Dr. Ohidujjaman Tuhin
Assistant Professor
Dept. of CSE, UIU

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function *T(n)* - time versus the input size *n*. We want to define time taken by an algorithm without depending on the implementation details. But you agree that *T(n)* does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: **processor speed**; **instruction set**, disk **speed**, **brand of compiler** and etc. The way around is to estimate efficiency of each algorithm *asymptotically*. We will measure time *T(n)* as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is time taken by addition of two bits. On different computers, addition of two bits might take different time, say c_1 and c_2, thus the addition of two n-bit integers takes $T(n) = c_1 * n$ and $T(n) = c_2 * n$ respectively. This shows that different machines result in different slopes, but time *T(n)* grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

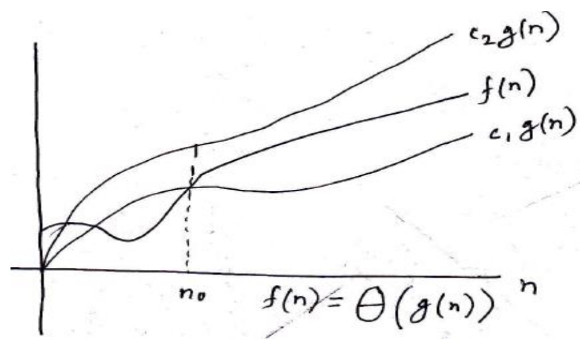
The analysis of an algorithm is a major task in computer science. The time and space complexity used by the algorithm are the two main measures for the efficiency of the algorithm. The time is measured by counting the number of **key operation** i.e. the **number of comparison**. The complexity of an algorithm is the function f(n) which gives the running time and /or storage space requirement of the algorithm in terms of the size n of the input data.

Definition of Theta(Θ) notation: The function $f(n)= \Theta(g(n))$ (read as "f of n is theta of g of n") if and only if there exist positive constants c_1,c_2 and n_0 such that $0<=c_1g(n)<=f(n)<=c_2g(n)$ for all n, $n>=n_0$.

Or,

$f(n)= \Theta(g(n))$

$\Theta(g(n))=\{ f(n)$: there exist positive constants c_1,c_2 and n_0 such that $0<=c_1g(n)<=f(n)<=c_2g(n)$ for all n, $n>=n_0 \}$



Example:

If $f(n) = \frac{1}{2} n^2 - 3n$ then, $f(n) = \Theta(n^2)$

Proof: Here, $f(n) = \frac{1}{2} n^2 - 3n$

$g(n) = n^2$

Therefore, $c_1 g(n) <= f(n) <= c_2 g(n)$

$=> c_1 n^2 <= \frac{1}{2} n^2 - 3n <= c_2 n^2$

$=> c_1 <= \frac{1}{2} - \frac{3}{n} <= c_2$

L. H. Inequality:

$=> c_1 <= \frac{1}{2} - \frac{3}{n}$

So, $n >= 7$, $c_1 <= \frac{1}{14}$

$n=1$, $c1 <= \frac{1}{2} - 3 => c_1 <= -\frac{5}{2}$

$n=2$, $c1 <= \frac{1}{2} - \frac{3}{2} => c_1 <= -1$

.

.

.

$n=7$, $c1 <= \frac{1}{2} - \frac{3}{7} => c_1 <= \frac{1}{14}$

R.H. inequality:

$\frac{1}{2} - \frac{3}{n} <= c_2$

So, $n >= 1$, $c_2 >= \frac{1}{2}$

$n=1$, $\frac{1}{2} - 3 <= c_2 => -\frac{5}{2} <= c_2$

$n=2$, $\frac{1}{2} - \frac{3}{2} <= c_2 => -1 <= c_2$

.

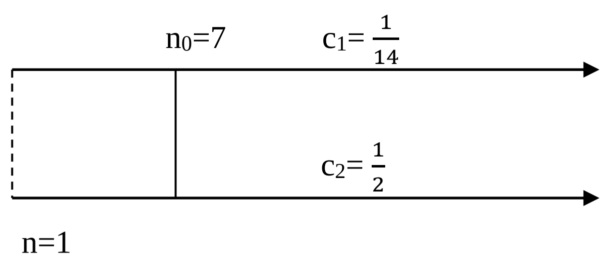
.

$n=\infty$, $\frac{1}{2} <= c_2$

Combining two inequalities:

$c_1 <= \frac{1}{14}$, $c_2 >= \frac{1}{2}$ $n >= 7$

$c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ $n_0 = 7$



Since, $\frac{1}{14}$ g(n)<=f(n)<= $\frac{1}{2}$ g(n)

- ⇨ $\frac{1}{14} n^2 <= \frac{1}{2} n^2 - 3n <= \frac{1}{2}$ g(n)
- ⇨ f(n)= Θ (g(n))
- ⇨ f(n)= Θ (n^2) proved

Example:
If f(n)=$6n^3$, the f(n) != Θ(n^2)

Proof:
c_1g(n)<=f(n)<=c_2 g(n)
=>$c_1n^2<=6n^3<=c_2n^2$
=> $c_1<=6n<=c_2$

L. H. inequality:

⇨ $c_1<=6n$	n=1, $c_1/6<=1$ =>$c_1<=6$
⇨ $c_1/6<=n$	n=2, $c_1/6<=2$ =>$c_1<=12$

So, $c_1<=6$ for n>=1

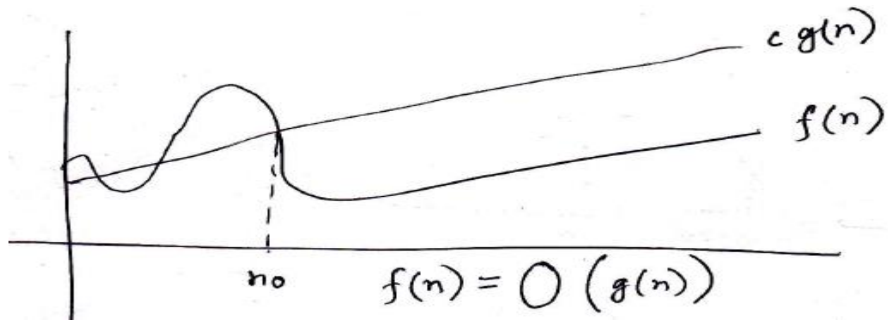
R. H. inequality:

⇨ $c_2>=6n$	n=1, $c_2/6>=1$ =>$c_2>=6$
⇨ $c_2/6>=n$	n=2, $c_2/6>=2$ =>$c_2>=12$
	.
	.
	n=∞, =>$c_2=\infty$

Hence, there is no positive constant for c_2
c_1g(n)<=f(n)<=c_2 g(n) not hold

f(n)!= Θ(n^2)

Definition Big oh(O) notation: The function f(n)= O(g(n)) (read as "f of n is big oh of g of n") if and only if there exist positive constants c and n_0 such that 0<=f(n)<=cg(n) for all n, n>=n_0.
Or,
f(n)= O(g(n))
O(g(n))={ f(n): there exist positive constants c and n_0 such that 0<=f(n)<=cg(n) for all n, n>=n_0 }



Example:

If $f(n) = \frac{1}{2}n^2 - 3n$ then, $f(n) = O(n^2)$

Proof: Here, $f(n) = \frac{1}{2}n^2 - 3n$

$g(n) = n^2$

Therefore, $f(n) <= cg(n)$

$=> \frac{1}{2}n^2 - 3n <= cn^2$

$=> \frac{1}{2} - \frac{3}{n} <= c$

⇨ $\frac{1}{2} - \frac{3}{n} <= c$

So, $n >= 1$, $c >= \frac{1}{2}$

$n=1$, $\frac{1}{2} - 3 <= c$ $=> -\frac{5}{2} <= c$

$n=2$, $\frac{1}{2} - \frac{3}{2} <= c$ $=> -1 <= c$

.

.

$n=\infty$, $\frac{1}{2} <= c$

$n_0=1$ $c=\frac{1}{2}$

$c=\frac{1}{2}$, $n_0=1$

Since, $f(n) <= cg(n)$

$=> \frac{1}{2}n^2 - 3n <= \frac{1}{2}n^2$

$=> f(n) = O(g(n))$

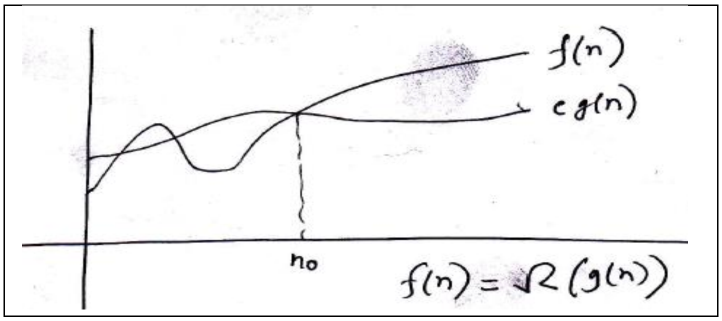
$=> f(n) = O(n^2)$

Definition Big Omega(Ω) notation: The function f(n)= Ω(g(n)) (read as "f of n is big omega of g of n") if and only if there exist positive constants c and n_0 such that 0<= cg(n)<= f(n) for all n, n>=n_0.

Or,

f(n)= Ω(g(n))

Ω(g(n))={ f(n): there exist positive constants c and n_0 such that 0<=cg(n)<=f(n) for all n, n>=n_0 }



Example:

If f (n) = $\frac{1}{2}$ n^2- 3n then, f (n) = **Ω**(n^2)

Proof: Here, f (n) = $\frac{1}{2}$ n^2- 3n

g(n)=n^2

Therefore, cg(n) <= f(n)

=> $cn^2 <= \frac{1}{2} n^2 - 3n$

=> $c <= \frac{1}{2} - \frac{3}{n}$

So, n>=7, $c <= \frac{1}{14}$

n=1, $c <= \frac{1}{2} - 3$ => $c <= -\frac{5}{2}$

n=2, $c <= \frac{1}{2} - \frac{3}{2}$ => c<= -1

.

.

n=7, $c <= \frac{1}{2} - \frac{3}{7}$ => $c <= \frac{1}{14}$

n_0= 7 $c = \frac{1}{14}$

$c = \frac{1}{14}$, n_0=7

Since, cg(n)<= f(n)

=> $\frac{1}{14} n^2 <= \frac{1}{2} n^2 - 3n$

=> f(n)= **Ω**(g(n))

=>f (n) =**Ω**(n^2) proved