Finding an item in a sequence
Search for item n in a sequence, seq. Return True if n is a member, else False.

Examples
'hello', 'i'
False
(4, 8, 2, 6), 8
True

Test-driven design

Finding an item in a sequence
def isIn(seq, t):
    ' ' '
    check each item in the sequence, compare to target (t) if item is the target, done (True)
    if item is not the target, keep looking at end of sequence, done (false)
def isIn(seq, t):
  """(sequence, item) -> boolean
  Search for item n in a sequence, seq.
  Return True if n is a member, else False.
  >>> isIn('hello', 'i')
  False
  >>> isIn((10, 20, 30, 40, 50, 60, 70, 80, 90), 80)
  True
  ...
  check each item in the sequence, compare to target (t)
  for item in seq:
      if item == t:
          found = True
      else:
          found = False
  return found

# exercise: what is needed to
# return index of found item?
for item in seq:
    if item == t:
        found = True  # developing code
    else:
        found = False
return found
Finding an item in a sequence

def isIn(seq, t):
    """(sequence, item) -> boolean
    Search for item n in a sorted sequence, seq.
    Return True if n is a member, else False.
    >>> isIn((10, 20, 30, 40, 50, 60, 70, 80, 90), 80)
    True
    """
    for item in seq:
        if item == t:
            return True
    return False

Binary Search: efficient search technique, as long as the list is already sorted.

Each step divides the remaining data into equal parts and discards one part:
If remaining part is empty, then done (not found).
Go to mid-point of remaining part and compare to target.
If mid-point is the target, then done (found).
Otherwise, keep the part of the list where n could be, and search that. (Discard the rest.)

For example,
nums = (1, 3, 4, 6, 8, 9, 11)
target value: 4

For example,
nums = (1, 3, 4, 6, 8, 9, 11, 13, 15, 25, 99, 100, 102)
target value: 42
CIS 210

For example,

nums = (1, 3, 4, 6, 8, 9, 11, 13, 15, 25, 99, 100, 102)
target value: 42

compare x to 11; x is larger, so repeat with
(13, 15, 25, 99, 100, 102)

compare x to 99; x is smaller, so repeat with (13, 15, 25)

compare x to 15; x is larger, so repeat with (25)

compare x to 25; not equal; done; false is returned

CIS 210

Sequential search  O(n)
Binary search  O(log n)

When n is 150 ...
When n is around 1000 ...
When n is around 1,000,000 ...

CIS 210

Should we sort the argument to the isIn function so we can always use binary search?

No. Good sort algorithms are O(nlogn) > O(n) linear search.

CIS 210

✓Binary Search
• A very short introduction to recursion
• Happy Thanksgiving!
Recall: Factorials

0! = 1
n! = n * (n-1)!

25

Factorials have a recursive (inductive) definition—elements in a set are defined in terms of other (more basic) elements in the set

0! = 1
n! = n * (n-1)!
= n * (n-1) * (n-2)!
= n * (n-1) * (n-2) * (n-3)!
= n * (n-1) * (n-2) * (n-3)! … * 0!

26

Factorials have a recursive (inductive) definition—elements in a set are defined in terms of other (more basic) elements in the set

0! = 1
n! = n * (n-1)!

4! = ??
4! = 4 * 3!
= 4 * 6
= 4 * (3 * 2)
= 4 * (3 * (2 * 1))
= 4 * (3 * (2 * (1 * 1)))

27

Factorials have a recursive (inductive) definition—elements in a set are defined in terms of other (more basic) elements in the set

0! = 1
n! = n * (n-1)!

4! = ??
4! = 4 * 3!
= 4 * 6
= 4 * (3 * 2)
= 4 * (3 * (2 * 1))
= 4 * (3 * (2 * (1 * 1)))
= 4 * (3 * (2 * (1 * 1))))

28

def factR(n):
    '''    '''
    if n == 0:
        # base case
        return 1
    else:
        # recursive call
        ??

29

30
def factR(n):
    if n == 0:  # base case
        return 1
    else:       # recursive call
        return n * factR(n-1)
CIS 210
What is recursion?
• defining things in simpler terms of themselves
• a function that calls itself
⇒ a problem solving approach where a task is divided into simpler and simpler versions of the original task (until it reaches one or more base cases)

CIS 210
Can binary search be defined recursively?

CIS 210
Can binary search be defined recursively?

CIS 210
Why recursion?
• elegant approach to problem solving for problems with a recursive structure
  underlying algorithm is clear – solutions (programs) are simpler to write, analyze, and understand
Computational Problem Solving: designing, implementing, checking, revising algorithms/programs.

Good programming style: function docstrings (type contract; description including parameters, returned value, and side effects); examples of function use, well-named variables, use of whitespace between operators and sections of code, judicious use of inline comments (why not what).

Python is a programming language and Python is an interpreter (program).

Python primitive elements: Objects - value/attributes, type, memory location (id).

Functions: type contract, parameter passing, variable scope, evaluation, side effects.

Naming values: Variables/assignment - assignment statements are not expressions and do not return a value.

Functions are an executable data type; what happens when a function is called.

Printing values: Variable assignment - assignment statements are not expressions and do not return a value.

Functions as arguments.

Iterative algorithms; accumulator pattern; Monte Carlo algorithms; data analysis; map and filter patterns; binary representation of characters; k-means cluster analysis; binary search.

Intro to recursive functions.

Systematic approaches to testing and debugging; automated testing; test-driven design.

What happens when an assignment statement is executed: memory allocation; reference semantics.

can binary search be defined recursively?

base case
length of sequence is 0 → return False
item in the middle of the sequence is match → return True

recursive rule
divide the sequence in half and recursively call binary search for the new sequence.