Orders of Growth Review

Owen Jow
CS 61A Spring 2017

What is an order of growth?

An order of growth is a **function** depicting how something (often runtime or memory usage) increases, or “grows”, with respect to some input.

We can use orders of growth to assert the efficiency of our code. If runtime and memory usage don’t scale up too quickly, our code is probably efficient!


Looks innocent enough, right?


Two hours later...

THEY’RE EVERYWHERE

Motivating Example

You have been dealt a hand of cards,[1] and you want to sort them by rank. Which algorithm would you rather use to do that? (Both are correct, at least if you subscribe to the infinite monkey theorem.)

- **Bogosort.** See if your cards are sorted; if not, shuffle them into a random order. Repeat until the cards do turn out to be sorted.
- **Insertion sort.** Go through your cards one by one and place each into the appropriate position within all of the already-processed cards.

[1] Such is life.

Motivating Example

You have been dealt a hand of cards, and you want to sort them by rank. Which algorithm would you rather use to do that? (Both are correct, at least if you subscribe to the infinite monkey theorem.)

Hopefully you chose insertion sort! *Intuitively* it's better, right? In more concrete terms, time complexity for insertion sort is $O(n^2)$, as opposed to bogosort's deterministic $O((n + 1)!)$.

To get a sense of the difference in scale, 10^2 is 100. $(10 + 1)!$ is 39,916,800.

Standard Approximations

We group our growth functions into “classes” (*orders*), without worrying too much about the specifics within each class.

Imagine you had the growth functions $4n$, $4n^2$, and $5n^2 + 1$. Asymptotically (meaning as n gets really big), the difference between $4n^2$ and $5n^2 + 1$ is *nothing* compared to the difference between $4n^2$ and $4n$. When grouping our functions into classes, we’ll want to divide them up according to this type of asymptotic disparity. Thus, we adhere to the following simplifications:

1. **Drop lower-order terms (the “+ 1” in “$5n^2 + 1$”).**
2. **Drop multiplicative constants (the “5” in “$5n^2 + 1$”).**

Standard Approximations

If the number of constant-time ops to perform were exactly...

- ...$n^3 + 40000n^{2.1}$, the time complexity would be $O(n^3)$.
- ...$25n^3$, the time complexity would be $O(n^3)$.
- ...$25n^3 + 40000n^{2.1}$, the time complexity would be $O(n^3)$.

These growth functions are all part of the same *order*, $O(n^3)$.
Note that everything here is *in terms of the input size n*.

Hope for the Best, Plan for the Worst

Don't make any assumptions that aren't explicitly stated. In other words, think about what happens for the **worst-case input**. This'll cover all the possibilities!

Say I have an algorithm to sort those cards from before. The *best-case input* is a hand of cards that's already sorted. But if I always assume that,[1] then even bogosort will appear to be O(1)!

If we want a order of growth that's truly representative, we're going to have to consider the most unfortunate cases: spindly binary trees, massive inputs, the like.

[1] If you're looking for the one true sort, check out intelligent design sort.

A Few Other Notes

- “Number of constant-time steps you have to execute” is just another way to describe runtime.
- “Time complexity” means “the order of growth of the **runtime** with respect to some input.”
- It’s really the large inputs we care about! Order of growth doesn’t matter for small inputs, since those can usually be run pretty quickly regardless of the algorithm.
- I use “O” throughout these slides, but I always mean it as Θ.
  (For those interested, I use O because typing Θ stresses me out.)

Question 0

Determine the order of growth that best describes the worst-case execution time (measured by the quantity of constant-time operations) of a call to mystery0 as a function of n.

^ Hilfinger wording. Example answers: O(1), O(n), O(n^2)...

```
def mystery0(n):
     total = 0
     for i in range(n):
          total *= i
     for i in range(n // 2):
          total += i
     return total
```

Question 0 Solutions

Determine the order of growth that best describes the worst-case execution time (measured by the quantity of constant-time operations) of a call to mystery0 as a function of n.

^ Hilfinger wording. Example answers: O(1), O(n), O(n^2)...

```
def mystery0(n):
    total = 0
    for i in range(n):
        total *= i
    for i in range(n // 2):
        total += i
    return total
```

O(n). We consider all of the lines with “total” in them to be constant time, and these happen 1 + n + n/2 times. Thus, the “overly detailed” growth function is 3/2 * n + 1 and the associated order is O(n).

In other words, we do O(1) work plus O(n) work plus O(n) work, which is just O(n) work after dropping constant multipliers and lower-order terms.

How to identify the time complexity of a function

Go through the function line by line, determining roughly how much time each block of code takes as a function of the input. Then sum all of your estimations together and drop constant multipliers / lower-order terms. That's pretty much it.

If there's recursion, figure out how much work there is to be done in each call and how many calls there'll be. Then multiply those values together.

(When calculating the order of growth, we're always just summing up the amount of work done overall. Think 1 + 1 + 1 + 1, where each 1 represents a constant-time step.)

The main orders of growth to know

Descriptions will assume we're interested in time complexity

- **O(1)**; constant time. Runtime is not affected by the input size.
  - "Theoretical runtime is upper-bounded by a constant"

```
def const(n):
    for _ in range(500):
        print('spam i am')
```

- **O(logn)**; logarithmic time. *Multiplying* the input size by some constant will only *add* some constant to the runtime.
  - "Make multiplicative progress / divide the problem size in half upon every step"

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

The main orders of growth to know

Descriptions will assume we're interested in time complexity

- **O(n)**; linear time. *Adding* a constant to the input size will *add* a constant to the runtime.
  - "Sequential scan through a list"

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

- **O(n^2)**; quadratic time.
  - "Double for-loops" (but not always; watch out for tricks!)

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

The main orders of growth to know

Descriptions will assume we're interested in time complexity

- **$O(c^n)$**; exponential time. *Adding* to the input size will *multiply* the runtime.
  - Denotes problems as "intractable"
  - Runtime increases very quickly relative to the input size
  - Often describes tree recursion

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

The main orders of growth to know

Descriptions will assume we're interested in time complexity

There's also stuff like **O(sqrt(n))** and **O(nlogn)**.

For **O(sqrt(n))** to happen, there'll usually have to be a square or a square root in the algorithm somewhere.

For **O(nlogn)** to happen, you'll generally just be doing n work logn times (or logn work n times).

```
def nlogn(n):
    for _ in range(n):
        _ = log(n) # the log-time function defined three slides ago
```

A General Timing Comparison

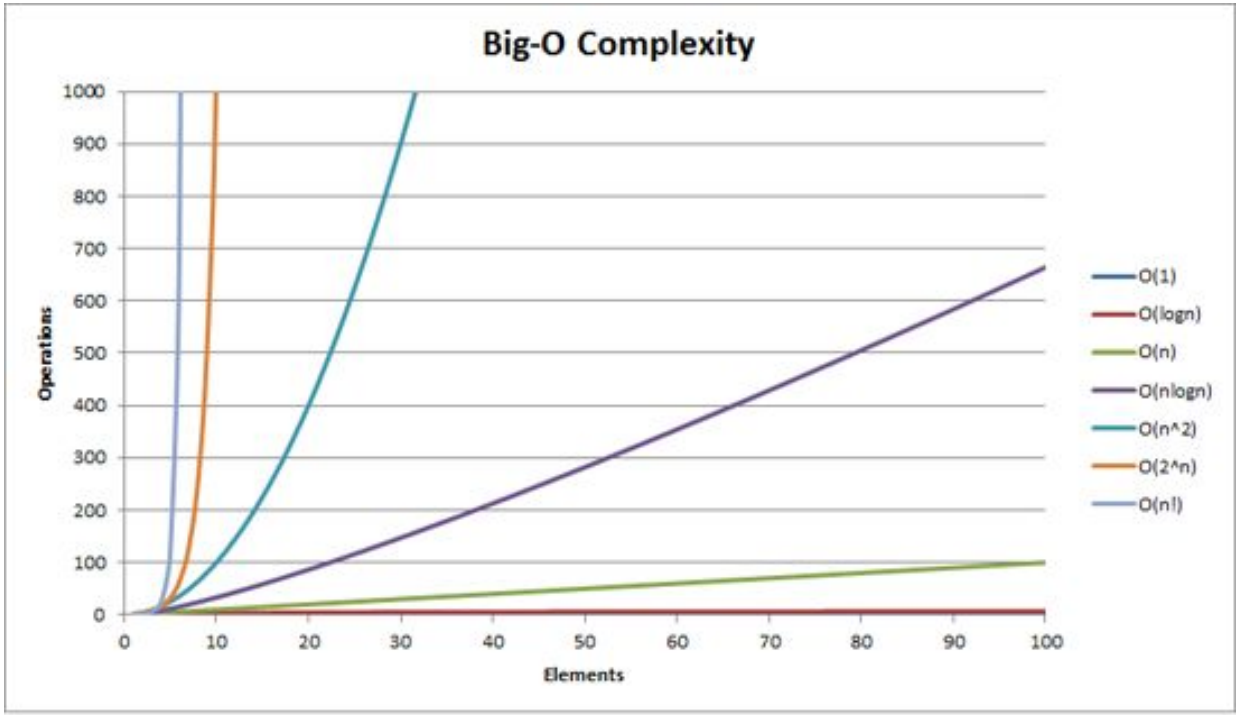
	n = 10	n = 50	n = 100	n = 1000
logn	0.0003s	0.0006s	0.0007s	0.0010s
sqrt(n)	0.0003s	0.0007s	0.0010s	0.0032s
n	0.0010s	0.0050s	0.0100s	0.1000s
nlogn	0.0033s	0.0282s	0.0664s	0.9966s
n^2	0.0100s	0.2500s	1.0000s	100.00s
n^6	1.6667m	18.102d	3.1710y	3171.0*c*
2^n	0.1024s	35.702*c*	$4x10^{16}$*c*	$1x10^{166}$*c*
n!	362.88s	$1x10^{51}$*c*	$3x10^{144}$*c*	$1x10^{2554}$*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



Questions

For each code segment, determine the order of growth of the runtime as a function of n.

Question 1

```
def mystery1(n):
    if n <= sqrt(abs(n)):
        return n
    return n + mystery1(n // 3)
```

Question 1 Solutions

```
def mystery1(n):
     if n <= sqrt(abs(n)):
          return n
     return n + mystery1(n // 3)
```

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

Question 2

```
def mystery2(n):
     while n > 1:
          x = n
          while x > 1:
               print(n, x)
               x = x // 2
          n -= 1
```

Question 2 Solutions

```
def mystery2(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.

Question 2 Follow-Up

What if we switch the updates from the inner and outer loop? (Updated version below.)

```
def mystery2f(n):
     while n > 1:
          x = n
          while x > 1:
               print(n, x)
               x -= 1
          n //= 2
```

Question 2 Follow-Up Solutions

What if we switch the updates from the inner and outer loop? (Updated version below.)

```
def mystery2f(n):
     while n > 1:
          x = n
          while x > 1:
               print(n, x)
               x -= 1
          n //= 2
```

O(n). This is O(n) (not O(nlogn)) because the total amount of work done is approximately n + n / 2 + n / 4 + ... + 1. Before this change, the total amount of work would be about log(n) + log(n - 1) + log(n - 2) + ... + 1, which is of course different.

Question 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
```

Question 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.

Question 4

```
def mystery4(n):
    total = 0
    for i in range(1, n):
        total *= 2
        if i % n == 0:
            total *= mystery4(n - 1)
            total *= mystery4(n - 2)
        elif i == n // 2:
            for j in range(1, n):
                total *= j
    return total
```

Question 4 Solutions

```
def mystery4(n):
    total = 0
    for i in range(1, n):
        total *= 2
        if i % n == 0:
            total *= mystery4(n - 1)
            total *= mystery4(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.

Question 5

———

```
def mystery5(n):
     n, result = str(n), ‘’
     num_digits = len(n)
     for i in range(num_digits):
          result += n[num_digits - i - 1]
     return result
```

Question 5 Solutions

```
def mystery5(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).
str(n) is O(logn); len(n) is O(1).

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.

Question 6

Here, the order of growth should be a function of m **and** n.

```
def mystery6(m, n):
     result = 0
     for i in range(1, m):
          j = i * i
          while j <= n:
               result, j = result + j, j + 1
     return result
```

Question 6 Solutions

Here, the order of growth should be a function of m **and** n.

```
def mystery6(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(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).

Useful Formulas

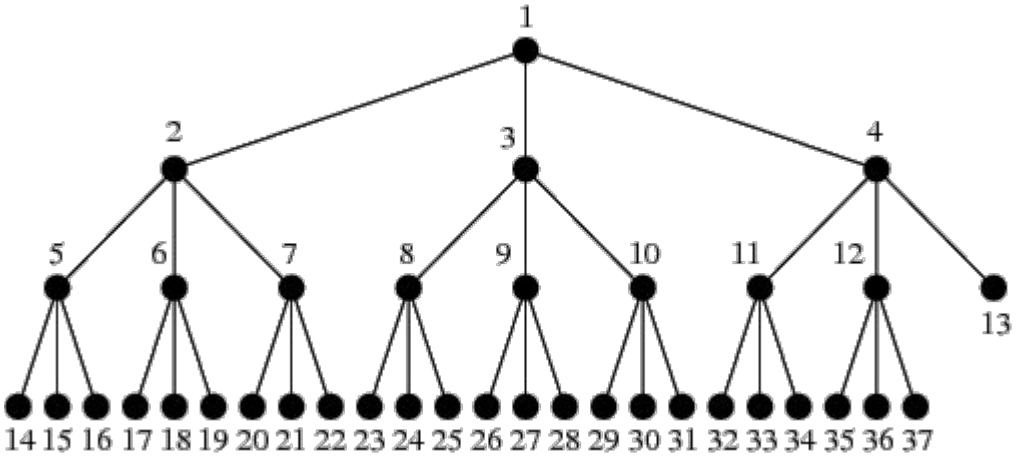
- $1 + 2 + 3 + \ldots + n = n(n + 1) / 2 = O(n^2)$
- The total number of nodes in a **full** tree with branching factor B and height H is $(B^{H + 1} - 1) / (B - 1)$.
  - The branching factor is the maximum number of children that any individual node can have (for a binary tree, B = 2).
  - Remember that a tree with only one level has height 0.
  - This means that for a full tree, the number of nodes is **exponential** in its height! [i.e. for height H, the number of nodes is $O(B^H)$]

One reason this is useful: you'll often want to think about recursive functions' orders of growth by drawing their associated tree of calls. This formula might allow you to determine the worst-case number of calls, and then you just have to multiply by the amount of work in each one. Notice that the **height** of the call tree is usually just the maximal length of a path from the initial call (the root) to the base case (a leaf)!

Useful Formulas

- The branching factor of this tree is 3 and the height is also 3, so the maximum (worst-case?) number of nodes in this tree is

  $(3^{3+1} - 1) / (3 - 1)$
  $= 80 / 2$
  $= 40$



You can see from the numbering that this checks out!

Question 7

———

```
def mystery7(n):
     if n < 1:
          return n
     def helper(n):
          i = 1
          while i < n:
               i *= 2
          return i
     return mystery7(n / 2) + mystery7(n / 2) + helper(n - 2)
```

Question 7 Solutions

```
def mystery7(n):
    if n < 1:
        return n
    def helper(n):
        i = 1
        while i < n:
            i *= 2
        return i
    return mystery7(n / 2) + mystery7(n / 2) + helper(n - 2)
```

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

WAIT WE MAKE $O(2^{logn})$ RECURSIVE CALLS? HOW DO WE KNOW THAT? MAYBE WE SHOULD CONSULT THE THIRD-TO-LAST SLIDE!!

Question 8

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)
```

Question 8 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 + … + n, which we know (through the summing formula from the last slide but four) is equal to $n(n + 1) / 2 = O(n^2)$.

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; **updated version 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; **updated version 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 order of growth of crash is O(n).

Spring 2014 Final | Q5(c)

Give worst-case asymptotic $\Theta(\cdot)$ 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 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(m^2).
    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!
Hope you have a good time during the final. :)