ORDERS OF GROWTH AND TREES 6

COMPUTER SCIENCE 61A

October 15, 2015

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	square(100)	100*100	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
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
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	factorial(100)	100*99*...*1*1	100
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	factorial(n)	n*(n-1)*...*1*1	n

Big-O notation is a way to denote an upper bound on the complexity of a function. For example, $O(n^2)$ states that a function's run time will be **no larger than the quadratic** of the input.

- If a function requires $n^3 + 3n^2 + 5n + 10$ operations with a given input n, then the runtime of this function is $O(n^3)$. As n gets larger, the lower order terms (10, $5n$, and $3n^2$) all become insignificant compared to n^3.
- If a function requires $5n$ operations with a given input n, then the runtime of this function is $O(n)$. The constant 5 only influences the runtime by a constant amount. In other words, the function still runs in linear time. Therefore, it doesn't matter that we drop the constant.

1.1 Kinds of Growth

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

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

When using big-O notation, we always want to find the "tightest bound". Recall that `factorial(n)` requires `n` multiplications. It's technically correct to say that `factorial(n)` is in $O(n^2)$, since $n^2 \geq n$ for all positive values of `n`, but it's not very informative. Instead, we want to find the smallest big-O that `factorial(n)` belongs to. Since our implementation of `factorial(n)` must use at least `n` multiplications in all cases, we say its tightest bound is $O(n)$.

1.2 Questions

1. What is the order of growth in time for the following functions? Use big-O notation.

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

Solution: $O(n^2)$, we will call factorial n times with arguments $n, n-1, n-2, ..., 0$. The sum from 0 to n is approximately n^2.

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

Solution: $O(\Phi^n)$, where Φ is the golden ratio. As long as you understand the runtime is exponential in n, we would accept your answer.

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

Solution: $O(n)$, since the while loop executes n times with each iteration taking a constant $O(1)$ time.

4.
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Solution: $O(1)$, since at worst it will require 6 recursive calls to reach the base case. So this is $O(6)$, which can be reduced to $O(1)$.

5.
```
def bonk(n):
```

```
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

Solution: $O(\log(n))$, because our while loop iterates at most $\log(n)$ times, due to n being halved in every iteration.

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))`?

Solution: $O(n^2)$

1.3 Extra Questions

1. Previously, we looked at the `is_prime` function. Here's the code for it:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

What is the order of growth of `is_prime`?

Solution: $O(n)$

How can we change `is_prime` so that it runs in $O(\sqrt{n})$?

```
def is_prime(n):
```

Solution: If `n` is not prime, it can be factored into at least two factors. If both of the factors are greater than the $\sqrt{n}$, then their product is greater than `n`. Therefore, one of the factors must be less than or equal to $\sqrt{n}$.

```
    if n == 1:
        return False
    k = 2
    while k * k <= n:
        if n % k == 0:
            return False
        k += 1
    return True
```

2 Object-Oriented Trees

Previously, we have seen trees defined as an abstract data type using lists. Let's look at another implementation using objects. With this implementation, we will be able to easily specify specialized tree types such as binary trees through inheritance.

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

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

Notice that with this implementation we are able to mutate the entry of a tree by reassigning `tree.entry`. This was not possible when using ADT's 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 entries 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
```

```
    Tree(2, [Tree(2, [Tree(4)]), Tree(4), Tree(6)])
    """
```

Solution:

```
    if t.entry % 2 != 0:
        t.entry += 1
    for branch in t.branches:
        make_even(branch)
```

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)])
    """
```

Solution:

```
    filled_branches = [fill_tree(b, n) for b in t.branches]
    return Tree(n, filled_branches)
```

3. Write a function that combines the entries of two identically-shaped trees `t1` and `t2` together by using the `combiner` function. 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
    Tree(4, [Tree(10, [Tree(18)])])
    """
```

Solution:

```
    combined = [combine_tree(b1, b2, combiner) for b1, b2
                in zip(t1.branches, t2.branches)]
    return Tree(combiner(t1.entry, t2.entry), combined)
```

4. Assuming that every entry in `t` is a number, let's define `average(t)`, which returns the average of all the entries in `t`. Hint: use a helper function. What two things do you need to know in order to compute an average? This helper function should help you compute these two things, so that you can then compute the average and return it from `average(t)`.

```
def average(t):
    """
    Returns the average value of all the entries 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
    """
```

Solution: It would help to write a helper function, because we cannot just add recursive calls of average together directly.

```
    def sum_helper(t):
        sum_entries, count = t.entry, 1
        for b in t.branches:
            branch_sum, branch_count = sum_helper(b)
            sum_entries += branch_sum
            count += branch_count
        return sum_entries, count
    sum_entries, count = sum_helper(t)
    return sum_entries / count
```

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)])
    """
```

Solution:

```
    def helper(t, depth):
        if depth % 2 == 0:
            entry = map_fn(t.entry)
        else:
            entry = t.entry
        branches = [helper(b, depth+1) for b in t.branches]
        return Tree(entry, branches)
    return helper(t, 0)
```

Alternate solution without a helper function:

```
def alt_tree_map(t, map_fn):
    entry = map_fn(t.entry)
    branches = []
    for b in t.branches:
        next_branches = [alt_tree_map(bb, map_fn) for bb in
            b.branches]
        branches.append(Tree(b.entry, next_branches))
    return Tree(entry, branches)
```

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.

Solution:

```
class Tree:
    """A tree with entry as its root value."""
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
```

```
            assert isinstance(branch, Tree)
            branch.parent = self
        self.branches = list(branches)
```

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):
```

Solution:

```
    assert len(self.branches) > 0 and
        len(other.branches) > 0,
        "Must have children to swap."
    self.branches[0], other.branches[-1] =
        other.branches[-1], self.branches[0]
    self.branches[0].parent, other.branches[-1].parent =
        self, other
```

The important part here is that the parent pointers must be updated as well.