The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

Donald Knuth, The Art of Computer Programming, vol. 1

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
Finding an item in a sequence

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

Examples

<table>
<thead>
<tr>
<th>$seq$</th>
<th>$n$</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'hello', 'i'</td>
<td>(4, 8, 2, 6)</td>
<td>False</td>
</tr>
<tr>
<td>(4, 8, 2, 6), 8</td>
<td>(8, 2)</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>99</td>
<td>False</td>
</tr>
<tr>
<td>(4, 8, 2, 6), 2</td>
<td>(99)</td>
<td>True</td>
</tr>
</tbody>
</table>

Test-driven design
Finding an item in a sequence

```python
def isIn(seq, t):
    """ (seq: sequence, t: item) -> bool
    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
    ""
    found = False
    for item in seq:
        if item == t:
            # developing code
    return found
```

For item in seq:
  if item == t:
    # developing code
  return False

Finding an item in a sequence

```python
def isIn(seq, t):
    """ (seq: sequence, t: item) -> bool
    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
    ""
    for item in seq:
        if item == t:
            # developing code
    return False
```

Finding an item in a sequence

```python
def isIn(seq, t):
    """ (seq: sequence, t: item) -> bool
    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
    ""
    for item in seq:
        if item == t:
            # developing code
    return False
```

Finding an item in a sequence

```python
def isIn(seq, t):
    """ (seq: sorted sequence, t: item) -> bool
    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
        elif item > t:
            return False
    return False
```

Finding an item in a sorted sequence

```python
def isIn(seq, t):
    """ (seq: sorted sequence, t: item) -> bool
    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
    return False
```

Finding an item in a sorted sequence

```python
def isIn(seq, t):
    """ (seq: sorted sequence, t: item) -> bool
    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
    return False
```

Finding an item in a sorted sequence

```python
def isIn(seq, t):
    """ (seq: sorted sequence, t: item) -> bool
    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
    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.)
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,

- \( \text{nums} = \{1, 3, 4, 6, 8, 9, 11\} \)
- target value: \( 4 \)

For example,

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

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 ...

When \( n \) is 150 ...

When \( n \) is around 1000 ...

When \( n \) is around 1,000,000 ...

When \( n \) is 1,000,000 v. 20

When \( n \) is 1,000,000,000 v. 30
Should we sort the argument to the `isin` function so we can always use binary search?

**No.** Good sort algorithms are $O(n \log n) > O(n)$ linear search.

**Yes.** If we can sort once and use binary search many times.

---

Binary Search

A very short introduction to big-O notation

• A very short introduction to recursion

• Happy Thanksgiving!

---

Recall: Factorials

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!

---

Factorials have a **recursive (inductive)** definition – elements in a set are defined in terms of other 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!
Factorials have a recursive (inductive) definition – elements in a set are defined in terms of other (more basic) elements in the set.

\[ 0! = 1 \]
\[ n! = n \cdot (n-1)! \]

\[
\begin{align*}
4! &= 24 \\
4! &= 4 \cdot 3! \\
4! &= 4 \cdot (3 \cdot 2)! \\
4! &= 4 \cdot (3 \cdot (2 \cdot 1!)) \\
4! &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!))) \\
4! &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1)))
\end{align*}
\]

```python
def factR(n):
    '''problem has recurring pattern → recursive solution'''
    if n == 0:  # base case
        return 1
    else:  # recursive call
        return n * factR(n-1)

>>> factR(4)
```

```python
def factR(n):
    '''problem has recurring pattern → recursive solution'''
    if n == 0:  # base case
        return 1
    else:  # recursive call
        return n * factR(n-1)

>>> factR(4)
```
def factR(n):
    if n == 0:  # base case
        return 1
    else:  # recursive call
        return n * factR(n-1)

>>> factR(4)
37

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)

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
can binary search be defined recursively?

base case(s)

recursive rule
A Structured Approach to Computational Problem Solving

1. Review the project specification thoroughly.
2. Write examples of expected results for specified inputs.
3. Review specification, if needed.
4. Develop, review, and/or revise a problem-solving approach, using natural language, pseudocode (not Python code).
5. Check algorithm using your examples — revise algorithm, re- review specification, if needed.
6. Starting with the lowest level function:
   - Write the function header.
   - Write the function docstring — type contract.
   - Write the function docstring — brief description.
   - Write the function docstring — examples of use (use ones developed earlier).
   - Write the return statement.
7. Using tools from the Python toolkit, start writing the body of the function.
   - Test, fix, revise as needed.
   - Test using examples in the docstring, and then project specification, and then others.

CIS 210 Learning Outcomes

• Understand, develop, implement algorithms for computational problem solving.
• Use structured design and testing methods to develop and implement programs.
• Read, write, review, document, test, and debug code.
• Demonstrate robust mental models of data representation and code execution.
• Demonstrate good understanding of high-level programming languages.
• Introduce and/or implement a sampling of classic computer science problem domains and algorithms.