Recursion and Search

Oliver W. Layton

CS151: Computational Thinking: Visual Media

Lecture 37, Fall 2020

Friday November 20
Linear search

How could we search myList to determine whether it contains the element 99 *(we can't use the in reserved word)*?

myList = [10, 3, 2, 3, 9, 99, 100, 15, 16]

- We can step through the list, element-by-element, check to see if each matches the one we're looking for.

- How would we indicate if we didn't find it in the list?
  - Return a 'special value' to indicate failure (e.g. -1).

- There's no reason why this won't work on data. Example: Looking for a name (e.g. 'Jennifer') in a long list, our function would work just fine.

- Let's write the code: lecture37_linear_search.py
Problems with linear search

- Does the time it takes to find the element depend on its position in the list?
- If list is length $n$, what's the worst case number of items we have to check before we find our element
  - $n$
- What if we had a list double the length?
  - $2n$
- **Example:** If we had 10,000,000 items, it could take 10,000,000 checks to find the element in the worst case! What does worst case mean?
- The element isn't in the list, or it's at the last position.
A faster way to search (with a caveat)

Another search strategy can be much faster, but requires that the list is sorted!

Here is how to sort a list in Python:

```python
myList = [10, 3, 2, 3, 9, 99, 100, 15, 16]
# Ascending order:
sorted(myList)  # [2, 3, 3, 9, 10, 15, 16, 99, 100]
```
Binary search (1/4)

**Goal:** Search for 99.

[2, 3, 3, 9, 10, 15, 16, 99, 100]

- First, divide our list into two equal halves around the middle element.

[2, 3, 3, 9, **10**, 15, 16, 99, 100]

- Is the element we're searching for the middle element? If so, we're done!
- If the element we're searching for is smaller, continue the search on the left half.
- If the element we're searching for is bigger, continue the search on the right half.
Goal: Search for 99.

Whichever half we end up with, the half becomes the "new list" that we search.

[2, 3, 3, 9, **10**, **>15**, 16, 99, **100**]

Repeat search on "new list" until we find the element.

Let's keep searching for the 99... What's the new middle element (let's round down)?
Binary search (3/4)

[2, 3, 3, 9, 10, >>15, **16**, 99, 100<<]

• Is 99 the middle element?

• Is 99 bigger or smaller than the middle element?

• Let's update our "new list"...
Binary search (4/4)

\[2, 3, 3, 9, 10, 15, **16**, >>99, 100<<\]

What's the new middle element (let's round down)?

- \[2, 3, 3, 9, 10, 15, 16, >>**99**, 100<<\]
- Is 99 the middle element?
- YES! We're done! *When we find the item, we return its index (7).*
When the item is not in the list (1/2)

[2, 3, 3, 9, 10, 15, 16, >>**99***, 100<<]

• How will we know whether the element we're looking for is not in the list?

• **We can't subdivide list anymore.** Let's say we wanted to find 99.5 (not in list) and kept searching. What's the next middle element (let's round down)?

• 99! No change: [2, 3, 3, 9, 10, 15, 16, >>**99***, 100<<]

• Is the 99.5 the middle element?

• Is 99.5 bigger or smaller than the middle element?

• We look at right sublist: [2, 3, 3, 9, 10, 15, 16, **99***, >>100<<]
When the item is not in the list (2/2)

[2, 3, 3, 9, 10, 15, 16, **99**, >>100<<]

• New middle is 100: [2, 3, 3, 9, 10, 15, 16, 99, >>**100**<<]

• Is 99.5 bigger or smaller than 100?

• Smaller! We move the right endpoint down by own to search the "left sublist":

  • [2, 3, 3, 9, 10, 15, 16, 99<<, >>**100**]

• This is weird! But it gives us a good "base case" to give up searching!

• **We give up binary search when rightIdx < leftIdx.**
Binary search vs. linear search

• How many checks did we have to do for the 99?
• Only 3. We used 8 in linear search.
• What was the worst case runtime for linear search for a list length $n$?
  • $n$
• For binary search it is only $\log_2(n)$

**Linear search:** 10,000,000 items -> **10,000,000 checks** (worst case)
**Binary search:** 10,000,000 items -> **24 checks** (worst case)

---

1 The Log is base 2 because we subdivide the list into two parts each time. Binary numbers are called binary numbers because they are base 2 (two states: 0 or 1).
$n \text{ vs } \log(n)$
Practice binary search by hand

Use binary search on the following list to find the element 32.

\[11, 12, 17, 20, 21, 32, 33, 39, 43, 50]\]

At each step indicate the: middle index, left bound, right bound
Solution

• \([11, 12, 17, 20, **21**, 32, 33, 39, 43, 50]\)
• \([11, 12, 17, 20, **21**, >>32, 33, 39, 43, 50 <<]\)
• \([11, 12, 17, 20, 21, >>32, 33, **39**, 43, 50 <<]\)
• \([11, 12, 17, 20, 21, >>32, 33<<, **39**, 43, 50] \# MATCH!\)
• \([11, 12, 17, 20, 21, >>**32**, 33<<, 39, 43, 50] \# MATCH!\)
• \([11, 12, 17, 20, 21, >>**32**, 33<<, **39**, 43, 50] \# MATCH!\)
• \([11, 12, 17, 20, 21, >>**32**, 33<<, 39, 43, 50] \# MATCH!\)
Recursion and binary search

- **Checklist item #1**: Can the problem can be built up in terms of solutions to simpler versions of the problem.
  - Yes! The half will have the item (if it's in the list). Like running binary search on smaller initial list.

- **Checklist item #2**: Does calling $f(\cdot)$ from within $f(\cdot)$ get us closer to the stopping condition?
  - Yes! We keep making the list half as long

- **Checklist item #3**: Always ask yourself, what's the stopping condition? How will the recursion stop?
  - Yes! The list will subdivide until it's (less than) a single item long.
  - Instead of modifying the list, let's modify the leftInd and rightInd that define the boundaries within the list that we search in.
  - Let's implement recursive binary search!
Coding recursive binary search

Let's code it up!