

# CS 61A Discussion 7

## - Orders of Growth & Trees -

March 09, 2017

# Announcements

— — —

MT2 is next Wednesday (3/15) from 8-10pm. Fill out the conflict form by Friday!

There will be a guerrilla section this Saturday from 12-3pm in 247 Cory.

- Topics covered: OOP (/inheritance), orders of growth

No section next week! We'll be grading.

**Today: orders of growth (a method for measuring efficiency), object-oriented Trees**

# Orders of Growth

“Are my programs  
efficient?”

(answer: no)  
(jk)



**Wait.**

**How do you know if a  
program is efficient?**

# *How do you know if a program is efficient?*

— — —

You might think, well, look at how long it takes to run.

If it takes a day, is that efficient? A day sounds like a lot at first.

But maybe you're immortal, like Paul Hilfinger, and a day is nothing to you. Or maybe your problem is HUGE... maybe it's the problem of counting every molecule in the universe, twice. And if that's the case, one day is actually miraculous. **Moral of the story: always define the size of the problem.** "Efficiency" only makes sense if you know the size of the problem!

## *How do you know if a program is efficient? (cont.)*

— — —

Moreover, maybe time is nothing to you. You could be a time traveler.

(...with only 1MB of RAM.)

Here, it's *space* you care about. How much **memory** is your program going to consume as a function of input size, compared to all other programs that do the same thing?

Programs can be efficient in different ways.

# so as a tl;dr

— — —

Efficiency is relative.

To define efficiency, we'll look at how many resources our program uses **in relation to the size of the problem**.

This is where orders of growth come in.



# Orders of Growth, formalized

— — —

An order of growth is a **function** describing how something scales (usually a resource; time or space) with respect to some input.

You'll hear things like this: “what is the *order of growth* of <function-name>'s running time, in terms of <input-name(s)>?”

Common orders of growth:

$O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^c)$ ,  $O(c^n)$ ... [pretty much any mathematical function of the input can be an order of growth]

# Orders of Growth, formalized (cont.)

Orders of growth allow us to assert that certain functions run more quickly (as a function of their input), or **scale better**, than others.

This is what happens when all of the 170 students run their garbage  $O(n^4)$ -time project code on the instructional machines<sup>1</sup> →

This site displays usage stats for the Berkeley EECS instructional computers.

The data below was gathered **a few seconds ago**. Refresh to check for updates.

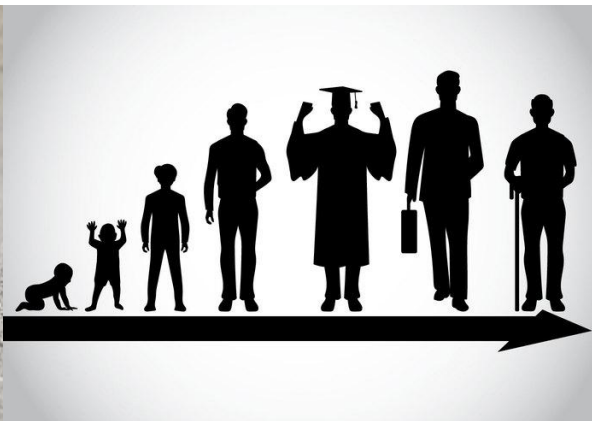
The least-busy Hive server is [hive17.cs.berkeley.edu](https://hive17.cs.berkeley.edu).

Server name ↕	Overall load? ↕	User count ↕	CPU usage? ↓
hive19	• HIGH	8 users	11076.13%
hive21	• HIGH	11 users	6857.75%
hive20	• HIGH	9 users	6010.38%
hive15	• HIGH	12 users	5967.38%
hive7	• HIGH	6 users	5697.88%
hive5	• HIGH	18 users	5680.63%
hive8	• HIGH	11 users	5678.63%
hive3	• HIGH	15 users	5677.13%
hive10	• HIGH	16 users	5640.13%

<sup>1</sup> This isn't actually that great a representation of runtime growth, since a bunch of people are probably just looping indefinitely and running lots of instances.

# Orders of Growth: Visual Examples

— — —



A rock; we could argue that its growth is an  $O(1)$  function of time

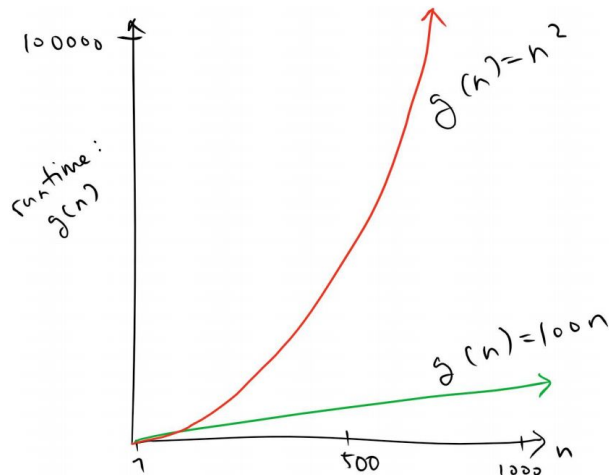
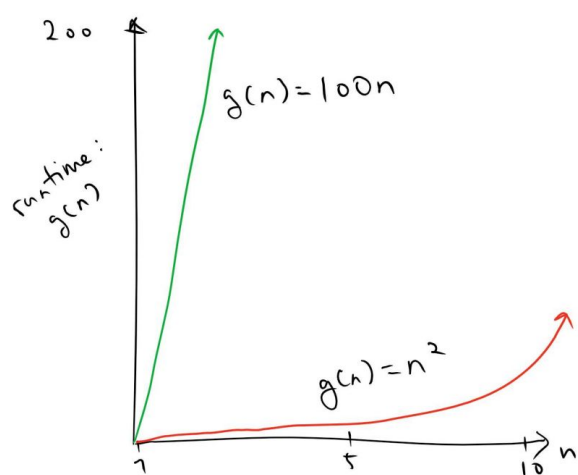
Age increases linearly over time, of course

The wheat and chessboard problem

# A Rough Approximation

— — —

We don't care about small inputs; we can always handle those pretty easily anyway. We care about what happens as the input size gets REALLY BIG (as it approaches infinity, even!). Small input sizes aren't necessarily representative anyway:



(Thanks to Jerome Baek for this great visual!)

# A Rough Approximation, continued

— — —

Asymptotically, only the highest-order term (or terms if there are multiple input variables) in the growth function matters. Therefore, that's the only term we retain.

ex.  $n^3 + 40000n^{2.1} + 26$  becomes  $n^3$

For similar reasons we'll also omit constant multipliers for that first term (it helps with standardization, and anyway we want to stay focused on the big - asymptotic - picture).

$65\sqrt{n} + \log n + \log(\log n)$  becomes  $\sqrt{n}$

# Summary So Far

— — —

An order of growth is just a function that depicts how stuff (like running time) scales.  $O(f(n))$  means that aforementioned “stuff” increases no faster than the  $f(n)$ -class of functions as  $n$  gets larger and larger. This can be useful for guaranteeing efficiency in the temporal or spatial domain.

When determining an order of growth, do two things:

1. Drop lower-order terms.
2. Drop multiplicative constants.

Neither of these descriptors is asymptotically relevant.

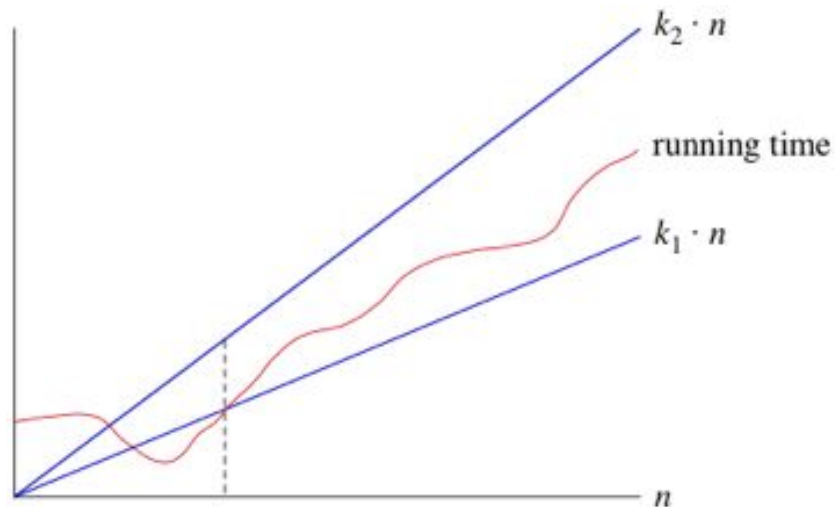
**Vaguely Mathematical Depiction**  
**ft. big theta**

# Big-Theta

— — —

The definition of Big-Theta:

If we say that the order of growth is  $f(n)$ , then there exist positive constants  $k_1$  and  $k_2$  such that the ACTUAL order of growth is sandwiched between  $k_1 \cdot f(n)$  and  $k_2 \cdot f(n)$  for sufficiently large values of  $n$ .



In this diagram, the “actual” order of growth is the red line. The blue lines are the upper and lower bounds.

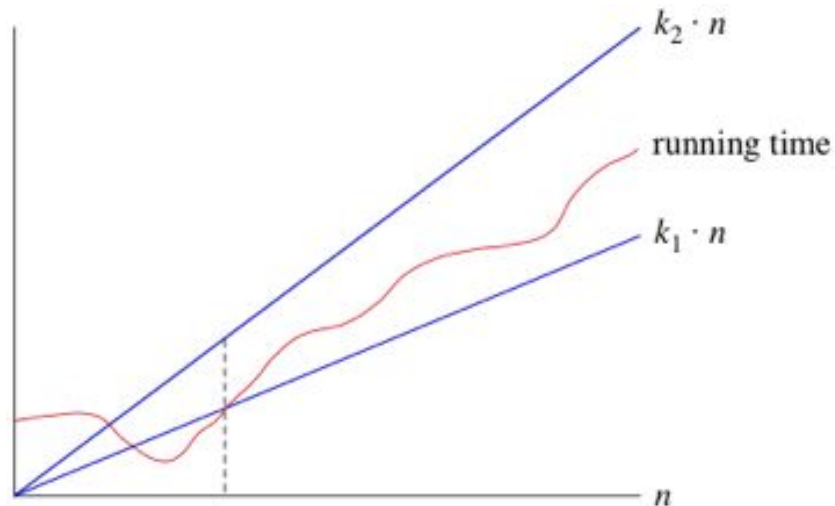


# Big-Theta, cont.

---

If the growth function (e.g. runtime as a function of input size) is ALWAYS sandwiched between  $0.5n$  and  $1n$  when  $n$  is really large, then the order of growth would be  $\Theta(n)$ .

In other words, if  $k_1 = 0.5$  and  $k_2 = 1$  in the diagram to the right, then the running time can be said to be  $\Theta(n)$ .



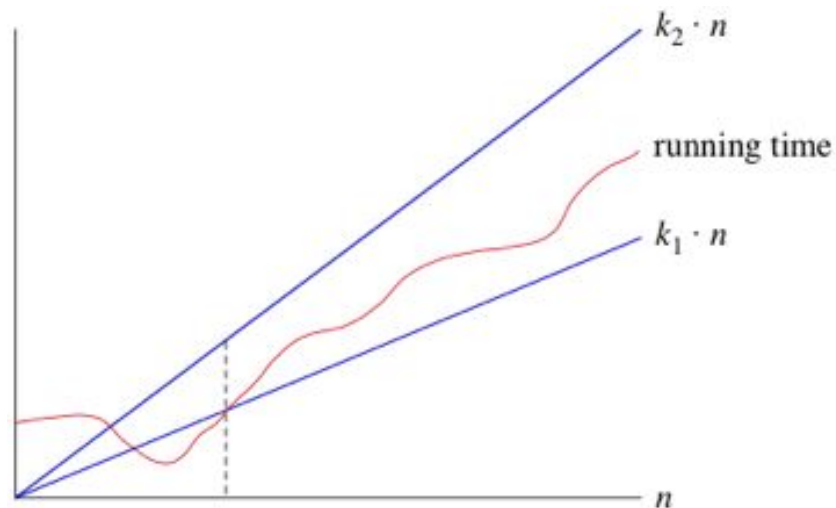
# Big-O

---

The definition of Big-O:

If we say that the order of growth is  $f(n)$ , then there exists a constant  $k_2$  for which the ACTUAL order of growth is BELOW  $k_2 \cdot f(n)$  for sufficiently large values of  $n$ .

(Basically, it's only the upper bound.)

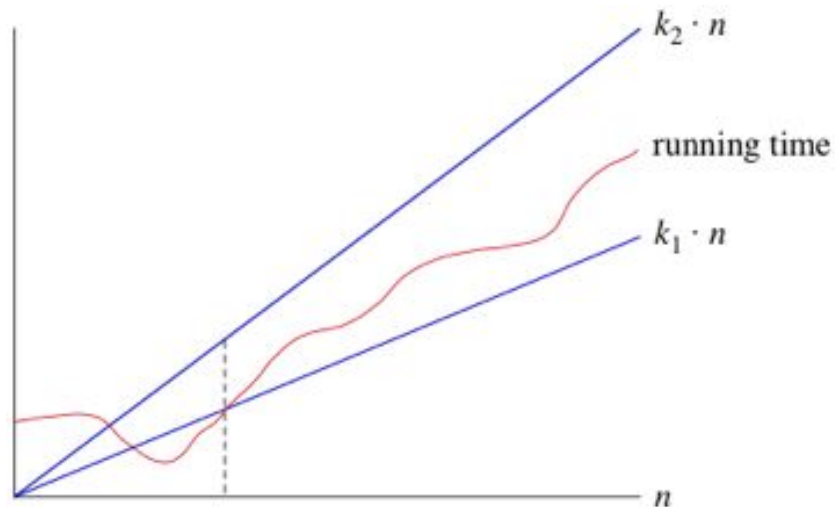


# Big-O vs Big-Theta

— — —

In this class (and in practice), you try to use Big-O as Big-Theta because it's not very informative otherwise (i.e. pretty much everything is *technically*  $O(n^n)$ , but who cares?).

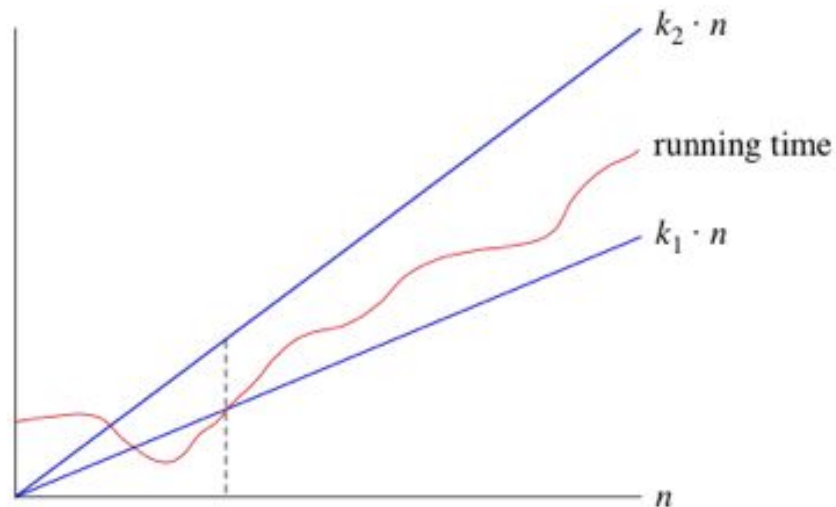
tl;dr Even though Big-O technically only refers to an upper bound, we want you to find the “tightest” bound.



# Big-O vs Big-Theta

— — —

So if  $k_1 = 0.5$  and  $k_2 = 1$  in our diagram, we'd say that the order of growth is  $\Theta(n) \approx O(n)$ .



# Determining Order of Growth

# Approach

— — —

*If faced with a function of unknown time complexity:*

Go through the function line-by-line, determining approximately how much time each block of code takes as a function of  $n$ . Then add them all together (by “them” I mean your estimation for each region of code) and drop lower-order terms. That’s pretty much it.

If there’s recursion (which there will be), figure out how much work there is to be done in each call and how many calls there’ll be. Then multiply those things together.

# An Example [Summer 2012 Final | Q2(c)]

— — —

```
def carpe_noctem(n):  
    if n <= 1:  
        return n  
    return carpe_noctem(n - 1) \  
        + carpe_noctem(n - 2)
```

```
def yolo(n):  
    if n <= 1:  
        return 5  
    sum = 0  
    for i in range(n):  
        sum += carpe_noctem(n)  
    return sum + yolo(n - 1)
```

**Question:** What is the order of growth in  $n$  of the runtime of `yolo`, where  $n$  is its input?

**Answer:**  
- Well, going through each line...

# An Example [Summer 2012 Final | Q2(c)]

— — —

```
def carpe_noctem(n):  
    if n <= 1:  
        return n  
    return carpe_noctem(n - 1) + carpe_noctem(n - 2)
```

```
def yolo(n):  
    if n <= 1: # this block is O(1) on its own  
        return 5  
    sum = 0  
    for i in range(n):  
        sum += carpe_noctem(n)  
    return sum + yolo(n - 1)
```



# An Example [Summer 2012 Final | Q2(c)]

— — —

```
def carpe_noctem(n):  
    if n <= 1:  
        return n  
    return carpe_noctem(n - 1) + carpe_noctem(n - 2)
```

```
def yolo(n):  
    if n <= 1: # this block is O(1) on its own  
        return 5  
    sum = 0 # O(1)  
    for i in range(n):  
        sum += carpe_noctem(n)  
    return sum + yolo(n - 1)
```

# An Example [Summer 2012 Final | Q2(c)]

— — —

```
def carpe_noctem(n):  
    if n <= 1:  
        return n  
    return carpe_noctem(n - 1) + carpe_noctem(n - 2)
```

```
def yolo(n):  
    if n <= 1: # this block is O(1) on its own  
        return 5  
    sum = 0 # O(1)  
    for i in range(n): # this block is O(n)  
        sum += carpe_noctem(n)  
    return sum + yolo(n - 1)
```

# An Example [Summer 2012 Final | Q2(c)]

— — —

```
def carpe_noctem(n):  
    if n <= 1:  
        return n  
    return carpe_noctem(n - 1) + carpe_noctem(n - 2)  
  
def yolo(n):  
    if n <= 1: # this block is O(1) on its own  
        return 5  
    sum = 0 # O(1)  
    for i in range(n): # this block is O(n),  
        sum += carpe_noctem(n) # times whatever the OOG of carpe_noctem is  
    return sum + yolo(n - 1)
```

# An Example [Summer 2012 Final | Q2(c)]

---

```
def carpe_noctem(n):  
    if n <= 1:  
        return n  
    return carpe_noctem(n - 1) + carpe_noctem(n - 2)  
  
def yolo(n):  
    if n <= 1: # this block is O(1) on its own  
        return 5  
    sum = 0 # O(1)  
    for i in range(n): # this block is O(n),  
        sum += carpe_noctem(n) # times whatever the OOG of carpe_noctem is  
    return sum + yolo(n - 1) # and then there's another recursive call
```

# An Example [Summer 2012 Final | Q2(c)]

---

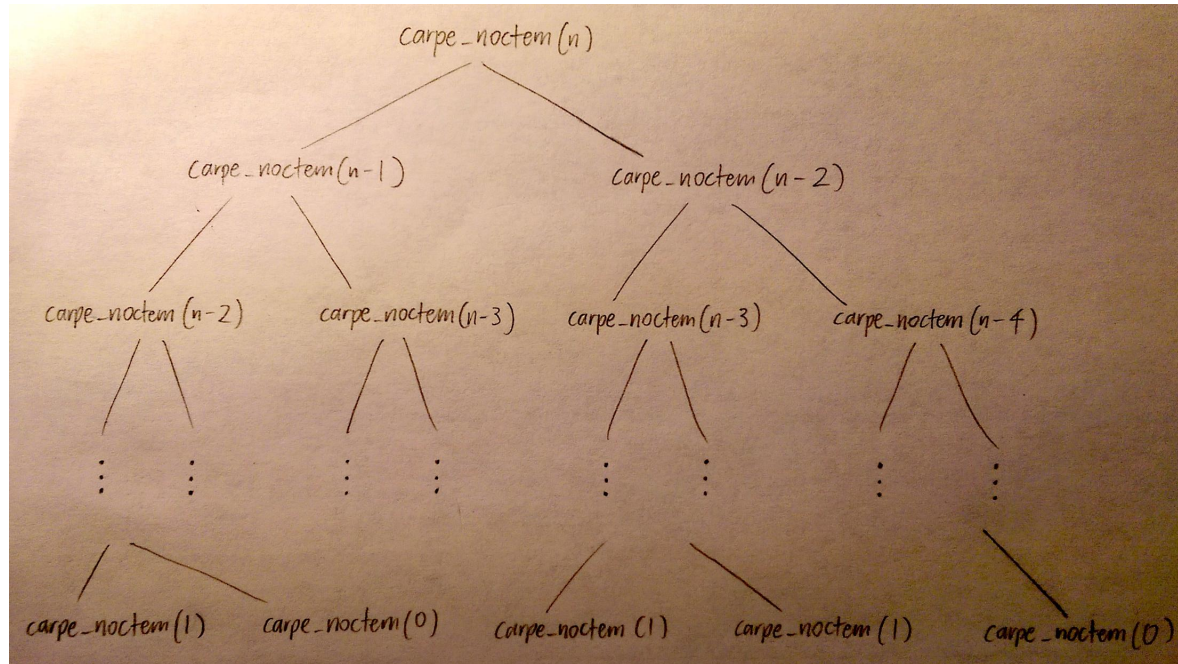
```
def carpe_noctem(n):
    if n <= 1: # this block is O(1) on its own
        return n
    return carpe_noctem(n - 1) + carpe_noctem(n - 2) # TWO recursive calls

def yolo(n):
    if n <= 1: # this block is O(1) on its own
        return 5
    sum = 0 # O(1)
    for i in range(n): # this block is O(n),
        sum += carpe_noctem(n) # times whatever the OOG of carpe_noctem is
    return sum + yolo(n - 1) # and then there's another recursive call
```

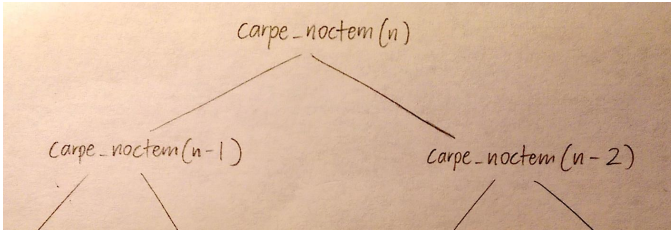
# An Example [Summer 2012 Final | Q2(c)]

---

Recursive call tree for `carpe_noctem` (authentic handwritten edition):



# An Example [Summer 2012 Final | Q2(c)]

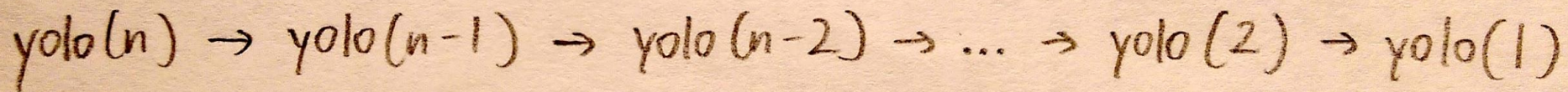


The call tree is a binary tree of depth  $(n - 1)$ . There are at most  $2^{k+1} - 1$  nodes in a binary tree of depth  $k$ , which means that there are at most  $2^{(n-1)+1} - 1 = 2^n - 1$  nodes in *this* tree.

Each of these nodes represents a call to `carpe_noctem`, and we do  $O(1)$  work in the body of each of these calls, so a single `carpe_noctem` call produces  $O(2^n)$  recursive `carpe_noctem` calls and  $O(1) * O(2^n) = O(2^n)$  work overall. **Conclusion:** `carpe_noctem`'s growth function is  $O(2^n)$ .

# An Example [Summer 2012 Final | Q2(c)]

---



A handwritten sequence of recursive calls for a function named 'yolo'. The sequence is written in brown ink on a light brown, textured background that resembles a piece of paper or a chalkboard. The sequence starts with 'yolo(n)' and follows a pattern of decreasing the argument by 1, indicated by arrows, until it reaches 'yolo(1)'. The sequence is:  $yolo(n) \rightarrow yolo(n-1) \rightarrow yolo(n-2) \rightarrow \dots \rightarrow yolo(2) \rightarrow yolo(1)$ .

The call sequence for yolo, meanwhile, is a lot simpler. It should be clear from the above sketch that  $yolo(n)$  involves  $n$  calls to yolo. Accordingly, we know that the stuff in the body of yolo happens  $n$  times in total.



# An Example [Summer 2012 Final | Q2(c)]

— — —

```
def carpe_noctem(n):  
    # stuff happens here, but all we need to know is that it's  $O(2^n)$   
  
def yolo(n):  
    #  $O(1)$  stuff  
    for i in range(n): okay, we do the stuff in the loop  $n$  times  
        sum += carpe_noctem(n) # carpe_noctem is  $O(2^n)$   
    return sum + yolo(n - 1) # and then all of this stuff happens  $n$  times again
```

In conclusion: the body of yolo is  $O(n * 2^n)$ , and then that code gets executed  $n$  times. Therefore yolo is, holistically speaking, an  $O(n * n * 2^n) = O(n^2 * 2^n)$  function because that's how much work we do as a result of one yolo( $n$ ) call.

# General Heuristics (Not Guarantees!)

— — —

Warning: this stuff definitely isn't always going to be true (especially in the case of exam questions designed to filter out confused students).

- When there are  $c$  recursive calls in the function body (tree recursion), there tends to be  $O(c^n)$  calls overall (exponential growth).
- Double-nested for-loops tend to indicate that you'll do the stuff in the inner loop  $n^2$  times.
- If you make multiplicative progress during every step (e.g. by dividing problem size by 2 or multiplying something by 3), it's likely logarithmic growth.

# Individual Function Descriptions (+ Exercises)

# Live Answer Submission Link

— — —

(Process: I review one classification at a time, and present a function after each one. Your job is to identify the order of growth of each function's running time.)

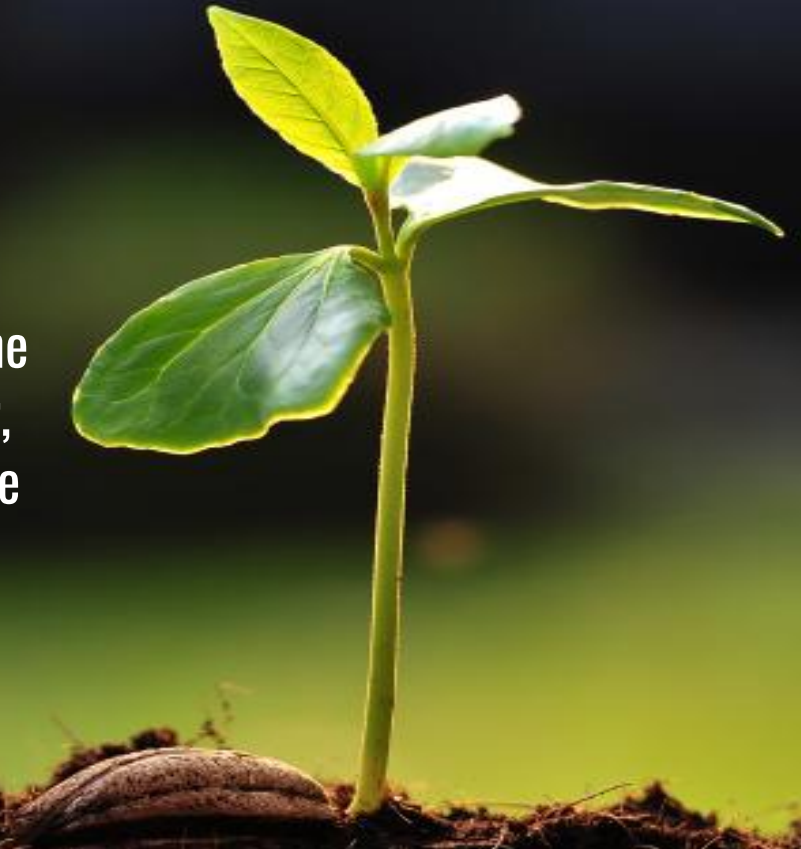
It'd be great if you guys can anonymously submit your answers as we go, so I can see how you're doing and figure out how difficult everything is.

Link to live / anonymous poll: [**edit: done through Poll Everywhere, see alt. version of slides**]

Don't worry if your answer is different from everyone else's.



**DISCLAIMER** In the following slides, we treat the growth function as a runtime descriptor. However, note that orders of growth can be used to describe *any* phenomena that scale as a function of their inputs (memory is another big one, for example).



**DISCLAIMER 2** I say Big-O in pretty much all of these slides, but I'm really only doing that because " $\Theta$ " is a damn pain to type.

I say Big-O, but I mean Big- $\Theta$ .



**DISCLAIMER 3** I have a lot of slides on OOG. But if you're primarily concerned with exams (not a perspective I recommend :/ ), don't expect this to be representative of a topical distribution on a MT. ...OOG is usually like two points on a test.





# $O(1)$

**Constant** time. Best order of growth for scalability; runtime is not affected by the input size.

```
def const(n):  
    n = 902 + 54  
    return 'hamburger'
```

# $O(\log n)$

**Logarithmic** time. Amazingly scalable; a multiplicative increase in input size leads to an additive increase in running time.

```
def loga(n):  
    if n <= 1:  
        return 1  
    return n * loga(n // 2)
```

# Exercise 1

— — —

```
def mystery1(n):  
    n, result = 5, 0  
    while n <= 3000:  
        result += const(n // 2)  
        n += 1  
    return result
```

**Reminder:** we want the order of growth of the runtime a function of  $n$ .

**Example answers:**  $O(1)$ ,  $O(n)$ ,  $O(n^2)$ ...

# Exercise 1 Solutions

— — —

```
def mystery1(n):  
    n, result = 5, 0  
    while n <= 3000:  
        result += const(n // 2)  
        n += 1  
    return result
```

**$O(1)$ .**

Notes: The input  $n$  doesn't even matter!

# $O(\sqrt{n})$

**Square-root** time, aka  
knockoff logarithmic time  
(runtime still increases  
slowly with input size).  
Better than  $O(n)$ , but  
rarely observed.

```
def sqroot(n):  
    lim = int(sqrt(n))  
    for i in range(lim):  
        n += 45  
    return n
```

# Exercise 2

— — —

```
def mystery2(n):  
    if n < 0 or sqrt(n) <= 50:  
        return 1  
    return n * mystery2(n // 2)
```

# Exercise 2 Solutions

— — —

```
def mystery2(n):  
    if n < 0 or sqrt(n) <= 50:  
        return 1  
    return n * mystery2(n // 2)
```

**$O(\log n)$ .**  $\text{sqrt}(n) \leq 50$  is equivalent to  $n \leq 2500$ , so this ends up being a standard logarithmic-time algorithm

# $O(n)$

**Linear** time. Still very scalable; adding a constant to the input size also adds a constant to the runtime.

```
def lin(n):  
    if n <= 1:  
        return 1  
    return n + lin(n - 1)
```



# Exercise 3

— — —

```
def mystery3(n):  
    result = 0  
    for i in range(n // 10):  
        result += 1  
        for j in range(10):  
            result += 1  
            for k in range(10 // n):  
                result += 1  
    return result
```

# Exercise 3 Solutions

— — —

```
def mystery3(n):  
    result = 0  
    for i in range(n // 10):  
        result += 1  
        for j in range(10):  
            result += 1  
            for k in range(10 // n):  
                result += 1  
    return result
```

**$O(n)$** . The number of iterations in the j-loop is based on a constant, and for large values of  $n$  (specifically when  $n > 10$ ) there are 0 iterations in the k-loop

# $O(n^2)$

**Quadratic** time. Still polynomial, so it could be worse; multiplying input size by a constant factor ends up multiplying the runtime by the square of that factor.

```
def quad(n):  
    if n <= 1:  
        return 1  
    r = lin(n) * quad(n - 1)  
    return r
```

# $O(2^n)$

**Exponential** time. Not scalable at all; identifies problems as intractable. Adding to the input size multiplies the runtime.

```
def expo(n):  
    if n <= 1:  
        return 1  
    r1 = expo(n - 1) + 1  
    r2 = expo(n - 1) + 2  
    return r1 * r2
```

# A General Timing Comparison

— — —

	n = 10	n = 50	n = 100	n = 1000
logn	0.0003s	0.0006s	0.0007s	0.0010s
sqrt(n)	0.0003s	0.0007s	0.0010s	0.0032s
n	0.0010s	0.0050s	0.0100s	0.1000s
nlogn	0.0033s	0.0282s	0.0664s	0.9966s
$n^2$	0.0100s	0.2500s	1.0000s	100.00s
$n^6$	1.6667m	18.102d	3.1710y	3171.0c
$2^n$	0.1024s	35.702c	$4 \times 10^{16}c$	$1 \times 10^{166}c$
$n!$	362.88s	$1 \times 10^{51}c$	$3 \times 10^{144}c$	$1 \times 10^{2554}c$

← Time required to process n items at a speed of 10,000 operations per second, using eight different algorithms

s = seconds

m = minutes

d = days

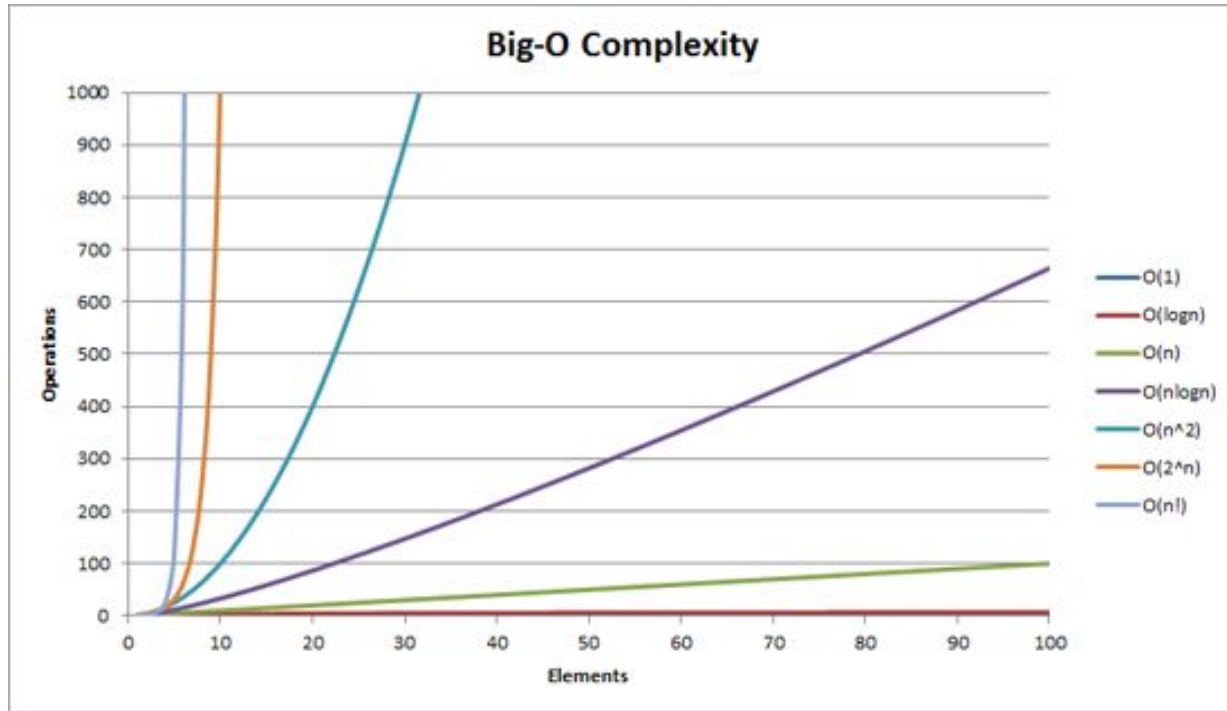
y = years

c = *centuries*

Source:

<http://www.ccs.neu.edu/home/jaa/CS7800.12F/Information/Handouts/order.html>

\_\_\_\_\_



# Exercise 4

— — —

```
def mystery4(n):  
    if n > 0:  
        r1 = mystery10(-n)  
        r2 = mystery10(n - 1)  
        return r1 + r2  
    return 1
```

# Exercise 4 Solutions

— — —

```
def mystery4(n):  
    if n > 0:  
        r1 = mystery10(-n)  
        r2 = mystery10(n - 1)  
        return r1 + r2  
    return 1
```

**0(n).** The first recursive call can never go anywhere



**Saving the best for last:  
a single slide on the Tree class**

# The Tree class

— — —

You guys already know trees. This is the same thing as the ADT version, except formalized using Python's object system. You can mutate these trees by modifying their attributes. Yay!

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```