CS 61A: Orders of Growth

6 December 2016

Orders of Growth: Boring Text Version

— — —

An order of growth is a **function** describing how something scales (usually a resource; time or space) according 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:

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

Orders of Growth: More Boring Text

———

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) project code on the instructional machines →

Hivemind

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**.

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%

Orders of Growth: (Also Boring) Visual Examples

— — —

A rock; we could argue that its growth is an O(1) function of time
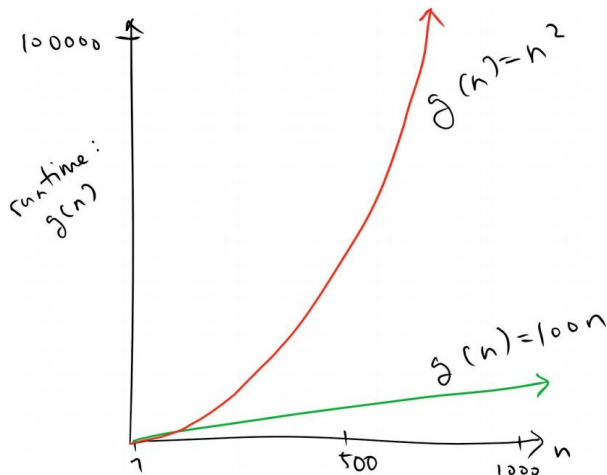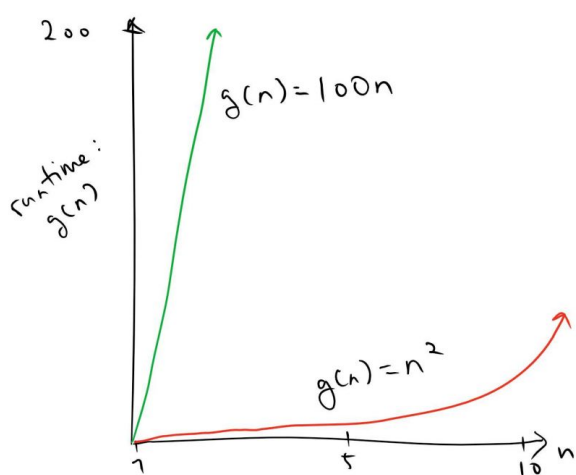
Age increases linearly over time, of course

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. Small input sizes aren't necessarily representative anyway:

(Thanks to Jerome Baek for this great visual!)

A Rough Approximation, continued

———

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

ex. **n^3 + 40000$n^{2.1}$ + 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$\sqrt{n}$ + logn + log(logn) becomes **$\sqrt{n}$**

Summary So Far

———

An order of growth is just a function that depicts how stuff (e.g. running time) scales.

When determining an order of growth, do two things:

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

Neither descriptor is asymptotically relevant.

Vaguely Mathematical Depiction
(Out of Scope for 61A)

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 for which 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.
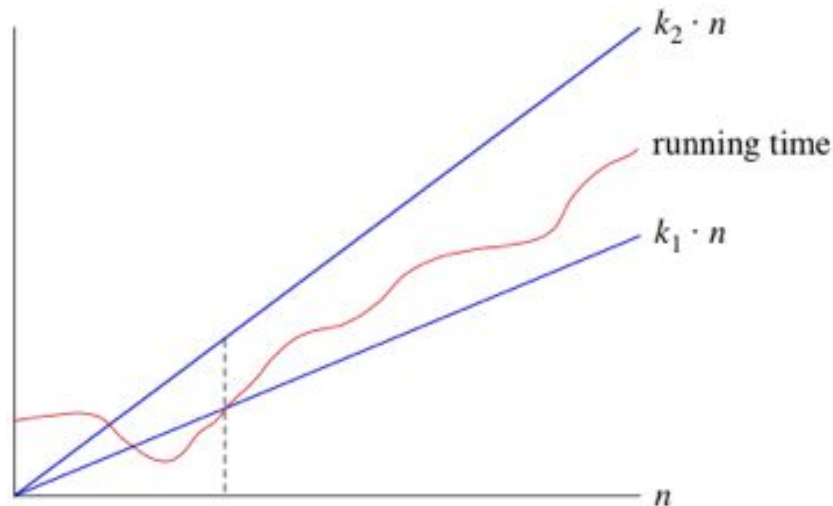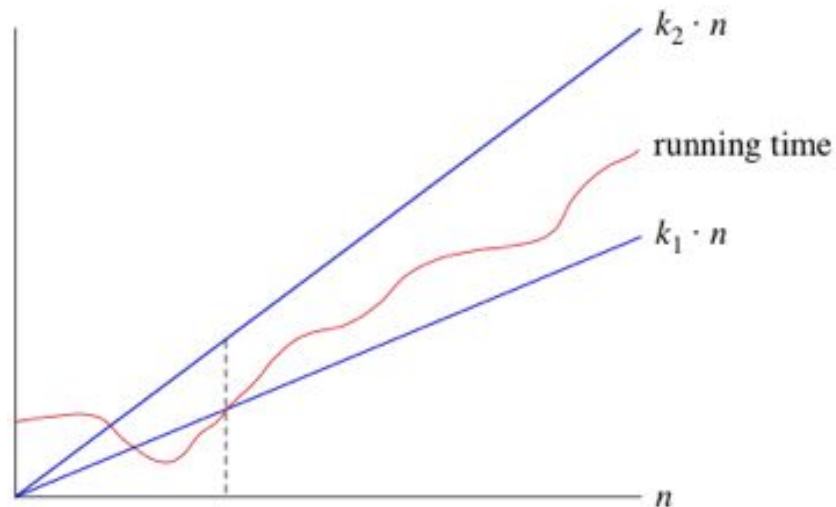

In this diagram, the "actual" order of growth is the red line. The blue lines are the upper and lower bounds.

Big-Theta, cont.

———

If the growth function (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 Θ(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 Θ(n).
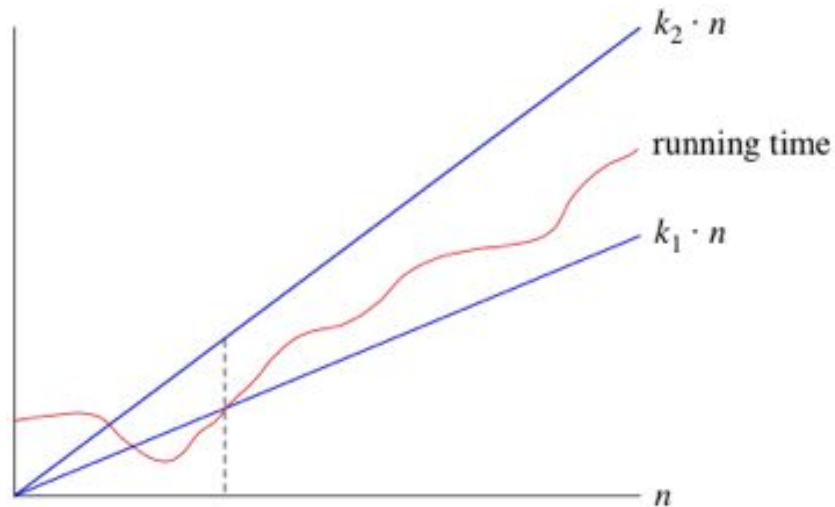
Big-O

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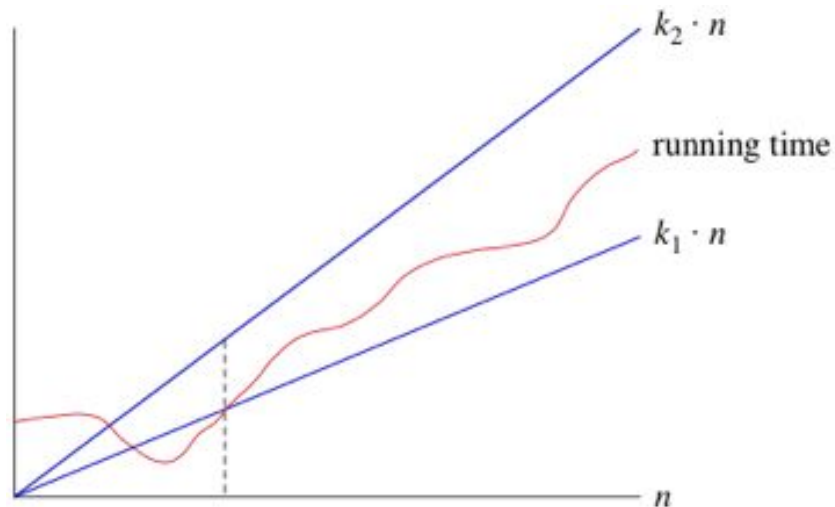
(Basically, it's only the upper bound.)

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 $\Theta(n) \approx O(n)$.
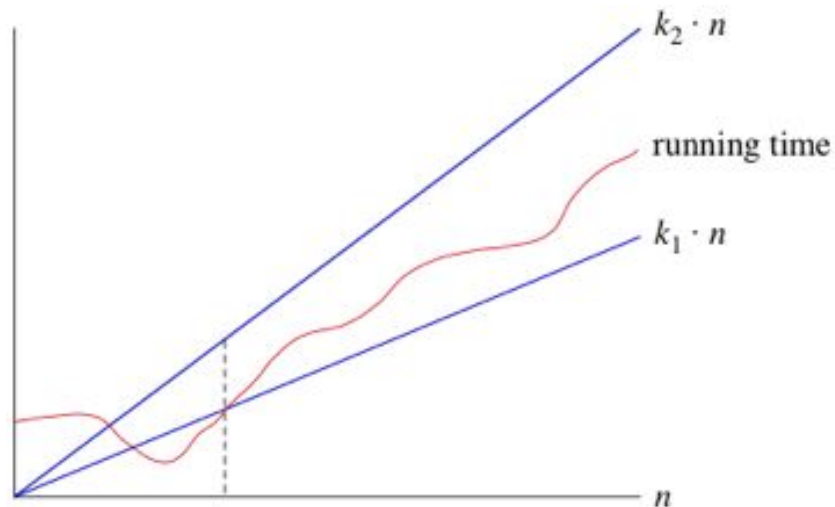
Determining Order of Growth

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 sum 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: # 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) times...
        sum += carpe_noctem(n) # 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):
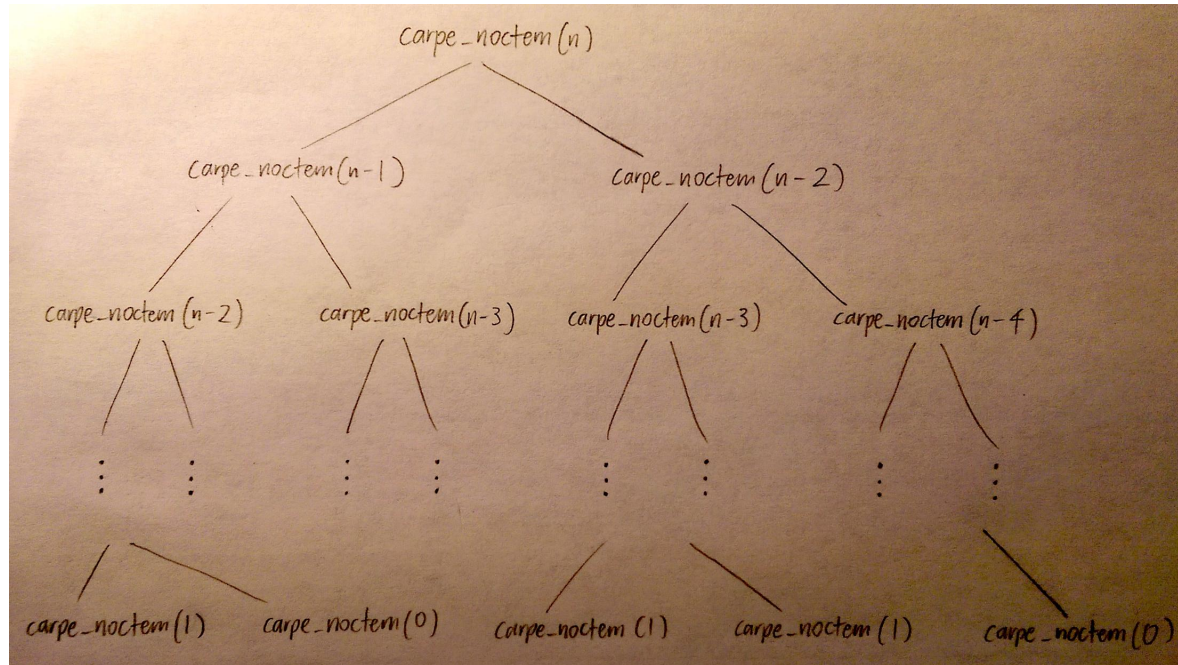
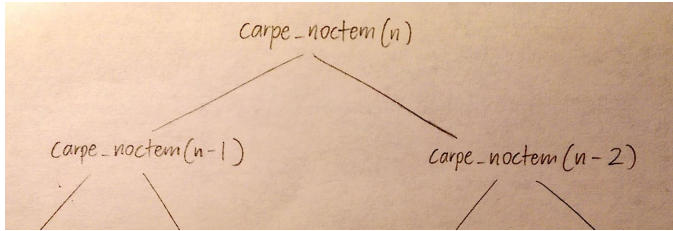# An Example [Summer 2012 Final | Q2(c)]

‒ ‒ ‒

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)]

$$yolo(n) \rightarrow yolo(n-1) \rightarrow yolo(n-2) \rightarrow ... \rightarrow yolo(2) \rightarrow yolo(1)$$

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.

Individual Function Descriptions (+ Exercises)

Live Answer Submission Link

———

(Process: I review one classification at a time, and present 2-3 functions after each one. Your job is to identify the order of growth of each function's running time, where the answer will tend to focus on the classifications I've already been through. There may or may not be exceptions, though.)

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: this'll be done through Pear Deck**]

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).
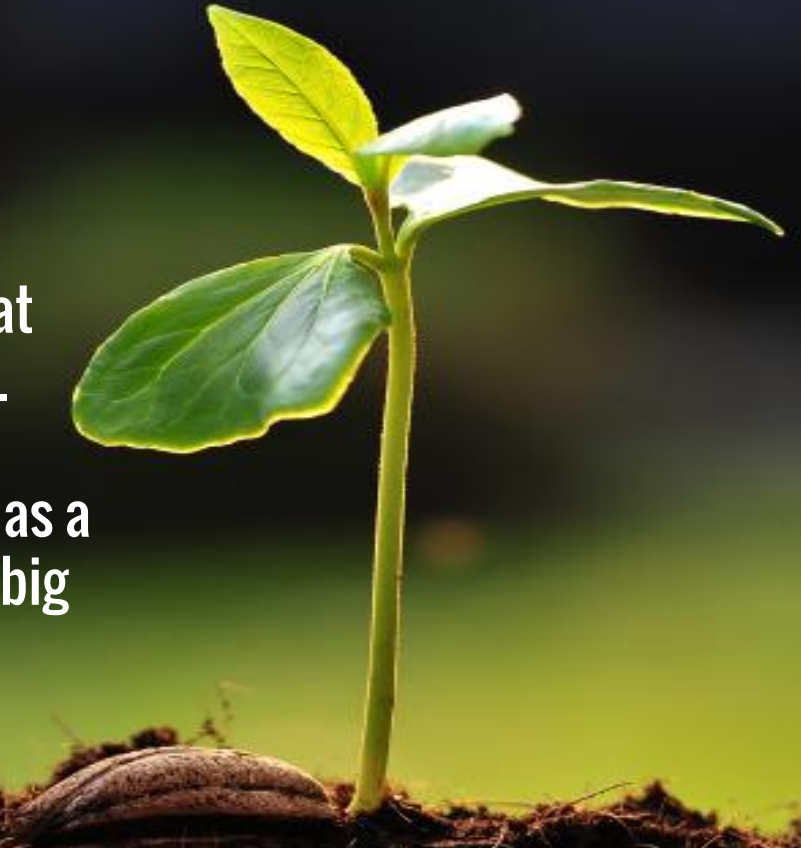
O(1)

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(logn)

Logarithmic time. Amazingly scalable; a <u>multiplicative</u> increase in input size leads to an <u>additive</u> increase in running time.

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

Exercise 1

— — —

```
def mystery1(n):
    n, result = str(n), ''
    num_digits = len(n)
    for i in range(num_digits):
        result += n[num_digits - i - 1]
    return result
```

```
def mystery2(n):
    n, result = 5, 0
    while n <= 3000:
        result += mystery1(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(n²)...

Exercise 1 Solutions

— — —

```
def mystery1(n):
    n, result = str(n), ''
    num_digits = len(n)
    for i in range(num_digits):
        result += n[num_digits - i - 1]
    return result
```

O(logn).
Notes: str(n) is O(logn), len(n) is O(1)

```
def mystery2(n):
    n, result = 5, 0
    while n <= 3000:
        result += mystery1(n // 2)
        n += 1
    return result
```

O(1).
Notes: The input n doesn't even matter!

Exercise 1 Solutions

— — —

```
def mystery1(n):
    n, result = str(n), ''
    num_digits = len(n)
    for i in range(num_digits):
        result += n[num_digits - i - 1]
    return result
```

O(logn).
I anticipate this one being confusing (although I would love for you guys to prove me wrong so I don't have to use this slide!). Let's go through it a bit.

1. str(n) is O(logn) because you have to MULTIPLY n by your radix in order to ADD one digit to your output string.

2. len(n) is O(1) because Python strings keep track of their own length, but you should realize that it's at worst O(logn) for the same reasons as 1 (there are only O(logn) digits).

3. Since there are only O(logn) digits, the loop simply performs constant-time[1] indexing and addition O(logn) times.

[1] We're not interested in bit-level complexity here.

O(√n)

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

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

Exercise 2

— — —

```python
def mystery3(n):
    if n < 0 or n <= sqrt(n):
        return n
    return n + mystery3(n // 3)
```

```python
def mystery4(n):
    if n < 0 or sqrt(n) <= 50:
        return 1
    return n * mystery4(n // 2)
```

```python
def mystery5(n):
    for _ in range(int(sqrt(n))):
        n = 1 + 1
    return n
```

Exercise 2 Solutions

———

```
def mystery3(n):
    if n < 0 or n <= sqrt(n):
        return n
    return n + mystery3(n // 3)
```

O(logn). n <= sqrt(n) will only be hit when
0 <= n <= 1

```
def mystery5(n):
    for _ in range(int(sqrt(n))):
        n = 1 + 1
    return n
```

O(sqrt(n)). Possible confusion: sqrt(n) is only computed once at the beginning

```
def mystery4(n):
    if n < 0 or sqrt(n) <= 50:
        return 1
    return n * mystery4(n // 2)
```

O(logn). sqrt(n) <= 50 is equivalent to
n <= 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.

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

Exercise 3

— — —

```
def mystery6(n):
    while n > 1:
        x = n
        while x > 1:
            print(n, x)
            x = x // 2
        n -= 1
```

(Thanks to Mark Miyashita for this problem!)

```
def mystery7(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 mystery6(n):
    while n > 1:
        x = n
        while x > 1:
            print(n, x)
            x = x // 2
        n -= 1
```

O(nlogn). Inner loop is O(logn), and it happens O(n) times

```
def mystery7(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 the k-loop won't happen once n is sufficiently large.

O(n²)

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

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

Exercise 4

———

```python
def mystery8(n):
    if n == 0: return ''
    result, stringified = '', str(n)
    for digit in stringified:
        for _ in range(n):
            result += digit
    result += mystery8(n - 1)
    return result
```

```python
def mystery9(n):
    total = 0
    for i in range(1, n):
        total *= 2
        if i % n == 0:
            total *= mystery9(n - 1)
            total *= mystery9(n - 2)
        elif i == n // 2:
            for j in range(1, n):
                total *= j
    return total
```

Exercise 4 Solutions

— — —

```
def mystery8(n):
    if n == 0: return ''
    result, stringified = '', str(n)
    for digit in stringified:
        for _ in range(n):
            result += digit
    result += mystery8(n - 1)
    return result
```

O(n²logn). The double-nested loop is nlogn work. And we run it n times (because of the recursive mystery8 call).

```
def mystery9(n):
    total = 0
    for i in range(1, n):
        total *= 2
        if i % n == 0:
            total *= mystery9(n - 1)
            total *= mystery9(n - 2)
        elif i == n // 2:
            for j in range(1, n):
                total *= j
    return total
```

O(n). The first if-statement never happens, and the second only happens once

$O(2^n)$

Exponential time. Not scalable at all; identifies problems as intractable. Adding to the input size <u>multiplies</u> the runtime.

```python
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
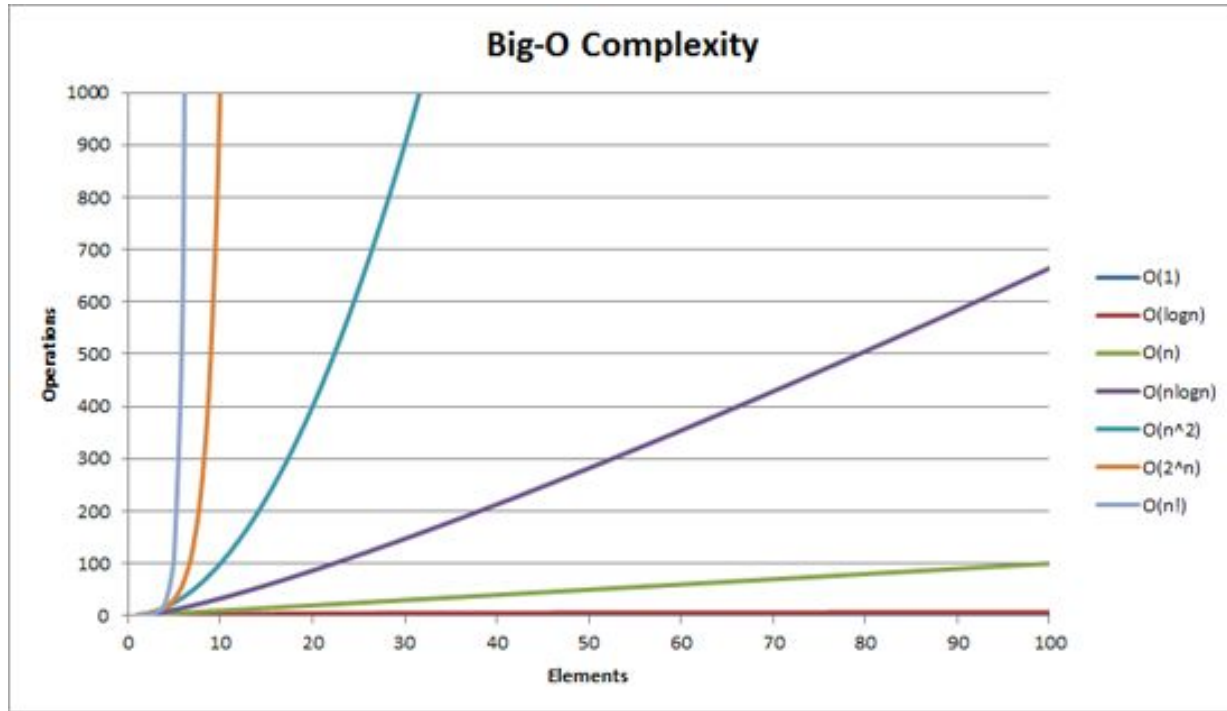
s = seconds
m = minutes
d = days
y = years
c = *centuries*

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

Graphical Summary

— — —

Exercise 5

— — —

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

```
def mystery11(n):
    if n < 1: return n
    def mystery12(n):
        i = 1
        while i < n:
            i *= 2
        return i
    return mystery11(n / 2) + mystery11(n / 2) \
        + mystery12(n - 2)
```

Exercise 5 Solutions

— — —

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

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

```
def mystery11(n):
    if n < 1: return n
    def mystery12(n):
        i = 1
        while i < n:
            i *= 2
        return i
    return mystery11(n / 2) + mystery11(n / 2) \
            + mystery12(n - 2)
```

O(nlogn). We make $O(2^{logn})$ = O(n) recursive calls, and each recursive call does logn work.

A Few More Exercises

Exercise 6

———

The orders of growth should now be functions of m **and** n.

```
def mystery13(m, n):
    if n <= 1:
        return 0
    result = 0
    for i in range(3 ** m):
        result += i // n
    return result + mystery13(m - 5, n // 3)
```

```
def mystery14(m, n):
    result = 0
    for i in range(1, m):
        j = i * i
        while j <= n:
            result, j = result + j, j + 1
    return result
```

Exercise 6 Solutions

— — —

The orders of growth should now be functions of m **and** n.

```
def mystery13(m, n):
    if n <= 1:
        return 0
    result = 0
    for i in range(3 ** m):
        result += i // n
    return result + mystery13(m - 5, n // 3)
```

```
def mystery14(m, n):
    result = 0
    for i in range(1, m):
        j = i * i
        while j <= n:
            result, j = result + j, j + 1
    return result
```

O(3^mlogn). Work done in body is O(3^m), with O($\log_3 n$) calls to the function (bases for logs don't matter because a change of base is just a constant multiplication)

O(m + n√n). The outer loop happens m times no matter what (doing guaranteed constant work), while the inner loop only runs when i <= √n (i.e. it does n work √n times).

Exercise 7

———

Define n to be the length of the input list. How much memory does the following program use as a function of n?

```
def weighted_random_choice(lst):
    temp = []
    for i in range(len(lst)):
        temp.extend([lst[i]] * (i + 1))
    return random.choice(temp)
```

Exercise 7 Solutions

— — —

Define n to be the length of the input list. How much memory does the following program use as a function of n?

```
def weighted_random_choice(lst):
    temp = []
    for i in range(len(lst)):
        temp.extend([lst[i]] * (i + 1))
    return random.choice(temp)
```

$O(n^2)$**.** The length of the temporary list is $1 + 2 + 3 + \dots + n$, which we know (through Gauss's summing trick) is equal to $n(n + 1) / 2 = O(n^2)$.

Exercise 8

— — —

Provide an algorithm that, given a sorted list A of distinct integers, determines whether there is an index i for which A[i] = i. Your algorithm should run in time O(logn), where n is the length of the list. *You don't have to write actual code for this; pseudocode or a general approach would be sufficient.*

Thanks to CS 170 for the question description!
If you want to write code, there's a skeleton to the right.

```
def index_exists(A):
    def helper(lower, upper):
        if _____:
            return _____
        mid_idx = (lower + upper) // 2
        if _____:
            return True
        elif _____:
            return _____
        else:
            return _____
    return _____
```

Exercise 8 Solutions

━ ━ ━

Provide an algorithm that, given a sorted list A of distinct integers, determines whether there is an index i for which A[i] = i. Your algorithm should run in time O(logn), where n is the length of the list. *You don't have to write actual code for this; pseudocode or a general approach would be sufficient.*

Thanks to CS 170 for the question description!
If you want to write code, there's a skeleton to the right.

```python
def index_exists(A):
    def helper(lower, upper):
        if lower >= upper:
            return A[upper] == upper
        mid_idx = (lower + upper) // 2
        if A[mid_idx] == mid_idx:
            return True
        elif A[mid_idx] > mid_idx:
            return helper(lower, mid_idx - 1)
        else:
            return helper(mid_idx + 1, upper)
    return helper(0, len(A) - 1)
```

NOTE I intended for these problems to be very tricky. (At the time of me writing this, I'm not sure whether I succeeded.) That being said, some of these questions are probably slightly above the level of difficulty you'd expect on the final. Even if you didn't get a lot of them, you might not have to worry TOO much.

Past Exam Questions

Summer 2013 MT2 | Q2(a)

— — —

```
def fizzle(n):
    if n <= 0:
        return n
    elif n % 23 == 0:
        return n
    return fizzle(n - 1)
```

What is the order of growth for a call to fizzle(n)?

Summer 2013 MT2 | Q2(a) Solutions

```
def fizzle(n):
    if n <= 0:
        return n
    elif n % 23 == 0: # this line ensures that fizzle will never be called more than 23 times
        return n
    return fizzle(n - 1)
```

What is the order of growth for a call to fizzle(n)?

Answer: O(1).

Summer 2013 MT2 | Q2(b)

— — —

```
def boom(n):
    if n == 0: return 'BOOM!'
    return boom(n - 1)

def explode(n):
    if n == 0: return boom(n)
    i = 0
    while i < n:
        boom(n)
        i += 1
    return boom(n)
```

What is the order of growth for a call to explode(n)?

Summer 2013 MT2 | Q2(b) Solutions

— — —

```
def boom(n):
    if n == 0: return 'BOOM!'
    return boom(n - 1)

def explode(n):
    if n == 0: return boom(n)
    i = 0
    while i < n:
        boom(n) # n work (happening n times because of the loop)
        i += 1
    return boom(n)
```

What is the order of growth for a call to explode(n)? $O(n^2)$.

Summer 2013 MT2 | Q2(c)

———

```
def dreams(n):
    if n <= 0:
        return n
    if n > 0:
        return n + dreams(n // 2)
```

What is the order of growth for a call to dreams(n)?

Summer 2013 MT2 | Q2(c) Solutions

— — —

```
def dreams(n):
    if n <= 0:
        return n
    if n > 0:
        return n + dreams(n // 2) # divide the problem in half every time
```

What is the order of growth for a call to dreams(n)?

Answer: O(logn).

Spring 2014 MT2 | Q6(a)

— — —

Consider the following function (assume that parameter S is a list):

```
def umatches(S):
    result = set()
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.add(item)
    return result
```

Fill in the blank: The function umatches returns the set of all

_____.

Spring 2014 MT2 | Q6(a) Solutions

———

Consider the following function (assume that parameter S is a list):

```
def umatches(S):
    result = set()
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.add(item)
    return result
```

Fill in the blank: The function umatches returns the set of all
values in S that occur an odd number of times.

Spring 2014 MT2 | Q6(b)

———

```
def umatches(S):
    result = set()
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.add(item)
    return result
```

Let's assume that the operations of adding to, removing from, or checking containment in a set each take roughly constant time. Give an asymptotic bound (the tightest you can) on the worst-case time for umatches as a function of N = len(S).

Spring 2014 MT2 | Q6(b) Solutions

— — —

```
def umatches(S):
    result = set()
    for item in S: # this is why it's O(N)
        if item in result:
            result.remove(item)
        else:
            result.add(item)
    return result
```

Let's assume that the operations of adding to, removing from, or checking containment in a set each take roughly constant time. Give an asymptotic bound (the tightest you can) on the worst-case time for umatches as a function of N = len(S).
Answer: O(N).

Spring 2014 MT2 | Q6(c)

— — —

```
def umatches(S):
    result = []
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.append(item)
    return result
```

Suppose that instead of having result be a set, we make it a list (so that it is initialized to [] and we use .append to add an item; **changes shown to the left**). What now is the worst-case time bound? You can assume that .append is a constant-time operation, and .remove and the in operator require time that is $\Theta(L)$ in the worst case, where L is the length of the list operated on. Since we never add an item that is already in the list, each value appears at most once, just as for a Python set.

Spring 2014 MT2 | Q6(c) Solutions

— — —

```
def umatches(S):
    result = []
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.append(item)
    return result
```

Suppose that instead of having result be a set, we make it a list (so that it is initialized to [] and we use .append to add an item; **changes shown to the left**). What now is the worst-case time bound? You can assume that .append is a constant-time operation, and .remove and the in operator require time that is $\Theta(L)$ in the worst case, where L is the length of the list operated on. Since we never add an item that is already in the list, each value appears at most once, just as for a Python set.

Answer: $O(N^2)$. In the worst case, where every item in S is the same, you have to do two $\Theta(L)$ operations (in and .remove) for N / 2 items in S. Since L is really $O(N)$, we have an $O(N^2)$ function overall.

Spring 2014 MT2 | Q6(d)

— — —

```
def umatches(S):
    result = []
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.append(item)
    return result
```

Now suppose that we consider only cases where the number of different values in list S is at most 100, and we again use a list for result. What is the worst-case time now?

Spring 2014 MT2 | Q6(d) Solutions

— — —

```
def umatches(S):
    result = []
    for item in S:
        if item in result:
            result.remove(item)
        else:
            result.append(item)
    return result
```

Now suppose that we consider only cases where the number of different values in list S is at most 100, and we again use a list for result. What is the worst-case time now?
Answer: O(N). L is now upper bounded by 100, so Θ(L) becomes Θ(1).

Summer 2015 MT2 | Q5(d)

———

```
def append(link, value):
    """Mutates LINK by adding VALUE to
    the end of LINK.
    """
    if link.rest is Link.empty:
        link.rest = Link(value)
    else:
        append(link.rest, value)
```

```
def extend(link1, link2):
    """Mutates LINK_1 so that all
    elements of LINK_2 are added to the
    end of LINK_1.
    """
    while link2 is not Link.empty:
        append(link1, link2.first)
        link2 = link2.rest
```

(i) What order of growth describes the runtime of calling append? Give your function in terms of n, where n is the number of elements in the input LINK.

(ii) Assuming the two input linked lists both contain n elements, what order of growth best describes the runtime of calling extend?

Summer 2015 MT2 | Q5(d) Solutions

— — —

```
def append(link, value):
    """Mutates LINK by adding VALUE to
    the end of LINK.
    """
    if link.rest is Link.empty:
        link.rest = Link(value)
    else:
        append(link.rest, value)
```

```
def extend(link1, link2):
    """Mutates LINK_1 so that all
    elements of LINK_2 are added to the
    end of LINK_1.
    """
    while link2 is not Link.empty:
        append(link1, link2.first)
        link2 = link2.rest
```

(i) What order of growth describes the runtime of calling append? Give your function in terms of n, where n is the number of elements in the input LINK. Answer: $O(n)$.

(ii) Assuming the two input linked lists both contain n elements, what order of growth best describes the runtime of calling extend? Answer: $O(n^2)$.

Summer 2012 Final | Q2(a)

———

```
def collide(n):
    lst = []
    for i in range(n):
        lst.append(i)
    if n <= 1:
        return 1
    if n <= 50:
        return collide(n - 1) + collide(n - 2)
    elif n > 50:
        return collide(50) + collide(49)
```

What is the order of growth in n of the runtime of collide, where n is its input?

Summer 2012 Final | Q2(a) Solutions

— — —

```
def collide(n):
    lst = []
    for i in range(n): # O(n) block of code right here
        lst.append(i)
    if n <= 1:
        return 1
    if n <= 50:
        return collide(n - 1) + collide(n - 2)
    elif n > 50: # this covers the case we're interested in (really large n)
        return collide(50) + collide(49)
```

What is the order of growth in n of the runtime of collide, where n is its input?
Answer: O(n). For large n, it performs an O(n) list initialization and then runs collide(50)
+ collide(49). Since 50 and 49 are constants, that part's runtime is irrespective of n.

Summer 2012 Final | Q2(b)

— — —

```
def crash(n):
    if n < 1:
        return n
    return crash(n - 1) * n


def into_me(n):
    lst = []
    for i in range(n):
        lst.append(i)
    sum = 0
    for elem in lst:
        sum = sum + crash(n) + crash(n)
    return sum
```

What is the order of growth in n of the runtime of into_me, where n is its input?

Summer 2012 Final | Q2(b) Solutions

— — —

```
def crash(n): # O(n) function
    if n < 1:
        return n
    return crash(n - 1) * n


def into_me(n):
    lst = []
    for i in range(n): # O(n)
        lst.append(i)
    sum = 0
    for elem in lst: # do n times:
        sum = sum + crash(n) + crash(n)
    return sum
```

What is the order of growth in n of the runtime of into_me, where n is its input?

Answer: $O(n^2)$. We make 2n crash calls per into_me call, and the growth function of crash is $O(n)$.

Spring 2014 Final | Q5(c)

— — —

Give worst-case asymptotic $\Theta(\cdot)$ bounds — **you guys can write them as O($\cdot$) bounds** — for the running time of the following code snippets. As a reminder, it is meaningful to write things with multiple arguments like $\Theta(a + b)$, which you can think of as "$\Theta(N)$ where N = a + b."

```
def a(m, n):
    for i in range(m):
        for j in range(n // 100):
            print("hi")
```

```
def b(m, n):
    for i in range(m // 3):
        print("hi")
    for j in range(n * 5):
        print("bye")
```

```
def d(m, n):
    for i in range(m):
        j = 0
        while j < i: j = j + 100
```

```
def f(m):
    i = 1
    while i < m:
        i = i * 2
    return i
```

Spring 2014 Final | Q5(c) Solutions

— — —

Give worst-case asymptotic $\Theta(\cdot)$ bounds — **you guys can write them as $O(\cdot)$ bounds** — for the running time of the following code snippets. As a reminder, it is meaningful to write things with multiple arguments like $\Theta(a + b)$, which you can think of as "$\Theta(N)$ where $N = a + b$."

```
def a(m, n): # Answer: O(mn).
    for i in range(m):
        for j in range(n // 100):
            print("hi")
```

```
def b(m, n): # Answer: O(m + n).
    for i in range(m // 3):
        print("hi")
    for j in range(n * 5):
        print("bye")
```

```
def d(m, n): # Answer: O(m²).
    for i in range(m): # essentially 1 + … + m work
        j = 0
        while j < i: j = j + 100
```

```
def f(m): # Answer: O(logm).
    i = 1
    while i < m:
        i = i * 2
    return i
```

Thanks, everyone!
Good luck on the final.