CS 1110:

Introduction to Computing Using Python

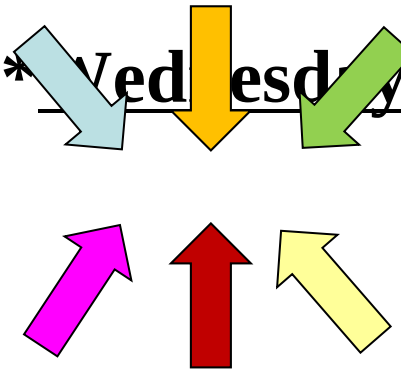
Lecture 23

Sorting and Searching

[Andersen, Gries, Lee, Marschner, Van Loan, White]

Announcements

- Final Exam conflicts due tonight at 11:59pm
- Final Exam review sessions on the 14th
- Labs on 5/9 and 5/10 will be office hours
- Assignment 5
  - Due 11:59pm on ***Wednesday*** May 10th
- Lab 13 is out



Recall: Accessing the "Original" Method

- What if you want to use the original version method?
  - New method = original+more
  - Do not want to repeat code from the original version
- Call old method **explicitly**
  - Use method as a function
  - Pass object as first argument
- **Example**:

  Employee.__str__(self)

```
class Employee(object):
    """An Employee with a salary"""
    …
    def __str__(self):
        return (self._name +
                ', year ' + str(self._start) +
                ', salary ' + str(self._salary))

class Executive(Employee):
    """An Employee with a bonus."""
    …
    def __str__(self):
        return (Employee.__str__(self)
                + ', bonus ' + str(self._bonus))
```

super

- Can also use super
- super(<class>, <instance>) returns the parent class of <class> and <instance>
- **Example**:

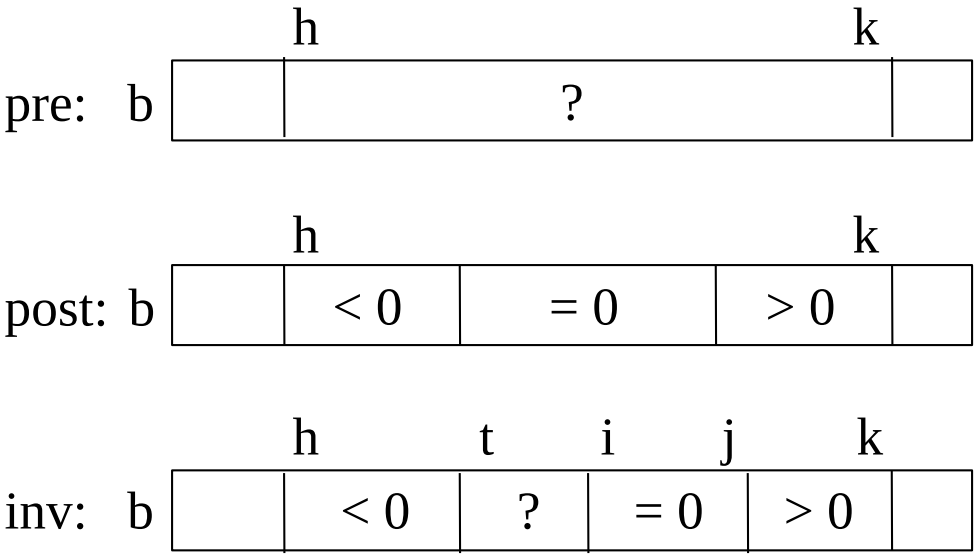
  super(Executive, self).__str__()

```
class Employee(object):
    """An Employee with a salary"""
    ...
    def __str__(self):
        return (self._name +
                ', year ' + str(self._start) +
                ', salary ' + str(self._salary))

class Executive(Employee):
    """An Employee with a bonus."""
    ...
    def __str__(self):
        return (super(Executive, self).__str__()
                + ', bonus ' + str(self._bonus))
```

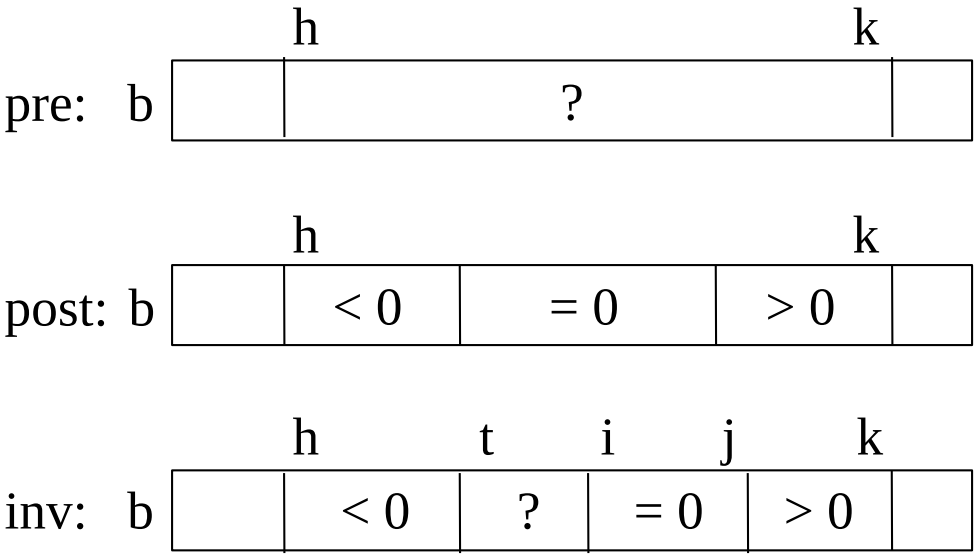
Dutch National Flag Variant

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



Dutch National Flag Variant

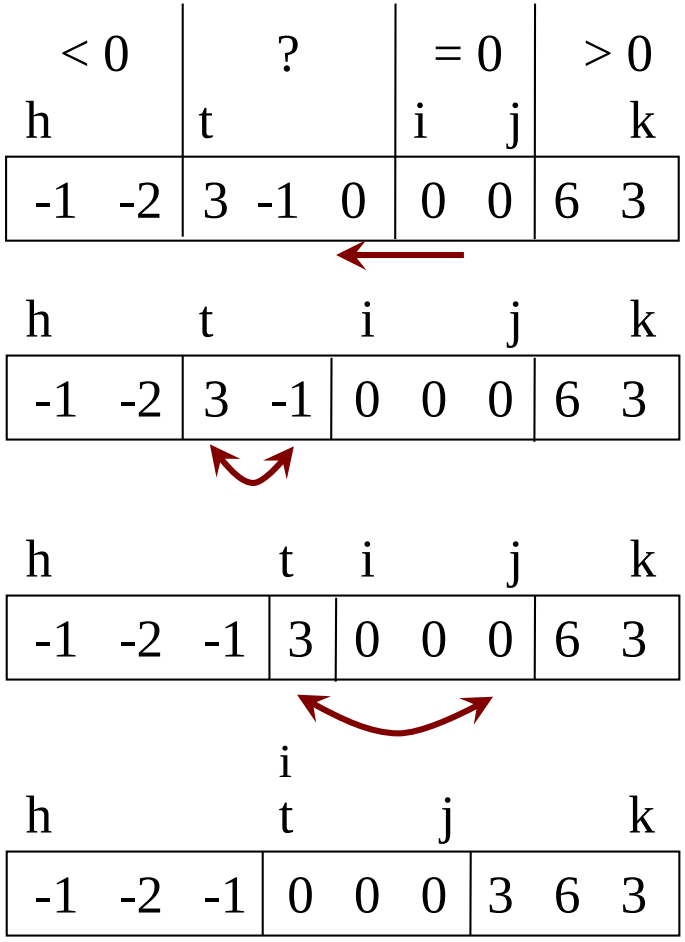
- Sequence of integer values
  - ‘red’ = negatives, ‘white’ = 0, ‘blues’ = positive
  - Only rearrange part of the list, not all



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

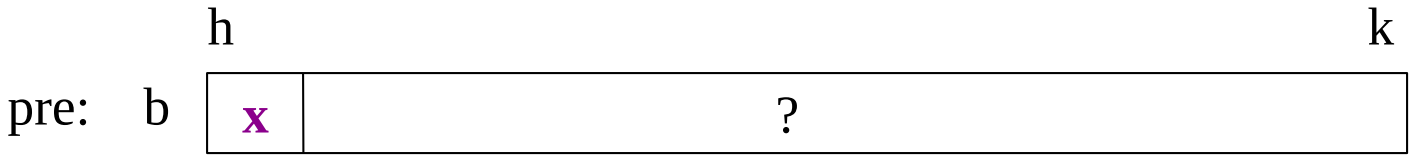
Dutch National Flag Algorithm

```
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)
```

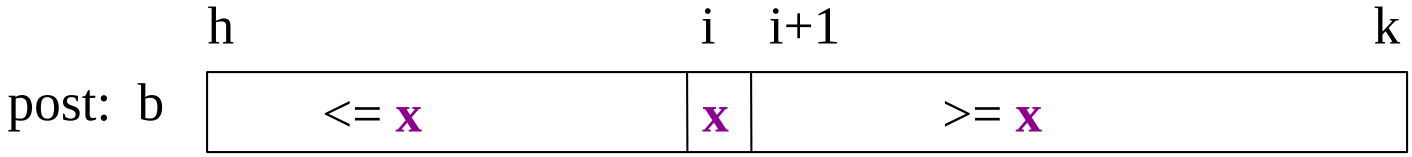


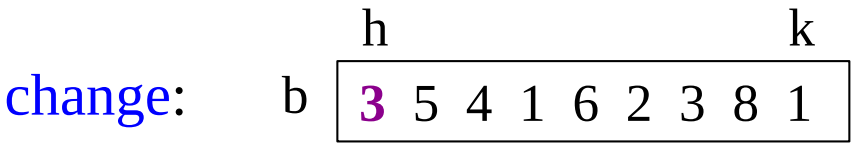
Partition Algorithm

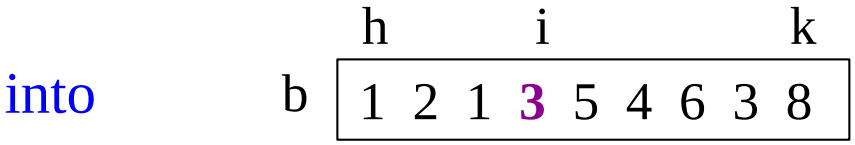
- Given a list segment b[h..k] with some pivot value x in b[h]:



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

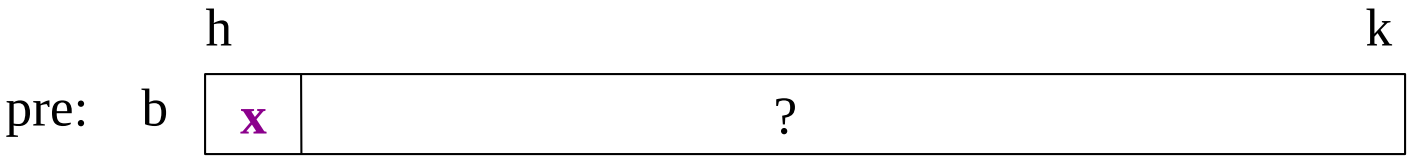




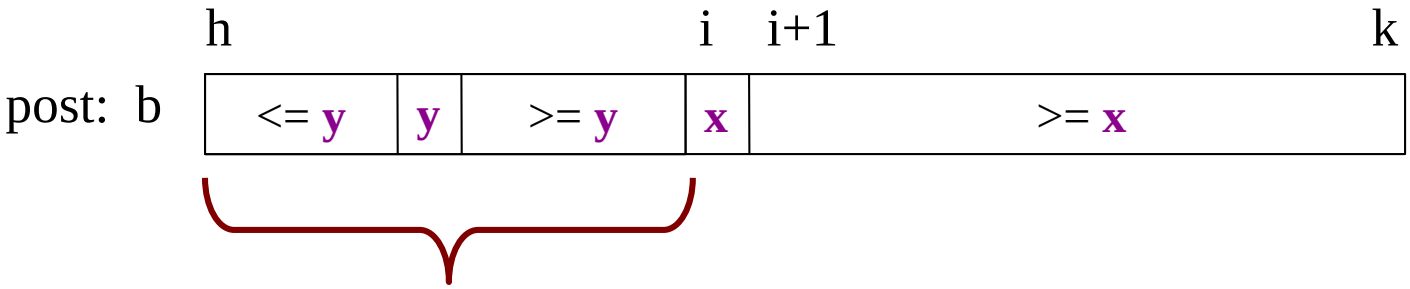


Sorting with Partitions

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



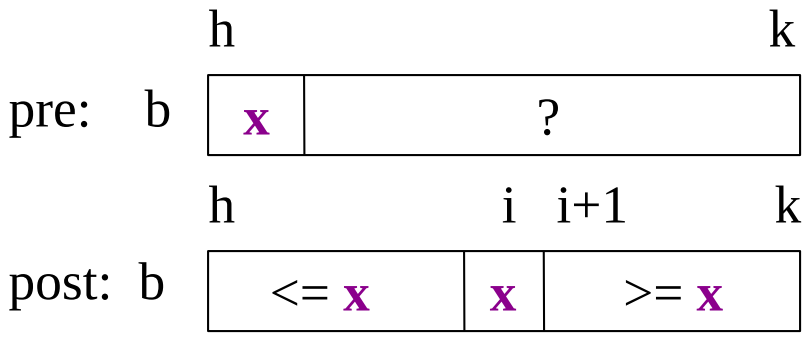
- Swap elements of b[h..k] and store in j to truthify post:



Recursive partitions = sorting

QuickSort

```
def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if  b[h..k] has fewer than 2 elements:
        return
    i = partition(b, h, k)
    # b[h..i–1] <= b[i] <= b[i+1..k]
    # Sort b[h..i–1]  and  b[i+1..k]
    quick_sort (b, h, i–1)
    quick_sort (b, i+1, k)
```



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]

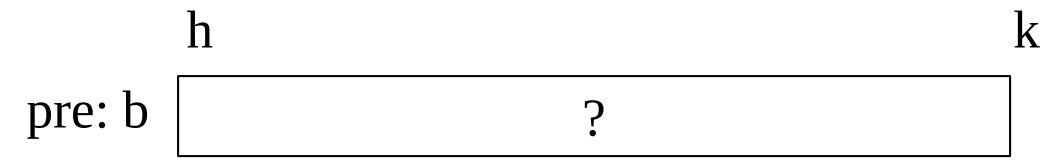
  2. i = k OR v = b[i]

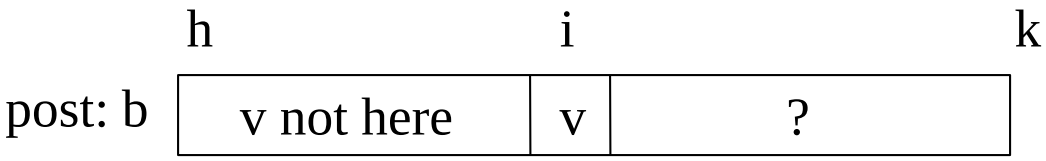
Linear Search

- **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]



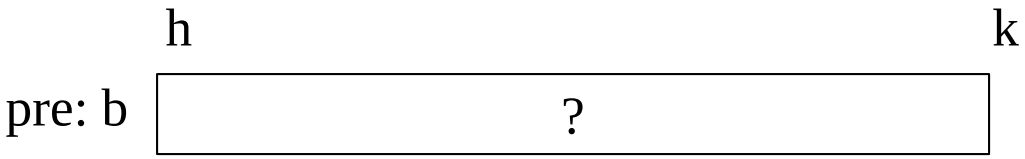


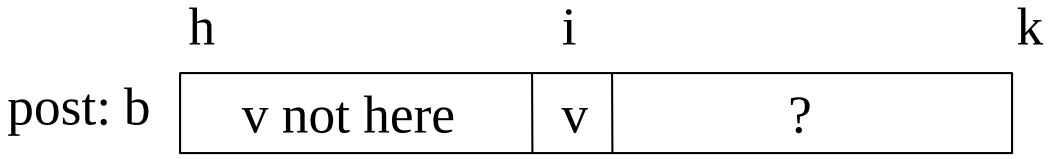
Linear Search

- **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]

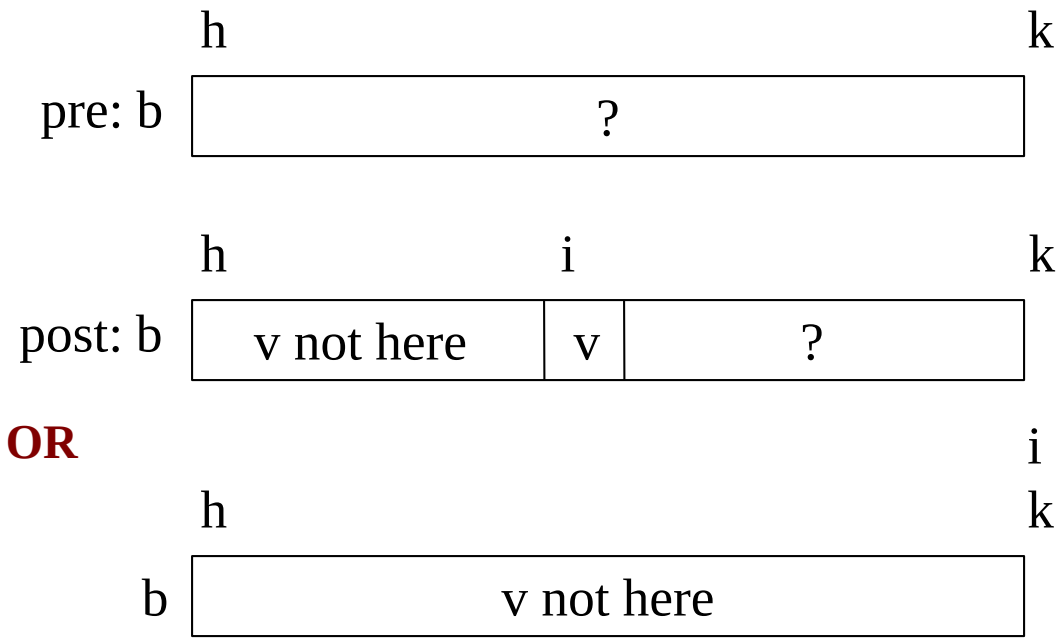
2. i = k OR v = b[i]

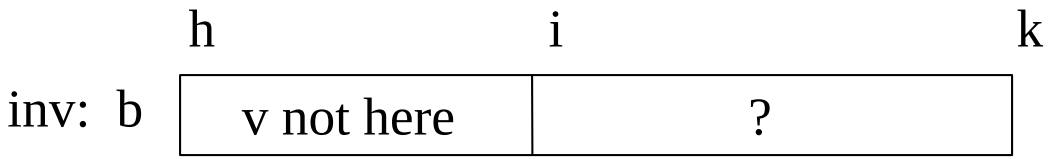




OR

Linear Search





Linear Search

```
def linear_search(b,v,h,k):
    """Returns: first occurrence of v in b[h..k-1]"""
    # Store in i index of the first v in b[h..k-1]
    i = h

    # invariant: v is not in b[0..i-1]
    while i < k and b[i] != v:
        i = i + 1

    # post: v is not in b[h..i-1]
    #          i >= k or b[i] == v
    return i if i < k 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?

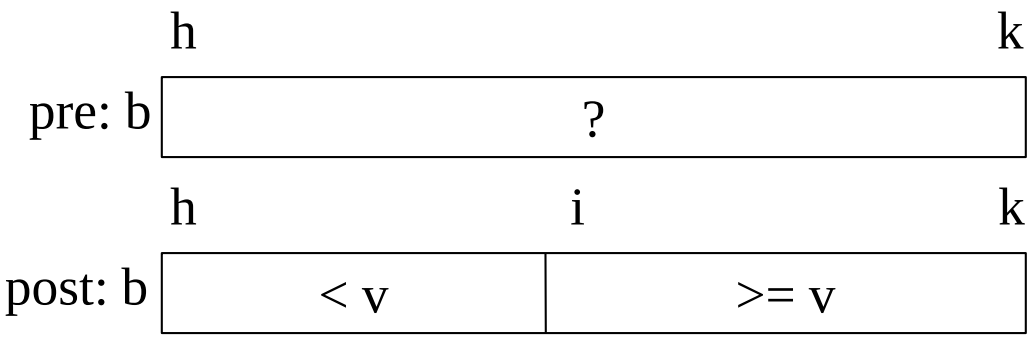
4. Does the repetend keep the invariant **inv** true?

Binary Search

- Look for v in **sorted** sequence segment b[h..k].

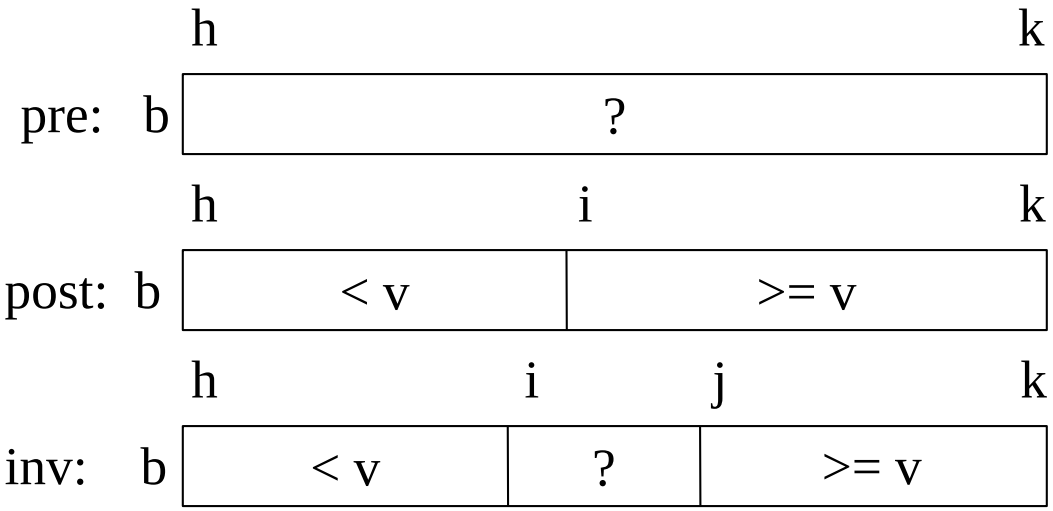
Binary Search

- Look for v in **sorted** sequence segment b[h..k].
  - Precondition: b[h..k-1] is sorted (in ascending order).
  - Postcondition: b[h..i-1] < v and v <= b[i..k]

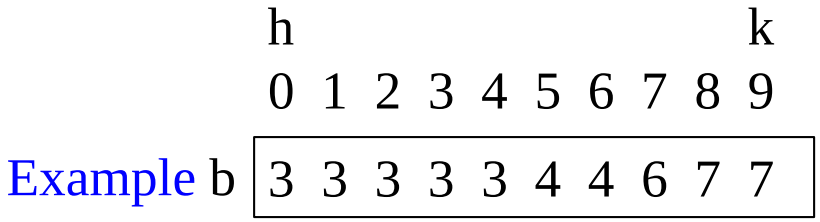


Binary Search

- Look for value v in **sorted** segment b[h..k]

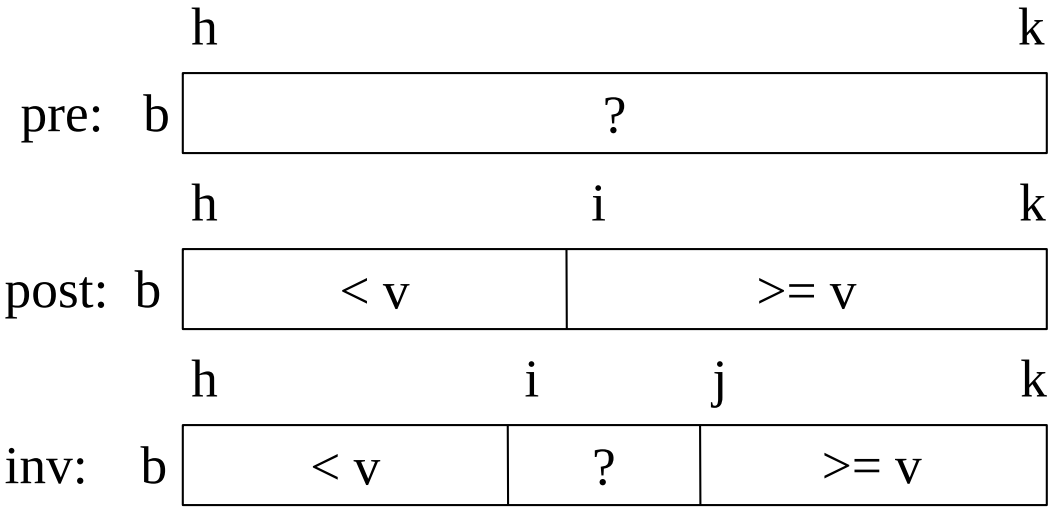


Called binary search because each iteration of the loop cuts the array segment still to be processed in half



- if v is 3, set i to 0
- if v is 4, set i to 5
- if v is 5, set i to 7
- if v is 8, set i to 10

Binary Search




```
i = h;  j = k+1;
while i != j:
```

Looking at b[i] gives linear search from left.

Looking at b[j-1] gives linear search from right.

Looking at middle: b[(i+j)/2] gives binary search.

Binary Search

```
def bsearch(b, v):
    i = 0
    j = len(b)
    # invariant; b[0..i-1] < v, b[i..j-1] unknown, b[j..] >= v
    while i < j:
        mid = (i+j)/2
        if b[mid] < v:
            i = mid+1
        else: #b[mid] >= v
            j = mid

    if i< len(b) and b[i] == v:
        return i
    else:
        return -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 the invariant **inv** true?

Binary Search Recursive

```
def rbsearch(b, v):
    """ len(b) > 0 """
    return rbsearch_helper(b, v, 0, len(b))
```

```
def rbsearch_helper(b, v, i, j):
    if i >= j:
        if i < len(b) and b[i] == v:
            return i
        else:
            return -1

    mid = (i + j) / 2

    if b[mid] < v:
        return rbsearch_helper(b, v, mid + 1, j)
    else: # b[mid] >= v
        return rbsearch_helper(b, v, i, mid)
```