Performance Analysis

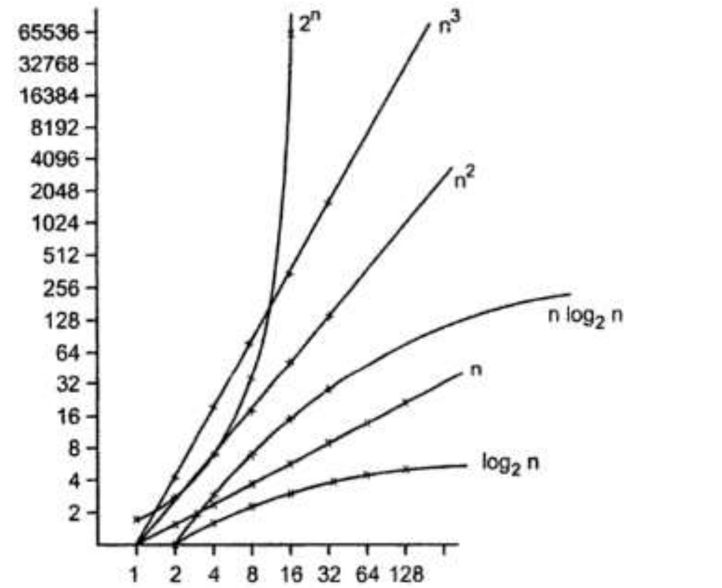
Order of Growth

Order of Growth

- Measuring the performance of an algorithm in relation with input size n is called order of growth.

Order of growth for varying input size of 'n'

n	Log n	n log n	n^2	2^n
1	0	0	1	1
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296



Rate of growth of common computing time function

How efficiency determines practical usability?

input	logn	n	nlogn	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}	10^{11}				

Acceptability: 10 sec *i.e* **10^9 Operations**

Orders of Magnitude

- When comparing T(n) across problems, focus on orders of magnitude and ignore constants
- Eg: $f(n)=n^3$ grows faster than $g(n)=500n^2$, How?
  - *Asymptotic complexity*
- T(n) proportional to

$\log n$	→	Logarithmic	
n	→	Linear	
n^2	→	Quadratic	Polynomial
n^3	→	Cubic	
2^n	→	Exponential	

Growth of Functions- Asymptotic Notations

- *Asymptotic complexity* is a short hand way to represent time complexity.

O –(Big “Oh “)
Ω –(Omega)
Θ – (Theta)

Big Oh Notation

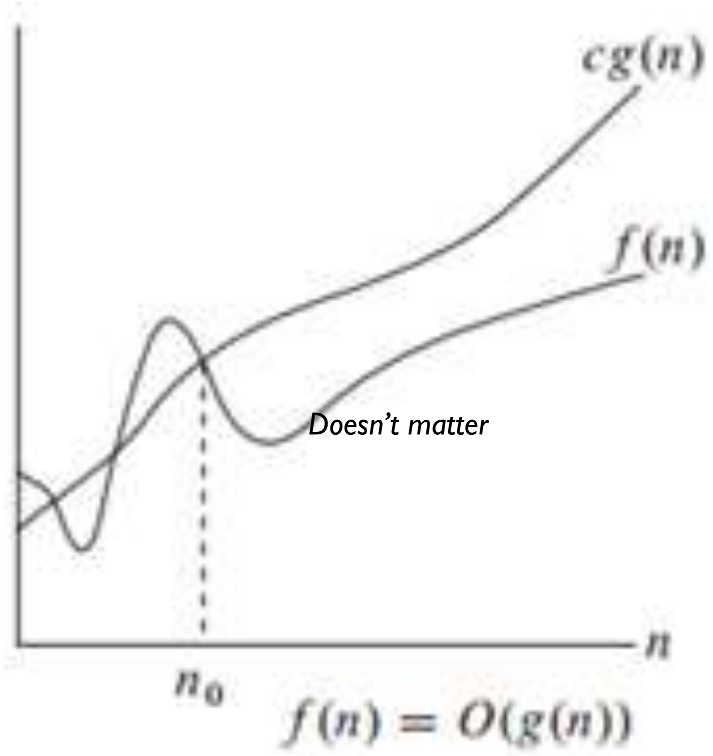
- Big-O is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

- The function $f(n)=O(g(n))$ iff there exists a positive constants c and n_0 such that $f(n)<=c*g(n)$ for all $n>=n_0$

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$


I WON'T QUIT UNTIL I FIND A Θ(n) SORTING ALGORITHM.

Big-oh


cg(n)
f(n)
Doesn't matter
n
n0
f(n) = O(g(n))

Example-Big O

- $3n+2 = O(n)$
- $3n+2 = O(n^2)$
- $3n+2 \neq O(1)$
- $10n^2+4n+2 = O(n^2)$
- $10n^2+4n+2 = O(n^4)$
- $10n^2+4n+2 \neq O(n)$

> ***For f(n)=O(g(n)) to be informative g(n) should be as small a function of n as one can come up with, for which f(n)=Og(n).***

- Since we're trying to generalize this for large values of n, and small values (1, 2, 3,…) aren't that important, we can say that f(n) is generally faster than g(n); that is, f(n) is bound by g(n), and will always be less than it.

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

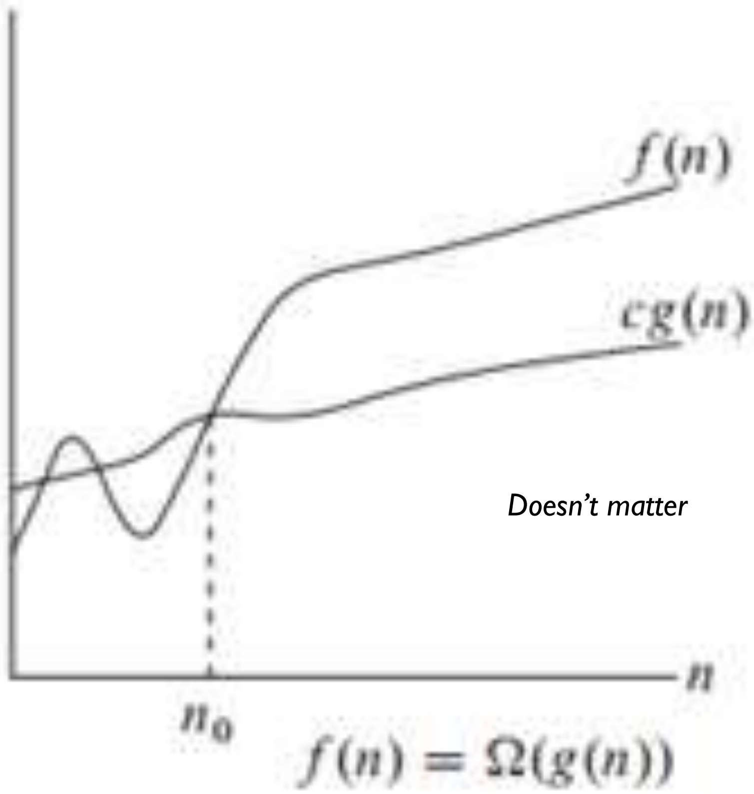
Big Ω- Omega

- For non-negative functions, *f(n)* and *g(n)*, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \geq cg(n)$, then *f(n)* is omega of *g(n)*.
- Denoted as "$f(n) = \Omega(g(n))$".
- This is almost the same definition as Big Oh, except that "$f(n) \geq cg(n)$", this makes *g(n)* a lower bound function, instead of an upper bound function. It describes the **best that can happen** for a given data size.

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$


I WON'T QUIT UNTIL I FIND A Θ(n) SORTING ALGORITHM.

Big- Ω omega


$f(n)$
$cg(n)$
Doesn't matter
n
n_0
$f(n) = \Omega(g(n))$

Ω-Notation: Examples

- $3n+2 = \Omega(n)$
- $3n+2 = \Omega(1)$
- $10n^2+4n+2 = \Omega(n^2)$
- $10n^2+4n+2 = \Omega(n)$
- $10n^2+4n+2 = \Omega(1)$

➢ ***For f(n)= Ω(g(n)) to be informative g(n) should be as large a function of n as possible, for which f(n)= Ω g(n) is true.***

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

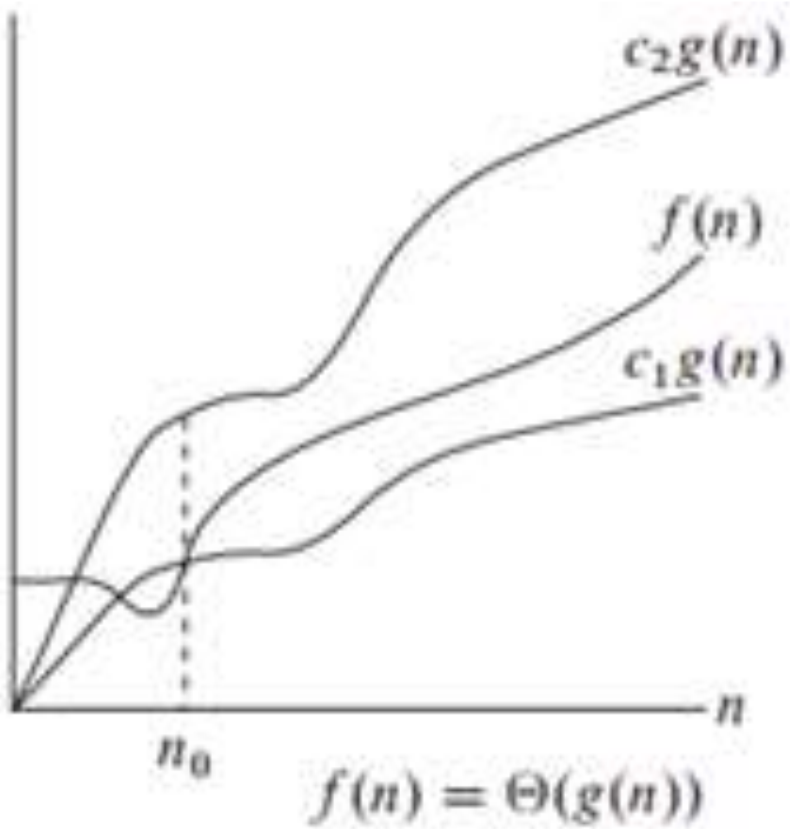
Big Theta

- For non-negative functions, $f(n)$ and $g(n)$, $f(n)$ is theta of $g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Denoted as "$f(n) = \theta(g(n))$".
- Basically, it says that the function, $f(n)$ is bounded both from the top and bottom by the same function, $g(n)$.

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$


I WON'T QUIT
UNTIL I FIND A
Θ(n) SORTING
ALGORITHM.

Big-theta


$c_2g(n)$
$f(n)$
$c_1g(n)$
n
n_0
$f(n) = \Theta(g(n))$

θ Notation-Examples

- $3n+2 = \theta(n)$
  - $3n+2>=3n$ *for* $n>=2$
  - $3n+2<=4n$ *for* $n>=2$
  - $c_1=3$
  - $c_2=4$
  - $n_0=2$

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < 2^n < 3^n < n^n$

Few Examples

- $f(n)=2n^2+3n+4$
- $f(n)=n!$
- $f(n)=n^2\log n+n$

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Practical Complexities

- When the size of n is large ,
  - the function 2^n grows very rapidly with n
  - so the utility of algorithms with exponential complexity is limited to small n
- Algorithms that have a complexity that is a polynomial of high degree are also of limited utility
- From a practical standpoint it is evident that for reasonably large n (say n>100) only algorithms of small complexity such as n,nlogn,n^2 etc .. are feasible

Properties of Order of Growth

1. If $F_1(n)$ is order of $g_1(n)$ and $F_2(n)$ is order of $g_2(n)$, then $F_1(n)+F_2(n) \in O(\max(g_1(n), g_2(n)))$.
2. Polynomials of degree $m \in \Theta(n^m)$.

   That means maximum degree is considered from the polynomial.

   For example : $a_1n^3 + a_2n^2 + a_3n + c$ has the order of growth $\Theta(n^3)$.
3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.
4. Exponential functions a^n have different orders of growth for different values of a.

Key Points:

i) $O(g(n))$ is a class of functions $F(n)$ that grows less fast than $g(n)$, that means $F(n)$ possess the time complexity which is always lesser than the time complexities that $g(n)$ have.

ii) $\Theta(g(n))$ is a class of functions $F(n)$ that grows at same rate as $g(n)$.

iii) $\Omega(g(n))$ is a class of functions $F(n)$ that grows faster than or at least as fast as $g(n)$. That means $F(n)$ is greater than $\Omega(g(n))$.

Name of efficiency class	Order of growth	Description	Example
Constant	1	As input size grows the we get larger running time.	Scanning array elements.
Logarithmic	logn	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration.	Performing binary search operation.
Linear	n	The running time of algorithm depends on the input size n.	Performing sequential search operation.
nlogn	nlogn	Some instance of input is considered for the list of size n.	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	n!	When an algorithm is computing all the permutations then this type of efficiency occurs.	Generating all permutations.


Thank you!