

# UNDERSTANDING PROGRAM EFFICIENCY: 1

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

---

6.0001 LECTURE 10

# 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

```
import time
```

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

- **start** clock → 






```
t0 = time.clock()
```
- **call** function → 

```
c_to_f(100000)
```
- **stop** clock → 

```
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
```

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

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  $\text{len}(L)$
- **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

*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**

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

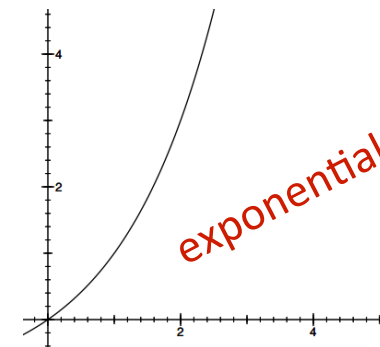
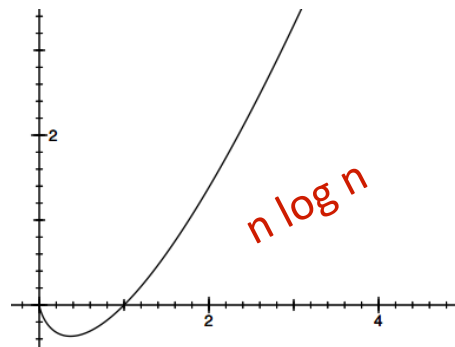
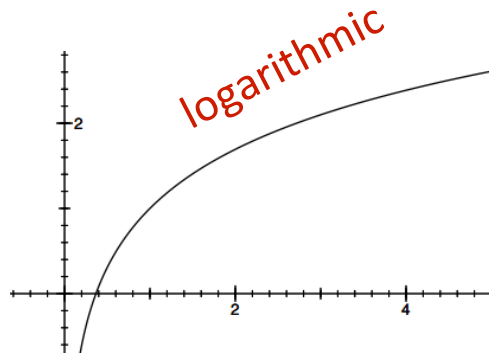
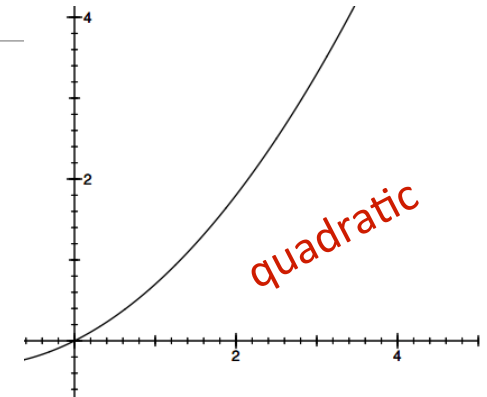
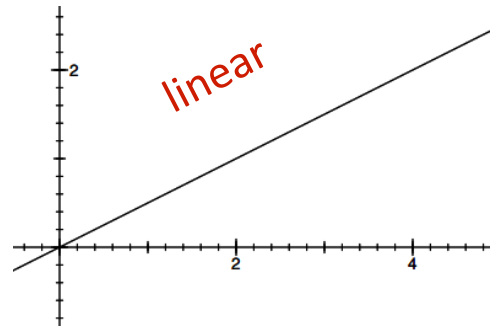
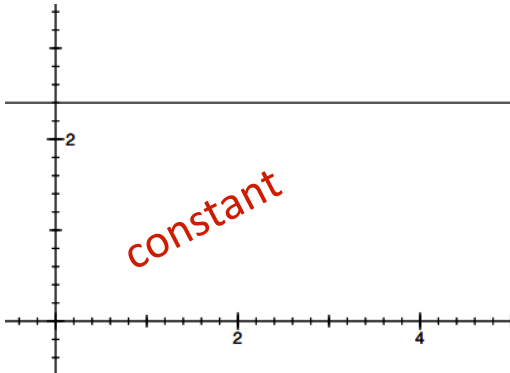
$$O(n^2) : n^2 + 100000n + 3^{1000}$$

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

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

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

# 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')
```

$O(n)$   
 $O(n^2)$

}

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

is  $O(n) + O(n^2) = O(n+n^2) = O(n^2)$  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  $O(n) \rightarrow O(n)*O(n)$

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

---

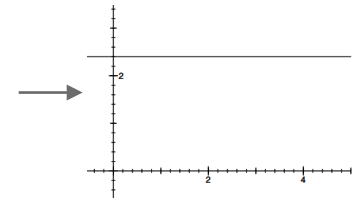
- $O(1)$  denotes constant running time
- $O(\log n)$  denotes logarithmic running time
- $O(n)$  denotes linear running time
- $O(n \log n)$  denotes log-linear running time
- $O(n^c)$  denotes polynomial running time ( $c$  is a constant)
- $O(c^n)$  denotes exponential running time ( $c$  is a constant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

# COMPLEXITY CLASSES ORDERED LOW TO HIGH

$O(1)$

:

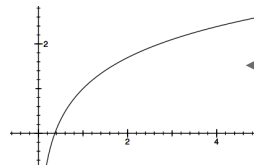
constant



$O(\log n)$

:

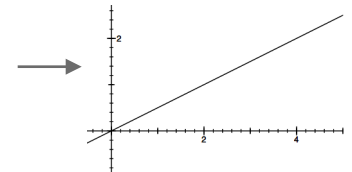
logarithmic



$O(n)$

:

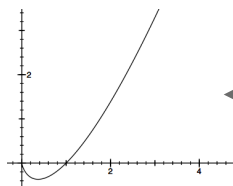
linear



$O(n \log n)$

:

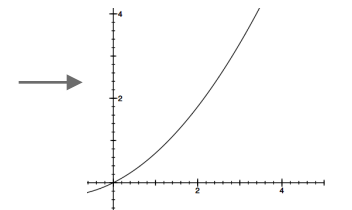
loglinear



$O(n^c)$

:

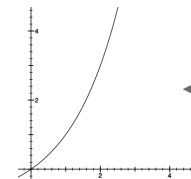
polynomial



$O(c^n)$

:

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	1071508607186267320948425049060 0018105614048117055336074437503 8837035105112493612249319837881 5695858127594672917553146825187 1452856923140435984577574698574 8039345677748242309854210746050 6237114187795418215304647498358 1941267398767559165543946077062 9145711964776865421676604298316 52624386837205668069376	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
- $O(\text{len}(L))$  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  $\text{len}(L)$**

Assumes we can  
retrieve element  
of list in constant  
time

# CONSTANT TIME LIST ACCESS

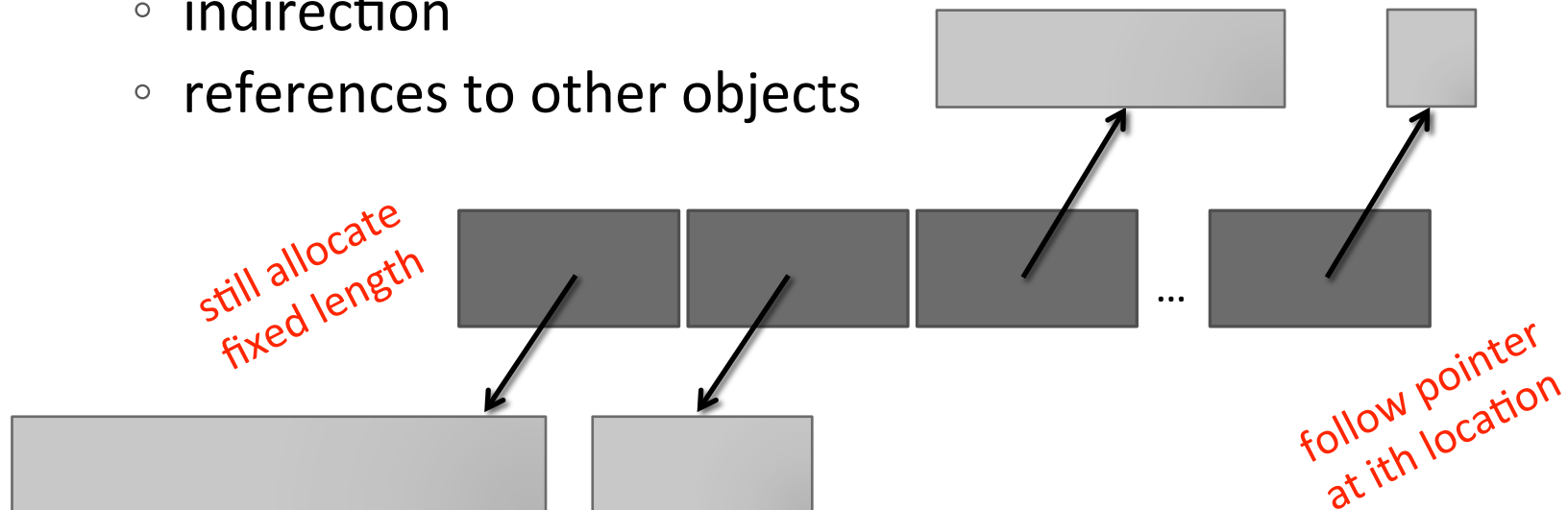
- if list is all ints

- $i^{\text{th}}$  element at
- $\text{base} + 4*i$



- 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
- $O(\text{len}(L))$  for the loop \*  $O(1)$  to test if  $e == L[i]$
- overall complexity is  **$O(n)$  – where  $n$  is  $\text{len}(L)$**
- **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
```

- $O(\text{len}(s))$

# 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  $\text{len}(L1)$  times

each iteration will execute inner loop up to  $\text{len}(L2)$  times, with constant number of operations

$O(\text{len}(L1) * \text{len}(L2))$

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

$O(\text{len}(L1)^2)$

# 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  
 $\text{len}(L1) * \text{len}(L2)$  steps

second loop takes at  
most  $\text{len}(L1)$  steps

determining if element  
in list might take  $\text{len}(L1)$   
steps

if we assume lists are of  
roughly same length,  
then

$O(\text{len}(L1)^2)$

# 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

MIT OpenCourseWare  
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python  
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.