

# PROGRAM EFFICIENCY

(download slides and .py files to follow along!)

---

6.0001 LECTURE 9

# TODAY

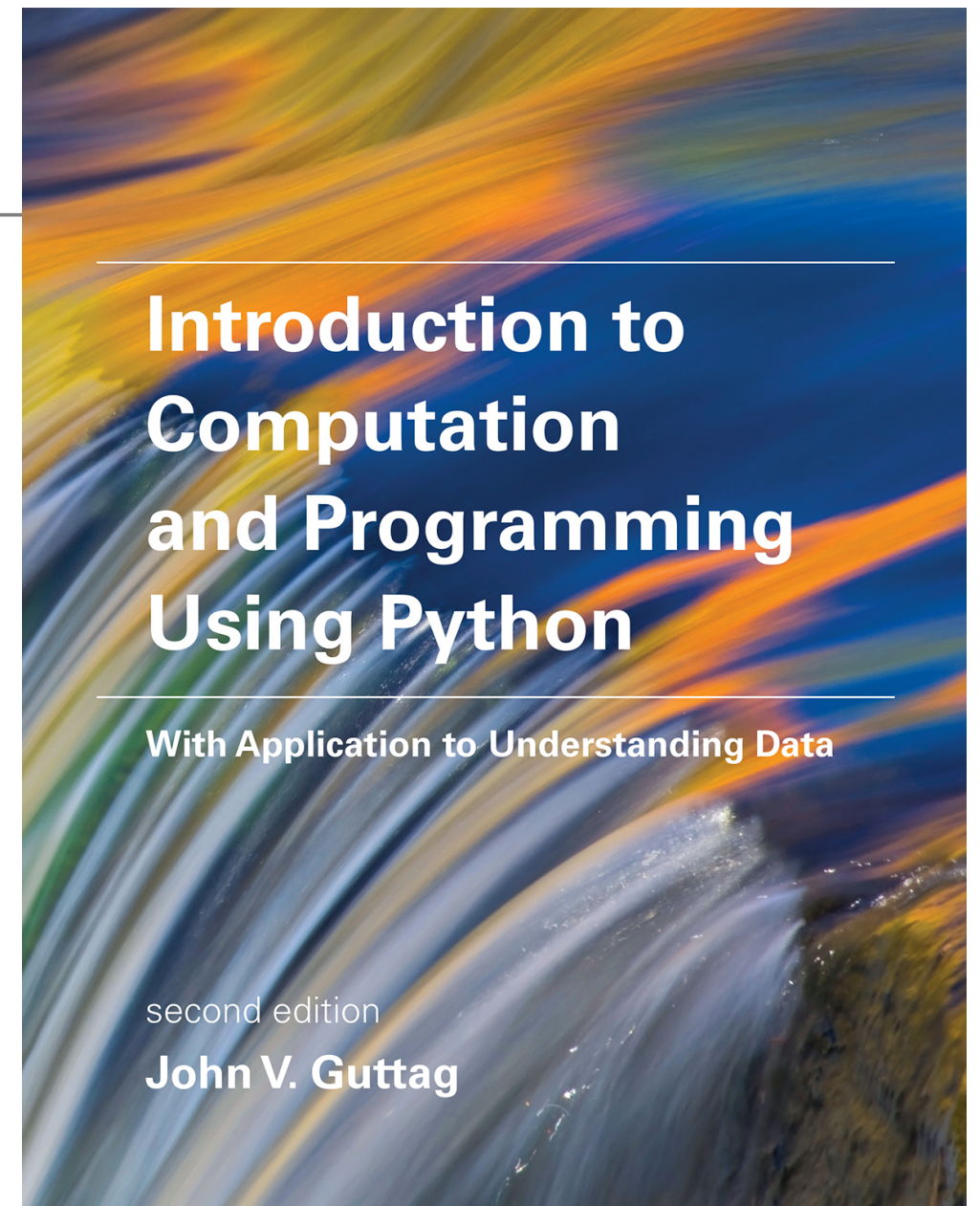
---

- Formally evaluate programs
- Efficiency in time
- Orders of growth, big Oh notation
- Examples of different complexity cases

# Assigned Reading

---

## ■ Chapter 9



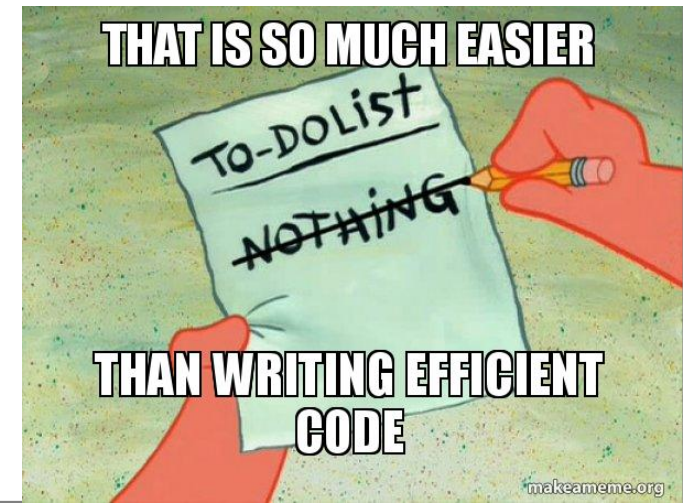
[https://mitpress.mit.edu/sites/default/files/Guttag\\_errata\\_revised\\_083117.pdf](https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf)

# PROGRAM EFFICIENCY

---



# WRITING EFFICIENT PROGRAMS



- So far, we have emphasized correctness. It is the first thing to worry about!
- But sometimes that is not enough
- Problems can be very complex (as we shall see when we get to optimization in 6.0002)

- But data sets can be very large: in 2014 Google served 30,000,000,000,000 pages covering 100,000,000 GB of data



# EFFICIENCY IS IMPORTANT

---

- Separate **time and space efficiency** of a program
- Tradeoff between them: can use up a bit more memory to store values for quicker lookup later
- Challenges in understanding efficiency
  - A program can be **implemented in many different ways**
  - You can solve a problem using only a handful of different **algorithms**
- Want to separate choice of implementation from choice of more abstract algorithm

A tester has the heart  
of a developer.....

•  
•  
•  
•

In a jar on the desk...



# EVALUATING PROGRAMS

---

- Measure with a **timer**
- **Count** the operations
- Abstract notion of **order of growth**

# Aside: MODULES

---

- A module is a set of python definitions and statements stored in a file
- You first need to “import” the module
- call functions inside the module using the module’s name and dot notation
- `module.function()`

# TIMING A PROGRAM



- Use time module

```
import time
```

- Recall that importing means to bring in that class into your own file

```
def convert_to_km(m):  
    return m * 1.609
```

- **Start** clock → `t0 = time.perf_counter()`

- **Call** function → ~~`c_to_f`~~ <sup>convert</sup> `(1000000)`

- **Stop** clock → `t1 = time.perf_counter() - t0`  
`print("t =", t1, "s,")`

# Example: Convert, compound

---

```
def convert_to_km(m):  
    return m * 1.609  
  
def compound(invest, interest, n_months):  
    total=0  
    for i in range(n_months):  
        total = total * interest + invest  
    return total
```

# Measure time: convert

---

```
L_N = [1]
for i in range(7):
    L_N.append(L_N[-1]*10)

for N in L_N:
    t = time.perf_counter()
    km = convert_to_km(N)
    dt = time.perf_counter()-t
    print ("convert(", N, ") took ", dt, "seconds")
```

# Measure time: convert multiple samples

---

*### run the computation n\_samples times for better measurement*

`n_samples = 50`

```
for N in L_N:
    t = time.perf_counter()
    for i in range(n_samples):
        km = convert_to_km(N)
    dt = (time.perf_counter()-t)/n_samples
    print ("convert(", N, ") took ", dt, "seconds")
```

```
convert( 1 ) took 2.1178973838686943e-07 seconds
convert( 10 ) took 2.1529034711420535e-07 seconds
convert( 100 ) took 2.0069011952728033e-07 seconds
convert( 1000 ) took 2.1399988327175378e-07 seconds
convert( 10000 ) took 2.2487947717308999e-07 seconds
convert( 100000 ) took 1.9868952222168446e-07 seconds
convert( 1000000 ) took 2.4397042579948903e-07 seconds
convert( 10000000 ) took 2.021400723606348e-07 seconds
```



# Measure time: compound

```
def compound(invest, interest, n_months):  
    total=0  
    for i in range(n_months):  
        total = total * interest + invest  
    return total
```

```
compound( 1 ) took 1.2168078683316707e-06 seconds  
compound( 10 ) took 2.0174076780676843e-06 seconds  
compound( 100 ) took 6.968004163354635e-06 seconds  
compound( 1000 ) took 6.973260315135121e-05 seconds  
compound( 10000 ) took 0.000683364795986563 seconds  
compound( 100000 ) took 0.006051766208838671 seconds  
compound( 1000000 ) took 0.04472687559900805 seconds  
compound( 10000000 ) took 0.429290418792516 seconds
```

*growth unclear*

*10x*

*10x*

# Measure time: sum

---

```
def sum_of(L):  
    total = 0.0  
    for elt in L:  
        total = total + elt  
    return total
```

```
L_N = [1]  
for i in range(8):  
    L_N.append(L_N[-1]*10)
```

```
for N in L_N:  
    L = range(N)  
    t = time.perf_counter()  
    s = sum_of(L)  
    print ("sum of ", N, "elements took ", time.perf_counter()-t, "seconds")
```

# Measure time: is\_in

---

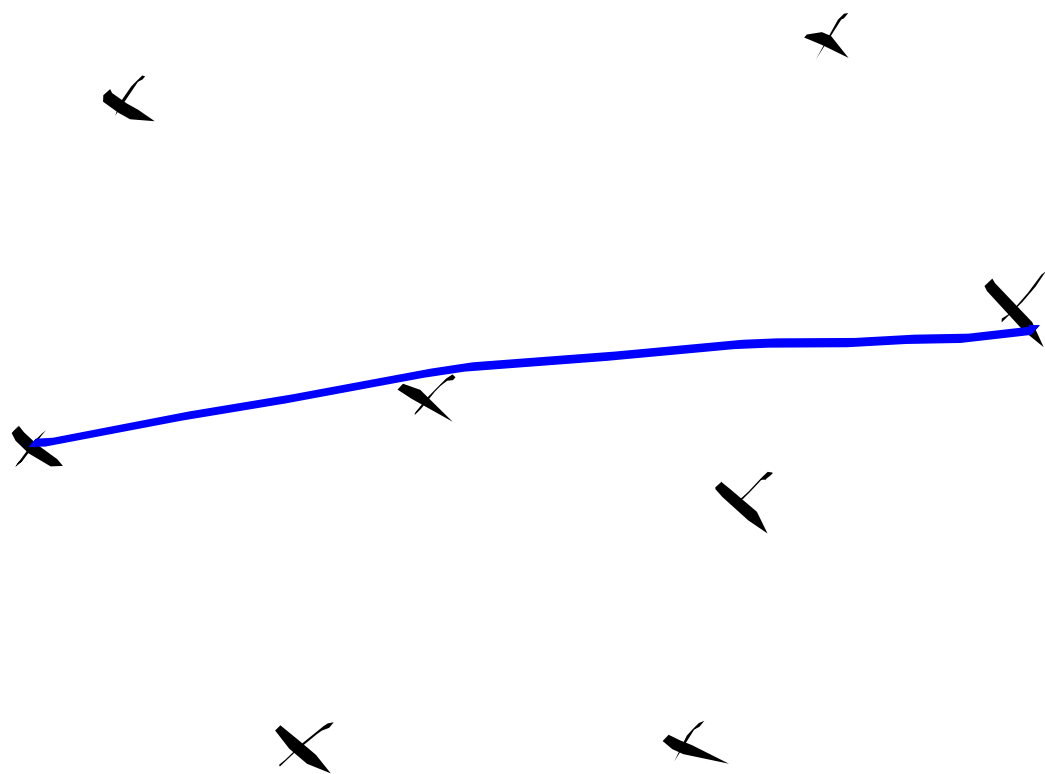
```
def is_in(L, x):
    for elt in L:
        if elt==x: return True
    return False

def binary_search(L, x):
    """ returns True if x is in L
        L must be sorted """
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        mid = (hi+lo) // 2
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
    return L[lo] == x
```

# Measure time: diameter

---

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                               + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```



# Measure time: diameter

---

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                               + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

# Measure time: diameter

---

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                           + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

```
diameter of 10 points took 7.76359811425209e-05 seconds
diameter of 100 points took 0.0028979270136915147 seconds
diameter of 1000 points took 0.18294400698505342 seconds
diameter of 10000 points took 15.902969222981483 seconds
```

# Measure time: binary numbers

---

```
def all_binary_numbers(N):  
    def helper(prefix, N):  
        if N==0:  
            return [prefix]  
        return helper(prefix+'0', N-1) + helper(prefix+'1', N-1)  
    return helper('', N)
```

```
.....  
binary numbers of 5 digits took 3.0381022952497005e-05 s  
binary numbers of 10 digits took 0.0007962230010889471 s  
binary numbers of 15 digits took 0.019941627979278564 s  
binary numbers of 20 digits took 0.4996888790046796 s  
binary numbers of 25 digits took 19.526876593008637 s
```



# Two different machines

---

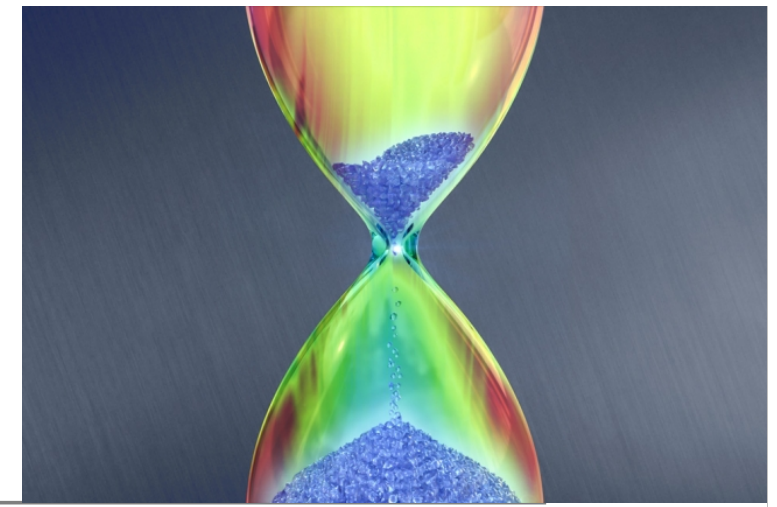
## Fredo's laptop

```
convert( 1 ) took 0.0919969081879 seconds
convert( 10 ) took 0.0812351703644 seconds
convert( 100 ) took 0.0810060501099 seconds
convert( 1000 ) took 0.0786969661713 seconds
convert( 10000 ) took 0.0776309967041 seconds
convert( 100000 ) took 0.0800149440765 seconds
convert( 1000000 ) took 0.0772659778595 seconds
convert( 10000000 ) took 0.0839469432831 seconds
convert( 100000000 ) took 0.0802690982819 seconds
convert( 1000000000 ) took 0.0796220302582 seconds
compound( 1 ) took 0.0781879425049 seconds
compound( 10 ) took 0.0791871547699 seconds
compound( 100 ) took 0.0802779197693 seconds
compound( 1000 ) took 0.0811159610748 seconds
compound( 10000 ) took 0.079794883728 seconds
compound( 100000 ) took 0.0803499221802 seconds
compound( 1000000 ) took 0.180749893188 seconds
compound( 10000000 ) took 0.713826179504 seconds
compound( 100000000 ) took 6.48052787781 seconds
compound( 1000000000 ) took 63.5682651997 seconds
```

## Fredo's (old) desktop

```
convert( 1 ) took 0.0651700496674 seconds
convert( 10 ) took 0.0838208198547 seconds
convert( 100 ) took 0.0830719470978 seconds
convert( 1000 ) took 0.0816540718079 seconds
convert( 10000 ) took 0.0824558734894 seconds
convert( 100000 ) took 0.0837979316711 seconds
convert( 1000000 ) took 0.0837349891663 seconds
convert( 10000000 ) took 0.0843281745911 seconds
convert( 100000000 ) took 0.0838270187378 seconds
convert( 1000000000 ) took 0.0844709873199 seconds
compound( 1 ) took 0.083487033844 seconds
compound( 10 ) took 0.0834701061249 seconds
compound( 100 ) took 0.083163022995 seconds
compound( 1000 ) took 0.0843181610107 seconds
compound( 10000 ) took 0.0845410823822 seconds
compound( 100000 ) took 0.099858045578 seconds
compound( 1000000 ) took 0.183917045593 seconds
compound( 10000000 ) took 1.38667988777 seconds
compound( 100000000 ) took 12.7653880119 seconds
compound( 1000000000 ) took 126.978576899 seconds
```

# TIMING PROGRAMS IS INCONSISTENT



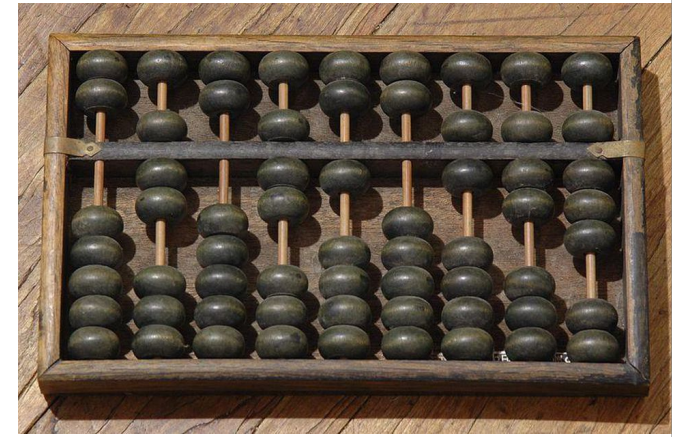
- GOAL: to evaluate different algorithms
- Running time **varies between algorithms** ✓
- Running time **varies between implementations** ✗
- Running time **varies between computers** ✗
- Running time is **not predictable** for small inputs ✗
- Time varies for different inputs but cannot really express a relationship between inputs and time ✗
- Can only be measured a-posteriori

# Don't get me wrong

---

- Timing is a critical tool to assess the performance of programs
  - At the end of the day, it is unreplaceable for real-world assessment
- But we will learn a complementary tool (asymptotic complexity) that has other advantages
  - A priori evaluation (before writing or running code)
  - Assesses algorithm independently of machine and implementation
  - Provides direct insight to the **design** of efficient algorithms

# COUNTING OPERATIONS



- Assume these steps take **constant time**:
  - Mathematical operations
  - Comparisons
  - Assignments
  - Accessing objects in memory
- Count number of operations executed as function of size of input

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

3 ops

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

1 op

loop  
x times

2 ops

1 op

$\text{mysum} \rightarrow 1+3(x+1) \text{ ops}$

# Count operations

```
def is_in_counter(L, x):  
    global count  
    for elt in L:  
        count += 1 #if == test  
        if elt==x: return True  
    return False
```

says I will be  
able to access  
variable outside  
function

usually frowned  
upon



# Count operations, binary search

```
def binary_search_counter(L, x):  
    """ returns True if x is in L  
        L must be sorted """  
    global count  
    lo = 0  
    hi = len(L)  
    while hi-lo > 1:  
        count += 1 #while test  
        mid = (hi+lo) // 2  
        count += 2 #mid + and /  
        if L[mid] <= x:  
            lo = mid  
        else:  
            hi = mid  
        count += 1 #if test  
    count += 1 #return == test  
    return L[lo] == x
```

```
for 10000 element, brute-force took on average 5000 ops  
9.988015978695074 times more than for 10 times fewer elements  
for 10000 element, binary search took on average 67 ops  
1.4195804195804198 times more than for 10 times fewer elements  
for 100000 element, brute-force took on average 50000 ops  
9.998800159978668 times more than for 10 times fewer elements  
for 100000 element, binary search took on average 82 ops  
1.2216748768472907 times more than for 10 times fewer elements  
for 1000000 element, brute-force took on average 500000 ops  
9.99988000159998 times more than for 10 times fewer elements  
for 1000000 element, binary search took on average 97 ops  
1.1814516129032258 times more than for 10 times fewer elements
```

# COUNTING OPERATIONS IS BETTER, BUT ...

---

- GOAL: to evaluate different algorithms
- Count **depends on algorithm** ✓
- Count **depends on implementations** ✗
- Count **independent of computers** ✓
- No real 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 and counting **evaluate machines**
  
- Want to **evaluate algorithm**
- Want to **evaluate scalability**
- Want to **evaluate in terms of input size**



Break

until

3:52

# A BETTER WAY

---



- Focus on idea of counting operations in an algorithm, but **not worry about small variations in implementation**
- 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

# HOW TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

---

- Want to express **efficiency in terms of input**, so need to decide what is your input
- Could be an **integer**  
`-- mysum (x)`
- Could be **length of list**  
`-- list_sum (L)`
- **You decide** when multiple parameters to a function  
`-- search_for_elt (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

---

- Consider that you are given a list  $L$  of some length  $\text{len}(L)$
- **Best case**: minimum running time over all possible inputs of a given size,  $\text{len}(L)$ 
  - Constant for `search_for_elmt`
  - First element in any list
- **Average case**: average running time over all possible inputs of a given size,  $\text{len}(L)$ 
  - Practical measure
- **Worst case**: maximum running time over all possible inputs of a given size,  $\text{len}(L)$ 
  - Linear in length of list for `search_for_elmt`
  - Must search entire list and not find it
  - Focus on **worst case** in this class

# ORDERS OF GROWTH

---

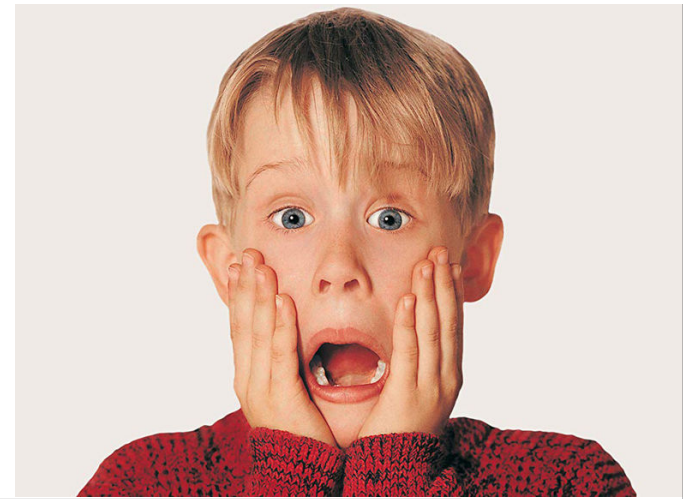
- Want to evaluate programs when **input is very big**
- Want to express the **growth of program's run time**
- Want to put an **upper bound** on growth
- Do not need to be precise: **“order of” not “exact”** growth
- We will look at **largest factors** in run time (which section of the program will take the longest to run?)

# MEASURING ORDER OF GROWTH: BIG O() 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 tends to occur often and is the bottleneck when a program runs
  - Express rate of growth of program relative to the input
  - Evaluate algorithm not machine or implementation
- A technicality
  - When we say that the complexity of  $f$  is  $O(n)$ , we mean that its asymptotic growth is not worse than linear in  $n$ .
  - It is an **upper bound**, not necessarily a **tight bound**
  - In practice, we are usually looking for something close to a tight bound

# EXACT STEPS vs $O()$



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

*temp = n-1  
n = temp*

- Computes factorial
- Number of steps:  $1 + 7n + 1 = O(n)$
- 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
- Given an expression for the number of operations needed to compute an algorithm, want to know **asymptotic behavior as size of problem gets large**
- Will focus on term that grows most rapidly
- 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 term**

$$: \textcircled{n^2} + 2n + 2$$

$$: \underline{n^2} + 1000000n + 3^{1000}$$

$$: \log(n) + \textcircled{n} + 4$$

# SIMPLIFICATION EXAMPLES

---

- Drop constants and multiplicative factors
- Focus on **dominant term**

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

$$O(n^2) : n^2 + 1000000n + 3^{1000}$$

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

# 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):     $O(n)$ 
    print('a')
for j in range(n*n):
    print('b')          $O(n^2)$ 
```

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 $O()$ :

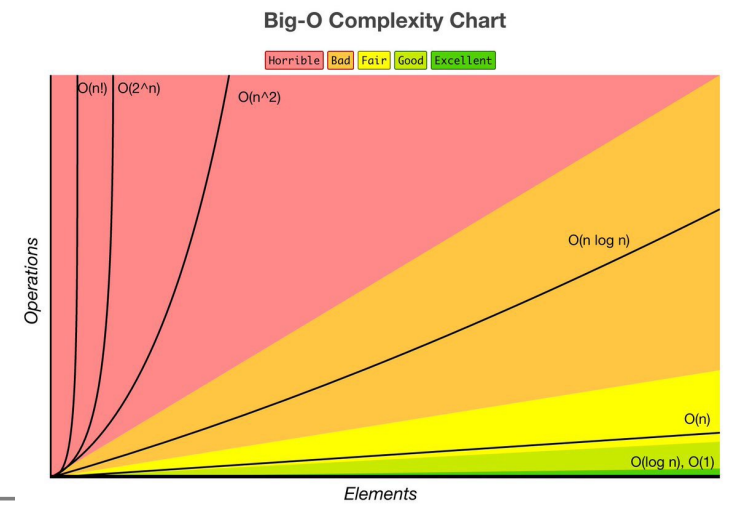
- Used with **nested** statements/loops
- $O(f(n)) * O(g(n))$  is  $O(f(n) * g(n))$
- For example,

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

$O(n)$  for each outer loop iteration

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



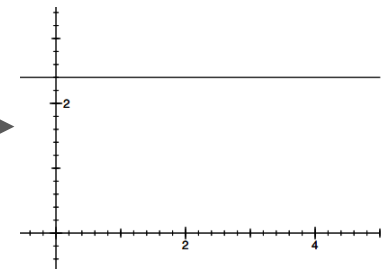
- $O(1)$  denotes **constant** running time
- $O(\log n)$  denotes **logarithmic** running time
- $O(n)$  denotes **linear** running time
- $O(n \log n)$  denotes **log-linear** running time
- $O(n^c)$  denotes **polynomial** running time ( $c$  is a constant)
- $O(c^n)$  denotes **exponential** running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

# COMPLEXITY CLASSES ORDERED LOW TO HIGH

$O(1)$

:

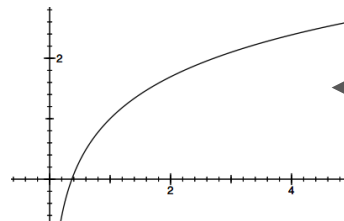
constant



$O(\log n)$

:

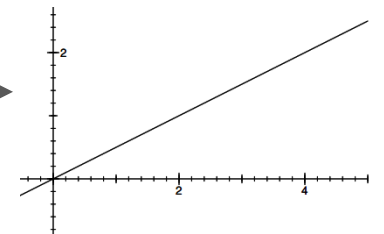
← logarithmic



$O(n)$

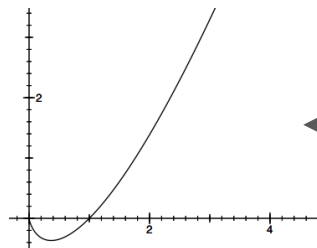
:

linear



$O(\underbrace{n \log n})$ :

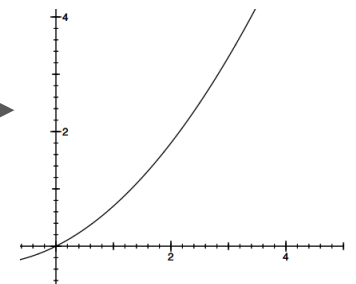
← log linear



$O(n^c)$

:

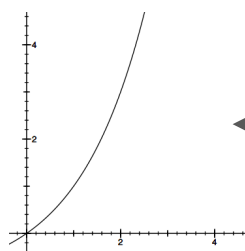
polynomial  
(quadratic)



$O(c^n)$

:

← exponential



*c is a  
constant*

