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:

```python
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.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>square(1)</td>
<td>1*1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>square(2)</td>
<td>2*2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>square(100)</td>
<td>100*100</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>square(n)</td>
<td>n*n</td>
<td>1</td>
</tr>
</tbody>
</table>
• 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.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>factorial(1)</td>
<td>1*1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>factorial(2)</td>
<td>2<em>1</em>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>factorial(100)</td>
<td>100<em>99</em>...<em>1</em>1</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( n )</td>
<td>factorial(( n ))</td>
<td>( n*(n-1)*...<em>1</em>1 )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

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
- \( \Theta(n^2), \Theta(n^3), \text{etc.} \) — 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)

Solution: $\Theta(n^2)$, 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: $\Theta(\phi^n)$, where $\phi$ 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: $\Theta(n)$, since the while loop executes $n$ times with each iteration taking a constant $\Theta(1)$ time.

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

Solution: $\Theta(1)$, since at worst it will require 6 recursive calls to reach the base case. So this is $\Theta(6)$, which can be reduced to $\Theta(1)$. 
4. def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total

Solution: $\Theta(\log(n))$, 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 $\text{foo}\left(\text{bar}(n)\right)$?

Solution: $\Theta(n^2)$
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.

```python
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.

```python
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
    """
```

Solution:

```python
if t.label % 2 != 0:
    t.label += 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 \( n \).

\[
def \text{fill_tree}(t, n):
    """
    >>> t0 = Tree(0, [Tree(1), Tree(2)])
    >>> t1 = \text{fill_tree}(t0, 5)
    >>> t1
    Tree(5, [Tree(5), Tree(5)])
    """
    Solution:
    filled_branches = [\text{fill_tree}(b, n) \text{ for } b \text{ in } t.\text{branches}]
    return Tree(n, filled_branches)
\]

3. Write a function that combines the labels of two trees \( t1 \) and \( t2 \) together with the \text{combiner} function. Assume that \( t1 \) and \( t2 \) have identical structure. This function should return a new tree.

\[
def \text{combine_tree}(t1, t2, \text{combiner}): 
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = \text{combine_tree}(a, b, \text{mul})
    >>> combined.\text{label}
    4
    >>> combined.\text{branches}[0].\text{label}
    10
    """
    Solution:
    combined = [\text{combine_tree}(b1, b2, \text{combiner}) \text{ for } b1, b2 \text{ in } \text{zip}(t1.\text{branches}, t2.\text{branches})]
    return Tree(\text{combiner}(t1.\text{label}, t2.\text{label}), combined)
4. Assuming that every label in $t$ is a number, let’s define $\text{average}(t)$, which returns the average of all the labels in $t$.

```python
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
    """
```

Solution:

```python
def sum_helper(t):
    sum_entries, count = t.label, 1
    for b in t.branches:
        child_sum, child_count = sum_helper(b)
        sum_entries += child_sum
        count += child_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.

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

    Alternate solution without a helper function:

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

   ```python
   Solution:
   class Tree:
       """A tree with label as its root value."""
       def __init__(self, label, branches=[]):
           self.label = label
           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!

```python
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.
```