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

Lecture 27: Searching

CS 1110

Introduction to Computing Using Python

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

Today's Plan of Attack

- Linear Search
- Binary Search
- Optional Binary Search Appendix

Linear Search Definition

h k

PRE: b | ? |

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

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

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

h i k

POST: b | v not here | v | ? |

OR

h k=i

b | v not here |

Linear Search: What's the Invariant?

	h		k
PRE: b	?		

Store an integer in **i** to make this post-condition true:

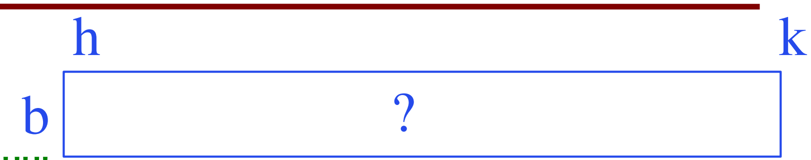
	h	i	k
INV: b	v not here	?	

	h	i		k
POST: b	v not here	v	?	

OR

	h	k=i
b	v not here	

Implementing Linear Search

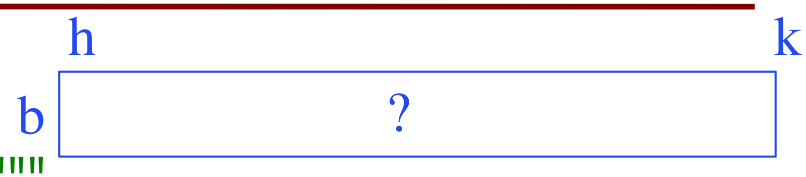



```
def linear_search(b,v,h,k):
    """Returns: first occurrence of v in b[h..k-1]"""
    # Store in i index of the first v in b[h..k-1]
    i = h

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

    # post: b[i] == v  OR
    #   v is not in b[h..i-1] and i >= k
    return i if i < k else -1
```

Analyzing 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: b[i] == v  OR
    #   v is not in b[h..i-1] and i >= k
    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?

How Fast is Linear Search?

No surprise: it's Linear!

(requires n steps to search though n elements)

What does linear time mean?

A: if you double the size of the list to 2n, it takes the original amount of time (~ n steps) to search for v

B: if you double the size of the list to 2n, it takes twice as long (~ 2n steps) to search for v

C: I don’t know

What if our list were sorted?

Then we could do **Binary Search**

Binary Search

Looking for the value **v** in a sorted list?

- Peek at the **middle** element of the list **m**
  - **v == x ?** Done!
  - **v > x ?** Go check the front half
  - **v < x** ? Go check the back half

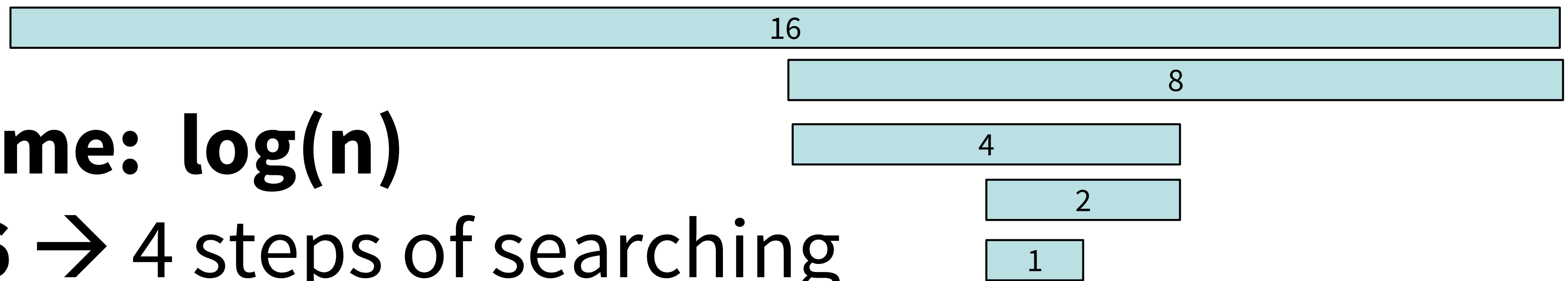
Example:

looking for 15? 15 > 8 → look in the 2nd half of list

1	2	3	5	6	7	8	12	15	21	33	37	38

How Fast is Binary Search?

With each step your list is cut in half.



Runtime: log(n)

n = 16 → 4 steps of searching

What does log(n) time mean?

A: if you double the size of the list to 32, it takes only the same time (~ 4 steps) to search for v

B: if you double the size of the list to 32, it takes twice as long (~ 8 steps) to search for v

C: if you double the size of the list to 32, it takes only slightly longer (~ 5 steps) to search for v

D: I don’t know

Is it worth it to sort the list?

Depends on how often you'll need to search it.

(Do we actually sort your Exams? Sort of…)

This is only the beginning of your foray into algorithm design and efficiency!

CS 1110 MATERIAL STOPS HERE!

CONGRATULATIONS!

(don't leave yet)

Appendix: Binary Search Details

You are **not** responsible for knowing the details of the following slides but they are a good (but difficult*) case study of how to develop an algorithm using loop invariants

* certainly more difficult than anything we would ask you on the Final Exam

OPTIONAL

Q: Binary Search Examples

	h									k
	0	1	2	3	4	5	6	7	8	9
Example b	3	3	3	3	3	4	4	6	7	7

- if v is 3, set i to ___?
- if v is 4, set i to ___?
- if v is 5, set i to ___?
- if v is 8, set i to ___?

A: 0
B: 3
C: 5
D: 7
E: None of the Above

	h	i	k
POST: b	v not here	v	?

OR

	h	k=i
b	v not here	

OPTIONAL

A: Binary Search Examples

	h									k
	0	1	2	3	4	5	6	7	8	9
Example b	3	3	3	3	3	4	4	6	7	7

- 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

A: 0
B: 3
C: 5
D: 7
E: None of the Above

	h	i	k
POST: b	v not here	v	?

OR

	h	k=i
b	v not here	

OPTIONAL

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]

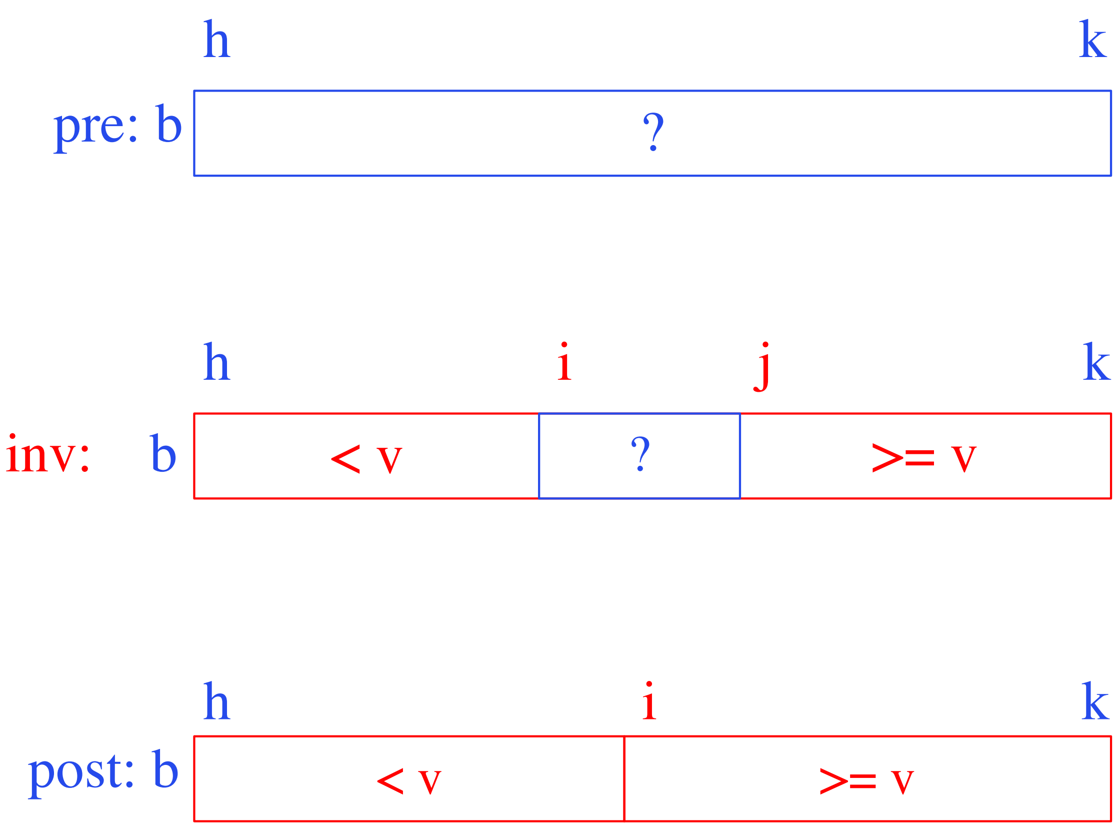
	h	k
pre: b	?	

	h	i	k
post: b	< v	>= v	

OPTIONAL

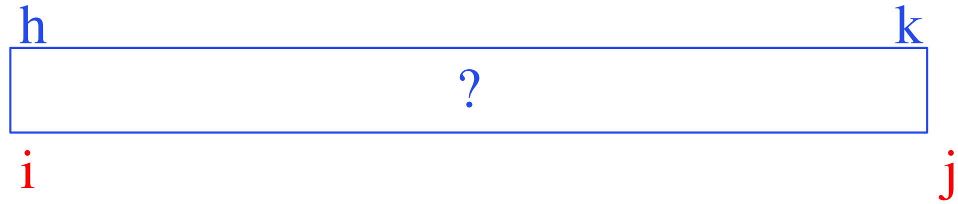
Binary Search: What's the Invariant?

- 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]

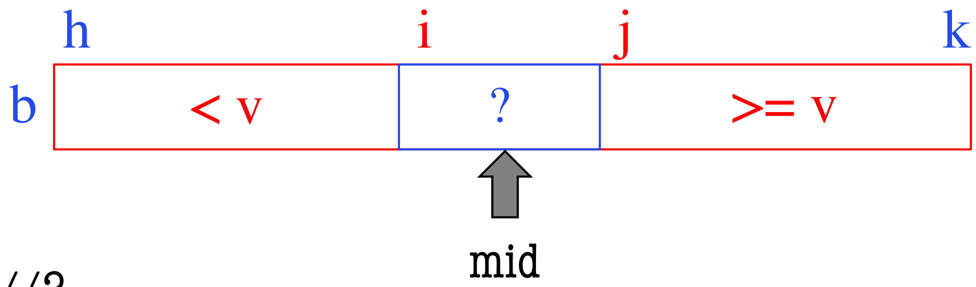


Called binary search because each iteration of the loop cuts the array segment still to be processed in half

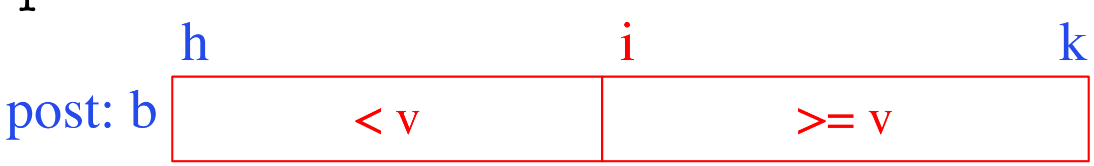
Implementing Binary Search




```
def bsearch(b, v):
    i = 0
    j = len(b)
```




```
    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
```



OPTIONAL

Analyzing 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?

OPTIONAL

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)
```

Notice that the recursive call needs more information than the original call, so we create a helper function and have it be recursive.