Lecture 22

Designing Sequence Algorithms

Announcements for This Lecture

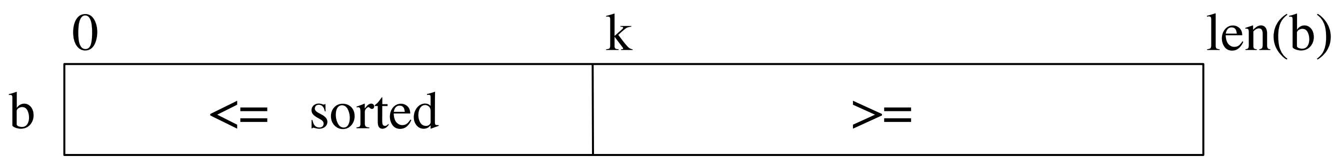
Assignments

- A5 graded by weekend
  - We just starting on it
- Should be working on A6
  - Due week from **Today**
  - Work on a method a day
  - Should start stenography no later than Sunday
  - **Friday** extension?
- A7 due after class ends

Prelim 2

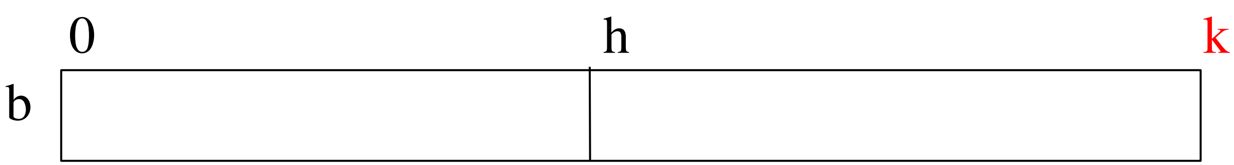
- High scores again
  - Mean: 83, **Median**: 86
  - 150/404 scored 90+
  - Historical mean: 76
  - For-loop, not recursion hard
- But good grade distribution
  - A: 90+
  - B: Mid-low 70s to high 80s
  - C: 50 to mid-low 70s

Horizontal Notation for Sequences

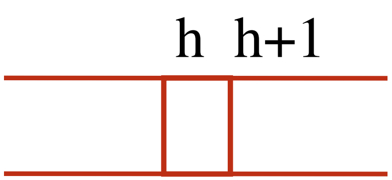


Example of an assertion about an sequence b. It asserts that:

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.



(h+1) – h = 1

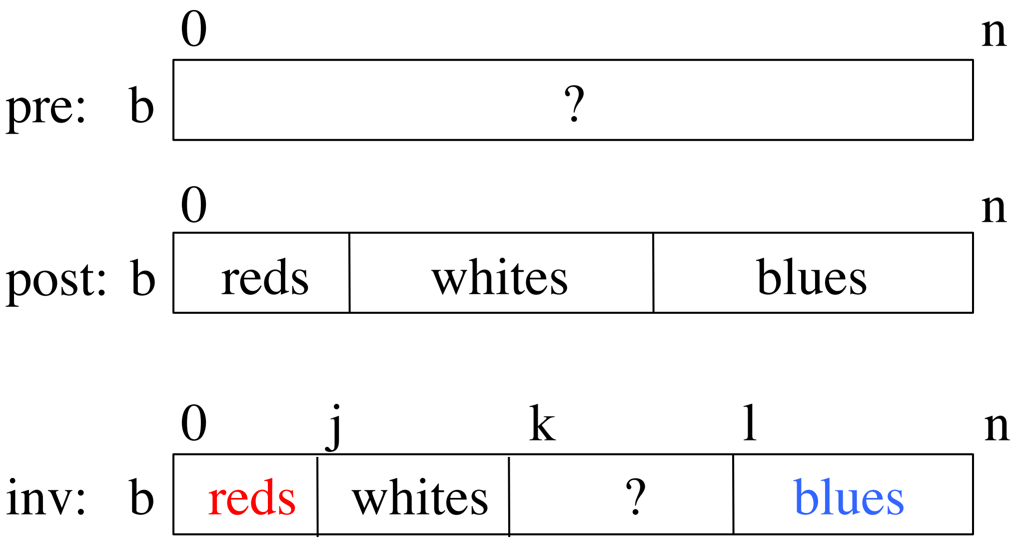
b[h .. k – 1] has k – h elements in it.

Developing Algorithms on Sequences

- Specify the algorithm by giving its precondition and postcondition as pictures.
- Draw the invariant by drawing another picture that "generalizes" the precondition and postcondition
  - The invariant is true at the beginning and at the end
- The four loop design questions (memorize them)
  1. How does loop start (how to make the invariant true)?
  2. How does it stop (is the postcondition true)?
  3. How does repetend make progress toward termination?
  4. How does repetend keep the invariant true?

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



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

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.

pre: b [0 .. n: ?] and n >= 0 (values in 0..n are unknown)

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

inv: b [0 .. j: x is min of this segment | j .. n: ?] (values in j..n are unknown)

pre: j = 0
post: j = n

- Put negative values before nonnegative ones.

pre: b [0 .. n: ?] and n >= 0 (values in 0..n are unknown)

post: b [0 .. k: < 0 | k .. n: >= 0]

inv: b [0 .. k: < 0 | k .. j: ? | j .. n: >= 0] (values in k..j-1 are unknown)

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

Partition Algorithm

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

pre: b

h		k
x	?	

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

post: b

h	i	i+1 k
<= x	x	>= x

change: b

h								k
3	5	4	1	6	2	3	8	1

into b

h			i					k
1	2	1	**3**	5	4	6	3	8

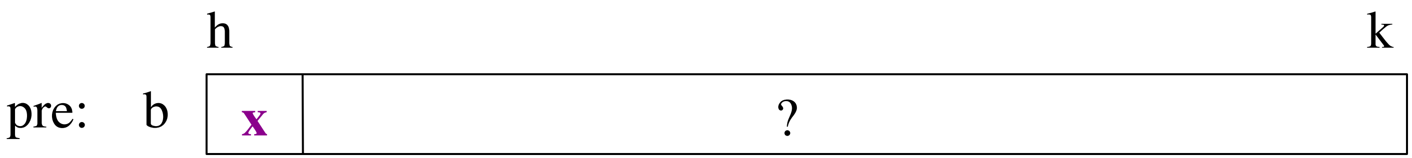
or b

h				i				k
1	2	3	1	**3**	4	5	6	8

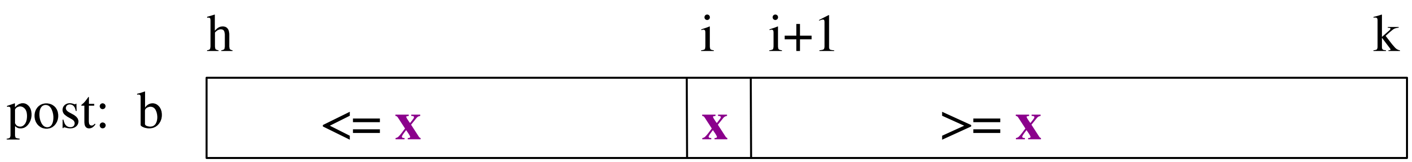
- x is called the pivot value
  - x is not a program variable
  - denotes value initially in b[h]

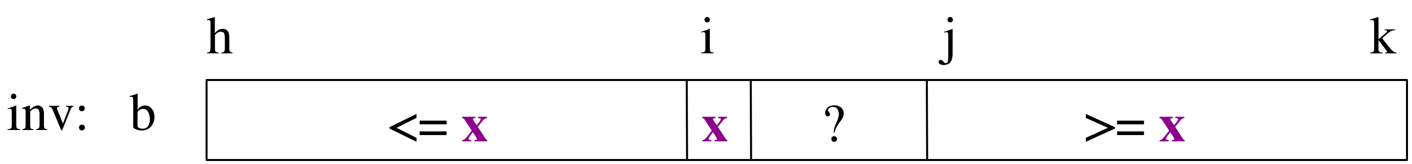
Partition Algorithm

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



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





- 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
```

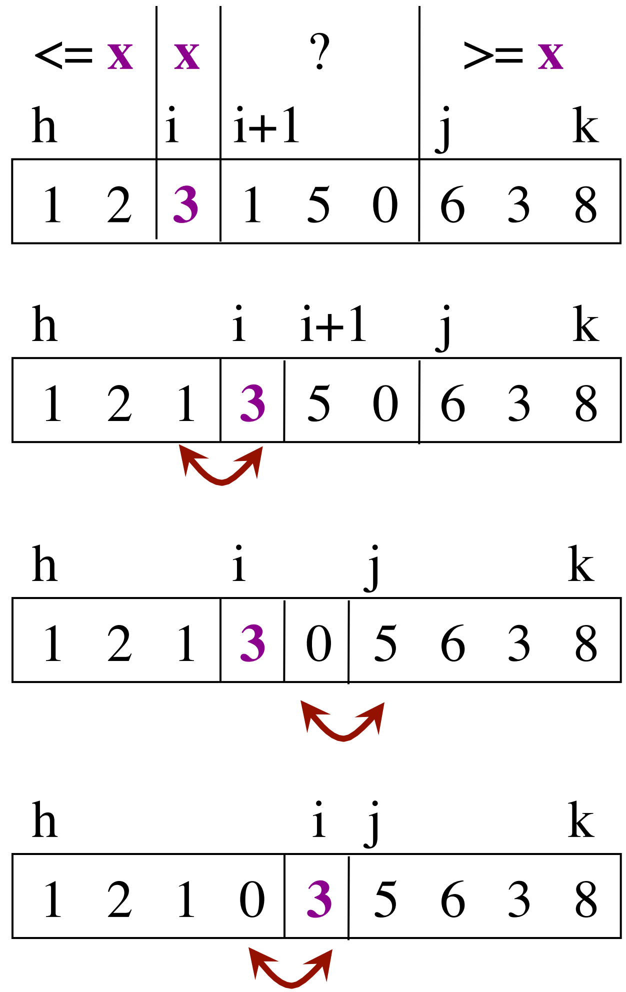
partition(b,h,k), not partition(b[h:k+1])

Remember, slicing always copies the list!

We want to partition the **original** list

Partition Algorithm Implementation

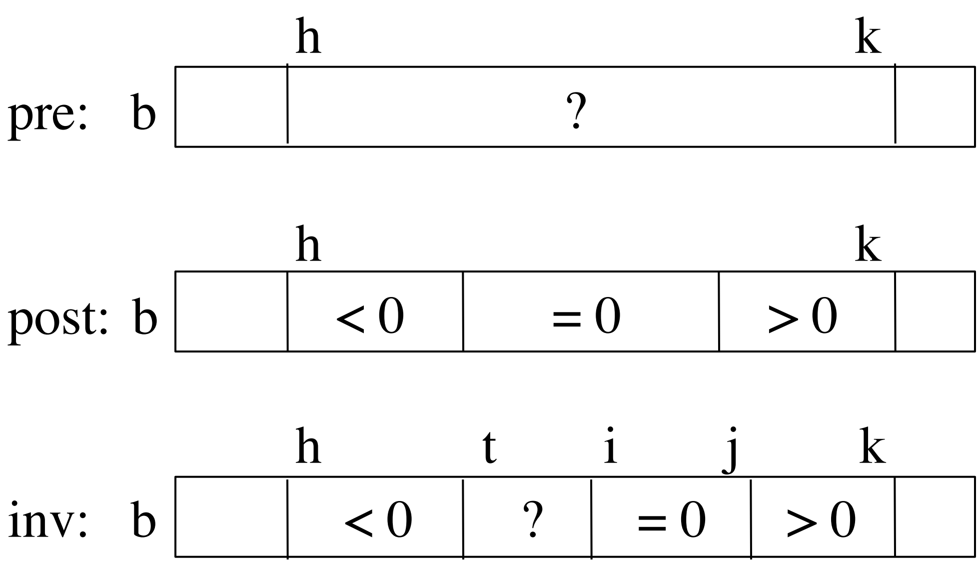
```
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            _swap(b,i+1,j-1)
            j = j - 1
 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



Dutch National Flag Variant

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

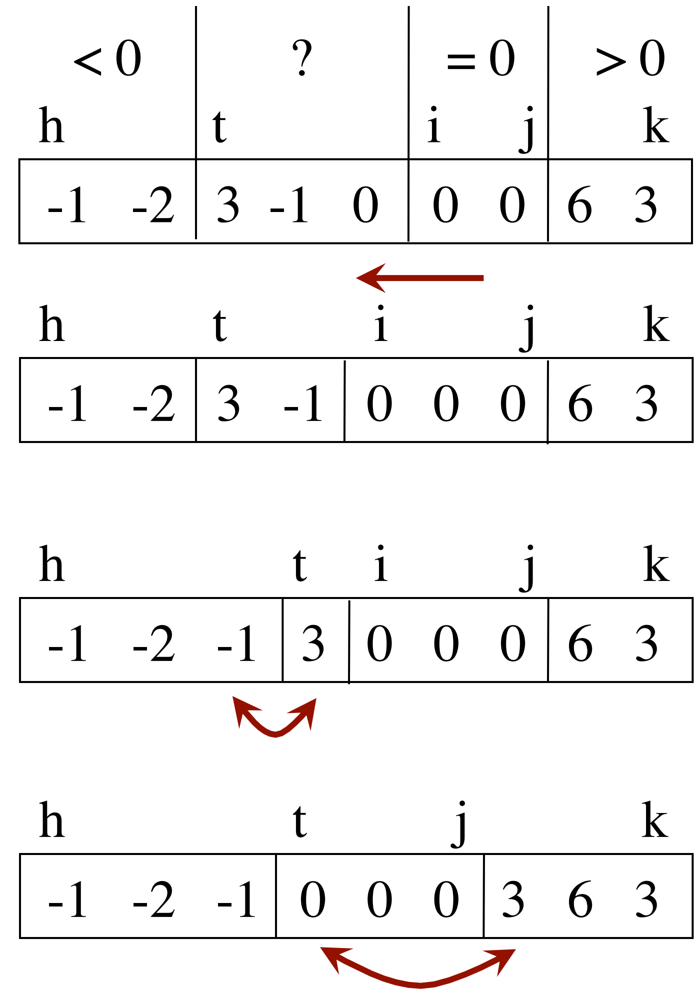
Final Exam:
Be prepared for variants



pre: $t = h$,
$i = k+1$,
$j = k$
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)
```

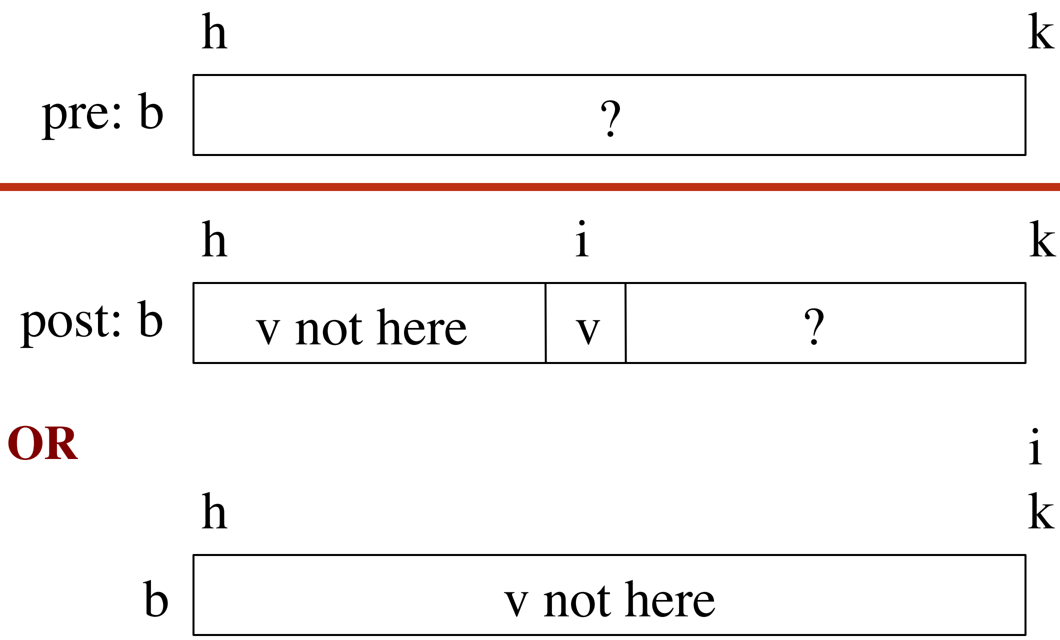


Linear Search

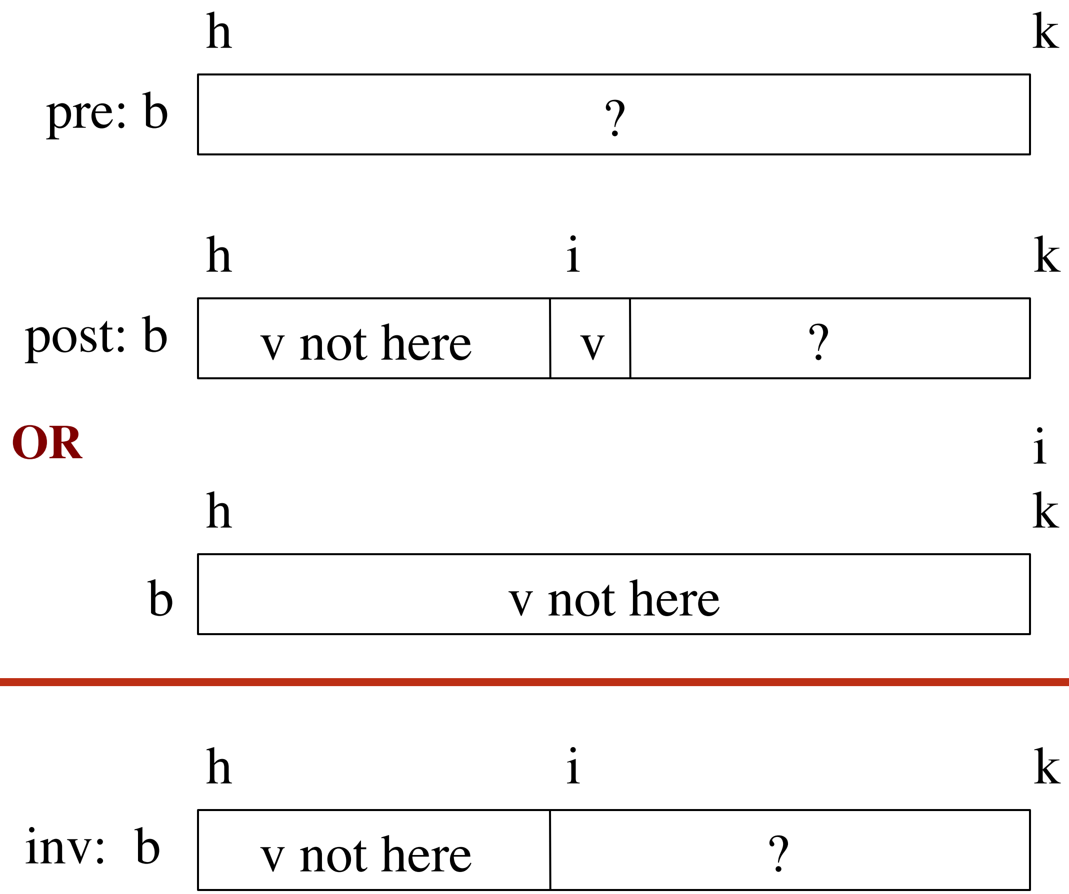
- **Vague**: Find first occurrence of v in b[h..k-1].
- **Better**: Store an integer in i to truthify result condition post:

  post: 1. v is not in b[h..i-1]

  2. i = k OR v = b[i]



Linear Search



Linear Search

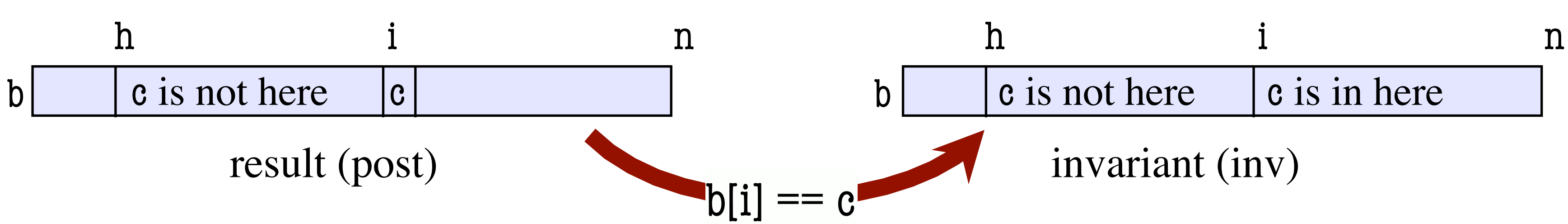
```
def linear_search(b,c,h):
    """Returns: first occurrence of c in b[h..]"""
    # Store in i the index of the first c in b[h..]
    i = h

    # 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
```

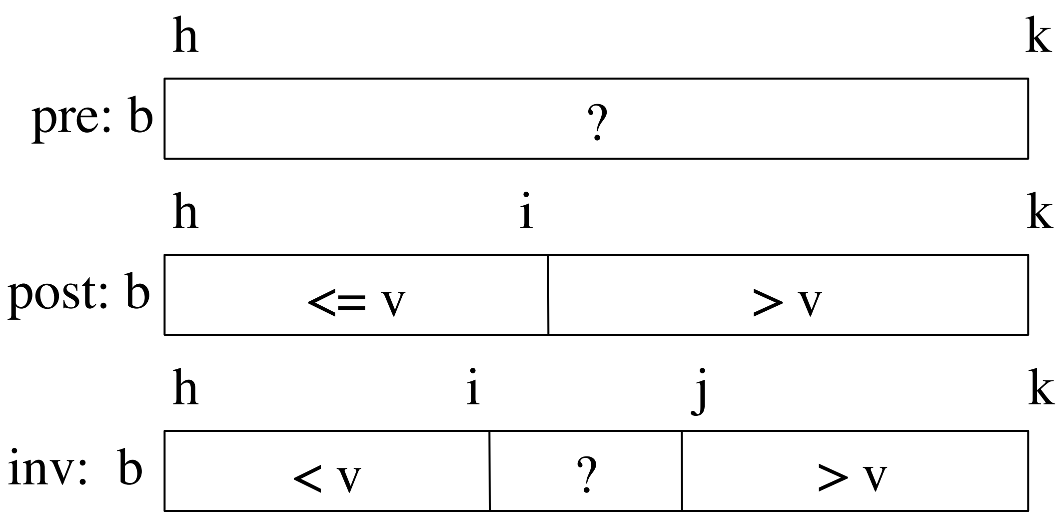
Analyzing the Loop

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?



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:



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

Extras Not Covered in Class

Loaded Dice

- Sequence p of length n represents n-sided die
  - Contents of p sum to 1
  - p[k] is probability die rolls the number k

1	2	3	4	5	6
0.1	0.1	0.1	0.1	0.3	0.3

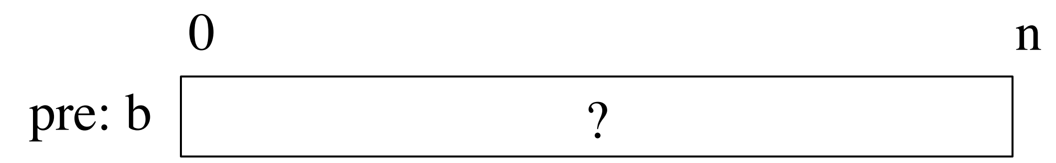
weighted d6, favoring 5, 6

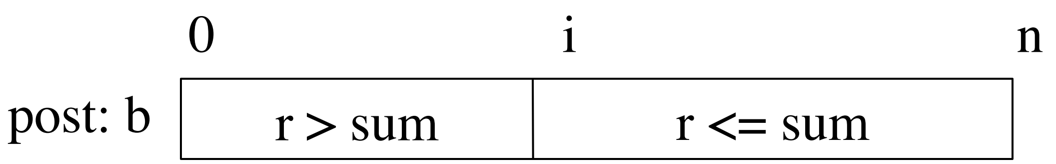
- Goal: Want to "roll the die"
  - Generate random number r between 0 and 1
  - Pick p[i] such that $p[i-1] < r \leq p[i]$

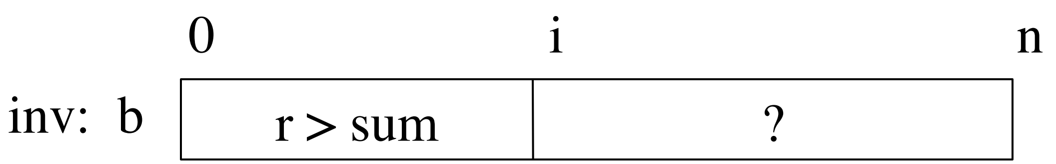
0.1	0.1	0.1	0.1	0.3	0.3
0.1	0.2	0.3	0.4	0.7	1.0

Loaded Dice

- **Want**: Value i such that p[i-1] < r <= p[i]







- Same as precondition if i = 0
- Postcondition is invariant + false loop condition

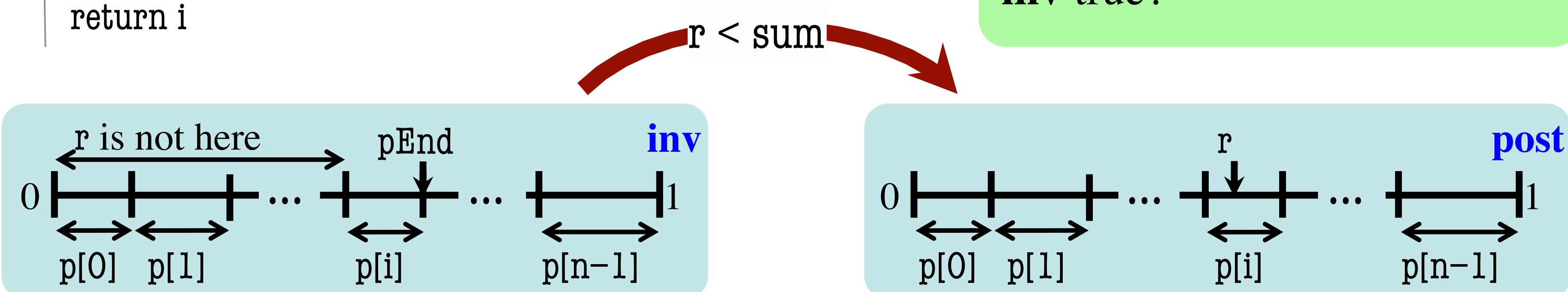
Loaded Dice

```
def roll(p):
    """Returns: randint in 0..len(p)-1; i returned with prob. p[i]
    Precondition: p list of positive floats that sum to 1."""
    r = random.random()     # r in [0,1)
    # Think of interval [0,1] divided into segments of size p[i]
    # Store into i the segment number in which r falls.
    i = 0;    sum_of = p[0]
    # inv: r >= sum of p[0] .. p[i–1]; pEnd = sum of p[0] .. p[i]
    while r >= sum_of:
        sum_of = sum_of + p[i+1]
        i = i + 1

    # post: sum of p[0] .. p[i–1] <= r < sum of p[0] .. p[i]
    return i
```

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?



Reversing a Sequence