Lecture 27: Searching

CS 1110
Introduction to Computing Using Python

http://www.cs.cornell.edu/courses/cs1110/2019sp
Today’s Plan of Attack

• Linear Search
• Binary Search
• Optional Binary Search Appendix
Linear Search Definition

• **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]
Linear Search: What’s the Invariant?

Store an integer in \( i \) to make this post-condition true:

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{PRE: } b )</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{INV: } b )</td>
<td>v not here</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{POST: } b )</td>
<td>v not here</td>
<td>v</td>
</tr>
</tbody>
</table>

OR

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{b} )</td>
<td>v not here</td>
<td>v not here</td>
</tr>
</tbody>
</table>
Implementing Linear Search

```python
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
```

```
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 ($\sim n$ steps) to search for $v$

B: if you double the size of the list to $2n$, it takes twice as long ($\sim 2n$ 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? $15 > 8 \rightarrow$ look in the 2$^{nd}$ half of list
How Fast is Binary Search?

With each step your list is cut in half.

Runtime: \( \log(n) \)

\( n = 16 \rightarrow 4 \) 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!
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
Q: Binary Search Examples

Example

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- 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

POST: b

v not here

OR

b

v not here

k = i
### A: Binary Search Examples

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: \[ b \rightarrow [3, 3, 3, 3, 3, 4, 4, 6, 7, 7] \]

- 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

Options:

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

**POST:**

**b**

| v not here | v | ? |

**OR**

**b**

v not here
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 \( v \leq b[i..k] \)

pre: \( b \)

\[ \begin{array}{c}
  \text{h} \\
  \text{?}
  \end{array} \]

post: \( b \)

\[ \begin{array}{ccc}
  \text{h} & \text{i} & \text{k} \\
  < v & \text{ } & \geq v
  \end{array} \]
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]

pre: b[h..k]

post: b[h..i-1] < v and v <= b[i..k]

inv: 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

```python
def bsearch(b, v):
    i = 0
    j = len(b)
    while i < j:
        mid = (i+j)//2
        if b[mid] < v:
            i = mid+1
        else:
            j = mid
    if i < len(b) and b[i] == v:
        return i
    else:
        return -1
```

**Pre:**
- `b` is a list of elements.
- `h` is a placeholder for the high index.
- `k` is a placeholder for the low index.
- `i` is the current index.
- `j` is the midpoint of the search space.

**Invariants:**
- `b[h] < v` (high end of search space is before `v`)
- `b[k] >= v` (low end of search space is at or after `v`)

**Post:**
- `b[i]` is the value `v` if found.
- `b` is partitioned such that all elements to the left are less than `v` and all elements to the right are greater than or equal to `v`.

**Diagram:**
- The binary search algorithm is visualized with a diagram showing the partitioning of the search space.
- The algorithm iterates over the list, halving the search space at each step until the value is found or the partitioning condition is met.

This diagram illustrates how the binary search algorithm works, showing the partitioning of the search space and the iterative process of narrowing down the search范围.
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
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.