Big-O and Big-Ω and Big θ

Suppose $f, g : \mathbb{N} \longrightarrow \mathbb{N}$

$f(n)$ is $O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

$f(n)$ is $\Omega(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$.

$f(n)$ is $\theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

$f(n)$ is $o(g(n))$ if $\lim\limits_{n \to \infty} f(n)/g(n) = 0$.

$f(n)$ is $\omega(g(n))$ if $\lim\limits_{n \to \infty} g(n)/f(n) = 0$.

$f(n) \sim g(n)$ if $\lim\limits_{n \to \infty} f(n)/g(n) = 1$.

$f(n) = O(g(n))$ is not an equality, rather a way of writing $f(n)$ is $O(g(n))$.

The proper defn of $O(g(n))$ is as a set
$O(g(n)) = \{ f(n) \mid$ there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n) \}$
and then $f(n) = O(g(n))$ can be interpreted as meaning $f(n) \in O(g(n))$.

Examples:

$3n+2 = O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$.

$3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$.

$100n+6 = O(n)$ as $100n+6 \leq 101n$ for all $n \geq 6$.

$10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$.

$1000n^2+100n-6 = O(n^2)$ as $1000n^2+100n-6 \leq 1001n^2$ for all $n \geq 100$.

$6*2^n+n^2 = O(2^n)$ as $6*2^n+n^2 \leq 7*2^n$ for all $n \geq 4$.

$3n+3 = O(n^2)$ as $3n+3 \leq 3n^2$ for all $n \geq 2$.

$10n^2+4n+2 = O(n^4)$ as $10n^2+4n+2 \leq 10n^4$ for all $n \geq 2$.

$3n+2 \neq O(1)$ as $3n+2$ is not less than or equal to c for any constant c and all $n \geq n_0$.

$10n^2+4n+2 \neq O(n)$.

$O(1)$ is called constant computing time; $O(n)$ is called linear; $O(n^2)$ is called quadratic; $O(n^3)$ is called cubic and $O(2^n)$ is called exponential.

If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n\log n)$ is better than $O(n^2)$ but not as good as $O(n)$.

It is now clear that $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$. But it does not say anything about how good this bound is. Note that $n = O(2^n)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = (n\log n)$ and so on. It should now be clear why these notations are called asymptotic.

__/__/__

As in the case of big-oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$.

The way $g(n)$ should be as small as possible for which $f(n) = O(g(n))$ to be informative or of any use, $f(n) = \Omega(g(n))$ is informative only when $g(n)$ is as large as possible.
So, while we say that $3n+3 = O(n)$, we almost never say $3n+3 = O(n^2)$ though it is correct.
Similarly, while we say that $3n+3 = \Omega(n)$ and $6*2^n + n^2 = \Omega(2^n)$, we almost never say that $3n+3 = \Omega(1)$ or $6*2^n + n^2 = \Omega(1)$, though they are correct.

We further observe that the theta notation is more precise than both the big-oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper bound and a lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$s used have been 1. This is only the practice. We don't say that $3n+3 = O(3n)$ or $10 = O(100)$ or $10n^2+4n+2 = \Omega(4n^2)$ or $6*2^n + n^2 = O(6*2^n)$ or $6*2^n + n^2 = \Theta(4*2^n)$ even though each one is true.

Asymptotic time complexity by step count:
Following are two algorithms, one iterative and one recursive, to add elements of an array:

```
Algo ISum(a,n){
    s = 0.0;
    for i = 0 to n-1
        s = s + a[i];
    return s;
}
```

```
Algo RSum(a,n){
    if (n ≤ 0) return 0.0;
    else return RSum(a, n-1) + a[n-1];
}
```

s/e is steps for one execution

Statement	s/e	frequency	total steps
1 Algo ISum(a,n){	0	—	$\theta(0)$
2 s = 0.0;	1	1	$\theta(1)$
3 for i = 0 to n-1	1	n+1	$\theta(n)$
4 s = s + a[i];	1	n	$\theta(n)$
5 return s;	1	1	$\theta(1)$
6 }	0	—	$\theta(0)$
Total			$\theta(n)$

Asymptotic complexity of ISum

__/__/__

Statement	s/e	frequency		total steps	
		$n<0$	$n\geq 0$	$n<0$	$n\geq 0$
1 Algo RSum(a,n){	0	—	—	0	$\Theta(0)$
2 if (n<0) return 0.0;	1	1	1	1	$\Theta(1)$
3 else return RSum(a,n−1) + a(n−1);	1+x	0	1	0	$\Theta(1+x)$
4 }	0	—	—	0	$\Theta(0)$
Total				1	$\Theta(1+x)$

$$x = t_{RSum}(n-1)$$

Asymptotic complexity of RSum

Asymptotic time complexity by analysis:

```
Algo Perm(a, k, n){
    if (k = n-1) write(a[0 : n-1]);
    else // a[k : n-1] has more than one permutation
        // Generate these recursively
        for i = k to n-1 {
            interchange(a[k], a[i]);
            Perm(a, k+1, n-1);
            // All permutations of a[k+1 : n-1]
            interchange(a[k], a[i]);
        }
}
```

__/__/__

Analysis:

When $k = n-1$, the time taken is $\theta(n)$.

When $k < n-1$, the else clause is entered. At this time, the second for loop is entered $n-k$ times. So,

$t_{perm}(k, n) = \theta((n-k)(n-1+t_{perm}(k+1, n-1)))$ when $k < n-1$.

Since, $t_{perm}(k+1, n-1)$ is at least $n-1$ when $k+1 \leq n-1$, we get

$t_{perm}(k, n) = \theta((n-k)\, t_{perm}(k+1, n-1))$ for $k < n-1$.

Using the substitution method, we obtain

$$t_{perm}(0, n-1) = \theta(n(n!)), \quad n \geq 0.$$