National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2013/2014

Recitation 2
Recursion & Orders of Growth

Definitions

Theta (Θ) notation:

$$f(n) = \Theta(g(n)) \rightarrow \text{There exist } k_1,\ k_2,\ n \text{ s.t.: } k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n), \text{for } n > n_0$$

Big-O notation:

$$f(n) = O(g(n)) \rightarrow \text{There exist } k,\ n \text{ s.t.: } f(n) \leq k \cdot g(n), \text{for } n > n_0$$

Adversarial approach: For you to show that $f(n) = \Theta(g(n))$, you pick k_1, k_2, and n_0, then I (the adversary) try to pick an n which doesn't satisfy $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$.

Implications

Ignore constants. Ignore lower order terms. For a sum, take the larger term. For a product, multiply the two terms. Orders of growth are concerned with how the effort scales up as the size of the problem increases, rather than an exact measure of the cost.

Typical Orders of Growth

- $\Theta(1)$ - Constant growth. Simple, non-looping, non-decomposable operations have constant growth.
- $\Theta(\log n)$ - Logarithmic growth. At each iteration, the problem size is scaled down by a constant amount: `(call-again (/ n c))`.
- $\Theta(n)$ - Linear growth. At each iteration, the problem size is decremented by a constant amount: `(call-again (- n c))`.
- $\Theta(n \log n)$ - Nifty growth. Nice recursive solution to normally $\Theta(n^2)$ problem.
- $\Theta(n^2)$ - Quadratic growth. Computing correspondence between a set of n things, or doing something of cost n to all n things both result in quadratic growth.
- $\Theta(2^n)$ - Exponential growth. Really bad. Searching all possibilities usually results in exponential growth.

What's n?

Order of growth is *always* in terms of the size of the problem. Without stating what the problem is, and what is considered primitive (what is being counted as a "unit of work" or "unit of space"), the order of growth doesn't have any meaning.

Problems:

1. Give order notation for the following:
   (a) $5n^2 + n$
   (b) $\sqrt{n} + n$
   (c) $3^n n^2$

2.
```
function fact(n) {
    if(n === 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```
   Running time? Space?

3. Write an iterative version of `fact`.

4.
```
function find_e(n) {
    if(n === 0) {
        return 1;
    } else {
        return (1 / fact(n)) + find_e(n - 1);
    }
}
```
   Running time? Space?

5. Assume you have a function `is_divisible(n, x)` which returns `true` if `n` is divisible by `x`. It runs in $O(n)$ time and $O(1)$ space. Write a function `is_prime` which takes a number and returns `true` if it's prime and `false` otherwise. You'll want to use a helper function.

Running time? Space?

6. **Homework:** Write an iterative version of `find_e`.

Running time? Space?