<u>Order Notation - Big Oh</u>

Since we want to simply count the number of simple statements and algorithm runs *in terms of Big-Oh* notation, we need to learn the formal definition of Big-Oh, Big-Omega, and Big-Theta, so that we properly use these technical terms.

Definition of O(g(n)) :

$f(n) = O(g(n))$ iff for all $n \geq n_0$ (n_0 is a constant.)

$f(n) \leq cg(n)$ for some constant c.
(Note: iff means "if and only if")

Here is an example:

Let $f(n) = 2n+1$ and $g(n) = n$. In this situation $f(n) = O(g(n))$.
Here is how to prove it:

Let $c=3$, and $n_0=2$. then $f(n) = 2n+1$ and $cg(n) = 3n$.

Thus, we need to show that $(2n+1) \leq 3n$ for all $n \geq 2$.

$2n+1 \leq 2n + n$, since $n > 1$.
$= 3n$.

Thus, if we say some algorithm takes O(n) time to execute (in the worst case), we are really saying that no matter what input of size n the algorithm receives, it will always complete in cn steps, where c is some constant. We will usually use big-Oh notation when we are describing a worst-case running time.

In general, a simple rule dealing with simple polynomial functions is the following:

If f(n) is a polynomial of degree k, then $f(n) = O(n^k)$.

Question : Is $2n+1 = O(n^{10})$? Answer yes, try $c=1$, $n_0=2$.

Big-Oh(O) is an upper bound. It simply guarantees that a function is no larger than a constant times a function g(n), for O(g(n)).

Here is a definition using a limit :
$f(n) = O(g(n))$
iff lim as $n\to\infty$ $f(n)/g(n) = c$, where c is a constant.

<u>Order Notation - Big Omega</u>

The opposite of big Oh, in some sense, is big Omega.

Definition of Ω:

$f(n) = \Omega(g(n))$ iff for all $n \geq n_0$ (n_0 is a constant.)

$f(n) \geq cg(n)$ for some constant c. (Notice that the ONLY difference here is the inequality sign.)

Here is a quick example:

Let $f(n) = n^2 - 3$
$g(n) = 10n$.

In this situation, we have $f(n) = \Omega(g(n))$. We can prove this as follows:

Let $c = .1$ and $n_0=3$.

Then we have
$f(n) = n^2 - 3$, $cg(n) = n$.

Thus, we need to show that
$n^2 - 3 \geq n$ for all $n \geq 3$.

$$n^2 - 3 \geq n^2 - n, \text{ since } n \geq 3.$$
$$= n(n-1)$$
$$\geq n(2), \text{ since } n\geq 3, n-1\geq 2.$$
$$\geq n.$$

Here is the limit definition of Ω:

$f(n) = \Omega(g(n))$
iff lim as $n\rightarrow\infty$ $f(n)/g(n) > 0$.

In essence, Ω establishes a lower bound for a function. f(n) has to grow at least as fast as g(n) to within a constant factor. With respect to an algorithm, when we say that an algorithm runs in $\Omega(n)$ for example, this means that whenever you run an algorithm with an input of size n, the number of small instructions executed is AT LEAST cn, where c is some positive constant.

Order Notation - Big Theta

Definition of Θ:

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

This simply means that g(n) is both an upper AND lower bound of f(n) within a constant factor. In essence, as n grows large, f(n) and g(n) are within a constant of each other.

Here's the limit definition:

$f(n) = \Theta(g(n))$
iff lim as $n \to \infty$ $f(n)/g(n) = c$, where c is a constant and $c > 0$.

Thus, if we can show that the an algorithm runs in O(f(n)) time for any input of size n, and also show that an algorithm runs in Ω(f(n)) time for any input of size n, we can conclude that both the WORST case running time and BEST case running time are proportional to f(n), (meaning that the number of small instructions run when the program using that algorithm is executed is always some constant times f(n).) If this is the case, we can then claim that the algorithm runs in Θ(f(n)) time.

Thus, we can think of each of these "operators" as comparing functions much like we compare real numbered values. Using this analogy, here is how each operator works:

O is like $\leq$.
Ω is like $\geq$.
Θ is like $=$.

Finally, another way to think about each of these is that they describe a class of functions.

If I say $f(n) = O(n)$, it's just like saying $f(n) \in O(n)$. This means that f(n) can be any one of a number of functions. In particular, f(n) can be any function that proportionate to n OR smaller.

Here is an example of analyzing the running time of an algorithm:

Consider a binary search on a sorted array A of size n for a value val:

```
public static boolean search(int[] A, int
val) {

     low = 0;
     high = A.length-1;
     while (low <= high) {
     mid = (low+high)/2;
     if (val == A[mid])
         return true;
     else if (val > A[mid])
         low = mid+1;
     else
         high = mid - 1;
     }
     return false;
}
```

Remember, we are only considered with the number of simple steps that are executed here within a constant factor.

In general, each loop iteration only contains at most 5 simple statements or comparisons. We can treat this as a constant. Thus, the real question is, how many times does the while loop that contains these 5 statements run?

You'll notice that the difference between high and low decreases by at least a factor of 2 for each iteration.

Essentially, we first are searching amongst n terms, and in the next iteration n/2 terms, then n/4 terms, then n/8 terms, etc.

In essence on the kth iteration, we are searching amongst $n/2^k$ terms. Thus, we want to find the value of k for which $n/2^k = 1$.

$n/2^k = 1$
$n = 2^k$
$k = \log_2 n$, using the definition of log.

Question: Can you prove the algorithm will always stop? Why will it?

Since there are a constant number of statements in a loop that runs at most $\log_2 n$ times, we can confidently say that this algorithm runs in $O(\log_2 n)$ time. The reason that I used O instead of Θ is that it is possible that the algorithm could end on the first iteration, which would mean in that instance the algorithm would run in $\Theta(1)$ time and not $\Theta(\log_2 n)$. This means that the best case running time is $\Omega(1)$. In essence, we bounded the worst case running time, but it's possible that the best case running time is far better. Thus, we just use a O bound instead of a Θ bound. However, it IS true that the average case running time of a binary search is $\Theta(\log_2 n)$, though this is more difficult to prove.

These methods can in general be used to determine the *theoretical run-time* of an algorithm. But, occasionally, an algorithm will prove too difficult to analyze theoretically. In these cases, we can experimentally gauge the run-time of an algorithm. (Furthermore, sometimes it is good to verify that an algorithm is *actually* running as fast as you expect it to do so. Thus, it makes sense to verify theoretical run-times with experiments.)

Verifying Algorithmic Analysis through running actual code

T(N) is the empirical (observed) running time of the code and the claim is made that $T(N) \in O(F(N))$.

Technique is to compute a series of values T(N)/F(N) for a range of N (commonly spaced out by a factors of two). Depending upon these values of T(N)/F(N) we can determine how accurate our estimation for F(N) is according to:

F(N) =
- **is a close answer(θ) if the values converge to a + const.**
- **is an overestimate if the values converge to zero.**
- **is an underestimate if the values diverge .**

Examples

Example 1

Consider the following table of data obtained from running an instance of an algorithm assumed to be cubic. Decide if the Big-Theta estimate, $\Theta(N^3)$ is accurate.

Run	N	T(N)	$F(N) = N^3$	T(N)/F(N)
1	100	0.017058 ms	10^6	1.0758×10^{-8}
2	1000	17.058 ms	10^9	1.0758×10^{-8}
3	5000	2132.2464 ms	$1.25x10^{11}$	1.0757×10^{-8}
4	10000	17057.971 ms	10^{12}	1.0757×10^{-8}
5	50000	2132246.375 ms	$1.25x10^{14}$	1.0757×10^{-8}

The calculated values converge to a positive constant (1.0757×10^{-8}) – so the estimate of Θ (n^3) is an accurate estimate. (In practice, this algorithm runs in $\theta(n^3)$ time.)

Example 2

Consider the following table of data obtained from running an instance of an algorithm assumed to be quadratic. Decide if the Big-Theta estimate, Θ (N^2) is accurate.

Run	N	T(N)	$F(N) = N^2$	T(N)/F(N)
1	100	0.00012 ms	10^4	1.6×10^{-8}
2	1000	0.03389 ms	10^6	3.389×10^{-8}
3	10000	10.6478 ms	10^8	1.064×10^{-7}
4	100000	2970.0177 ms	10^{10}	2.970×10^{-7}
5	1000000	938521.971 ms	10^{12}	9.385×10^{-7}

The values diverge, so the code runs in $\Omega(N^2)$, and has a larger theta bound.

Limitations of Big-Oh Notation

1) not useful for small sizes of input sets
2) omission of the constants can be misleading – example 2NlogN and 1000N, even though its growth rate is larger the first function is probably better. Constants also reflect things like memory access and disk access.
3) assumes an infinite amount of memory – not trivial when using large data sets
4) accurate analysis relies on clever observations to optimize the algorithm.

Growth Rates of Various Functions

The table below illustrates how various functions grow with the size of the input *n*.

Assume that the functions shown in this table are to be executed on a machine which will execute a million instructions per second. A linear function which consists of one million instructions will require one second to execute. This same linear function will require only 4×10^{-5} seconds (40 microseconds) if the number of instructions (a function of input size) is 40. Now consider an exponential function.

log n	√n	n	n log n	n^2	n^3	2^n
0	1	1	0	1	1	2
1	1.4	2	2	4	8	4
2	2	4	8	16	64	16
3	2.8	8	24	64	512	256
4	4	16	64	256	4096	65,536
5	5.7	32	160	1024	32,768	4.294×10^{9}
≈5.3	6.3	40	≈212	1600	64000	1.099×10^{12}
6	8	64	384	4096	262,144	1.844×10^{19}
~10	31.6	1000	9966	10^6	10^9	NaN =)

The Growth Rate of Functions (in terms of steps in the algorithm)

When the input size is 32 approximately 4.3×10^9 steps will be required (since $2^{32} = 4.29 \times 10^9$). Given our system performance this algorithm will require a running time of approximately 71.58 minutes. Now consider the effect of increasing the input size to 40, which will require approximately 1.1×10^{12} steps (since $2^{40} = 1.09 \times 10^{12}$). Given our conditions this function will require about 18325 minutes (12.7 days) to compute. If n is increased to 50 the time required will increase to about 35.7 years. If n increases to 60 the time increases to 36558 years and if n increases to 100 a total of 4×10^{16} years will be needed!

Suppose that an algorithm takes T(N) time to run for a problem of size N – the question becomes – how long will it take to solve a larger problem? As an example, assume that the algorithm is an $O(N^3)$ algorithm. This implies:

$T(N) = cN^3$.

**If we increase the size of the problem by a factor of 10 we have:
$T(10N) = c(10N)^3$. This gives us:
$T(10N) = 1000cN^3 = 1000T(N)$ (since $T(N) = cN^3$)**

Therefore, the running time of a cubic algorithm will increase by a factor of 1000 if the size of the problem is increased by a factor of 10. Similarly, increasing the problem size by another factor of 10 (increasing N to 100) will result in another 1000 fold increase in the running time of the algorithm (from 1000 to 1×10^6).

$T(100N) = c(100N)^3 = 1 \times 10^6 cN^3 = 1 \times 10^6 T(N)$

A similar argument will hold for quadratic and linear algorithms, but a slightly different approach is required for logarithmic algorithms. These are shown below.

For a quadratic algorithm, we have $T(N) = cN^2$. This implies: $T(10N) = c(10N)^2$. Expanding produces the form: $T(10N) = 100cN^2 = 100T(N)$. Therefore, when the input size increases by a factor of 10 the running time of the quadratic algorithm will increase by a factor of 100.

For a linear algorithm, we have $T(N) = cN$. This implies: $T(10N) = c(10N)$. Expanding produces the form: $T(10N) = 10cN = 10T(N)$. Therefore, when the input size increases by a factor of 10 the running time of the linear algorithm will increase by the same factor of 10.

In general, an *f*-fold increase in input size will yield an f^3-fold increase in the running time of a cubic algorithm, an f^2-fold increase in the running time of a quadratic algorithm, and an *f*-fold increase in the running time of a linear algorithm.

The analysis for the linear, quadratic, cubic (and in general polynomial) algorithms does not work when in the presence of logarithmic terms. When an O(N logN) algorithm experiences a 10-fold increase in input size, the running time increases by a factor which is only slightly larger than 10. For example, increasing the input by a factor of 10 for an O(N logN) algorithm produces: $T(10N) = c(10N)\log(10N)$. Expanding this yields: $T(10N) = 10cN\log(10N) = 10cN\log 10 + 10cN\log N = 10T(N) + c'N$ (where $c' = 10c\log 10$). As N gets very large, the ratio $T(10N)/T(N)$ gets closer to 10 (since $c'N/T(N) \approx (10\log 10)/\log N$ gets smaller and smaller as N increases.

The above analysis implies, for a logarithmic algorithm, if the algorithm is competitive with a linear algorithm for a sufficiently large value of N, it will remain so for slightly larger N.