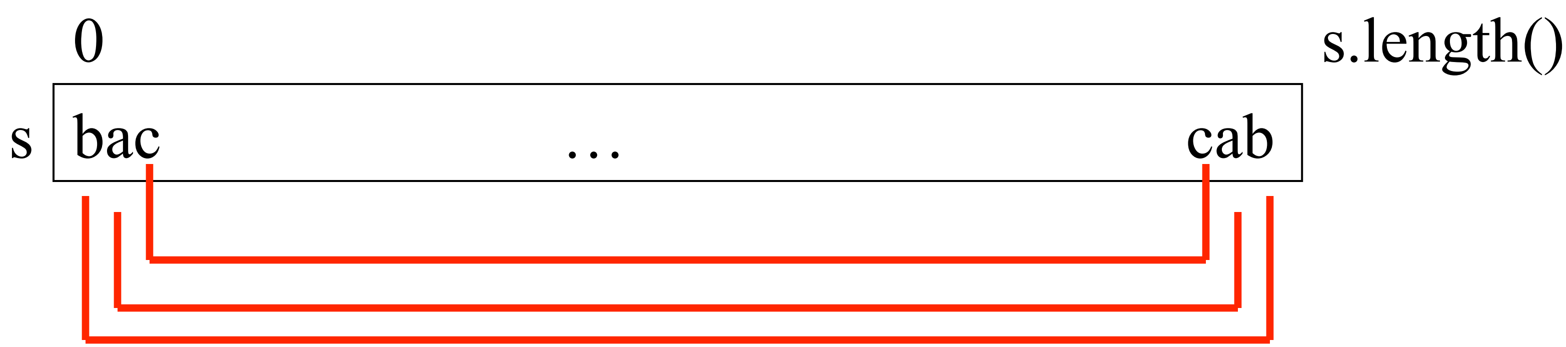
Think of checking equality of outer chars, then chars inside them, then chars inside them, etc.

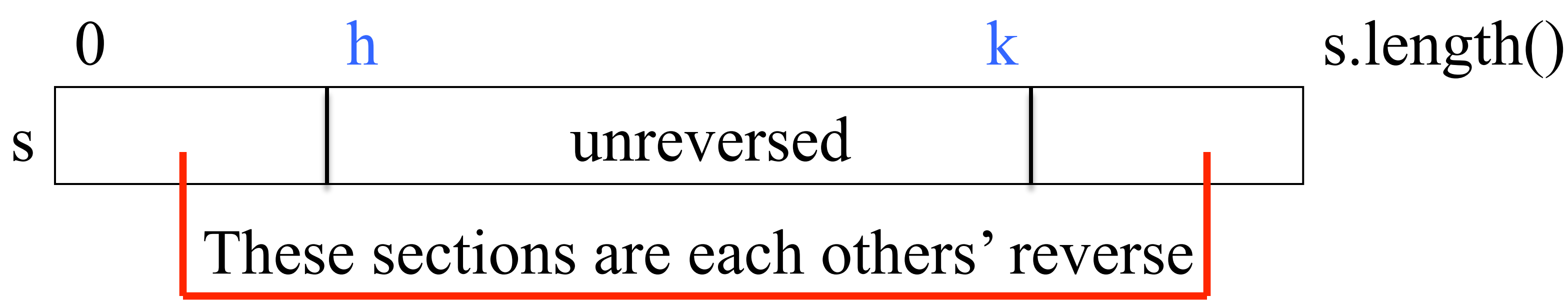


Key idea:
Generalize this to a picture that is true before/after each iteration

isPalindrome: Set ispal to “s is a palindrome” (forget about returns for now. Store value in ispal.

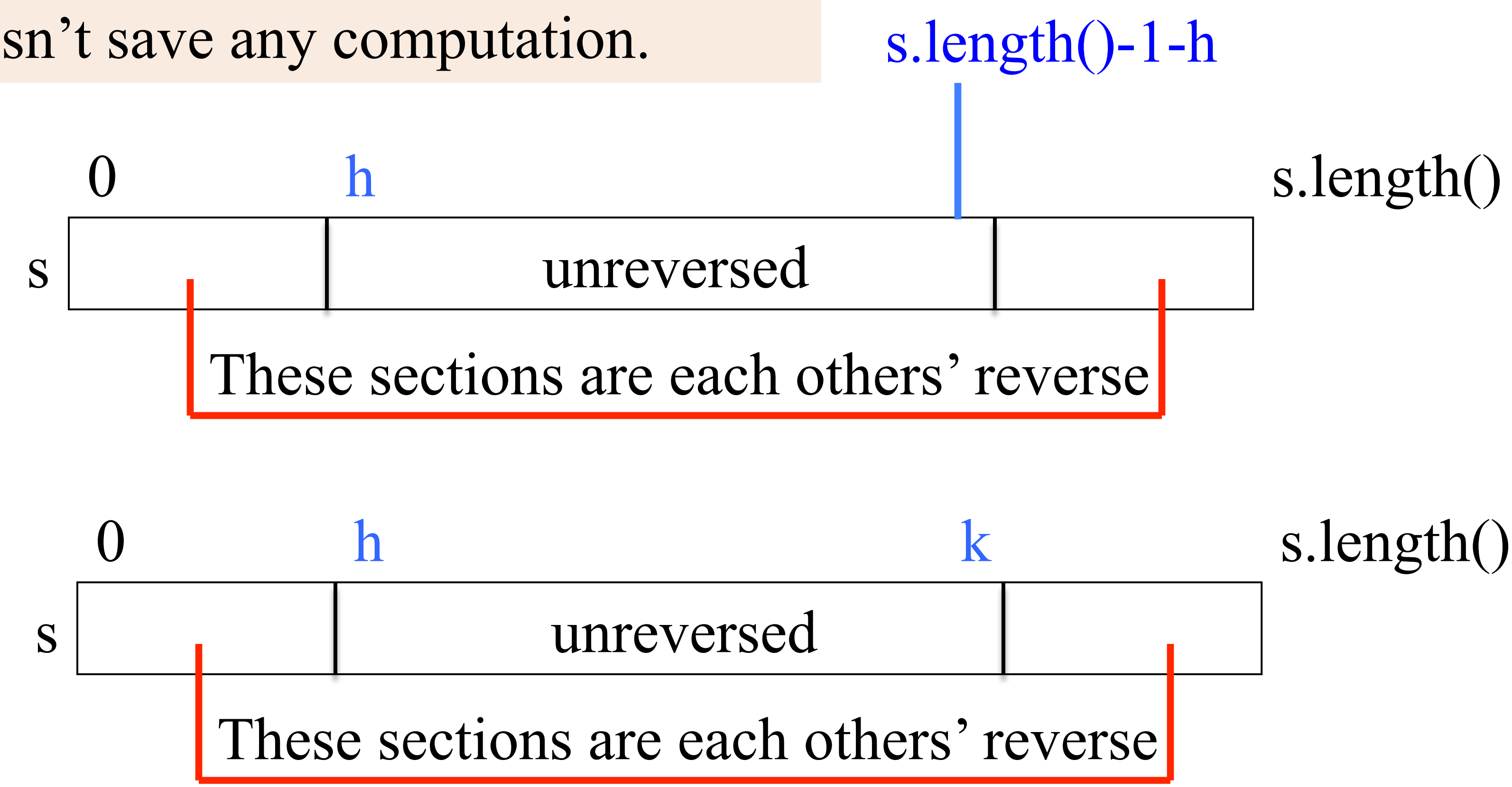
Generalize to a picture that is true before/after each iteration





Do it with one variable?

Using only h makes it more difficult to figure out when to stop and doesn't save any computation.



isPalindrome: Set ispal to “s is a palindrome”

Initialization to make picture true

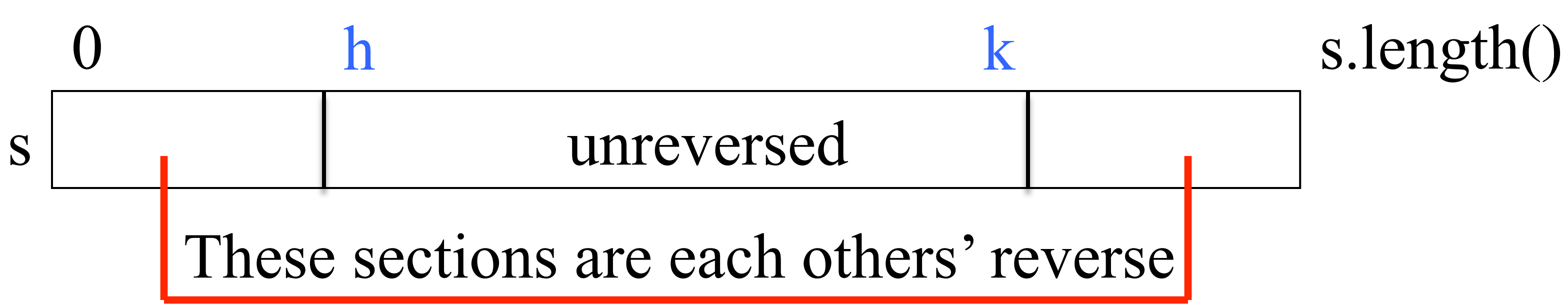
```
int h= 0;
int k= s.length() – 1;
// s[0..h-1] is the reverse of s[k+1..]
```

Stop when result is known
Continue when it’s not

```
while ( h < k && s.charAt(h) == s.charAt(k) ) {
      h= h+1; k= k-1;
}
```

Make progress toward termination
AND keep picture true

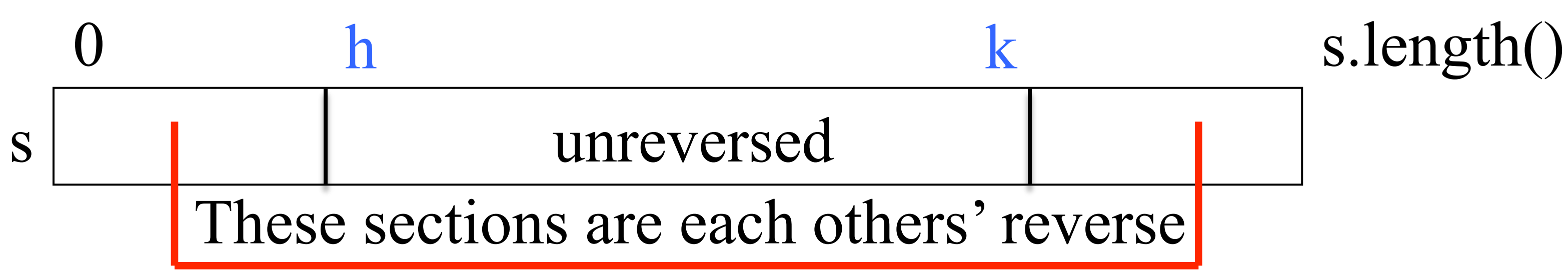
```
ispal= h >= k;
```



isPalindrome: written as a function.
Return when answer known


```
/** Return true iff s is a palindrome */
public static boolean isPal(String s) {
    int h=  0;  int k=  s.length() – 1;
    // invariant: s[0..h-1] is reverse of s[k+1..]
    while (h < k) {
        if (s.charAt(h) != s.charAt(k))
            return false;
        h=  h+1;  k=  k-1;
    }
    return true;
}
```

Loop invariant — invariant because it’s true before/ after each loop iteration



Engineering principle

Break a project up into parts, making them as independent as possible. When the parts are constructed, put them together.

Each part can be understood by itself, without mentioning the others.

Reason for introducing loop invariants

Given c >= 0, store b^c in x

```
z= 1;  x= b;  y=  c;
while (y != 0) {
    if (y is even) {
        x= x*x;  y= y/2;
    } else {
        z= z*x;  y= y - 1;
    }
}
{z = b^c}
```

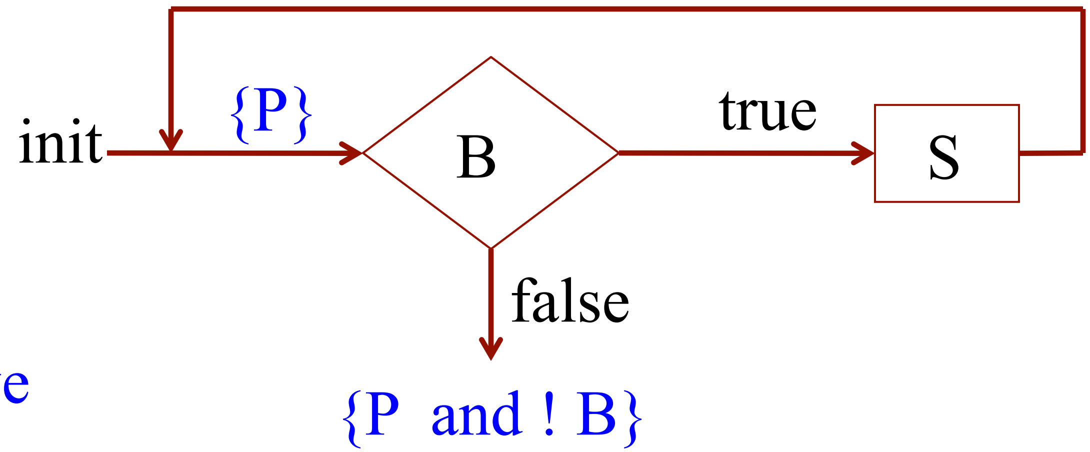
Algorithm to compute b^c.

Can't understand any piece of it without understanding all.
In fact, only way to get a handle on it is to execute it on some test case.

Need to understand initialization without looking at any other code.
Need to understand condition y != 0 without looking at loop body
Etc.

Invariant: is true before and after each iteration


```
initialization;
// invariant P
 while (B) {S}
```



Upon termination, we know P true, B false

"invariant" means unchanging. Loop invariant: an assertion —a true-false statement— that is true before and after each iteration of the loop —every time B is to be evaluated.

Help us understand each part of loop without looking at all other parts.

Simple example to illustrate methodology

Store sum of 0..n in s
Precondition: n >= 0

```
// { n >= 0}
k= 1; s= 0;
// inv:  s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

First loopy question.
Does it start right?
Does initialization make invariant true?

Yes!

    s = sum of 0..k-1
=   <substitute initialization>
    0 = sum of 0..1-1
=   <arithmetic>
    0 = sum of 0..0

We understand initialization without looking at any other code

Simple example to illustrate methodology


```
Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
k= 1; s= 0;
// inv:  s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

Second loopy question.

Does it stop right?

Upon termination, is postcondition true?

Yes!

inv && ! k <= n

=> <look at inv>

inv && k = n+1

=> <use inv>

s = sum of 0..n+1-1

We understand that postcondition is true without looking at init or repetend

Simple example to illustrate methodology

Store sum of 0..n in s
Precondition: n >= 0

```
// { n >= 0}
k= 1; s= 0;
// inv:  s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

Third loopy question.
Progress?
Does the repetend make progress toward termination?

Yes! Each iteration increases k, and when it gets larger than n, the loop terminates

We understand that there is no infinite looping without looking at init and focusing on ONE part of the repetend.

Simple example to illustrate methodology

Store sum of 0..n in s
Precondition: n >= 0

```
// { n >= 0}
k= 1; s= 0;
// inv: s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

Fourth loopy question.
Invariant maintained by each iteration?

Is this property true?

{inv && k <= n} repetend {inv}

Yes!

```
{s = sum of 0..k-1}
s= s + k;
{s =  sum of 0..k}
k= k+1;
{s = sum of 0..k-1}
```

4 loopy questions to ensure loop correctness


```
{precondition Q}
init;
// invariant P
while (B) {
    S
}
{R}
```

Four loopy questions: if answered yes, algorithm is correct.

First loopy question;
Does it start right?
Is {Q} init {P} true?

Second loopy question:
Does it stop right?
Does P && ! B imply R?

Third loopy question:
Does repetend make progress?
Will B eventually become false?

Fourth loopy question:
Does repetend keep invariant true?
Is {P && ! B} S {P} true?

Note on ranges m..n

Range m..n contains n+1–m ints: m, m+1, ..., n
(Think about this as “Follower (n+1) minus First (m)”)

2..4 contains 2, 3, 4: that is 4 + 1 – 2 = 3 values

2..3 contains 2, 3: that is 3 + 1 – 2 = 2 values

2..2 contains 2: that is 2 + 1 – 2 = 1 value

2..1 contains : that is 1 + 1 – 2 = 0 values

Convention: notation m..n implies that m <= n+1

Assume convention even if it is not mentioned!

If m is 1 larger than n, the range has 0 values

array segment b[m..n]:

Can't understand this example without invariant!

Given c >= 0, store b^c in z

```
z= 1;  x= b;  y=  c;
// invariant  y >= 0  &&
//              z*x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x;  y= y/2;
    } else {
        z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

First loopy question.
Does it start right?
Does initialization make invariant true?

Yes!

z*x^y
= <substitute initialization>
1*b^c
= <arithmetic>
b^c

We understand initialization without looking at any other code

For loopy questions to reason about invariant

Given c >= 0, store b^c in x

```
z= 1;  x= b;  y=  c;
// invariant y >= 0  AND
//                z*x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x;  y= y/2;
    } else {
        z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

Second loopy question.
Does it stop right?
When loop terminates,
is z = b^c?

Yes! Take the invariant, which is true, and use fact that y = 0:

```
    z*x^y  =  b^c
=     <y = 0>
    z*x^0 = b^c
=     <arithmetic>
    z = b^c
```

We understand loop condition without looking at any other code

For loopy questions to reason about invariant

Given c >= 0, store b^c in x

```
z= 1;  x= b;  y=  c;
// invariant y >= 0  AND
//                z*x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x;  y= y/2;
    } else {
        z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

Third loopy question.
Does repetend make progress toward termination?

Yes! We know that y > 0 when loop body is executed. The loop body decreases y.

We understand progress without looking at initialization

For loopy questions to reason about invariant

Given c >= 0, store b^c in x

```
z= 1;  x= b;  y=  c;
// invariant y >= 0  AND
//                z*x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x;  y= y/2;
    } else {
        z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

Fourth loopy question.
Does repetend keep invariant true?

Yes! Because of properties:

- For y even, x^y = (x*x)^(y/2)
- z*x^y = z*x*x^(y-1)

We understand invariance without looking at initialization

Develop binary search for v in sorted array b

pre: b

0	b.length
?	

Store in h to make this true:

post: b

0	h	b.length
<= v	> v	

Example:

pre: b

0				4	5	6	7				b.length
2	2	4	4	4	4	7	9	9	9	9	

If v is 4, 5, or 6, h is 5

If v is 7 or 8, h is 6

If v in b, h is index of rightmost occurrence of v.
If v not in b, h is index before where it belongs.

Develop binary search in sorted array b for v

		0		b.length
pre:	b	?		

Store a value in h to make this true:

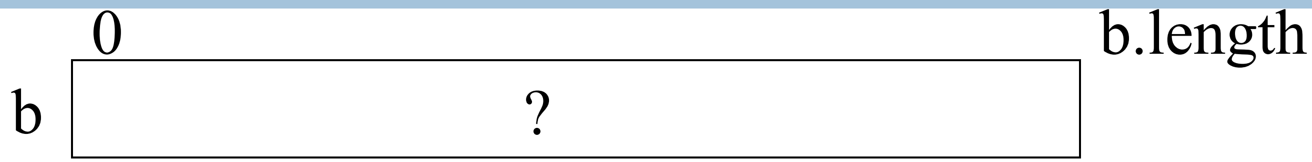
		0	h	b.length
post:	b	<= v	> v	

Get loop invariant by combining pre- and post-conditions, adding variable t to mark the other boundary

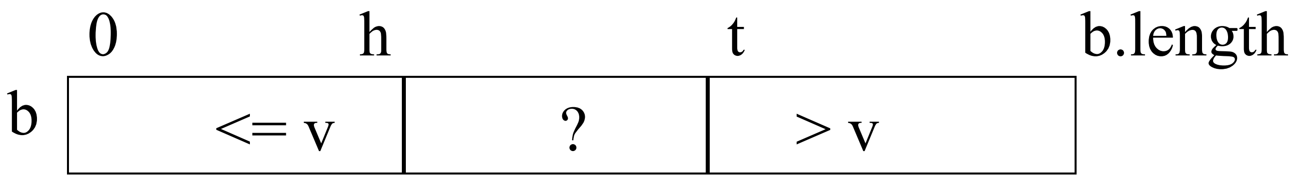
		0	h	t	b.length
inv:	b	<= v	?	> v	

How does it start (what makes the invariant true)?

pre:



inv:



Make first and last partitions empty:

h= -1; t= b.length;

When does it end (when does invariant look like postcondition)?

post: b

0	h	b.length
<= v	> v	

inv: b

0	h	t	b.length
<= v	?	> v	

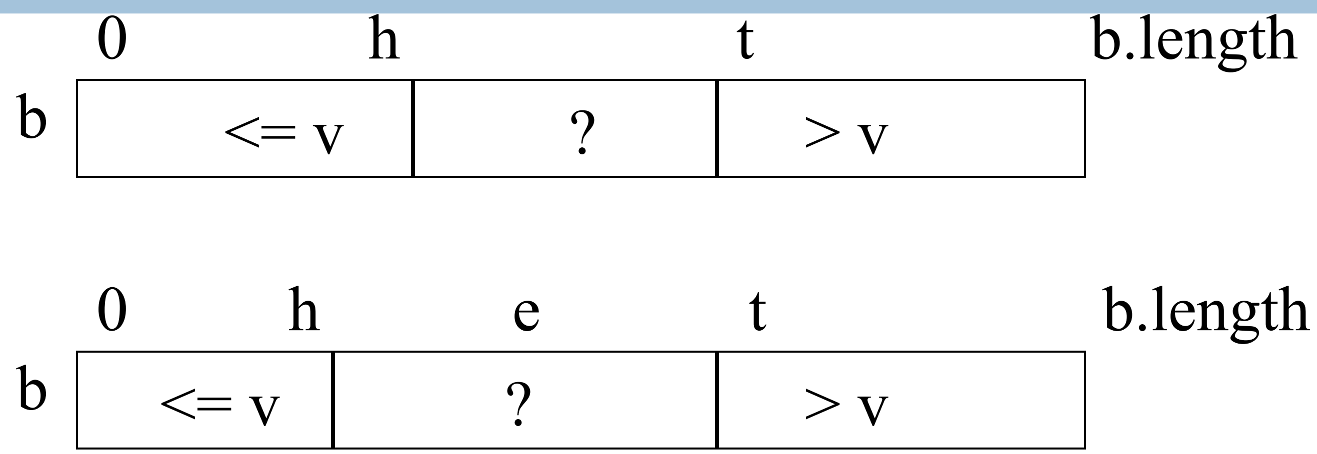
```
h= -1;  t= b.length;
while ( h  !=  t-1 ) {

}
```

Stop when ? section is empty. That is when h = t-1.
Therefore, continue as long as h != t-1.

How does body make progress toward termination (cut ? in half) and keep invariant true?

inv:




```
h= -1;  t= b.length;
while ( h != t-1 ) {
    int  e= (h+t)/2;
}
```

Let e be index of middle value of ? Section.
Maybe we can set h or t to e, cutting ? section in half

How does body make progress toward termination (cut ? in half) and keep invariant true?

inv:

	0	h	t	b.length
b	<= v	?	> v	

	0	h	e		t	b.length
b	<= v	?		?	> v	

	0	h	e		t	b.length
b	<= v	<=	v	?	> v	

```
h= -1;  t= b.length;
while ( h != t-1 ) {
    int  e= (h+t)/2;
    if (b[e] <= v)  h= e;

}
```

If b[e] <= v, then so is every value to its left, since the array is sorted. Therefore, h= e; keeps the invariant true.

How does body make progress toward termination (cut ? in half) and keep invariant true?

inv:

	0	h	t	b.length
b	<= v	?	> v	

	0	h	e		t	b.length
b	<= v	?		?	> v	

	0	h	e		t	b.length
b	<= v	?		> v	> v	

```
h= -1;  t= b.length;
while ( h != t-1 ) {
    int  e= (h+t)/2;
    if (b[e] <= v)  h= e;
    else  t= e;
}
```

If b[e] > v, then so is every value to its right, since the array is sorted. Therefore, t= e; keeps the invariant true.

Develop binary search in sorted array b for v

	0	b.length
pre: b	?	

Store a value in h to make this true:

	0	h	b.length
post: b	<= v	> v	

DON'T TRY TO MEMORIZE CODE!

Instead, learn to derive the loop invariant from the pre- and post-condition and then to develop the loop using the pre- and post-condition and the loop invariant.

PRACTICE THIS ON KNOWN ALGORITHMS!

Processing arrays from beg to end (or end to beg)

Many loops process elements of an array b (or a String, or any list) in order: b[0], b[1], b[2], …

If the postcondition is

R: b[0..b.length-1] has been processed

Then in the beginning, nothing has been processed, i.e.

b[0..-1] has been processed

After k iterations, k elements have been processed:

P: b[0..k-1] has been processed

		0	k	b.length
invariant P:	b	processed	not processed	

Processing arrays from beg to end (or end to beg)

Replace b.length in postcondition
by fresh variable k to get invariant

b[0..k-1] has been processed

or draw it as a picture

Task: Process b[0..b.length-1]

```
k= 0;
{inv P}
while (   k != b.length    ) {
    Process b[k];      // maintain invariant
    k=  k + 1;         // progress toward termination
}
{R:  b[0..b.length-1]   has been processed}
```

		0	k	b.length
inv P:	b	processed	not processed	

Processing arrays from beg to end (or end to beg)

Most loops that process the elements of an array in order will have this loop invariant and will look like this.

```
Task: Process b[0..b.length-1]
k= 0;
{inv P}
while (   k != b.length     ) {
     Process b[k];        // maintain invariant
     k=  k + 1;           // progress toward termination
}
{R:  b[0..b.length-1]   has been processed}
```

		0	k	b.length
inv P:	b	processed	not processed	

Count the number of zeros in b.
Start with last program and refine it for this task

Task: Set s to the number of 0's in b[0..b.length-1]

```
k= 0;     s=  0;
{inv P}
while (   k != b.length     ) {
    if (b[k] == 0)  s= s + 1;
    k=  k + 1;          // progress toward termination
}
{R: s = number of 0's in b[0..b.length-1]}
```

		0	k	b.length
inv P:	b	s = # 0's here	not processed	

Be careful. Invariant may require processing elements in reverse order!

This invariant forces processing from beginning to end

		0	k	b.length
inv P:	b	processed	not processed	

This invariant forces processing from end to beginning

		0	k	b.length
inv P:	b	not processed	processed	

Process elements from end to beginning


```
k= b.length–1;           // how does it start?

while (k >= 0) {         // how does it end?
    Process b[k];        // how does it maintain invariant?

    k= k – 1;            // how does it make progress?
}
```

{R: b[0..b.length-1] is processed}

		0	k	b.length
inv P:	b	not processed	processed	

Process elements from end to beginning


```
k= b.length–1;
while (k >= 0) {
      Process b[k];
      k= k – 1;
}
```

Heads up! It is important that you can look at an invariant and decide whether elements are processed from beginning to end or end to beginning!

For some reason, some students have difficulty with this. A question like this could be on the prelim!

{R: b[0..b.length-1] is processed}

		0	k	b.length
inv P:	b	not processed	processed	