Horizontal Notation for Sequences

	0	k	len(b)
b	<= sorted	>=	

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]

	0	h	k
b			

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 h+1

(h+1) – h = 1

Generalizing Pre- and Postconditions

- Dutch national flag: tri-color
  - Sequence of 0..n-1 of red, white, blue "pixels"
  - Arrange to put reds first, then whites, then blues

	0			n
pre: b	?			

(values in 0..n-1 are unknown)

	0			n
post: b	reds	whites	blues	

	0	j	k	l	n
inv: b	reds	whites	?	blues	

Make the red, white, blue sections initially **empty**:
- Range i..i-1 has 0 elements
- Main reason for this trick

Changing loop variables turns invariant into postcondition.

Generalizing Pre- and Postconditions

- Finding the minimum of a sequence.

	0		n
pre: b	?		

and n >= 0 (values in 0..n are unknown)

	0		n
post: b	x is the min of this segment		

	0	j	n
inv: b	x is min of this segment	?	

pre: j = 0
post: j = n

(values in j..n are unknown)

- Put negative values before nonnegative ones.

	0		n
pre: b	?		

and n >= 0 (values in 0..n are unknown)

	0	k	n
post: b	< 0	>= 0	

	0	k	j	n
inv: b	< 0	?	>= 0	

pre: k = 0, j = n
post: k = j

(values in k..j-1 are unknown)

Partition Algorithm

- Given a sequence b[h..k] with some value x in b[h]:

	h		k
pre: b	x	?	

- Swap elements of b[h..k] and store in j to truthify post:

	h	i	i+1	k
post: b	<= x	x	>= x	

	h								k
change: b	3	5	4	1	6	2	3	8	1
	h			i					k
into b	1	2	1	3	5	4	6	3	8
	h				i				k
or b	1	2	3	1	3	4	5	6	8

- x is called the pivot value
  - x is not a program variable
  - denotes value initially in b[h]

Partition Algorithm

- Given a sequence b[h..k] with some value x in b[h]:

	h		k
pre: b	x	?	

- Swap elements of b[h..k] and store in j to truthify post:

	h	i	i+1	k
post: b	<= x	x	>= x	

	h	i		j	k
inv: b	<= x	x	?	>= x	

- Agrees with precondition when i = h, j = k+1
- Agrees with postcondition when j = i+1

Partition Algorithm Implementation

```
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            _swap(b,i+1,j-1)
            j = j - 1
        else:   # b[i+1] < x
            _swap(b,i,i+1)
            i = i + 1
    # post: b[h..i-1] < x, b[i] is x, and b[i+1..k] >= x
    return i
```

<= x | x | ? | >= x

h		i	i+1			j		k
1	2	3	1	5	0	6	3	8
h			i	i+1		j		k
1	2	1	3	5	0	6	3	8
h			i		j			k
1	2	1	3	0	5	6	3	8
h				i	j			k
1	2	1	0	3	5	6	3	8

Dutch National Flag Variant

- Sequence of integer values
  - 'red' = negatives, 'white' = 0, 'blues' = positive
  - Only rearrange part of the list, not all

Final Exam:
Be prepared for variants

pre: b (h … k): ?

post: b (h … k): < 0 | = 0 | > 0

inv: b (h, t, i, j, k): < 0 | ? | = 0 | > 0

pre: t = h,
i = k+1,
j = k
post: t = i

Dutch National Flag Algorithm

```
def dnf(b, h, k):
    """Returns: partition points as a tuple (i,j)"""
    t = h; i = k+1, j = k;
    # inv: b[h..t-1] < 0, b[t..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)
```

< 0	?	= 0	> 0
h	t	i j	k
-1 -2	3 -1 0	0 0	6 3

h t i j k: -1 -2 | 3 -1 | 0 0 0 | 6 3

h t i j k: -1 -2 -1 | 3 | 0 0 0 | 6 3

h t j k: -1 -2 -1 | 0 0 0 | 3 6 3

Linear Search

- **Vague**: Find first occurrence of v in b[h..k-1].
- **Better**: Store an integer in i to truthify result condition post:

post: 1. v is not in b[h..i-1]
2. i = k OR v = b[i]

pre: b (h … k): ?

post: b (h, i, k): v not here | v | ?

OR

b (h … k, i = k): v not here

Linear Search

pre: b (h … k): ?

post: b (h, i, k): v not here | v | ?

OR

b (h … k, i = k): v not here

inv: b (h, i, k): v not here | ?

Linear Search

```
def linear_search(b,c,h):
    """Returns: first occurrence of c in b[h..]"""
    # Store in i the index of the first c in b[h..]
    i = h

    # invariant: c is not in b[0..i-1]
    while i < len(b) and b[i] != c:
        i = i + 1

    # post: b[i] == c and c is not in b[h..i-1]
    return i if i < len(b) else -1
```

Analyzing the Loop

1. Does the initialization make **inv** true?
2. Is **post** true when **inv** is true and **condition** is false?
3. Does the repetend make progress?
4. Does the repetend keep **inv** true?

b (h, i, n): c is not here | c | — result (post)

b (h, i, n): c is not here | c is in here — invariant (inv)

b[i] == c

Binary Search

- **Vague:** Look for v in **sorted** sequence segment b[h..k].
- **Better:**
  - Precondition: b[h..k-1] is sorted (in ascending order).
  - Postcondition: b[h..i] <= v and v < b[i+1..k-1]
- Below, the array is in non-descending order:

pre: b (h … k): ?

post: b (h, i, k): <= v | > v

inv: b (h, i, j, k): < v | ? | > v

Called binary search because each iteration of the loop cuts the array segment still to be processed in half