PYTHON CLASSES and INHERITANCE

LAST TIME

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

Objective

- 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

1 year old
b/w

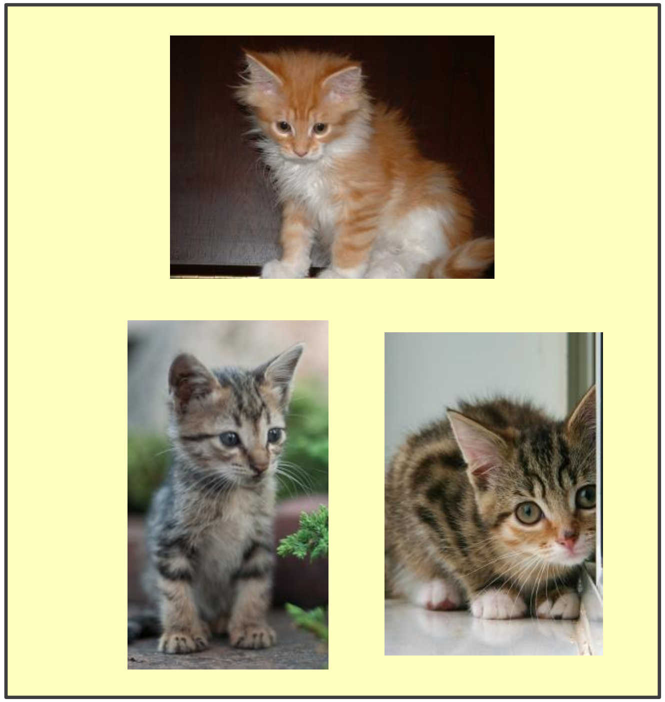
Bean
0 years old
black

2 years old
white

Tiger
2 years old
brown

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)

class definition

name

class parent

variable to refer to an instance of the class

what data initializes an Animal type

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None

myanimal = Animal(3)
```

special method to create an instance

name is a data attribute even though an instance is not initialized with it as a param

one instance

mapped to self.age in class def

GETTER AND SETTER METHODS

getter

setter

```
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)
```

- **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
```

```
a.get_age()
```

- access data attribute
- allowed, but not recommended

- 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

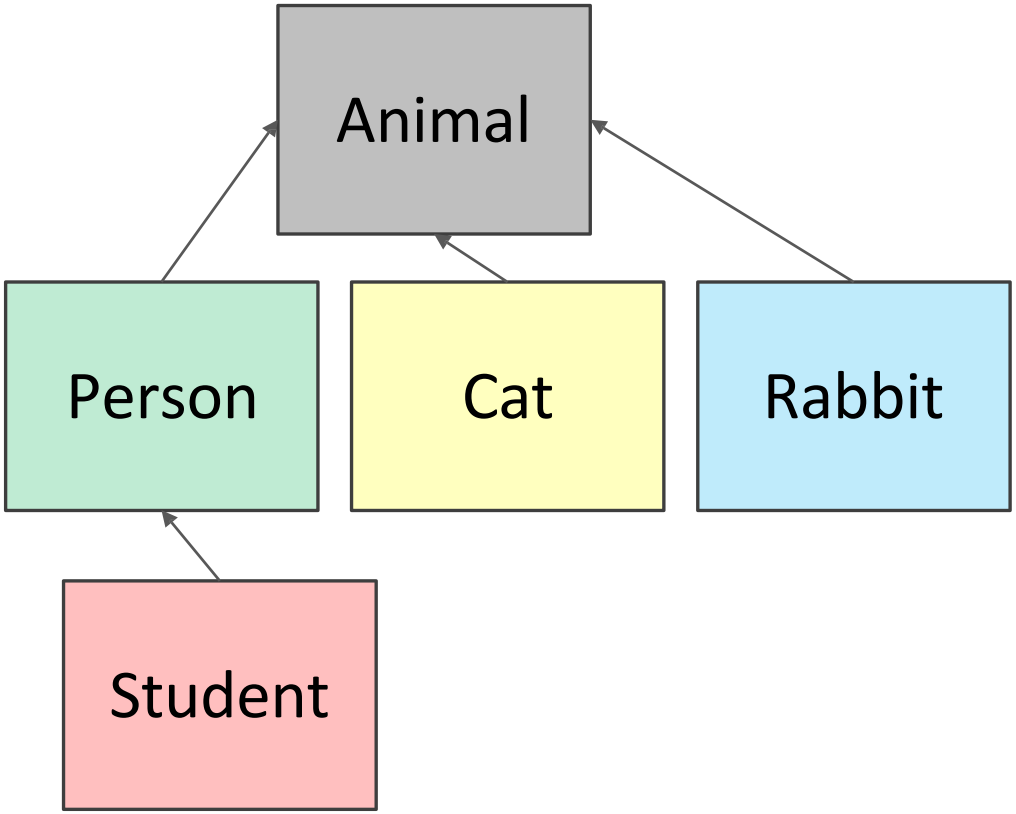
```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

HIERARCHIES

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

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

```python
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)

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

access class variable

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

instance variable

- `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**

booleans

```
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
```

- 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`

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

UNDERSTANDING PROGRAM EFFICIENCY: 1

Today

- Measuring orders of growth of algorithms
- Big “Oh” notation
- Complexity classes

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,”)
```

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) : $n^2 + 2n + 2$

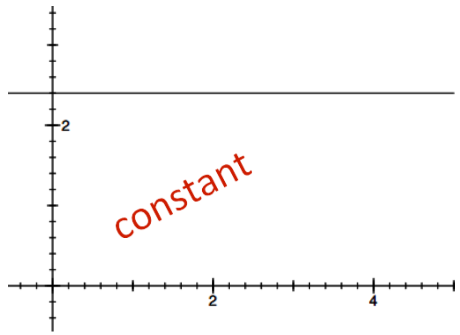
O(n^2) : $n^2 + 100000n + 3^{1000}$

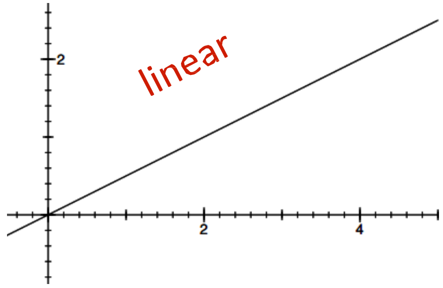
O(n) : `log(n) + n + 4`

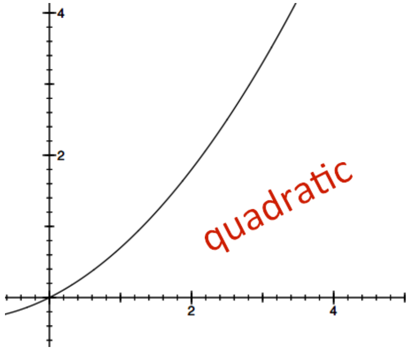
O(n log n) : `0.0001*n*log(n) + 300n`

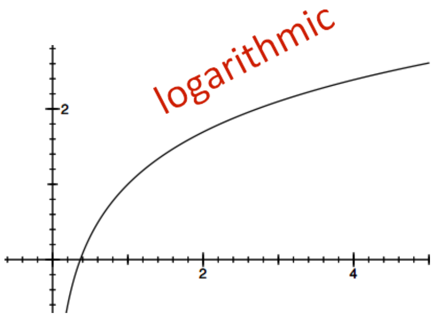
O(3^n) : $2n^{30} + 3^n$

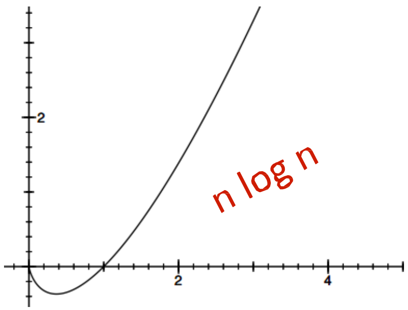
TYPES OF ORDERS OF GROWTH

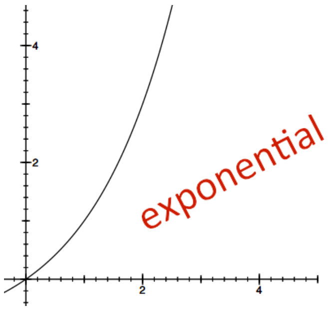












ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **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+n²) = O(n²) 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)

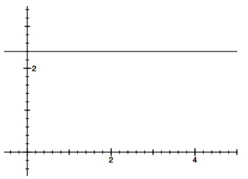
is O(n)*O(n) = O(n*n) = O(n^2) because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

COMPLEXITY CLASSES

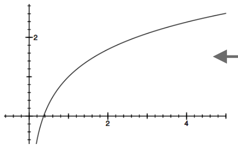
- *$O(1)$* denotes constant running time
- *$O(\log n)$* denotes logarithmic running time
- *$O(n)$* denotes linear running time
- *$O(n \log n)$* denotes log-linear running time
- *$O(n^c)$* denotes polynomial running time (c is a constant)
- *$O(c^n)$* denotes exponential running time (c is a constant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

COMPLEXITY CLASSES ORDERED LOW TO HIGH

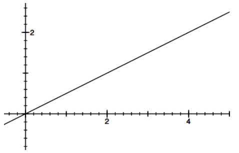
O(1) : constant



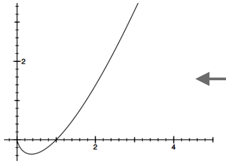
O(log n) : logarithmic



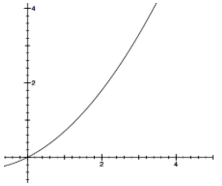
O(n) : linear



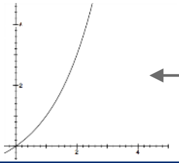
O(n log n): loglinear



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	12676506 00228229 40149670 3205376	107150860718626732094842504906 000181056140481170553360744375038 837035105112493612249319837881 569585812759467291755314682518 714528569231404359845775746985 748039345677748242309854210746050 623711418779541821530464749835 819412673987675591655439460770629 145711964776865421676604298316 526243868372056680693 76	**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
```

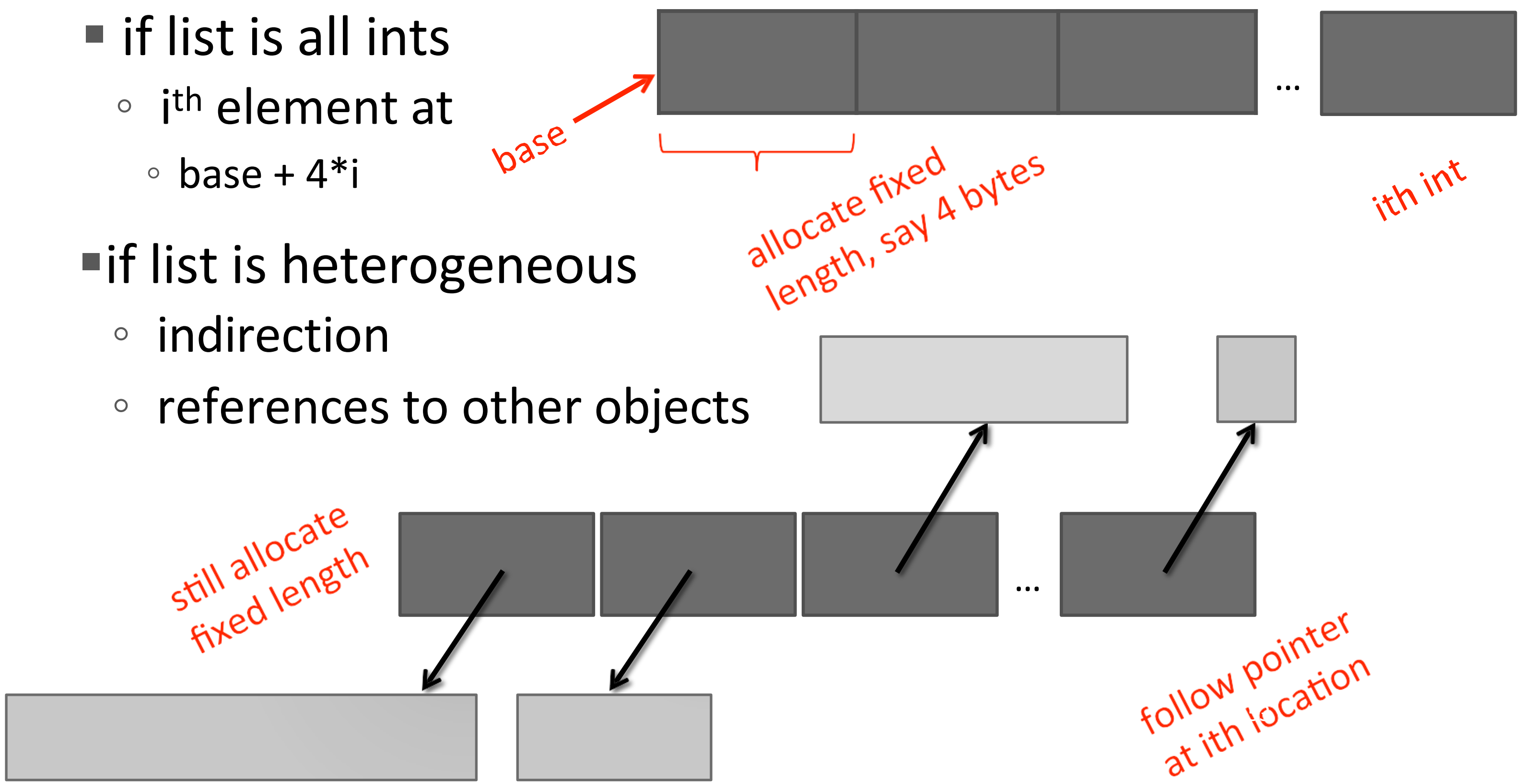
speed up a little by returning True here, but speed up doesn't impact worst case

- 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
- $\boxed{\text{O(len(L))}}$ 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 var
```

- *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 False
```

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

UNDERSTANDING PROGRAM EFFICIENCY: 2

Objective

- Classes of complexity
- Examples characteristic of each class

WHY WE WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

- 특정 크기의 문제를 해결하는 데 필요한 시간을 예측하기 위해 알고리즘에 대해 어떻게 추론 할 수 있습니까?
- 알고리즘 설계의 선택을 결과 알고리즘의 시간 효율성과 어떻게 관련시킬 수 있습니까?
- 특정 문제를 해결하는 데 필요한 시간에 근본적인 제한이 있습니까?

ORDERS OF GROWTH: Summary

Goals:

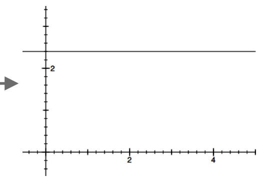
- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth – as tight as possible
- do not need to be precise: **"order of" not "exact"** growth
- we will look at **largest factors** 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**

COMPLEXITY CLASSES: Summary

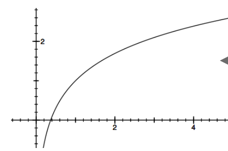
- *O(1)* denotes constant running time
- *O(log n)* denotes logarithmic running time
- *O(n)* denotes linear running time
- *O(n log n)* denotes log-linear running time
- *O(n^c)* denotes polynomial running time (c is a constant)
- *O(c^n)* denotes exponential running time (c is a constant being raised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

COMPLEXITY CLASSES ORDERED LOW TO HIGH

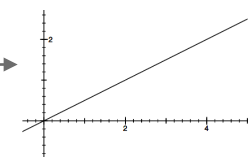
$O(1)$: constant



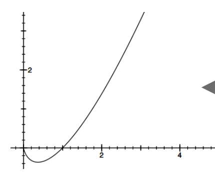
$O(\log n)$: logarithmic



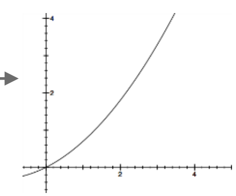
$O(n)$: linear



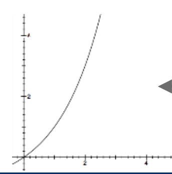
$O(n \log n)$: loglinear



$O(n^c)$: 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	12676506 00228229 40149670 3205376	107150860718626732094842504906 000181056140481170553360744375038 837035105112493612249319837881 569585812759467291755314682518714 528569231404359845775746985748039 345677748242309854210746050623711 418779541821530464749835819412673 987675591655439460770629145711964 776865421676604298316526243868372056680693 76	**Good luck!!**

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?

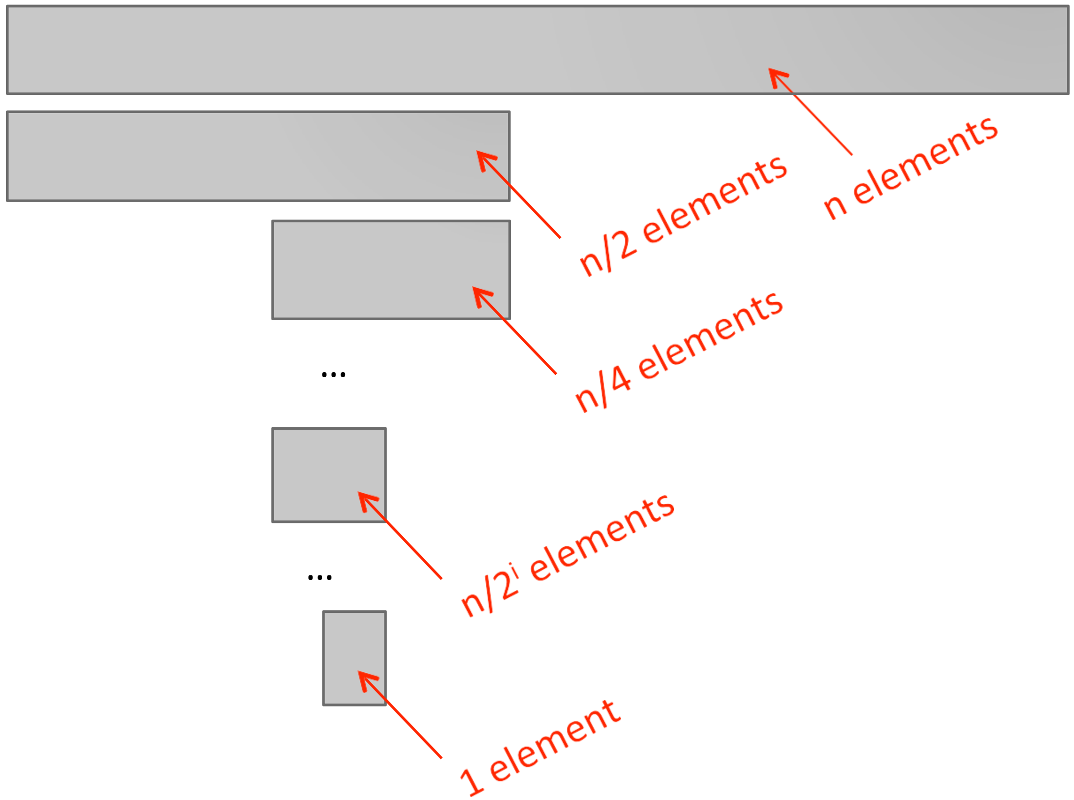
BISECTION SEARCH

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

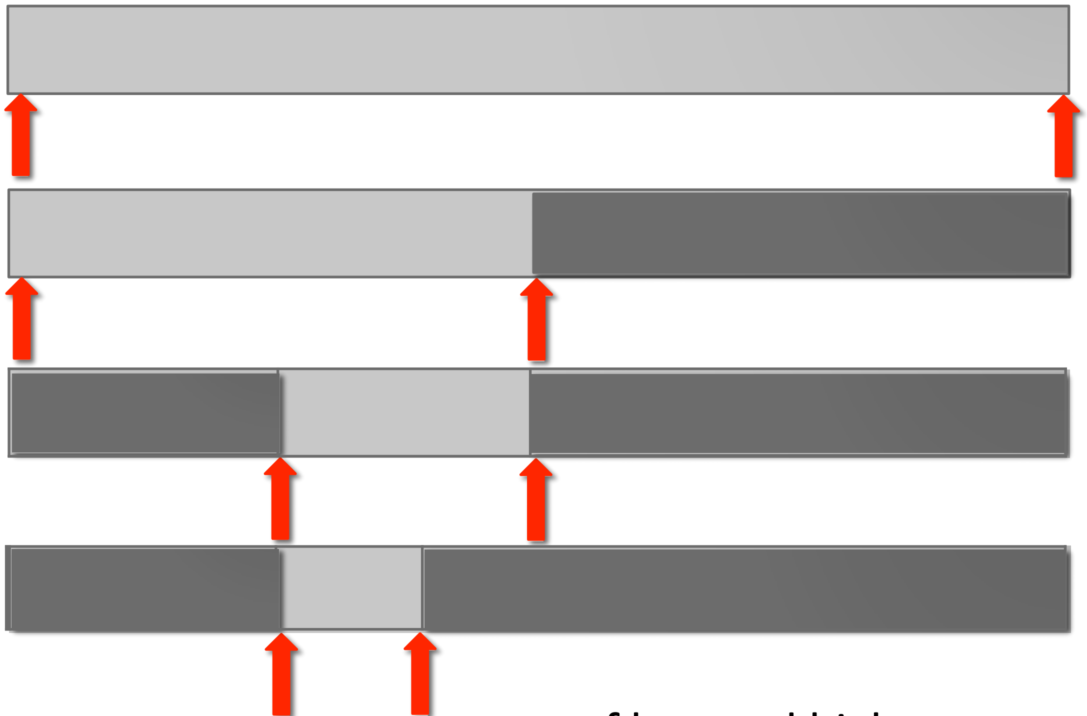
NOT constant

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 in 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'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        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

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

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

$a = 2^n - 1$

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:

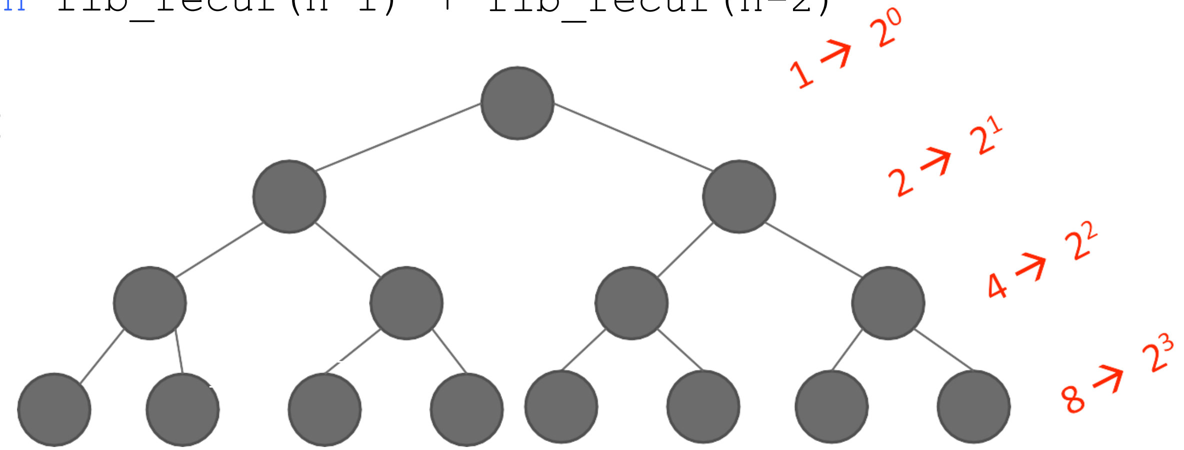
  O(1) + O(n) + O(1) → **O(n)**

COMPLEXITY OF RECURSIVE FIBONACCI

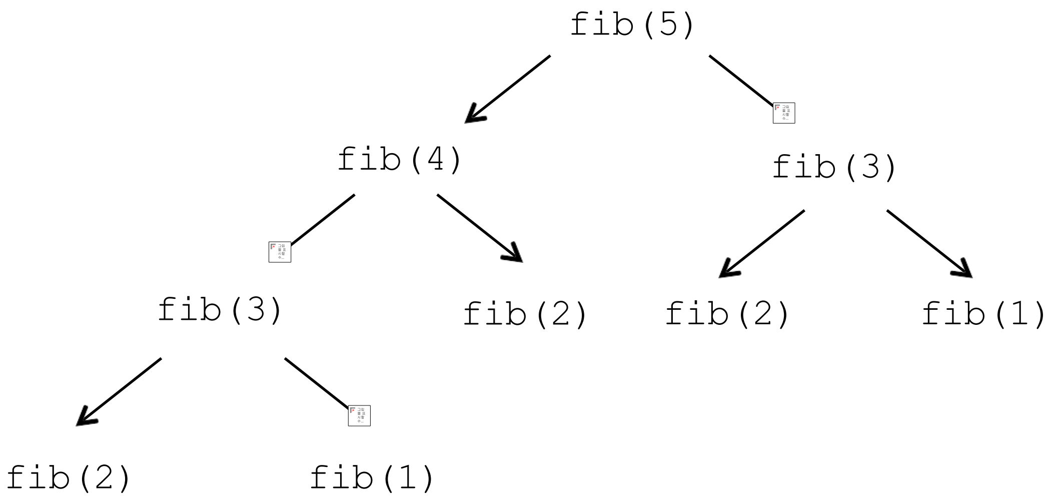
```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- 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)

SEARCHING AND SORTING ALGORITHMS

SEARCH ALGORITHMS

- search algorithm – method for finding an item or group of items with specific properties within a collection of items
- collection could be implicit
  - example – find square root as a search problem
    - exhaustive enumeration
    - bisection search
    - Newton-Raphson
- collection could be explicit
  - example – is a student record in a stored collection of data?

SEARCHING ALGORITHMS

- linear search
  - **brute force** search (aka British Museum algorithm)
  - list does not have to be sorted
- bisection search
  - list **MUST be sorted** to give correct answer
  - saw two different implementations of the algorithm

LINEAR SEARCH ON UNSORTED LIST: RECAP

```
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]
- overall complexity is **O(n) – where n is len(L)**

Assumes we can retrieve element of list in constant time

LINEAR SEARCH ON SORTED LIST: RECAP

```
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)**

USE BISECTION SEARCH: RECAP

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 IMPLEMENTATION: RECAP

```
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)
```

COMPLEXITY OF BISECTION SEARCH: RECAP

- **bisect_search2** and its helper
  - O(log n) bisection search calls
    - reduce size of problem by factor of 2 on each step
  - pass list and indices as parameters
  - list never copied, just re-passed as pointer
  - constant work inside function
  - → **O(log n)**

SEARCHING A SORTED LIST -- n is len(L)

- using **linear search**, search for an element is **O(n)**
- using **binary search**, can search for an element in **O(log n)**
  - assumes the **list is sorted**!
- when does it make sense to **sort first then search**?
  - SORT + O(`log n`) < O(`n`) → SORT < O(`n`) – O(`log n`)
  - when sorting is less than O(`n`)
- **NEVER TRUE!**
  - **to sort a collection of n elements must look at each one at least once!**

AMORTIZED COST -- n is len(L)

- why bother sorting first?
- in some cases, may **sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches
- SORT + `K`*O(`log n`) < `K`*O(`n`)

→ for large `K`, **SORT time becomes irrelevant,** if cost of sorting is small enough

SORT ALGORITHMS

- Want to efficiently sort a list of entries (typically numbers)
- Will see a range of methods, including one that is quite efficient

MONKEY SORT

- aka bogosort, stupid sort, slowsort, permutation sort, shotgun sort
- to sort a deck of cards
  - throw them in the air
  - pick them up
  - are they sorted?
  - repeat if not sorted

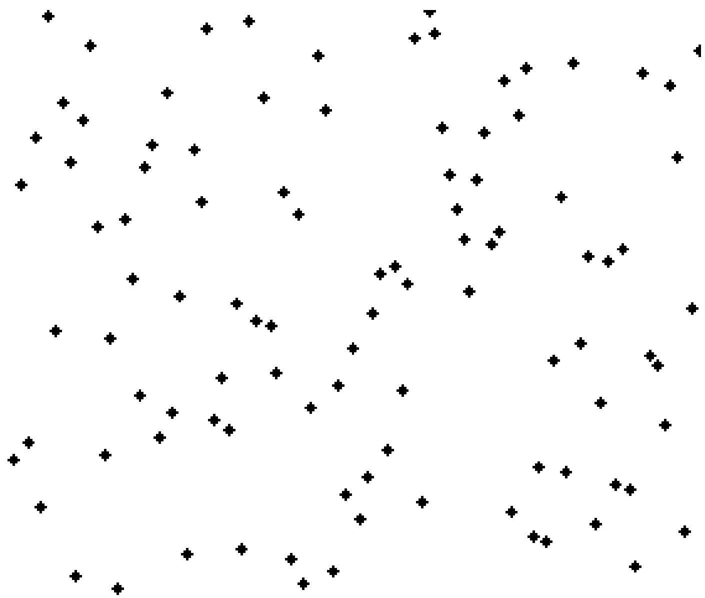
COMPLEXITY OF BOGO SORT

```
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

- best case: **O(n) where n is len(L)** to check if sorted
- worst case: O(?) it is **unbounded** if really unlucky

BUBBLE SORT

- **compare consecutive pairs** of elements
- **swap elements** in pair such that smaller is first
- when reach end of list, **start over** again
- stop when **no more swaps** have been made
- largest unsorted element always at end after pass, so at most n passes



COMPLEXITY OF BUBBLE SORT

```
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

O(len(L))

O(len(L))

- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **O(n^2) where n is len(L)** to do len(L)-1 comparisons and len(L)-1 passes

SELECTION SORT

- first step
  - extract **minimum element**
  - **swap it** with element at **index 0**
- subsequent step
  - in remaining sublist, extract **minimum element**
  - **swap it** with the element at **index 1**
- keep the left portion of the list sorted
  - at i’th step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

ANALYZING SELECTION SORT

- loop invariant
  - given prefix of list L[0:i] and suffix L[i+1:len(L)], then prefix is sorted and no element in prefix is larger than smallest element in suffix
    1. base case: prefix empty, suffix whole list – invariant true
    2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
    3. when exit, prefix is entire list, suffix empty, so sorted

COMPLEXITY OF SELECTION SORT

```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1
```

len(L) times
→ O(len(L))

len(L) – suffixSt times
→ O(len(L))

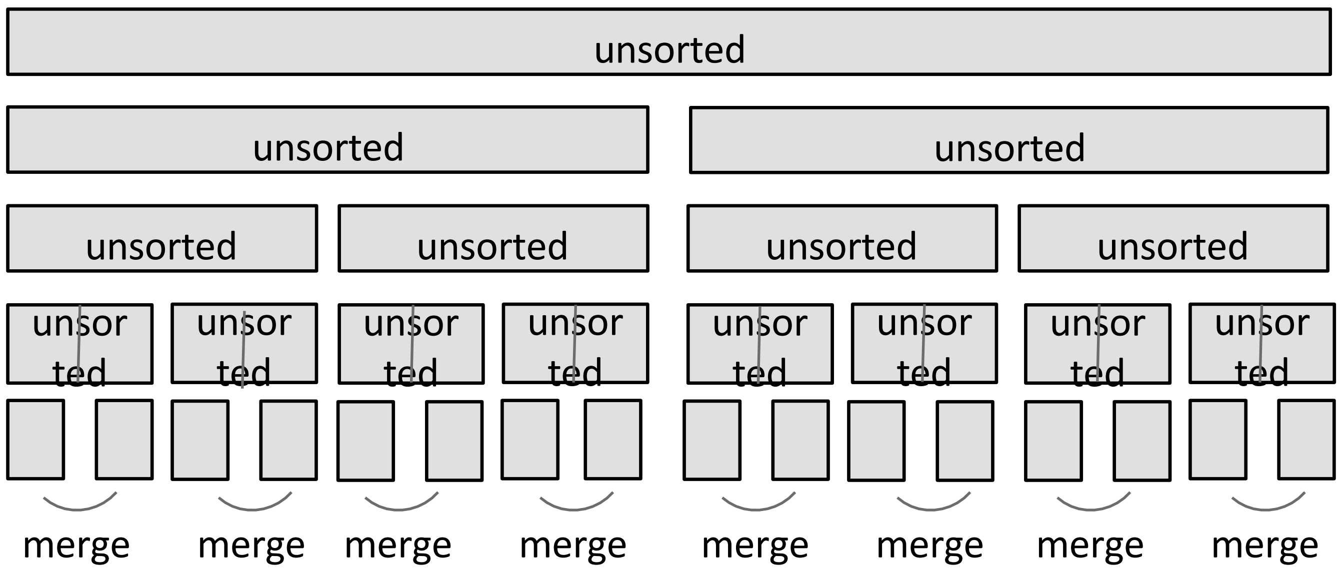
- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **$O(n^2)$ where n is len(L)**

MERGE SORT

- use a divide-and-conquer approach:
    1. if list is of length 0 or 1, already sorted
    2. if list has more than one element, split into two lists, and sort each
    3. merge sorted sublists
        1. look at first element of each, move smaller to end of the result
        2. when one list empty, just copy rest of other list

MERGE SORT

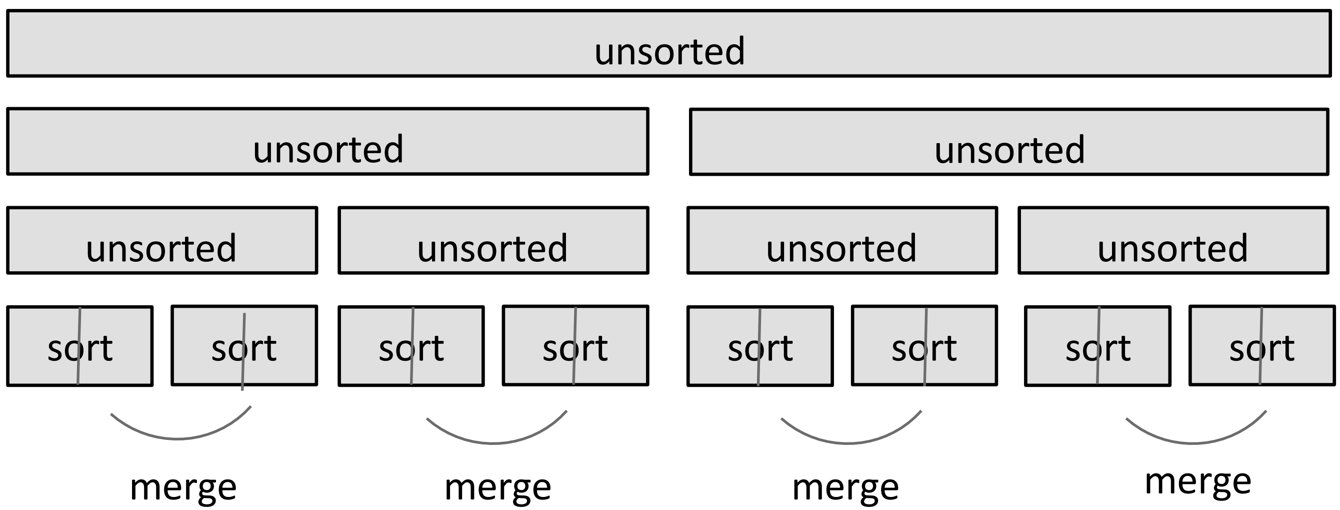
- divide and conquer



- **split list in half** until have sublists of only 1 element

MERGE SORT

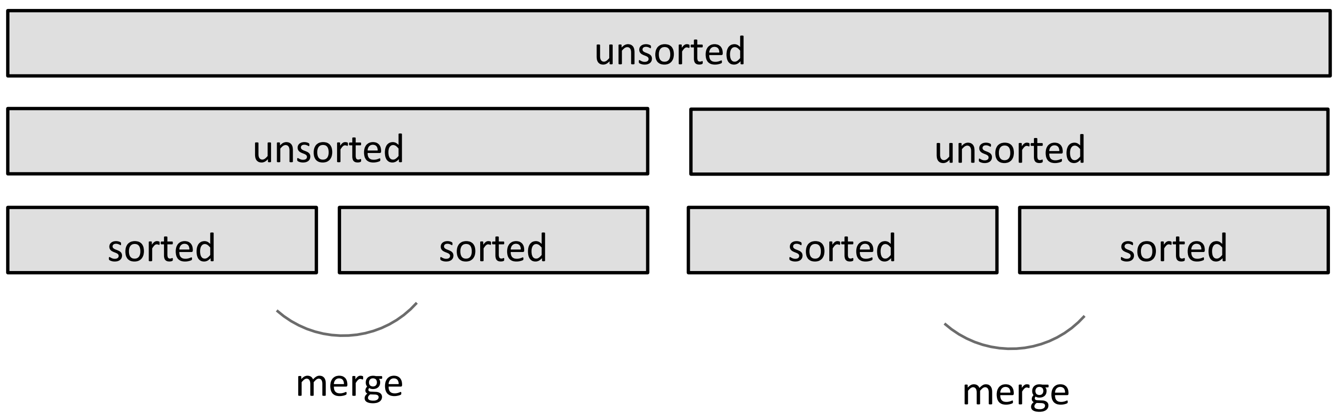
- divide and conquer



- merge such that **sublists will be sorted after merge**

MERGE SORT

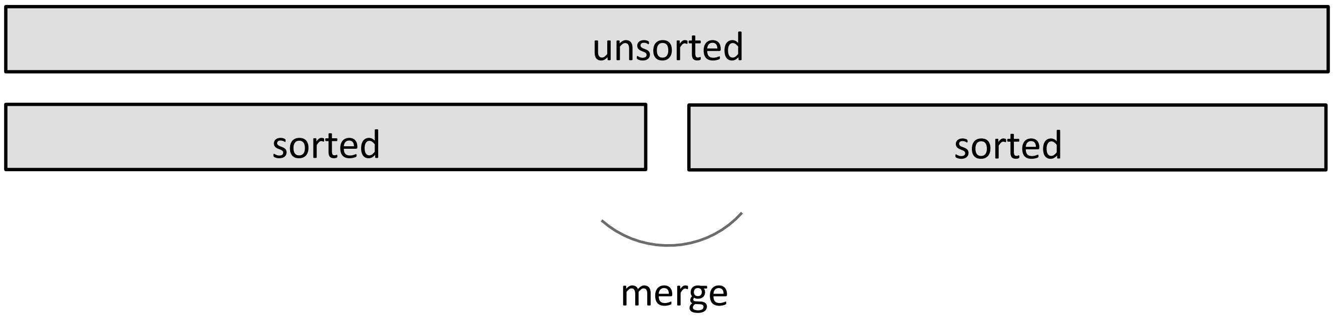
- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer – done!

sorted

EXAMPLE OF MERGING

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1, 2	[]
[5,12,18,19,20]	[2,3,4,17]	5, 2	[1]
[5,12,18,19,20]	[3,4,17]	5, 3	[1,2]
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

MERGING SUBLISTS STEP

```
def merge(left, right):
    result = []
    i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

- left and right sublists are ordered
- move indices for sublists depending on which sublist holds next smallest element

when right sublist is empty

when left sublist is empty

COMPLEXITY OF MERGING SUBLISTS STEP

- go through two lists, only one pass
- compare only **smallest elements in each sublist**
- O(len(left) + len(right)) copied elements
- O(len(longer list)) comparisons
- **linear in length of the lists**

MERGE SORT ALGORITHM -- RECURSIVE

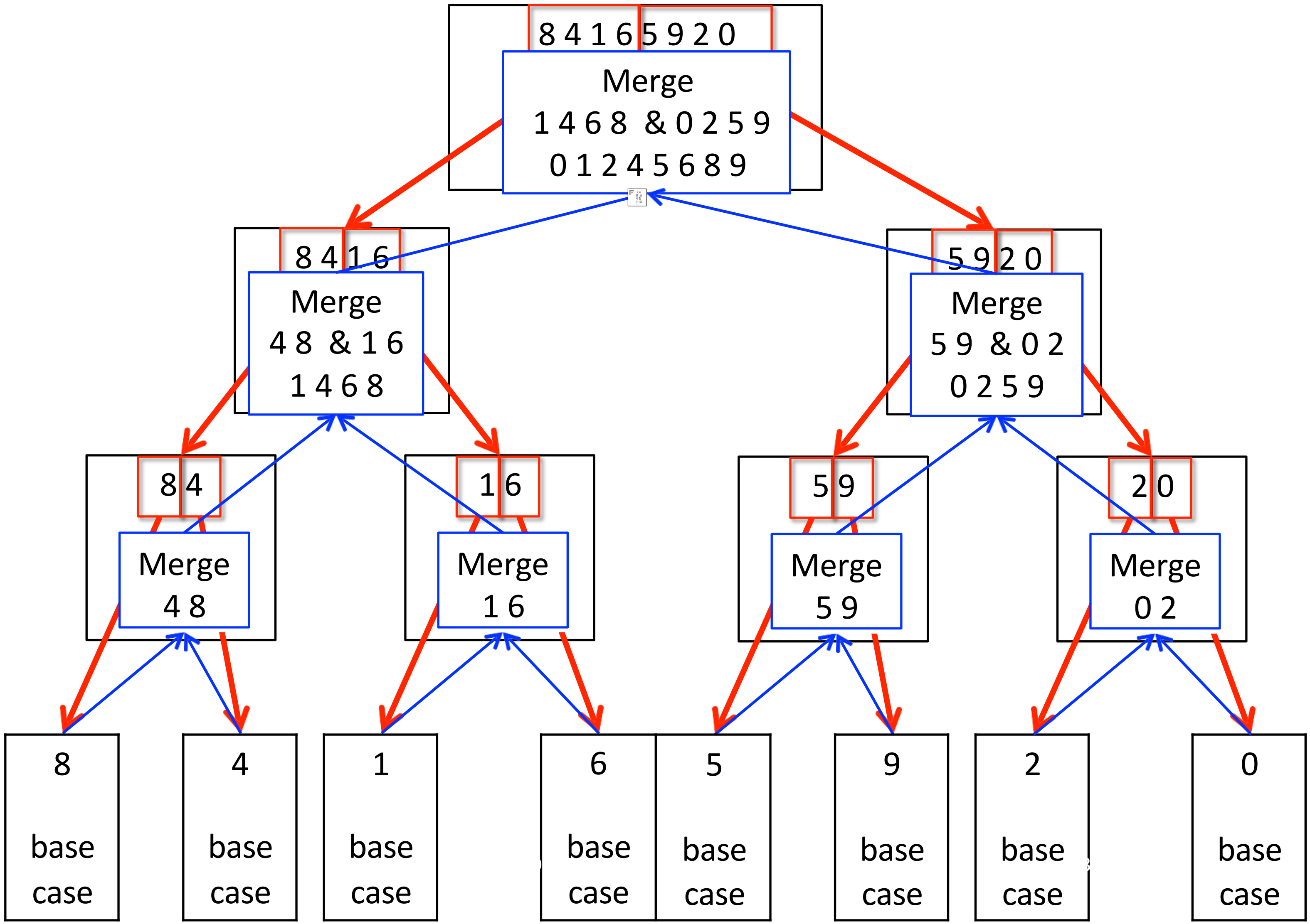
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

base case

divide

conquer with the merge step

- **divide list** successively into halves
- depth-first such that **conquer smallest pieces down one branch** first before moving to larger pieces



8 4 1 6 5 9 2 0
Merge
1 4 6 8 & 0 2 5 9
0 1 2 4 5 6 8 9
8 4 1 6
Merge
4 8 & 1 6
1 4 6 8
5 9 2 0
Merge
5 9 & 0 2
0 2 5 9
8 4
Merge
4 8
1 6
Merge
1 6
5 9
Merge
5 9
2 0
Merge
0 2
8 base case
4 base case
1 base case
6 base case
5 base case
9 base case
2 base case
0 base case

COMPLEXITY OF MERGE SORT

- at **first recursion level**
  - n/2 elements in each list
  - O(n) + O(n) = O(n) where n is len(L)
- at **second recursion level**
  - n/4 elements in each list
  - two merges → O(n) where n is len(L)
- each recursion level is O(n) where n is len(L)
- **dividing list in half** with each recursive call
  - O(log(n)) where n is len(L)
- overall complexity is **O(n log(n)) where n is len(L)**

SORTING SUMMARY -- n is len(L)

- bogo sort
  - randomness, unbounded O()
- bubble sort
  - $O(n^2)$
- selection sort
  - $O(n^2)$
  - guaranteed the first i elements were sorted
- merge sort
  - $O(n \log(n))$
- $O(n \log(n))$ is the fastest a sort can be

WHAT HAVE WE SEEN?

KEY TOPICS

- represent knowledge with **data structures**
- **iteration and recursion** as computational metaphors
- **abstraction** of procedures and data types
- **organize and modularize** systems using object classes and methods
- different classes of **algorithms**, searching and sorting
- **complexity** of algorithms

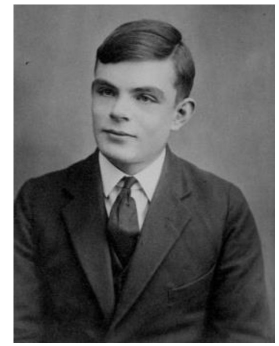
OVERVIEW OF COURSE

- learn computational modes of thinking
- begin to master the art of computational problem solving
- make computers do what you want them to do

Hope we have started you down the path to being able to think and act like a computer scientist

WHAT DO COMPUTER SCIENTISTS DO?

- they think computationally
  - abstractions, algorithms, automated execution
- just like the three r's: reading, 'riting, and 'rithmetic – computational thinking is becoming a fundamental skill that every well-educated person will need



Alan Turing

Image in the Public Domain, courtesy of Wikipedia Commons.

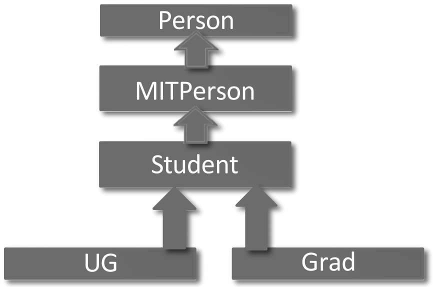
Ada Lovelace

Image in the Public Domain, courtesy of Wikipedia Commons.

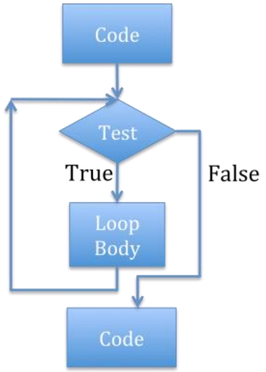
I ♥ 6.0001

THE THREE A'S OF COMPUTATIONAL THINKING

- abstraction
  - choosing the right abstractions
  - operating in multiple layers of abstraction simultaneously
  - defining the relationships between the abstraction layers



- automation
  - think in terms of mechanizing our abstractions
  - mechanization is possible – because we have precise and exacting notations and models; and because there is some "machine" that can interpret our notations



- algorithms
  - language for describing automated processes
  - also allows abstraction of details
  - language for communicating ideas & processes

```
def mergeSort(L, compare = operator.lt):
    if len(L) < 2:
        return L[:]
    else:
        middle = int(len(L)/2)
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```

ASPECTS OF COMPUTATIONAL THINKING

- how difficult is this problem and how best can I solve it?
  - theoretical computer science gives precise meaning to these and related questions and their answers
- thinking recursively
  - reformulating a seemingly difficult problem into one which we know how to solve
  - reduction, embedding, transformation, simulation

$\boxed{O(\log n)\ ;\ O(n)\ ;\ O(n \log n)\ ;\ O(n^2);\ O(c^n)}$

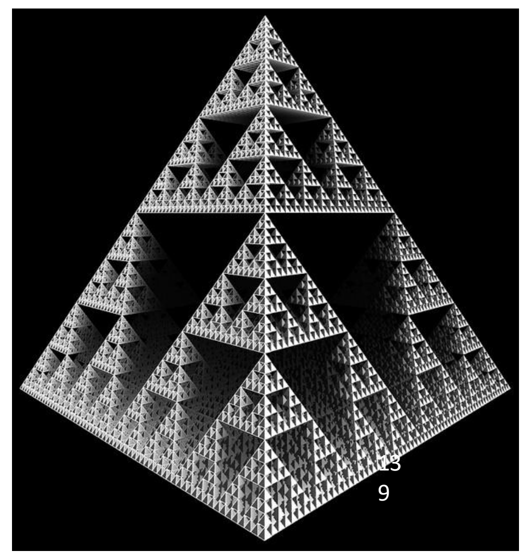


Image Licensed CC-BY, Courtesy of Robson# on Flickr.