ORDERS OF GROWTH AND TREES 7

COMPUTER SCIENCE 61A

March 9, 2017

1 Orders of Growth

When we talk about the efficiency of a function, we are often interested in the following: if the size of the input grows, how does the runtime of the function change? And what do we mean by "runtime"? Let's look at the following examples first:

```
def square(n):
    return n * n

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

- `square(1)` requires one primitive operation: `*` (multiplication). `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes one operation.

input	function call	return value	number of operations
1	square(1)	1*1	1
2	square(2)	2*2	1
⋮	⋮	⋮	⋮
100	square(100)	100*100	1
⋮	⋮	⋮	⋮
n	square(n)	n*n	1

- `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases linearly proportional to the input.

input	function call	return value	number of operations
1	factorial(1)	1*1	1
2	factorial(2)	2*1*1	2
⋮	⋮	⋮	⋮
100	factorial(100)	100*99*…*1*1	100
⋮	⋮	⋮	⋮
n	factorial(n)	n*(n-1)*…*1*1	n

Here are some general guidelines for orders of growth:

- If the function is recursive or iterative, you can subdivide the problem as seen above:
  - Count the number of recursive calls/iterations that will be made, given input n.
  - Count how much time it takes to process the input per recursive call/iteration.

  The answer is usually the product of the above two, but pay attention to control flow!
- If the function calls helper functions that are not constant-time, you need to take the orders of growth of the helper functions into consideration.
- We can ignore constant factors. For example, $\Theta(1000000n) = \Theta(n)$.
- We can also ignore lower-order terms. For example, $\Theta(n^3 + n^2 + 4n + 399) = \Theta(n^3)$. This is because the n^3 term dominates as n gets larger.

1.1 Kinds of Growth

Here are some common orders of growth, ranked from no growth to fastest growth:

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

1.2 Questions

What is the order of growth for the following functions?

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

```
def fib_recursive(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)
```

2.
```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

5.
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

6.
```
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))`?

2 Object-Oriented Trees

Trees are also data abstractions that can have multiple implementations. Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead. With this implementation, we can easily specify specialized tree types, such as binary trees , using inheritance.

```
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
```

Notice that with this implementation we can mutate the label of a tree by reassigning `tree.label`. In the previous implementation using lists, this was not possible, because the abstraction barrier prevented us from seeing how the tree was implemented.

2.1 Questions

1. Define a function `make_even` which takes in a tree `t` whose labels are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
```

2. Create and return a new tree with the same shape as `t`, but where all elements are `n`.

```
def fill_tree(t, n):
    """
    >>> t0 = Tree(0, [Tree(1), Tree(2)])
    >>> t1 = fill_tree(t0, 5)
    >>> t1
    Tree(5, [Tree(5), Tree(5)])
    """
```

3. Write a function that combines the labels of two trees `t1` and `t2` together with the `combiner` function. Assume that `t1` and `t2` have identical structure. This function should return a new tree.

```
def combine_tree(t1, t2, combiner):
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = combine_tree(a, b, mul)
    >>> combined.label
    4
    >>> combined.branches[0].label
    10
    """
```

4. Assuming that every label in `t` is a number, let's define `average(t)`, which returns the average of all the labels in `t`.

```
def average(t):
    """
    Returns the average value of all the labels in t.
    >>> t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
    >>> average(t0)
    1.5
    >>> t1 = Tree(8, [t0, Tree(4)])
    >>> average(t1)
    3.0
    """
```

2.2 Extra Questions

1. Implement the `alt_tree_map` function that, given a function and a `Tree`, applies the function to all of the data at every other level of the tree, starting at the root.

```
def alt_tree_map(t, map_fn):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
    >>> negate = lambda x: -x
    >>> alt_tree_map(t, negate)
    Tree(-1, [Tree(2, [Tree(-3)]), Tree(4)])
    """
```

2. How would we modify the Tree class so that each node remembers its parent? Write out the new Tree class with the necessary modifications.

   Now write a method `first_to_last` for the `Tree` class that swaps a tree's own first child with the last child of `other` (another instance of the `Tree` class). Don't forget to make sure the parents are still correct after the swap!

```
def first_to_last(self, other):
```