CIS 210

TODAY
• Binary Search Cont’d
• Recursion Cont’d
• Intro into user-defined classes
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,

\[ \text{nums} = (1, 3, 4, 6, 8, 9, 11) \]
\[ \text{target value: 4} \]
For example,

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

target value: 4

compare x to 6; x is smaller, so repeat with (1, 3, 4)

compare x to 3; x is larger, so repeat with (4)

compare x to 4; x == 4, so True is returned
For example,

```python
ums = (1, 3, 4, 6, 8, 9, 11, 13, 15, 25, 99, 100, 102)
target value: 42
```
For example,

\[ \text{nums} = (1, 3, 4, 6, 8, 9, 11, 13, 15, 25, 99, 100, 102) \]

target value: 42

compare \( x \) to 11; \( x \) is larger, so repeat with
\( (13, 15, 25, 99, 100, 102) \) (is 25 or 99 the midpoint?)

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
If \( n \) is number of items in sequence:

Sequential search \( \mathcal{O}(n) \)

Binary search \( \mathcal{O}(\log n) \)

When \( n \) is 150 ...

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

When \( n \) is around 1,000,000 ... 1,000,000,000 ...
Sequential search \( O(n) \)

Binary search \( O(\log n) \)

When \( n \) is 150 ... \( 150 \) v. 8

When \( n \) is around 1000 ... \( 1000 \) v. 10

When \( n \) is around 1,000,000 ... \( 1,000,000 \) v. 20 \( 1,000,000,000 \) v. 30
Should we sort the argument to the `isIn` function so we can always use binary search?
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.
✓ Binary Search
• A very short introduction to recursion
Recall: Factorials

0! = 1

n! = n * (n-1)!
Factorials have a recursive (inductive) definition – elements in a set are defined in terms of other elements in the set

\[ 0! = 1 \]

\[ n! = n \times (n-1)! \]
\[ = n \times (n-1) \times (n-2)! \]
\[ = n \times (n-1) \times (n-2) \times (n-3)! \]
\[ = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 0! \]
Factorials have a **recursive (inductive)** definition – elements in a set are defined in terms of other *(more basic)* elements in the set

\[
0! = 1 \\
n! = n \times (n-1)! \\
4! = 4 \times 3! \\
4! = 4 \times (3 \times 2!) \\
4! = 4 \times (3 \times (2 \times 1!)) \\
4! = 4 \times (3 \times (2 \times (1 \times 0!))) \\
4! = 4 \times (3 \times (2 \times (1 \times 1)))
\]
Factorials have a recursive (inductive) definition – elements in a set are defined in terms of other (more basic) elements in the set

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

\[ 4! = ?? \]
\[ 4! = 4 \times 3! \]
\[ 4! = 4 \times (3 \times 2!) \]
\[ 4! = 4 \times (3 \times (2 \times 1!)) \]
\[ 4! = 4 \times (3 \times (2 \times (1 \times 0!))) \]
\[ 4! = 4 \times (3 \times (2 \times (1 \times 1))) \]

\[ 4! = 24 \]
\[ 4! = 4 \times 6 \]
\[ 4! = 4 \times (3 \times 2) \]
\[ 4! = 4 \times (3 \times (2 \times 1)) \]
\[ 4! = 4 \times (3 \times (2 \times (1 \times 1))) \]
Factorials have a recursive (inductive) definition – elements in a set are defined in terms of other (more basic) elements in the set

\[ 0! = 1 \quad \# \text{ base case} \]
\[ n! = n \times (n-1)! \quad \# \text{ recursive (inductive) rule} \]

\[ 4! = 4 \times 3! = 4 \times 6 = 24 \]
\[ 3! = 3 \times 2! = 3 \times 2 = 6 \]
\[ 2! = 2 \times 1! = 2 \times 1 = 2 \]
\[ 1! = 1 \times 0! = 1 \times 1 = 1 \]
def factR(n):
    
    if n == 0:  # base case
        return 1
    else:       # recursive call
        ??
def factR(n):
    '''
    # base case
    if n == 0:
        return 1
    else:  # recursive call
        return n * factR(n-1)
def factR(n):
    '''
    
    if n == 0:  # base case
        return 1
    else:  # recursive call
        return n * factR(n-1)
def factR(n):
    '''    '''
    if n == 0:       # base case
        return 1
    else:            # recursive call
        return n * factR(n-1)
def factR(n):
    '''    '''
    if n == 0:    # base case
        return 1
    else:         # recursive call
        return n * factR(n-1)
def factR(n):
    '''
    # base case
    return 1
    # recursive call
    return n * factR(n-1)
def factR(n):
    '''    '''
    if n == 0:  # base case
        return 1
    else:  # recursive call
        return n * factR(n-1)
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)
can binary search be defined recursively?
can binary search be defined recursively?

base case(s)
??

recursive rule
??
can binary search be defined recursively?

base case
length of sequence is 0 \(\rightarrow\) ??
item in the middle of the sequence is match \(\rightarrow\) ??

recursive rule
??
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
assignment statement  
expressions  
Python repetition – for, while  
Python conditionals – if

numeric data types (int, float) and operations (e.g., +, **, round, abs)  
Boolean data type and operations (e.g., <, and)  
string data type and operations (e.g., +, len, count, find, format), formatted strings  
Python collections data types and operations – tuples, lists, dictionaries  
Python files and file processing  
data type “coercion” functions (e.g., str, int, list, float)  
NoneType (None)  
print/input

Python Standard Library – math, turtle, random modules; import (if __name__ == ‘__main__’)  
user-defined functions; function design; docstrings  
IDLE interactive development environment  
Python introspection: help, dir, type, id  
run-time checking of data and code: assert; try/except
Programming/Computer Science Concepts

Computational Problem Solving: designing, implementing, checking, revising algorithms/pgms.

Good programming style: function docstrings (type contract; description including parameters, returned value, and side effects if any; 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 Shell is a REPL (read-evaluate-print loop)
Python primitive elements: Objects - value/attributes, type, memory location (id)
Combining primitive elements: Expressions - expressions evaluate to a value; overloaded operators; methods/functions/operators; short circuit evaluation of boolean expressions
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces – builtins and global (__main__); scope; dynamic typing; reference semantics.
Other language considerations – strong typing, mutable and immutable data types
Functions are an executable data type; what happens when a function is called:
  Activation record/stack frame added to call stack for local namespace; return address
  Call-by-assignment parameter passing
  Functions always return a value (sometimes None)
  Functions sometimes have side effects
Functions as arguments
Iterative algorithms; accumulator pattern; Monte Carlo algorithms; data analysis; map and filter patterns
parity bits; 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
A Structured Approach to Computational Problem-Solving

-- review the project specification thoroughly
-- write examples of expected results for specified inputs – re-review spec, if needed
-- develop, review, and/or revise a problem-solving approach, using natural language, algorithm, pseudocode (not Python code)
-- check algorithm using your examples – revise algorithm, re-review spec, if needed

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

-- using tools from the Python toolkit, start writing the body of the function
-- test often, revise as needed
-- test using examples in the docstring, and then project spec, 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, revise, document, test, and debug code;

• demonstrate robust mental models of data representation and code execution;

• demonstrate good understanding of a high level programming language;

• introduce and/or implement a sampling of classic computer science problem domains and algorithms.
SOME HELP:  
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
CIS 210

Finishing / Starting

• Intro to user-defined classes
• Programming environments

NEXT WEEK:

• Summing up
• Final Exam Review – all labs (Q/A)

“When you express your understanding in code, you debug your brain.”
Python User-Defined Classes
A Short Intro

Recall: Python data types (objects):

- type – range of values and operations
- value(s)
- id
>>> y = 'abc'
>>> type(y)
<class 'str'>
>>> y
'abc'
>>> id(y)
4320804568
>>> y.count('a')
??

>>> y = str(97403)
>>> type(y)
<class 'str'>
>>> y
'97403'
>>> id(y)
4298926768
>>> y - 4
??
A **type**, or **class**, is a template for objects.

**Objects** are **instances** of a class.

Every Python object is an instance of a class:

```python
g>>> y = str(97403) # y is an instance of class str
>>> type(y) # returns class info
<class 'str'>
>>> y # returns value info
'97403'
>>> id(y) # returns memory location info
4298926768
>>> y - 4 # class/type restricts operations
TypeError
```
>>> y = str(97403)
>>> type(y)
<class 'str'>
>>> y
'97403'
>>> id(y)
4298926768

• str is a **constructor** method for class str

• str() **instantiates** a str object

• y is an **instance (object)** of class (type) str with **value** and **id**
class Turtle

>>> t1 = Turtle()
>>> t2 = Turtle()

• Turtle is a constructor method for class Turtle

• Turtle() instantiates a Turtle object

• t1 and t2 are instances of the class Turtle (Turtle objects)
CIS 210

```python
>>> t1
<turtle.Turtle object at 0x102692a90>

>>> type(t1)
<class 'turtle.Turtle'>

>>> id(t1)
4335413904
```
CIS 210

```python
>>> t1.heading()
>>> t1.pos()
>>> t2.heading()
>>> t2.pos()
>>> t1.color()
>>> t2.shape()

>>> t1.fd(100)
>>> t2.fd(200)
```
Recall: Python data types (objects):

- **type**
  - range of values → **attributes**
    - object methods (operations)
    - object descriptors

- value(s)

- id
>>> dir(Turtle)
>>> dir(t1)

>>> t1.heading()
>>> t1.pos()
>>> t2.heading()
t1.shape('turtle')
>>> t2.pos()
>>> t1.color()
>>> t2.shape()

>>> t1.seth(75)
>>> t1.setpos(0, 0)

>>> t2.color('blue')
primitive objects – single value

collections objects – strings, lists, tuples, dicts

other Python objects – functions, turtle

user-defined objects
Classes and Objects

Creating our own classes –

Another way to extend the language as with functions: names encapulates

now data types: **Object-oriented programming**
Classes and Objects

Creating our own data types, or classes –

To create our own objects, or instances –

Example: Point

```python
>>> class Point(object):
    """Represents a point in 2-d space."""

>>> p1 = Point()
>>> p1
<__main__.Point object at 0x15695d0>
```
Classes and Objects

special constructor method \_\_init\_\_ initializes class names and values:

for example, x and y

class Point():
    """Represents a point in 2-d space."""
    def \_\_init\_\_(self, x = 0, y = 0):
        self.x = x
        self.y = y

>>> p1 = Point()
>>> p2 = Point(3, 4)
special constructor method __init__ initializes class names and values:

class Point():
    '''Represents a point in 2-d space.'''

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

>>> p1 = Point()
>>> p2 = Point(3, 4)
Add methods to **get** and **set (mutate)** attribute values:

```python
class Point(object):
    '''Represents a point in 2-d space.'''

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def getx(self):
        return self.x

    def gety(self):
        return self.y
```

```python
>>> p2 = Point(3, 4)
>>> p2.getx()
4
>>> p2.gety()
4
```
Classes and Objects

```python
def getx(self):
    return self.x

>>> p1 = Point()
0

>>> p2 = Point(3, 4)
>>> p1.getx()     >>> Point.getx(p2)
0              3

>>> p1.sety(100)

>>> p1.gety()     >>> p1.gety()
100
```

Classes and Objects

class Point(object):
    """Represents a point in 2-d space."""

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def getx(self):
        return self.x

    def setx(self, newx):
        self.x = newx
        return None

    def gety(self):
        return self.y

    def sety(self, newy):
        self.y = newy
        return None
Classes and Objects

```python
>>> p1 = Point()
```

```python
>>> type(p1)
<class '__main__.Point'>
```

```python
>>> p2 = Point(3, 4)
```

```python
>>> id(p2)
4389818152
```

```python
>>> p1.getx()
0
```

```python
>>> p1.sety(100)
```

```python
>>> p1.gety()
100
```

```python
>>> Point.getx(p2)
3
```

```python
>>> p1.gety()
100
```
Classes and Objects

Besides `__init__`, other special (‘magic’) methods: `__str__` and `__add__`, for example

```python
>>> ctr = 1
>>> ctr + 1  # >>> ctr.__add__(1)

>>> s = 'abc'
>>> s + 'def'  # >>> s.__add__('def')
```

`__add__` is a special method used by the `+` operator

its behavior varies per object type (overloaded operators)
Classes and Objects

Other special methods: __str__ and __add__, for example

__add__ is a special method to allow “+” to be used with the object

class Point(object):
    """Represents a point in 2-d space."""

    def __add__(self, i):
        self.x += i
        self.y += i
        return None
CIS 210

Classes and Objects

class Point(object):
    """Represents a point in 2-d space."""

    def __add__(self, i):
        self.x += i
        self.y += i
        return None

>>> p1 = Point()
>>> p1 + 99
>>> p1.getx()
99
>>> p1.gety()
99