CS 61A Discussion 7

March 10, 2016

Today’s Agenda

+ Quiz
+ Orders of Growth
+ Object-Oriented Trees

Announcements

+ If you think you’re going to have a MT2 conflict (**March 30 @ 7-9pm**), you should file an alternate exam request sooner than later! The deadline for this is next Friday.

The `list` constructor

You can’t call `list` on an object or a number. Its argument must be an iterable.

WWWP?

```
>>> list(4) # error!
>>> list([4])
>>> list(Link(2)) # error!
>>> list([Link(2)])
>>> [Link(2)] # yeah, just use bracket notation. This works
>>> list()
```

Orders of Growth

Orders of Growth

Growth: how much of a resource (TIME or SPACE) our program consumes as the input size gets bigger and bigger

Order: how we quantify that growth. Also known as *time/space complexity*.

- Main orders to know: O(1), O(log(n)), O(n), O(n^2), O(n^3), and O(x^n)
- You should also be able to recognize O($\sqrt{n}$) and O(nlog(n))
- Drop constants and lower-order terms! Growth-wise, they’re not important
- Note: big-O is only one of several notations. In this class, we’ll almost always use it as big-Θ

Order of growth is an important thing to consider when designing algorithms.

Remember: complexity is relative!

It’s about how the program scales, not how fast the program is at any given point.

For example, if it takes ten seconds to sort 1000 items, how long would it take to sort 10000000 items (in *comparison* to sorting 1000 items)?

Basically, you have to think about the growth function as a whole.

The difference between big-O and big-Θ

+ Big-Θ refers to both an lower bound and an upper bound. It forces you to choose an order of growth that is as accurate as possible.

+ Big-O **only** refers to the upper bound. Technically, pretty much all of your functions will be $O(n^n)$ (since that's going to be an upper bound for everything), but this isn't very useful to know. So we want to make that upper bound as close to the theta bound as possible.

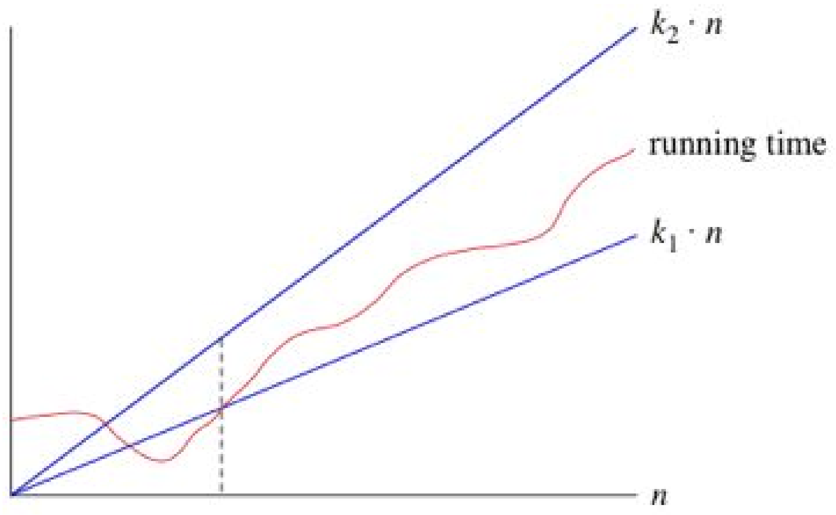
+ In the real world, a lot of people use Big-O as Big-Θ, which is basically what we're doing here.

Growth, *continued*

In this class, we're usually concerned with **time** (instead of space). However, you should be able to think about both in a similar way.

Even though big-O technically refers to an upper bound, we want you to find the "tightest" bound ➔

Ex. If the growth function (time plotted against input size) is ALWAYS sandwiched between 2n and 4n when n is large, then the program is Θ(n) ≂ O(n).



How do you figure out time/space complexity?

Think about what happens when you keep adding one unit to the input size. **How many more resources are you going to use each time? (ex. one more, twice as many as you had before…)**

Then think about what happens if you keep doubling the input size. **How many more resources are you going to use?**

You can synthesize the answers to the above questions in order to arrive at a finalized order of growth.

Translations (using time complexity as an example)

+ **O(1)**: changing the input size doesn’t change computation time.
The algorithm will perform about the same on an input size of 10000000000 as it would on an input size of 10.

In the following examples, n represents the input. Bigger n = bigger input size.

```
def constant(n):
    return 21
```

Translations, pt. 2

+ **O(log(n))**: A multiplicative increase in input size leads to an additive increase in runtime.

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

Translations, pt. 3

+ **O(n)**: adding to the input size also adds to the amount of computation time. Add a constant to the input size, add a constant to the runtime.

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

Translations, pt. 4

+ **O(n^2)**: multiplying the input size by some constant factor will also multiply the amount of computation time by some constant factor.

Classic example: nested for loops.

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

Translations, pt. 5

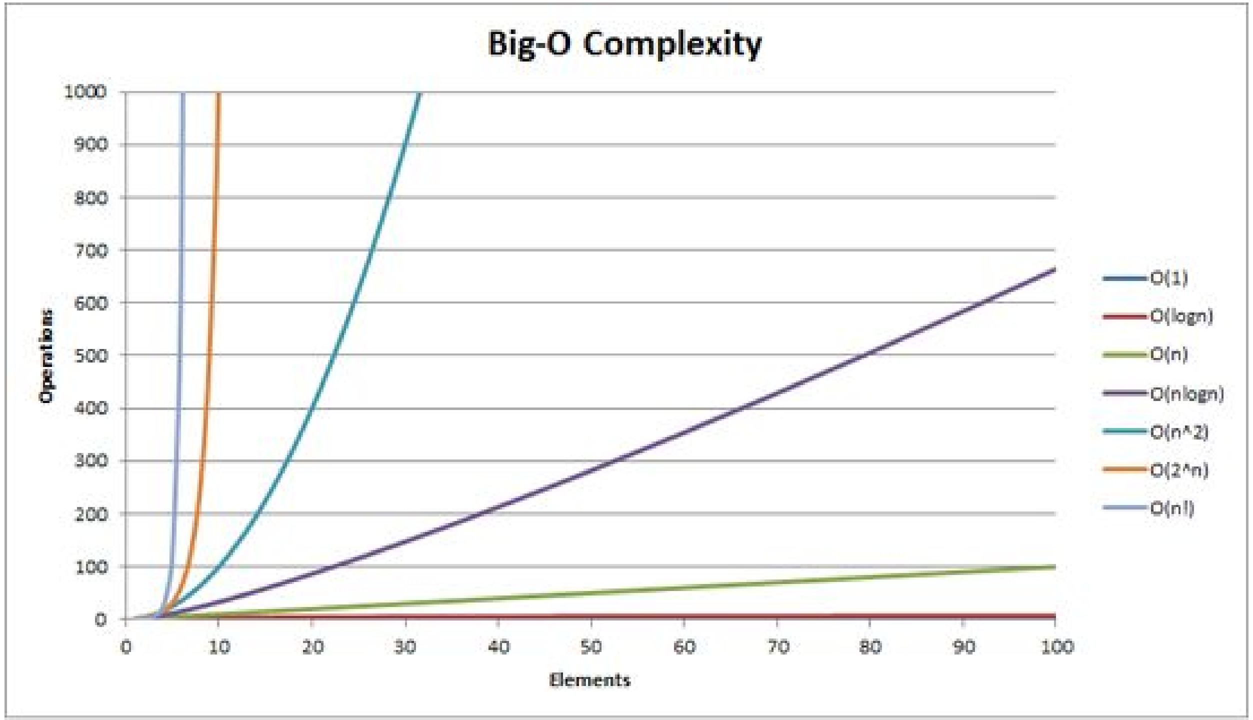
+ **O(2^n)**: Adding to the input size multiplies the runtime.

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

All of the last five slides have been describing the growth functions themselves

Just think about how `f(n) = 1` or `f(n) = n` grows as `n` increases. That’s really all this is – runtime as a function of `n`.

Complexity Visualization



On Patterns and Formulas

>>> *“Nested for loops? Must be O(n^2)...”*

Do NOT resort to a formulaic approach (at least, not without considering the program yourself as well). This may “generally” work, but in many cases there may be something going on that will make nested for loops into constant, linear, or even nlog(n) time.

In the end, you really have to look at the program line-by-line and think about what it’s doing.

Order of Growth: A Basic Example

```
def mystery(n):
    total = 0
    for i in range(n):
        total += constant(i)
    return total
```

What is the order of growth of the above function?

Order of Growth: A Basic Example

```
def mystery(n):
    total = 0
    for i in range(n): # loop n times
        total += constant(i) # each iteration, do constant work
    return total
```

What is the order of growth of the above function? O(n)

Order of Growth: A Harder Example

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

Order of Growth: O(n) [linear work + linear work!]

```
def mystery(n):
    total = 0
    for i in range(1, n):
        total *= 2
        if i % n == 0: # this will never happen
            total *= mystery(n - 1) * mystery(n - 2)
        elif i == n // 2: # this will only ever happen ONCE
            for j in range(1, n):
                total *= j
    return total
```

Object-Oriented Trees

Object-Oriented Trees

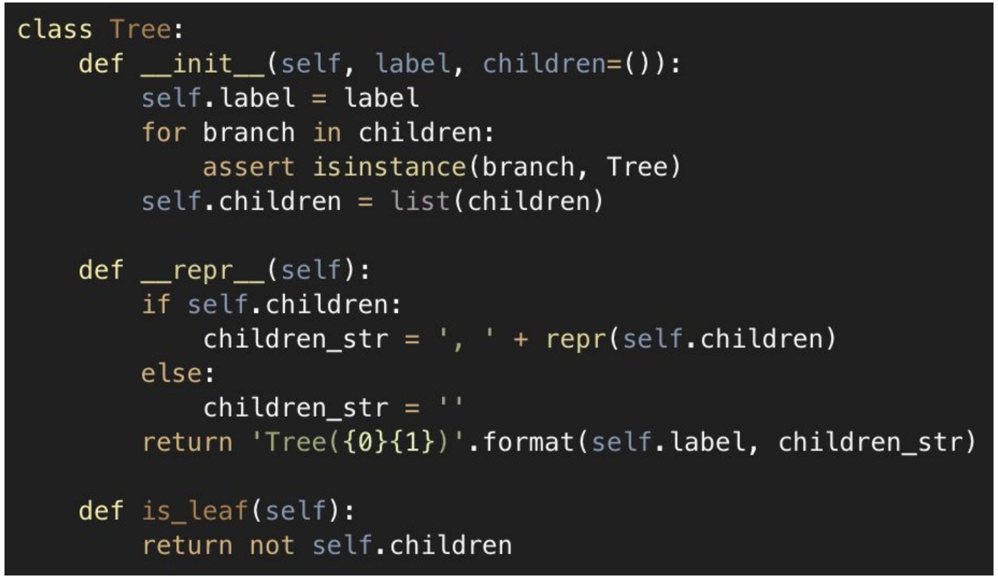
The *idea* of trees has not changed. They're still the same. The only difference is that we're representing them as objects now.

OO tree attributes

+ **`tree.label`** gives you the label at the top of the tree

+ **`tree.children`** gives you a list of children (which, again, are trees)

+ **`tree.is_leaf()`** tells you whether or not the tree is a leaf



```python
class Tree:
    def __init__(self, label, children=()):
        self.label = label
        for branch in children:
            assert isinstance(branch, Tree)
        self.children = list(children)

    def __repr__(self):
        if self.children:
            children_str = ', ' + repr(self.children)
        else:
            children_str = ''
        return 'Tree({0}{1})'.format(self.label, children_str)

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