Algorithms

---

1 Limits

The limit as n approaches infinity of a given complexity function will allow you to determine if the given equation is Theta, Little-O, or Little-Omega of another function. By dividing the given function (g(n)) by the function that you are looking to see if it exists within (f(n)), we can determine what kind of relationship the two functions have by examining the output.

If the limit is a constant, g(n) exists within Theta of f(n)

If the limit is 0, g(n) exists within Little-O of f(n)

If the limit is infinity, g(n) exists within Little-Omega of f(n)

Example 1: Show that $n^2/2$

Equation	Explanation
$g(n)/f(n)$	First we must setup the function in this form
$\frac{n^2}{2}/n^3$	setup the equation
$\frac{n^2}{2} \cdot \frac{1}{n^3}$	We can arrange the function like this
$\frac{1}{2n}$	We can simplify the equation to this
$1/\infty$	As n approaches infinity, the denominator approaches infinity
0	As n increases, the equation approaches 0

As mentioned before the example, if the limit of n approaching infinity equals 0, then g(n) exists within little omega of f(n), thus proving that (n2) /2 exists within little-O of (n3)

2 Properties of Order

Symmetry

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$
$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$
$f(n) \in o(g(n))$ if and only if $g(n) \in \omega(f(n))$

Transitivity

If f(n) is upper bounded by g(n), and g(n) is upper bounded by h(n) then h(n) would also be an upper bound of f(n)

Reflexivity

If f(n) is in O(f(n)) and f(n) is in Omega(f(n)) then f(n) is also in Theta(f(n))

3 Order of Logarithms

If $b > 1$ and $a > 1$ then $log_a n \in \Theta log_b n$

As long as the values of a and b are greater than 0, the above statement remains true. For example.

$$log_2 n \in \Theta log_8 n \text{ is true, however}$$

$$log_8 n \in \Theta log_2 n \text{ is also true}$$

4 Order of a^n

if $b > a > 0$ then $a^n \in O(b^n)$

For example, let $a = 3$ and $b = 3$, then $3^n \in O(5^n)$

5 Order of Different Time Complexities

This is a list of time complexities going from the slowest growing at the top and the fastest growing at the bottom. A function in the list is "little o" of the functions below it. For example, $n \in o(n^2)$

Let $j < k$ and $m < n$
$\Theta(1)$
$\Theta(log(n))$
$\Theta(lg(n))$
$\Theta(n)$
$\Theta(nlg(n))$
$\Theta(n^2)$
$\Theta(n^j)$
$\Theta(n^k)$
$\Theta(a^m)$
$\Theta(a^n)$
$\Theta(n!)$
$\Theta(n^n)$

6 Order of Sums

For $c \geq 0$ and $d \geq 0$
if:

$$f_1(n) \in \Theta(g(n))$$

$$f_2(n) \in \Theta(g(n))$$

then,

$$c \cdot f_1(n) + d \cdot f_2(n) \in \Theta(g(n))$$

In other words,if there are two functions that both belong to $\Theta(g(n))$, then the sum of those two functions still belongs to $\Theta(g(n))$

7 Order of Code Examples

Example 1:

For $(i = 1; i < n; i++)$:
    $total = total + i$

This snippet of code is $\Theta(n)$ because it is doing a constant time operation n times. No matter how many constant time operations there are in the body of the loop, it would remain $\Theta(n)$

Example 2:
For $(i = 0; i \leq n; i++)$:
    $\Theta(1)$
For $(i = 0; i \leq n; i++)$:
    $\Theta(1)$

This snippet of code is $\Theta(n)$ because the two loops do not affect the runtime of the other. The code snippet runs a constant time operation $2n$ times, however the definition of Θ cancels out multiplicative factors, so we just say the snippet is $\Theta(n)$

Example 3:
For $(i = 0; i \leq n\%1000; i++)$:
    $\Theta(1)$

Although it is tempting to say that this loop is $\Theta(n)$, it's actually $\Theta(1)$ because $n\%1000$ will always be a number between 0 and 999 no matter the n. So this loop's runtime isn't actually dependent on n, it is $\Theta(1)$

Example 4:
For $(i = 0; i \leq n; i++)$:
    For $(i = 0; i \leq n; i++)$:
        $\Theta(1)$
This code snippet is $\Theta(n^2)$ Because n constant time operations is being done n times.

Example 5:
For $(i = 0; i \leq n; i++)$:
    j = n
    while($j \leq 1$):
        Sum += i
        $j = j/2$

This code snippets outer loop belongs to $\Theta(n)$ because it runs n times, but the inner loop belongs to $\Theta(lgn)$ because its run time is being divided by 2. So all together the code snippet is $\Theta(nlgn)$ because the loop is running a function with a lgn run time, n times.