ORDERS OF GROWTH, DATA ABSTRACTION 4

COMPUTER SCIENCE 61A

June 28, 2012

1 Orders of Growth

When we talk about the efficiency of a procedure (at least for now), we are often interested in how much more expensive it is to run the procedure with a larger input. That is, as the size of the input grows, how do the speed of the procedure and the space its process occupies grow?

For expressing all of these, we use what is called the Big-Theta notation. For example, if we say the running time of a procedure `foo` is in $\Theta(n^2)$, we mean that the running time of the process, `R(n)`, will grow proportionally to the square of the size of the input `n`. More generally, we can say that `foo` is in some $\Theta(f(n))$ if there exist some constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \tag{1}$$

for $n > N$, where N is sufficiently large.

This is a mathematical definition of big-Theta notation. To prove that `foo` is in $\Theta(f(n))$, we only need to find constants k_1 and k_2 where the above holds.

There is also another way to express orders of growth: big-Oh notation. This denotes the worst case complexity of a procedure, whereas big-Theta notation gives a rough approximation of the actual complexity. Still, big-Oh notation can be useful when it is not possible to find a big-Theta. The mathematical definition of big-Oh is, for some values k_1 and n,

$$R(n) \leq k_1 \times f(n) \tag{2}$$

for $n > N$, where N is sufficiently large.

For example, $O(n^2)$ states that a function's worst case run time would be in quadratic time. This does not mean the function will never be slower than quadratic time; in fact, it might very well run in linear or even constant time!

Fortunately, in CS61A, we're not that concerned with rigorous mathematical proofs (you'll get the painful details in CS61B!). What we want you to develop in CS61A is the intuition to guess the orders of growth for certain procedures.

1.1 Kinds of Growth

Here are some common orders of growth, ranked from best to worst:

- $\Theta(1)$ — constant time takes the same amount of time regardless of input size
- $\Theta(\log n)$ — logarithmic time
- $\Theta(n)$ — linear time
- $\Theta(n^3)$, $\Theta(n^3)$, etc. — polynomial time
- $\Theta(2^n)$ — exponential time ("intractable"; these are really, really horrible)

1.2 Orders of Growth in Time

"Time," for us, basically refers to the number of recursive calls or the number of times the suite of a `while` loop executes. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

- If the function contains only primitive procedures like $+$ or $*$, then it is constant time – $\Theta(1)$.
- If the function is recursive, you need to:
  - Count the number of recursive calls that will be made, given input n.
  - Count how much time it takes to process the input per recursive call.

  The answer is usually the product of the above two. For example, given a fruit basket with 10 apples, how long does it take for me to process the whole basket? Well, I'll recursively call my `eat` procedure, which eats one apple at a time (so I'll call the procedure 10 times). Each time I eat an apple, it takes me 30 minutes. So the total amount of time is just $30 \times 10 = 300$ minutes!
- If the function contains calls of helper functions that are not constant-time, then you need to take orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be included.

- When we talk about orders of growth, we don't really care about constant factors. So if you get something like $\Theta(1000000n)$, this is really $\Theta(n)$. We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3 + n^2 + 4n + 399)$, we can take it to be $\Theta(n^3)$.

1.3 Questions

What is the order of growth in time for the following functions?

1.
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

Solution: $\Theta(n^2)$

2.
```
def fibonacci(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Solution: $\Theta(\Phi^n)$, where Φ is the golden ratio.

3.
```
def fib_iter(n):
    prev, cur, i = 0, 1, 1
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return curr
```

Solution: $\Theta(n)$

4.
```
def mod_7(n):
    if n % 8 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Solution: $\Theta(1)$

5. Given:

```
def bar(n):
    if n % 2 == 1:
        return n + 1
    return n

def foo(n):
    if n < 1:
        return 2
    if n % 2 == 0:
        return foo(n - 1) + foo(n - 2)
    else:
        return 1 + foo(n - 2)
```

What is the order of growth of `foo(bar(n))`?

Solution: $\Theta(n^2)$

6.
```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

Solution: $\Theta(\log(n))$

2 Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects — for example, car objects, chair objects, people objects, etc. That way, programmers don't have to worry about *how* code is implemented — they just have to know *what* it does.

This is especially important when programming with other people: with data abstraction, your group members won't have to read through every line of your code to understand how it works before they use it — they can just assume that it does work.

Data abstraction mimics how we think about the world. For example, when you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

To facilitate data abstraction, you will need to create two types of functions: constructors and selectors. Constructors are functions that build the abstract data type. Selectors are functions that retrieve information from the data type.

For example, say we have an abstract data type called `city`. This `city` object will hold the `city`'s name, and its latitude and longitude. To create a `city` object, you'd use a function like

```
make_city(name, lat, lon)
```

To extract the information of a `city` object, you would use functions like

```
get_name(city)
get_lat(city)
get_lon(city)
```

You would use similarly-named functions to extract the latitude and the longitudes.

The following code will compute the distance between two city objects:

```
def distance(city1, city2):
    from math import sqrt

    lat_1, lon_1 = get_lat(city_1), get_lon(city_1)
    lat_2, lon_2 = get_lat(city_2), get_lon(city_2)

    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Notice that we don't need to know how these functions were implemented. We are assuming that someone else (e.g. Jom Magrotker) has defined them for us.

It's okay if the end user doesn't know how functions were implemented. However, the functions still have to be defined by someone. How did Jom Magrotker define those constructors and selectors? You'll do just that in the following questions.

2.1 Questions

1. First, implement `closer_city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is closer.

   You may only use selectors and constructors (introduced above) for this question. You may also use the `distance` function defined above.

   ```
   def closer_city(lat, lon, city1, city2):
   ```

   Solution:

   ```
   new_city = make_city('arb', lat, lon)
   dist1 = distance(city1, new_city)
   dist2 = distance(city2, new_city)
   if dist1 < dist2:
       return get_name(city1)
   return get_name(city2)
   ```

2. Now, implement `make_city`, which takes in three arguments: `name`, which is a string; `lat`, the latitude of the `city`; and `lon`, the longitude of the `city`. `make_city` should then return a tuple that represents the `city` object.

   ```
   def make_city(name, lat, lon):
   ```

   Solution:

   ```
   return (name, lat, lon)
   ```

3. Implement the following selector functions:

   ```
   def get_name(city):
   ```

   Solution:

   ```
   return city[0]
   ```

   ```
   def get_lat(city):
   ```

Solution:

```
    return city[1]
```

```
def get_lon(city):
```

Solution:

```
    return city[2]
```

3 Extra: Immutability of Tuples

Tuples have a special property called *immutability*. Immutability means without change, or unchanging. What that means is that tuples *never change*. We've already seen some immutable values; numbers, Booleans, and strings mainly. What immutability means for all of these data types is that there is only one version of the number "1", and only one version of the number"1" will ever exist. In the same way, there is only one version of the value "True", and there will only be one version of the string "hi". Again, in the same vein of thought, there will only be one version of the tuple (1, 2).

What do I mean by one version?

Let me explain with some code!

```
x = 3
x = 2 * x
```

What's happening here is subtle. First, `x` is assigned the value of 3. Then, `x` is assigned to the *value x evaluates to multiplied by 2*. The value `3` is never changed, because `x` simply has the value `3`; the value `3` is the value `3` regardless of what has that value. The value `3` is never changed during the process of evaluating those lines; `x` changes values.

What consequences does this immutability have for tuples? There is one main one, as illustrated by the following code snippet:

```
tup = (2, 3, 4)
print(tup[0])                 # Prints 2
tup[0] = 1                    # Python will error on this line
tup = (1, tup[1], tup[2])     # tup is now (1, 3, 4)
```

As the previous code snippet illustrates, you can look at the individual values in a tuple with no problem, but you *cannot* change them. You can't change the value `(2, 3, 4)`.

It has that value, and there's nothing you can do about it. You can look at that value and make another tuple based on the contents of that tuple and change what a variable containing a tuple will evaluate to, as in the fourth line, however.

Why bother with immutability in Python? There a couple reasons the designers of Python made this decision.

One is performance. It turns out that for certain operations on data that you want to do, especially something called *hashing*, which you will learn about in CS61B, immutable data types are much easier to handle. This leads to easier code for them to write, and it also leads to a performance benefit for Python in general.

Another has to do with the style of what Python wants you to use mutable nd immutable data structures for. The idea with differentiating between tuples, which are immutable, and lists, which are mutable (we haven't learned about them yet), is to differentiate what type of data you would store in each.

Here's a code snippet to help us out!

```
>>> import time
>>> time.localtime()
<<<<<<< HEAD
(2012, 6, 28, 0, 35, 58, 3, 180, 1) #formatted as (year, month, day,
 hour, minute, second, weekday, yearday, is_daylist_savings_time)
=======
(2012, 6, 28, 0, 35, 58, 3, 180, 1)
<<<<<<< HEAD
#formatted as (year, month, day, hour, minute, second,
               weekday, yearday, is_daylist_savings_time)
>>>>>>> 0c44511477011b98fc4ddbb46e29f34e5213f11b
=======
# Formatted as (year, month, day, hour, minute, second,
                weekday, yearday, is_daylist_savings_time)
>>>>>>> 8fc7680bb6a9c204b95f44489c3ac639e0aa0318
```

We can see that the time is represented as a tuple. This is because the time has a specific format that we don't want to change. We don't want to be able to accidently change the first value to `3`, for example, because that would totally mess up any program that tries to read the time.