Lecture 11
Object Oriented Programming

Dr. Muhammad Jawad Khan

Robotics and Intelligent Machine Engineering Department,
School of Mechanical and Manufacturing,
National University of Sciences and Technology,
Islamabad, Pakistan.

\

Reading for next week

Guttag, John. *Introduction to Computation and Programming Using Python: With Application to Understanding Data Second Edition*. MIT Press, 2016. ISBN: 9780262529624

Chapters 8, 8.1, 8.1.1 and 8.1.2

Quiz 5

- Load an image using open cv library
- Convert the image into numpy array
- Apply reshaping.
- Save the image

OBJECTS

- Python supports many different kinds of data

```
1234        3.14159        "Hello"        [1, 5, 7, 11, 13]

{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a string

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just "forget" about them
  - python system will reclaim destroyed or inaccessible objects – called "garbage collection"

WHAT ARE OBJECTS?

- objects are **a data abstraction** that captures…

(1) an **internal representation**

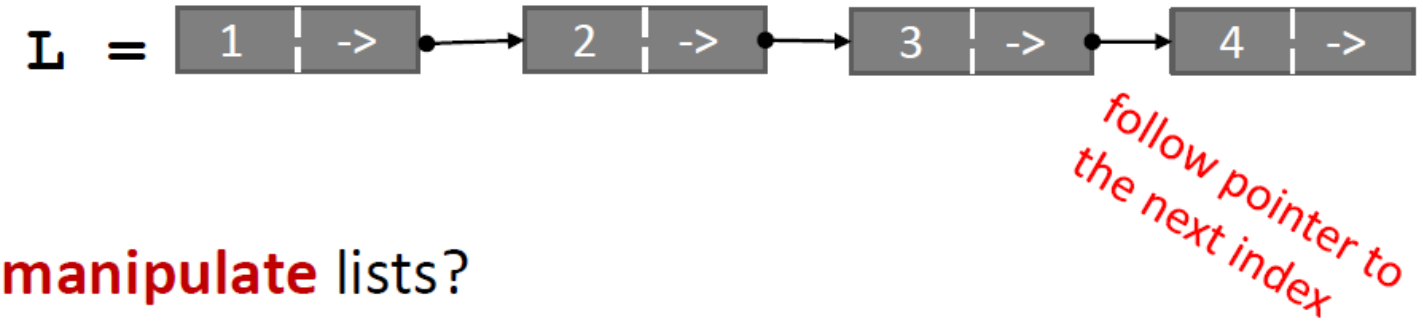
- through data attributes

(2) an **interface** for interacting with object

- through methods (aka procedures/functions)
- defines behaviors but hides implementation

EXAMPLE: [1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

ADVANTAGES OF OOP

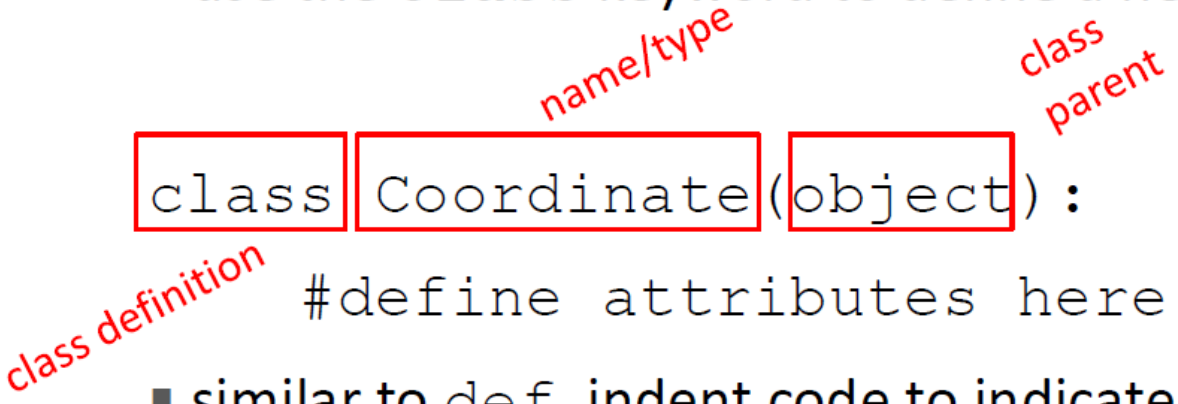
- **bundle data into packages** together with procedures that work on them through well-defined interfaces
- **divide-and-conquer** development
  - implement and test behavior of each class separately
  - increased modularity reduces complexity
- classes make it easy to **reuse** code
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- make a distinction between **creating a class** and **using an instance** of the class
- **creating** the class involves
  - defining the class name
  - defining class attributes
  - *for example, someone wrote code to implement a list class*
- **using** the class involves
  - creating new **instances** of objects
  - doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type



- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance next lecture)
  - `Coordinate` is a subclass of `object`
  - `object` is a superclass of `Coordinate`

WHAT ARE ATTRIBUTES?

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

special method to create an instance — is double underscore

what data initializes a `Coordinate` object

parameter to refer to an instance of the class

two data attributes for every `Coordinate` object

ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use `self` as the name of the first argument of all methods
- the "." **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

```
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def __str__(self):
        """ Returns a string representation of self """
        return "<" + str(self.x) + "," + str(self.y) + ">"
    def distance(self, other):
        """ Returns the euclidean distance between two points """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5

c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x, origin.x)
print(c.distance(origin))
print(Coordinate.distance(c, origin))
print(origin.distance(c))
print(c)
```

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class

a Coordinate class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

SPECIAL OPERATORS

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print self
```

... and others

EXAMPLE: FRACTIONS

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers
  - numerator
  - denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - add, subtract
  - print representation, convert to a float
  - invert the fraction

THE POWER OF OOP

- **bundle together objects** that share
  - common attributes and
  - procedures that operate on those attributes
- use **abstraction** to make a distinction between how to implement an object vs how to use the object
- build **layers** of object abstractions that inherit behaviors from other classes of objects
- create our **own classes of objects** on top of Python's basic classes

LAST TIME

- abstract data types through classes
- `Coordinate` example
- `Fraction` example

TODAY

- more on classes
  - getters and setters
  - information hiding
  - class variables
- inheritance

IMPLEMENTING THE CLASS vs USING THE CLASS

- write code from two different perspectives

implementing a new object type with a class

- **define** the class
- define **data attributes** (WHAT IS the object)
- define **methods** (HOW TO use the object)

using the new object type in code

- create **instances** of the object type
- do **operations** with them

CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- class name is the **type**

```
class Coordinate(object)
```

- class is defined generically
  - use `self` to refer to some instance while defining the class

```
(self.x - self.y)**2
```

  - `self` is a parameter to methods in class definition

- class defines data and methods **common across all instances**

- instance is **one specific** object

```
coord = Coordinate(1,2)
```

- data attribute values vary between instances

```
c1 = Coordinate(1,2)
c2 = Coordinate(3,4)
```

  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects

- instance has the **structure of the class**

WHY USE OOP AND CLASSES OF OBJECTS?

- mimic real life
- group different objects part of the same type

Jelly
1 year old
brown

5 years old
brown

Tiger
2 years old
brown

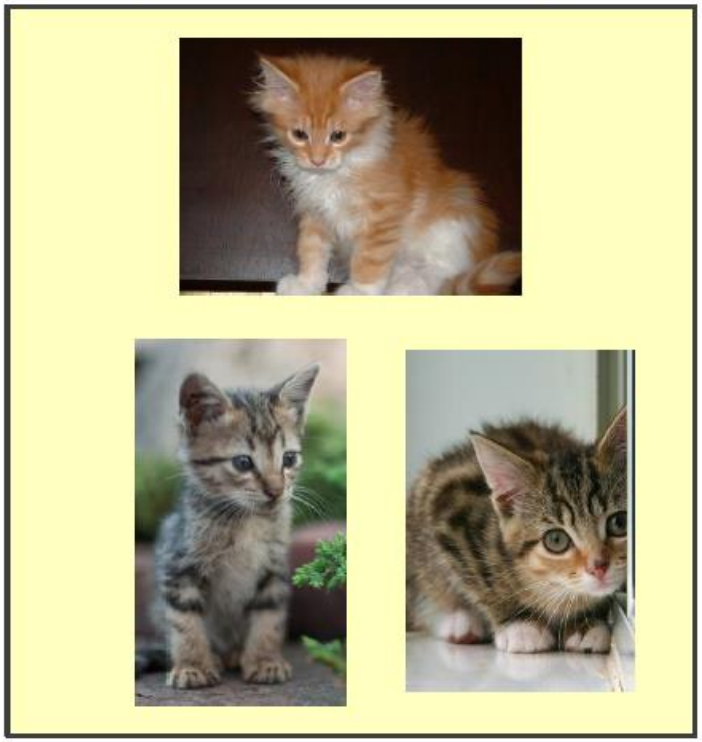
Bean
0 years old
black

2 years old
white

1 year old
b/w

WHY USE OOP AND CLASSES OF OBJECTS?

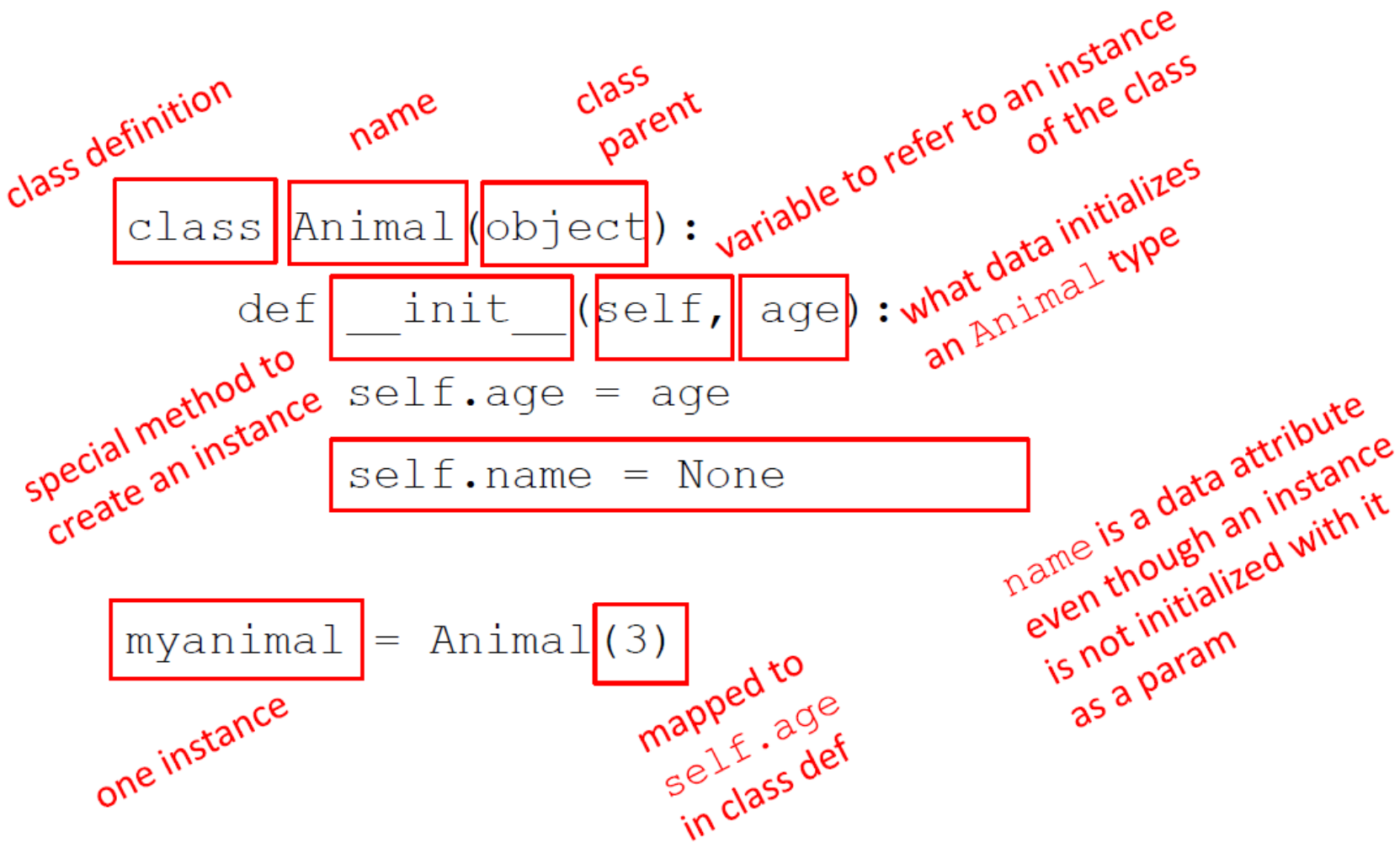
- mimic real life
- group different objects part of the same type



GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **data attributes**
  - how can you represent your object with data?
  - **what it is**
  - *for a coordinate: x and y values*
  - *for an animal: age, name*
- **procedural attributes** (behavior/operations/**methods**)
  - how can someone interact with the object?
  - **what it does**
  - *for a coordinate: find distance between two*
  - *for an animal: make a sound*

HOW TO DEFINE A CLASS (RECAP)



GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

- **getters and setters** should be used outside of class to access data attributes

AN INSTANCE and DOT NOTATION (RECAP)

- instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **dot notation** used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

- access data attribute
- allowed, but not recommended

```
a.get_age()
```

- access method
- best to use getters and setters

INFORMATION HIDING

- author of class definition may **change data attribute** variable names

replaced age data attribute by years

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- if you are **accessing data attributes** outside the class and class **definition changes**, may get errors

- outside of class, use getters and setters instead
  use `a.get_age()` NOT `a.age`
  - good style
  - easy to maintain code
  - prevents bugs

PYTHON NOT GREAT AT INFORMATION HIDING

- allows you to **access data** from outside class definition

```
print(a.age)
```

- allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- it's **not good style** to do any of these!

DEFAULT ARGUMENTS

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

```
print("\n---- animal tests ----")
a = Animal(4)
print(a)
print(a.get_age())
a.set_name("fluffy")
print(a)
a.set_name()
print(a)
```

HIERARCHIES

Animal

People

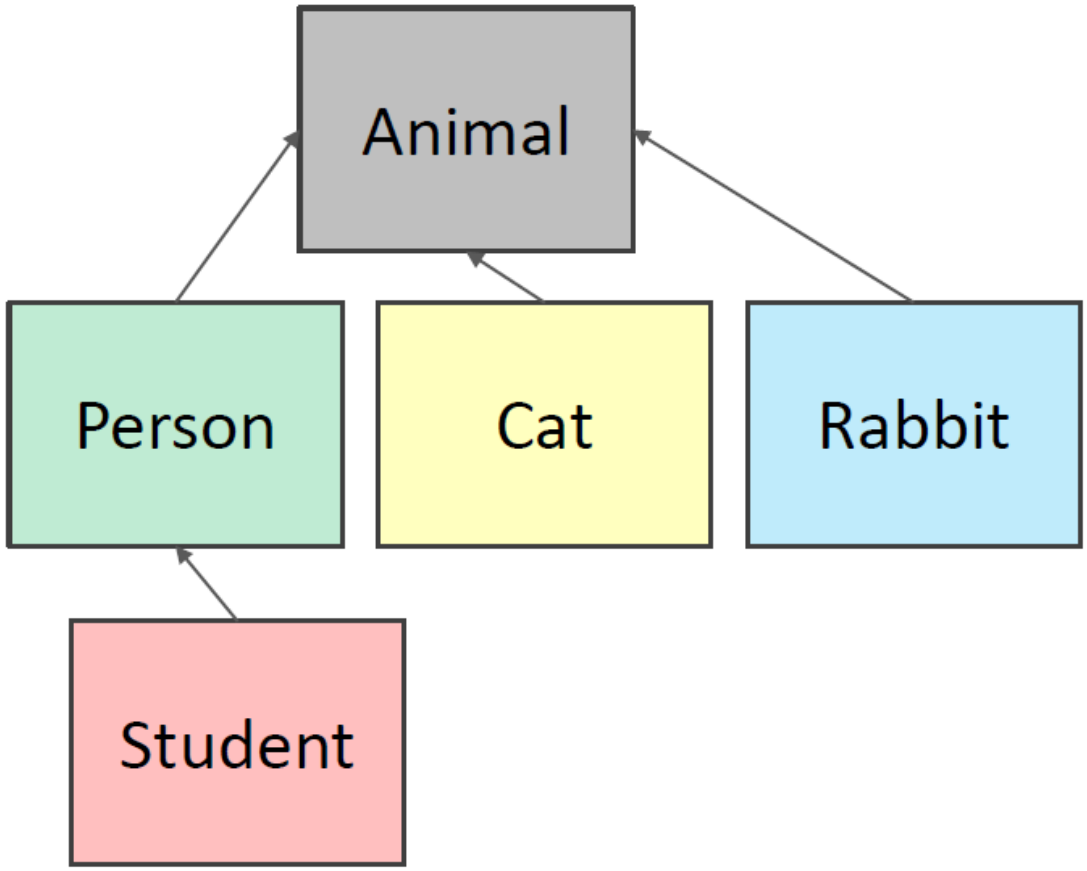
Student

Cat

Rabbit

HIERARCHIES

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior



INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

INHERITANCE: SUBCLASS

inherits all attributes of `Animal`:
`__init__()`
`age, name`
`get_age(), get_name()`
`set_age(), set_name()`
`__str__()`

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

add new functionality via speak method

overrides `__str__`

- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)


print("\n---- cat tests ----")
c = Cat(5)
c.set_name("fluffy")
print(c)
c.speak()
print(c.get_age())

#a.speak() # error because there is no speak method for Animal class
```

WHICH METHOD TO USE?

- subclass can have **methods with same name** as superclass
- for an instance of a class, look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

new methods

override Animal's __str__ method

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def speak(self):
        print("hello")
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

```
print("\n---- person tests ----")
p1 = Person("jack", 30)
p2 = Person("jill", 25)
print(p1.get_name())
print(p1.get_age())
print(p2.get_name())
print(p2.get_age())
print(p1)
p1.speak()
p1.age_diff(p2)
```

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from `random` class

inherits `Person` and `Animal` attributes

adds new data

- I looked up how to use the `random` class in the python docs
- `random()` method gives back float in [0, 1)

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
```

```
print("\n---- student tests ----")
s1 = Student('alice', 20, "CS")
s2 = Student('beth', 18)
print(s1)
print(s2)
print(s1.get_name(),"says:", end=" ")
s1.speak()
print(s2.get_name(),"says:", end=" ")
s2.speak()
```

CLASS VARIABLES AND THE `Rabbit` SUBCLASS

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

instance variable

access class variable

incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

Rabbit GETTER METHODS

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

method on a string to pad the beginning with zeros for example, 001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

WORKING WITH YOUR OWN TYPES

```
def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`

- define **+ operator** between two `Rabbit` instances
  - define what something like this does: `r4 = r1 + r2` where `r1` and `r2` are `Rabbit` instances
  - `r4` is a new `Rabbit` instance with age 0
  - `r4` has `self` as one parent and `other` as the other parent
  - in `__init__`, **`parent1` and `parent2` are of type `Rabbit`**

SPECIAL METHOD TO COMPARE TWO `Rabbits`

- decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

booleans

- compare ids of parents since **ids are unique** (due to class var)
- note you can't compare objects directly
  - for ex. with `self.parent1 == other.parent1`
  - this calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

```
class Rabbit(Animal):
    # a class variable, tag, shared across all instances
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        # zfill used to add leading zeroes 001 instead of 1
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
    def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

```
def __eq__(self, other):
    # compare the ids of self and other's parents
    # don't care about the order of the parents
    # the backslash tells python I want to break up my line
    parents_same = self.parent1.rid == other.parent1.rid \
                and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                    and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
 def __str__(self):
    return "rabbit:"+ self.get_rid()
```

```
print("\n---- rabbit tests ----")
print("---- testing creating rabbits ----")
r1 = Rabbit(3)
r2 = Rabbit(4)
r3 = Rabbit(5)
print("r1:", r1)
print("r2:", r2)
print("r3:", r3)
print("r1 parent1:", r1.get_parent1())
print("r1 parent2:", r1.get_parent2())


print("---- testing rabbit addition ----")
r4 = r1+r2   # r1.__add__(r2)
print("r1:", r1)
print("r2:", r2)
print("r4:", r4)
print("r4 parent1:", r4.get_parent1())
print("r4 parent2:", r4.get_parent2())
```

```
print("---- testing rabbit equality ----")
r5 = r3+r4
r6 = r4+r3
print("r3:", r3)
print("r4:", r4)
print("r5:", r5)
print("r6:", r6)
print("r5 parent1:", r5.get_parent1())
print("r5 parent2:", r5.get_parent2())
print("r6 parent1:", r6.get_parent1())
print("r6 parent2:", r6.get_parent2())
print("r5 and r6 have same parents?", r5 == r6)
print("r4 and r6 have same parents?", r4 == r6)
```

OBJECT ORIENTED PROGRAMMING

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

- computers are fast and getting faster – so maybe efficient programs don't matter?
  - but data sets can be very large (e.g., in 2014, Google served 30,000,000,000,000 pages, covering 100,000,000 GB – how long to search brute force?)
  - thus, simple solutions may simply not scale with size in acceptable manner
- how can we decide which option for program is most efficient?
- separate **time and space efficiency** of a program
- tradeoff between them:
  - can sometimes pre-compute results are stored; then use "lookup" to retrieve (e.g., memoization for Fibonacci)
  - will focus on time efficiency

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

Challenges in understanding efficiency of solution to a computational problem:

- a program can be **implemented in many different ways**
- you can solve a problem using only a handful of different **algorithms**
- would like to separate choices of implementation from choices of more abstract algorithm

HOW TO EVALUATE EFFICIENCY OF PROGRAMS

- measure with a **timer**
- **count** the operations
- abstract notion of **order of growth**

will argue that this is the most appropriate way of assessing the impact of choices of algorithm in solving a problem; and in measuring the inherent difficulty in solving a problem

TIMING A PROGRAM

- use time module
- recall that importing means to bring in that class into your own file
- **start** clock
- **call** function
- **stop** clock

```
import time

def c_to_f(c):
    return c*9/5 + 32

t0 = time.clock()
c_to_f(100000)
t1 = time.clock() - t0
Print("t =", t, ":", t1, "s,")
```

time.clock() is removed from python 3.8 and above use time.time() instead.

TIMING PROGRAMS IS INCONSISTENT

- GOAL: to evaluate different algorithms
- running time **varies between algorithms** ✓
- running time **varies between implementations** ✗
- running time **varies between computers** ✗
- running time is **not predictable** based on small inputs ✗

- time varies for different inputs but cannot really express a relationship between inputs and time ✗

COUNTING OPERATIONS

- assume these steps take **constant time**:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory
- then count the number of operations executed as function of size of input

```
def c_to_f(c):
    return c*9.0/5 + 32

def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

3 ops

1 op

loop x times

1 op

2 ops

mysum → 1+3x ops

COUNTING OPERATIONS IS BETTER, BUT STILL...

- GOAL: to evaluate different algorithms
- count **depends on algorithm** ✓
- count **depends on implementations** ✗
- count **independent of computers** ✓
- no clear definition of **which operations** to count ✗

- count varies for different inputs and can come up with a relationship between inputs and the count ✓

STILL NEED A BETTER WAY

- timing and counting **evaluate implementations**
- timing **evaluates machines**

- want to **evaluate algorithm**
- want to **evaluate scalability**
- want to **evaluate in terms of input size**

STILL NEED A BETTER WAY

- Going to focus on idea of counting operations in an algorithm, but not worry about small variations in implementation (e.g., whether we take 3 or 4 primitive operations to execute the steps of a loop)
- Going to focus on how algorithm performs when size of problem gets arbitrarily large
- Want to relate time needed to complete a computation, measured this way, against the size of the input to the problem
- Need to decide what to measure, given that actual number of steps may depend on specifics of trial

NEED TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

- want to express **efficiency in terms of size of input**, so need to decide what your input is
- could be an **integer**
  -- `mysum(x)`
- could be **length of list**
  -- `list_sum(L)`
- **you decide** when multiple parameters to a function
  -- `search_for_elmt(L, e)`

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- a function that searches for an element in a list

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- when e is **first element** in the list → BEST CASE
- when e is **not in list** → WORST CASE
- when **look through about half** of the elements in list → AVERAGE CASE
- want to measure this behavior in a general way

BEST, AVERAGE, WORST CASES

- suppose you are given a list `L` of some length `len(L)`
- **best case**: minimum running time over all possible inputs of a given size, `len(L)`
  - constant for `search_for_elmt`
  - first element in any list
- **average case**: average running time over all possible inputs of a given size, `len(L)`
  - practical measure
- **worst case**: maximum running time over all possible inputs of a given size, `len(L)`
  - linear in length of list for `search_for_elmt`
  - must search entire list and not find it

generally will focus on this case

ORDERS OF GROWTH

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth – as tight as possible
- do not need to be precise: **"order of" not "exact"** growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

MEASURING ORDER OF GROWTH: BIG OH NOTATION

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth

- **Big Oh or *O()*** is used to describe worst case
  - worst case occurs often and is the bottleneck when a program runs
  - express rate of growth of program relative to the input size
  - evaluate algorithm **NOT** machine or implementation

EXACT STEPS vs O()

```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

answer = answer * n

temp = n-1
n = temp

- computes factorial
- number of steps: 1 + 5n + 1
- worst case asymptotic complexity: O(n)
  - ignore additive constants
  - ignore multiplicative constants

WHAT DOES *O(N)* MEASURE?

- Interested in describing how amount of time needed grows as size of (input to) problem grows
- Thus, given an expression for the number of operations needed to compute an algorithm, want to know asymptotic behavior as size of problem gets large
- Hence, will focus on term that grows most rapidly in a sum of terms
- And will ignore multiplicative constants, since want to know how rapidly time required increases as increase size of input

SIMPLIFICATION EXAMPLES

- drop constants and multiplicative factors
- focus on **dominant terms**

$O(n^2)$: $n^2 + 2n + 2$

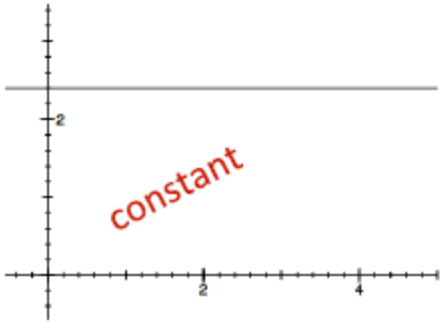
$O(n^2)$: $n^2 + 100000n + 3^{1000}$

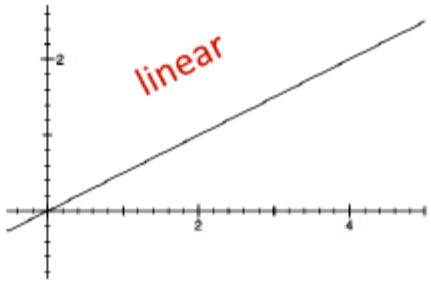
$O(n)$: $\log(n) + n + 4$

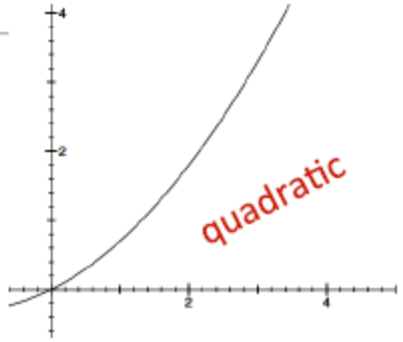
$O(n \log n)$: $0.0001*n*\log(n) + 300n$

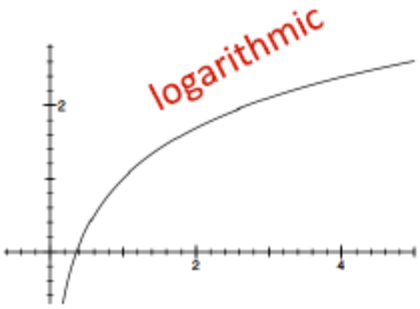
$O(3^n)$: $2n^{30} + 3^n$

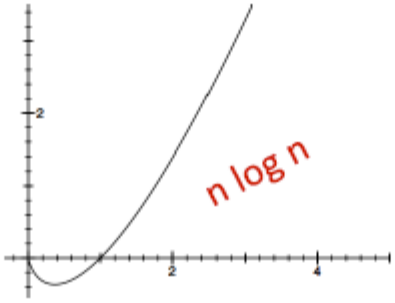
TYPES OF ORDERS OF GROWTH

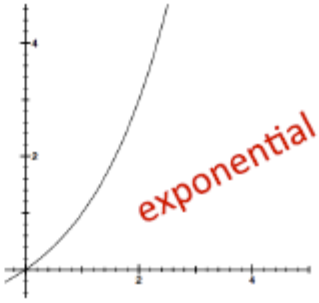












ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **combine** complexity classes
  - analyze statements inside functions
  - apply some rules, focus on dominant term

Law of Addition for O():

- used with **sequential** statements
- O(f(n)) + O(g(n)) is O(f(n) + g(n))
- for example,

```
for i in range(n):
    print('a')
for j in range(n*n):
    print('b')
```

O(n)

O(n*n)

O(n) + O(n*n)

is O(n) + O(n*n) = O(n+n^2) = O(n^2) because of dominant term

ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **combine** complexity classes
  - analyze statements inside functions
  - apply some rules, focus on dominant term

Law of Multiplication for O():

- used with **nested** statements/loops
- O(f(n)) * O(g(n)) is O(f(n) * g(n))
- for example,

```
for i in range(n):
    for j in range(n):
        print('a')
```

O(n)

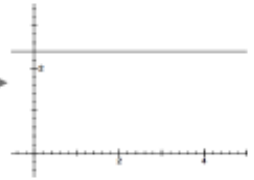
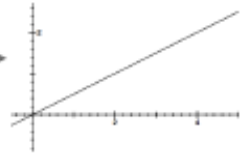
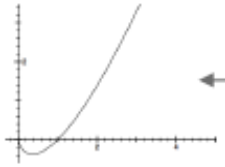
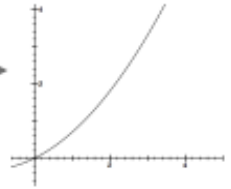
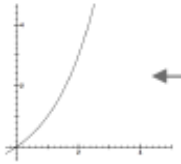
n loops, each O(n) → O(n)*O(n)

is O(n)*O(n) = O(n*n) = $O(n^2)$ because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

COMPLEXITY CLASSES

- *O(1)* denotes constant running time
- *O(log n)* denotes logarithmic running time
- *O(n)* denotes linear running time
- *O(n log n)* denotes log-linear running time
- $O(n^c)$ denotes polynomial running time (c is a constant)
- $O(c^n)$ denotes exponential running time (c is a constant being raised to a power based on size of input)

COMPLEXITY CLASSES ORDERED LOW TO HIGH

$O(1)$	:	constant 
$O(\log n)$	:	 logarithmic
$O(n)$	:	linear 
$O(n \log n)$	:	 loglinear
$O(n^c)$	:	polynomial 
$O(c^n)$	:	 exponential

c is a constant

COMPLEXITY GROWTH

CLASS	n=10	= 100	= 1000	= 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
O(n^2)	100	10000	1000000	1000000000000
O(2^n)	1024	1267650600228229401496703205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858127594672917553146825187145285692314043598457757469857480393456777482423098542107460506237114187795418215304647498358194126739876755916554394607706291457119647768654216766042983165262438683720566806937 6	Good luck!!

LINEAR COMPLEXITY

- Simple iterative loop algorithms are typically linear in complexity

LINEAR SEARCH ON UNSORTED LIST

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

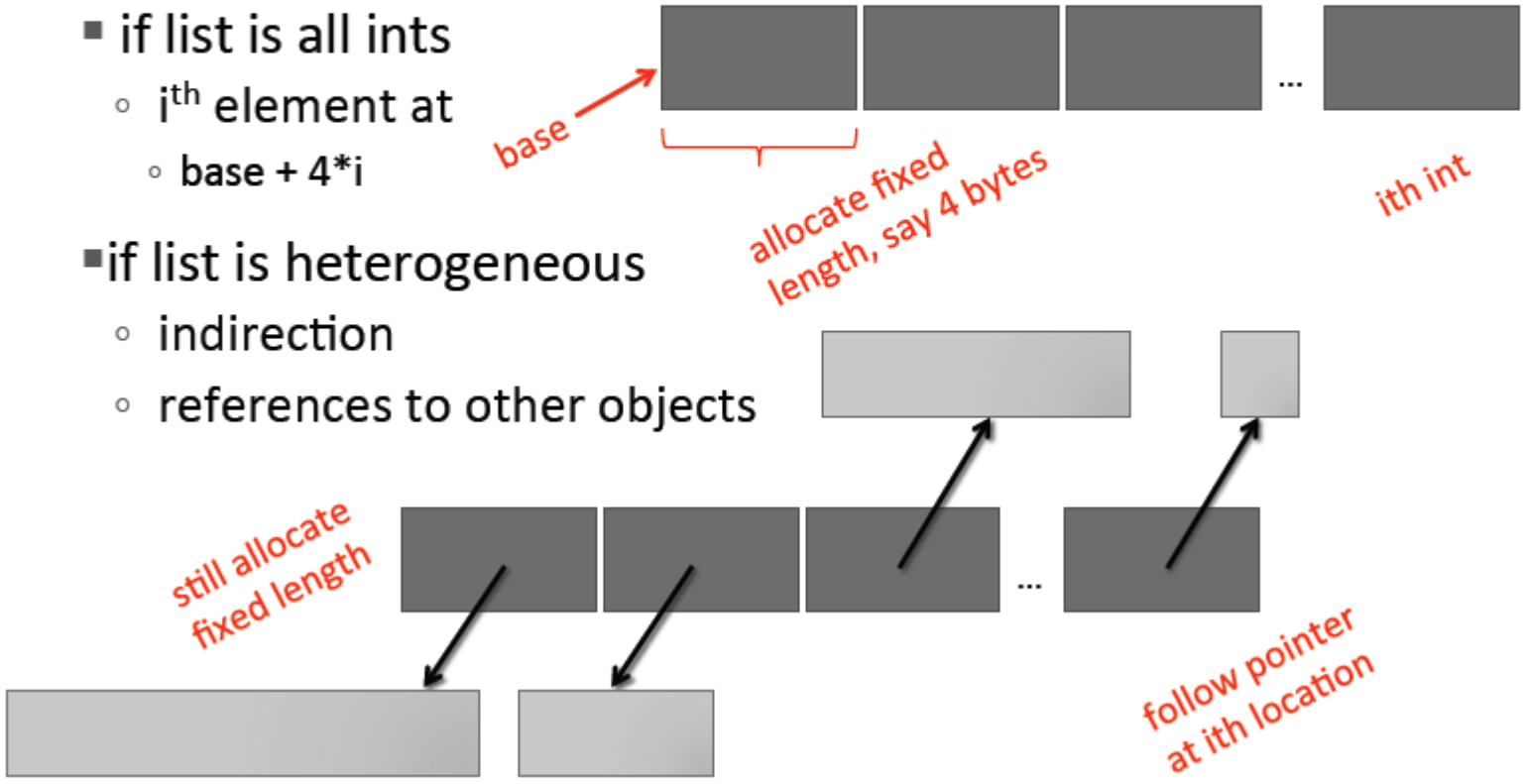
speed up a little by returning True here, but speed up doesn't impact worst case

- must look through all elements to decide it's not there
- O(len(L)) for the loop * O(1) to test if e == L[i]
  - O(1 + 4n + 1) = O(4n + 2) = O(n)
- overall complexity is **O(n) – where n is len(L)**

Assumes we can retrieve element of list in constant time

CONSTANT TIME LIST ACCESS

- if list is all ints
  - i^{th} element at
    - base + 4*i
- if list is heterogeneous
  - indirection
  - references to other objects



LINEAR SEARCH ON SORTED LIST

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- must only look until reach a number greater than e
- O(len(L)) for the loop * O(1) to test if e == L[i]
- overall complexity is **O(n) – where n is len(L)**
- **NOTE:** order of growth is same, though run time may differ for two search methods

worst case will need to look at whole list

LINEAR COMPLEXITY

- searching a list in sequence to see if an element is present
- add characters of a string, assumed to be composed of decimal digits

```
def addDigits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

- *O(len(s))*

LINEAR COMPLEXITY

- complexity often depends on number of iterations

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

- number of times around loop is n
- number of operations inside loop is a constant (in this case, 3 – set i, multiply, set prod)
  - O(1 + 3n + 1) = O(3n + 2) = O(n)
- overall just *O(n)*

NESTED LOOPS

- simple loops are linear in complexity
- what about loops that have loops within them?

QUADRATIC COMPLEXITY

determine if one list is subset of second, i.e., every element of first, appears in second (assume no duplicates)

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

QUADRATIC COMPLEXITY

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

outer loop executed len(L1) times

each iteration will execute inner loop up to len(L2) times, with constant number of operations

*O(len(L1)*len(L2))*

worst case when L1 and L2 same length, none of elements of L1 in L2

$O(len(L1)^2)$

QUADRATIC COMPLEXITY

find intersection of two lists, return a list with each element appearing only once

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

QUADRATIC COMPLEXITY

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

first nested loop takes *len(L1)*len(L2)* steps

second loop takes at most *len(L1)* steps

determining if element in list might take *len(L1)* steps

if we assume lists are of roughly same length, then

O(len(L1)^2)

O() FOR NESTED LOOPS

```
def g(n):
    """ assume n >= 0 """
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

- computes n^2 very inefficiently
- when dealing with nested loops, **look at the ranges**
- nested loops, **each iterating n times**
- ***O(n^2)***

THIS TIME AND NEXT TIME

- have seen examples of loops, and nested loops
- give rise to linear and quadratic complexity algorithms
- next time, will more carefully examine examples from each of the different complexity classes

CONSTANT COMPLEXITY

- complexity independent of inputs
- very few interesting algorithms in this class, but can often have pieces that fit this class
- can have loops or recursive calls, but ONLY IF number of iterations or calls independent of size of input

LOGARITHMIC COMPLEXITY

- complexity grows as log of size of one of its inputs
- example:
  - bisection search
  - binary search of a list

BISECTION SEARCH

- suppose we want to know if a particular element is present in a list
- saw last time that we could just "walk down" the list, checking each element
- complexity was linear in length of the list
- suppose we know that the list is ordered from smallest to largest
  - saw that sequential search was still linear in complexity
  - can we do better?

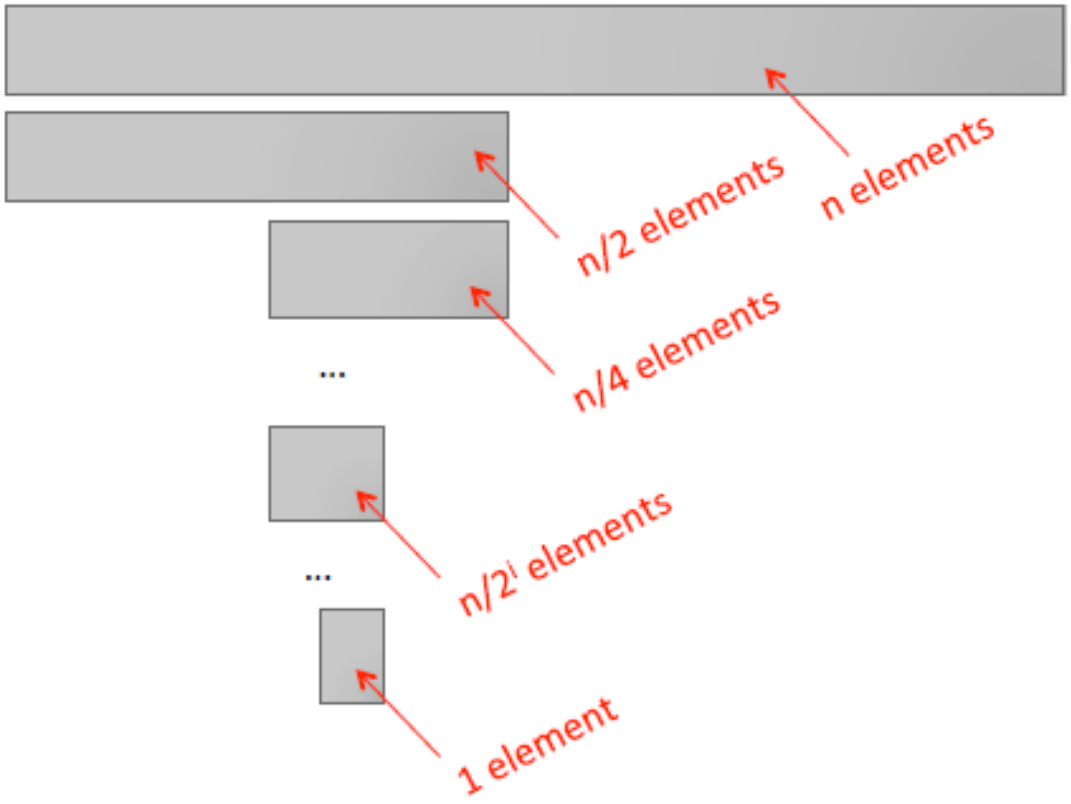
BISECTION SEARCH

1. pick an index, `i`, that divides list in half
2. ask if `L[i] == e`
3. if not, ask if `L[i]` is larger or smaller than `e`
4. depending on answer, search left or right half of `L` for `e`

A new version of a divide-and-conquer algorithm

- break into smaller version of problem (smaller list), plus some simple operations
- answer to smaller version is answer to original problem

BISECTION SEARCH COMPLEXITY ANALYSIS



- finish looking through list when

  $1 = n/2^i$

  so $i = \log n$

- complexity of recursion is **O(log n) – where n is len(L)**

BISECTION SEARCH IMPLEMENTATION 1

```
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1(L[:half], e)
        else:
            return bisect_search1(L[half:], e)
```

constant O(1)

constant O(1)

constant O(1)

NOT constant, copies list

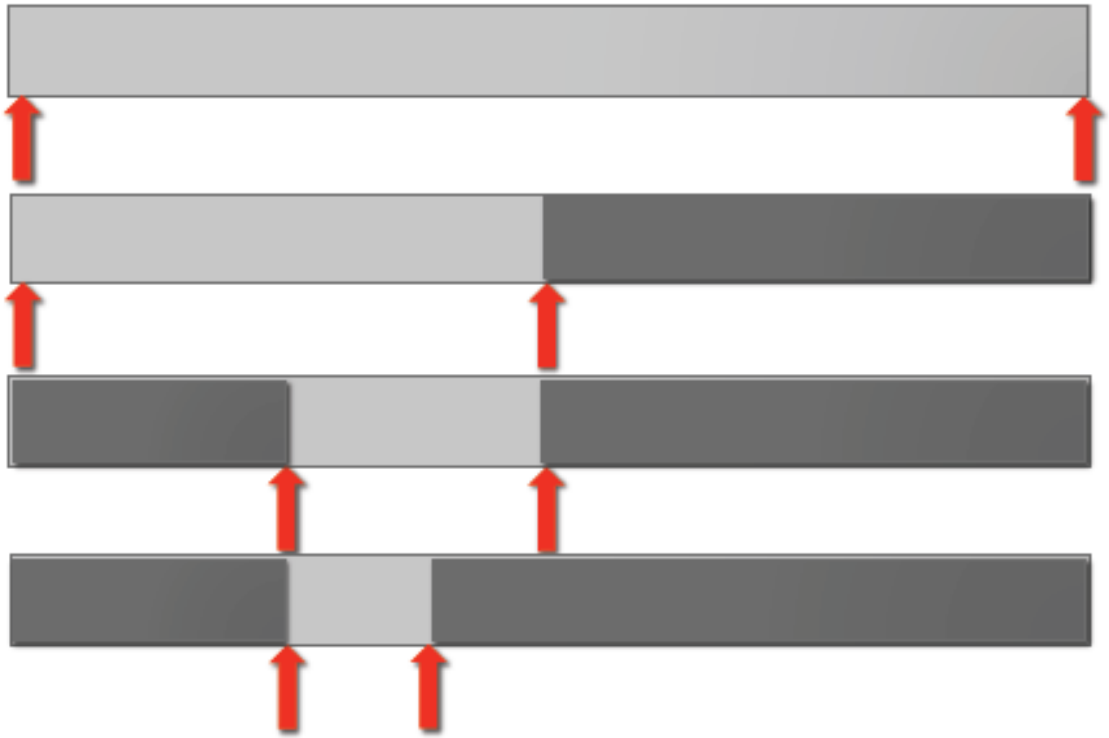
NOT constant

NOT constant

COMPLEXITY OF FIRST BISECTION SEARCH METHOD

- **implementation 1 – bisect_search1**
  - O(log n) bisection search calls
    - On each recursive call, size of range to be searched is cut in half
    - If original range is of size n, in worst case down to range of size 1 when n/(2^k) = 1; or when k = log n
  - O(n) for each bisection search call to copy list
    - This is the cost to set up each call, so do this for each level of recursion
  - O(log n) * O(n) → **O(n log n)**
  - if we are really careful, note that length of list to be copied is also halved on each recursive call
    - turns out that total cost to copy is **O(n)** and this dominates the log n cost due to the recursive calls

BISECTION SEARCH ALTERNATIVE



- still reduce size of problem by factor of two on each step
- but just keep track of low and high portion of list to be searched
- avoid copying the list
- complexity of recursion is again **O(log n) – where n is len(L)**

BISECTION SEARCH IMPLEMENTATION 2

```
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

constant other than recursive call

constant other than recursive call

COMPLEXITY OF SECOND BISECTION SEARCH METHOD

- **implementation 2 – bisect_search2** and its helper
  - O(log n) bisection search calls
    - On each recursive call, size of range to be searched is cut in half
    - If original range is of size n, in worst case down to range of size 1 when n/(2^k) = 1; or when k = log n
  - pass list and indices as parameters
  - list never copied, just re-passed as a pointer
  - thus O(1) work on each recursive call
  - O(log n) * O(1) → **O(log n)**

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10
    return result
```

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return result
```

only have to look at loop as no function calls

within while loop, constant number of steps

how many times through loop?

- how many times can one divide i by 10?
- *O(log(i))*

LINEAR COMPLEXITY

- saw this last time
  - searching a list in sequence to see if an element is present
  - iterative loops

O() FOR ITERATIVE FACTORIAL

- complexity can depend on number of iterative calls

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

- overall ***O(n)*** – n times round loop, constant cost each time

O() FOR RECURSIVE FACTORIAL

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

- computes factorial recursively
- if you time it, may notice that it runs a bit slower than iterative version due to function calls
- still ***O(n)*** because the number of function calls is linear in n, and constant effort to set up call
- **iterative and recursive factorial** implementations are the **same order of growth**

LOG-LINEAR COMPLEITY

- many practical algorithms are log-linear
- very commonly used log-linear algorithm is merge sort
- will return to this next lecture

POLYNOMIAL COMPLEXITY

- most common polynomial algorithms are quadratic, i.e., complexity grows with square of size of input
- commonly occurs when we have nested loops or recursive function calls
- saw this last time

EXPONENTIAL COMPLEXITY

- recursive functions where more than one recursive call for each size of problem
  - Towers of Hanoi
- many important problems are inherently exponential
  - unfortunate, as cost can be high
  - will lead us to consider approximate solutions as may provide reasonable answer more quickly

COMPLEXITY OF TOWERS OF HANOI

- Let t_n denote time to solve tower of size n
- $t_n = 2t_{n-1} + 1$
- $= 2(2t_{n-2} + 1) + 1$
- $= 4t_{n-2} + 2 + 1$
- $= 4(2t_{n-3} + 1) + 2 + 1$
- $= 8t_{n-3} + 4 + 2 + 1$
- $= 2^k t_{n-k} + 2^{k-1} + \ldots + 4 + 2 + 1$
- $= 2^{n-1} + 2^{n-2} + \ldots + 4 + 2 + 1$
- $= 2^n - 1$
- so order of growth is $O(2^n)$

Geometric growth

$a = 2^{n-1} + \ldots + 2 + 1$

$2a = 2^n + 2^{n-1} + \ldots + 2$

$a = 2^n - 1$

EXPONENTIAL COMPLEXITY

- given a set of integers (with no repeats), want to generate the collection of all possible subsets – called the power set
- {1, 2, 3, 4} would generate
  - {}, {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}, {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}
- order doesn't matter
  - {}, {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}, {4}, {1, 4}, {2, 4}, {1, 2, 4}, {3, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}

POWER SET – CONCEPT

- we want to generate the power set of integers from 1 to n
- assume we can generate power set of integers from 1 to n-1
- then all of those subsets belong to bigger power set (choosing not include n); and all of those subsets with n added to each of them also belong to the bigger power set (choosing to include n)
- {}, {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}, {4}, {1, 4}, {2, 4}, {1, 2, 4}, {3, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}
- nice recursive description!

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]] #list of empty list
    smaller = genSubsets(L[:-1]) # all subsets without
last element
    extra = L[-1:] # create a list of just last element
    new = []
    for small in smaller:
        new.append(small+extra)  # for all smaller
solutions, add one with last element
    return smaller+new  # combine those with last
element and those without
```

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

assuming append is constant time

time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

but important to think about size of smaller

know that for a set of size k there are 2^k cases

how can we deduce overall complexity?

EXPONENTIAL COMPLEXITY

- let t_n denote time to solve problem of size n
- let s_n denote size of solution for problem of size n
- $t_n = t_{n-1} + s_{n-1} + c$ (where c is some constant number of operations)
- $t_n = t_{n-1} + 2^{n-1} + c$
- $= t_{n-2} + 2^{n-2} + c + 2^{n-1} + c$
- $= t_{n-k} + 2^{n-k} + \ldots + 2^{n-1} + kc$
- $= t_0 + 2^0 + \ldots + 2^{n-1} + nc$
- $= 1 + 2^n + nc$

Thus computing power set is ***O(2^n)***

COMPLEXITY CLASSES

- $O(1)$ – code does not depend on size of problem
- $O(\log n)$ – reduce problem in half each time through process
- $O(n)$ – simple iterative or recursive programs
- $O(n \log n)$ – will see next time
- $O(n^c)$ – nested loops or recursive calls
- $O(c^n)$ – multiple recursive calls at each level

SOME MORE EXAMPLES OF ANALYZING COMPLEXITY

COMPLEXITY OF ITERATIVE FIBONACCI

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

constant O(1)

constant O(1)

linear O(n)

constant O(1)

- Best case:

  O(1)

- Worst case:

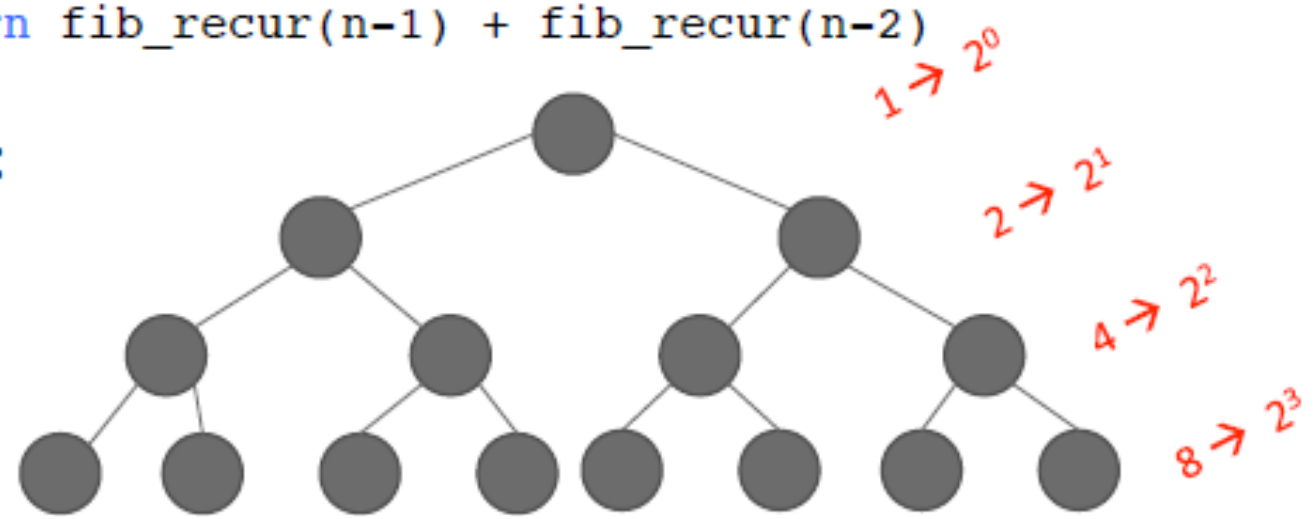
  O(1) + O(n) + O(1) → **O(n)**

COMPLEXITY OF RECURSIVE FIBONACCI

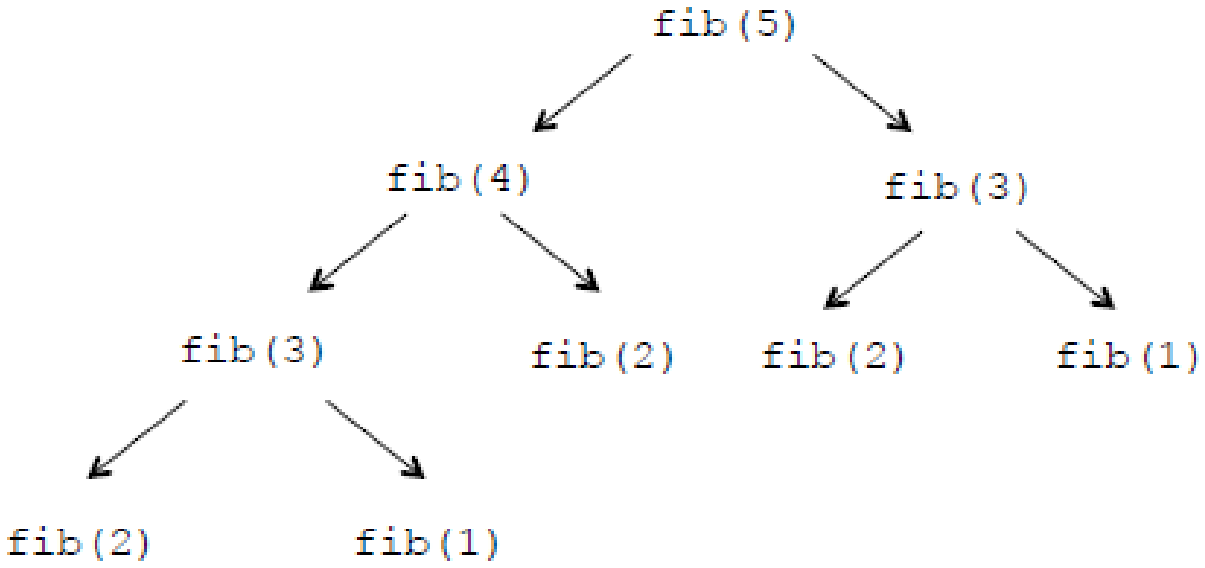
```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:

  $O(2^n)$



COMPLEXITY OF RECURSIVE FIBONACCI



- actually can do a bit better than 2^n since tree of cases thins out to right
- but complexity is still exponential

BIG OH SUMMARY

- compare **efficiency of algorithms**
  - notation that describes growth
  - **lower order of growth** is better
  - independent of machine or specific implementation

- use Big Oh
  - describe order of growth
  - **asymptotic notation**
  - **upper bound**
  - **worst case** analysis

COMPLEXITY OF COMMON PYTHON FUNCTIONS

- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)