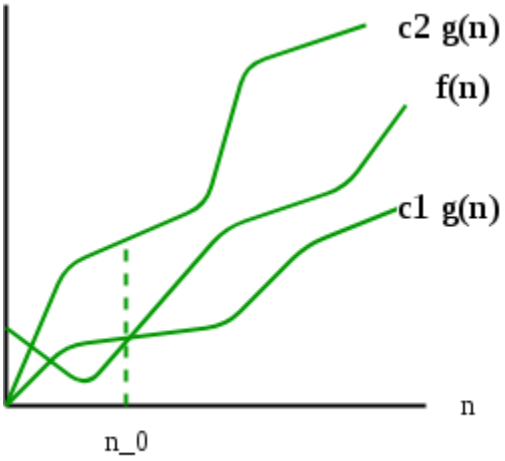
ASYMPTOTIC ANALYSIS

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared.

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

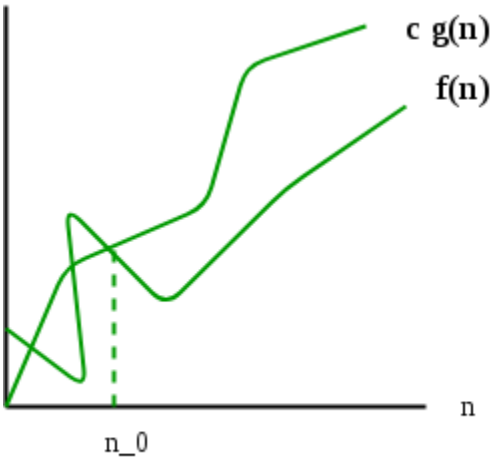


f(n) = theta(g(n))

1) Θ Notation: The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.
A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, consider the following expression.
$3n^3 + 6n^2 + 6000 = \Theta(n^3)$
Dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.
For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



f(n) = O(g(n))

Examples :
{ 100 , log (2000) , 10^4 } belongs to **Θ(1)**
{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Θ(n)**
{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **Θ(n^2)**
Θ provides exact bounds .

2) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
Ω Notation can be useful when we have a lower bound on the time complexity of an algorithm. The Omega notation is the least used notation among all three.
For a given function g(n), we denote by Ω(g(n)) the set of functions.

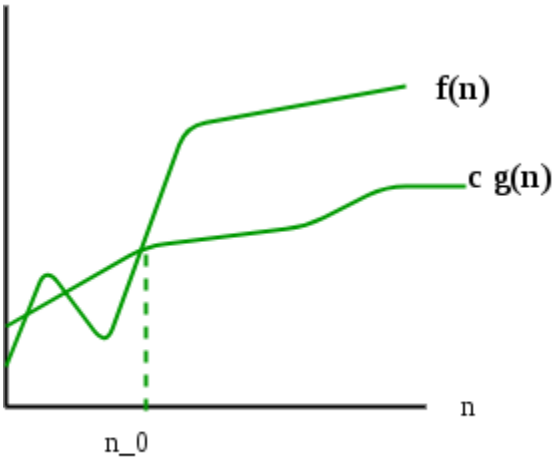
Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= c*g(n) <= f(n) for
all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.
Examples :
{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **Ω(n^2)**
U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Ω(n)**
U { 100 , log (2000) , 10^4 } belongs to **Ω(1)**

Here **U represents union ,** we can write it in these manner because **Ω provides exact or lower bounds .**

3) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
1. The worst-case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).
The Big O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and
n0 such that 0 <= f(n) <= c*g(n) for
all n >= n0}



f(n) = Omega(g(n))

Examples :
{ 100 , log (2000) , 10^4 } belongs to **O(1)**
U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **O(n)**
U { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **O(n^2)**
Here U represents union , we can write it in these manner because **O provides exact or upper bounds .**

BIG-O ANALYSIS OF ALGORITHMS

We can express algorithmic complexity using the big-O notation. For a problem of size N:

- A constant-time function/method is "order 1" : O(1)
- A linear-time function/method is "order N" : O(N)
- A quadratic-time function/method is "order N squared" : O(N 2)

Definition: Let g and f be functions from the set of natural numbers to itself. The function f is said to be O(g) (read big-oh of g), if there is a constant **c > 0** and a natural number $\mathbf{n_0}$ such that f (n) ≤ cg(n) for all n >= n_0 .
Note: O(g) is a set!

Abuse of notation: f = O(g) does not mean f ∈ O(g).
The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described below:

f(n) = O(g(n)) if there exists a positive integer n_0 and a positive constant c, such that $f(n) \leq c.g(n)$ $\forall n \geq n_0$

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Some of the useful properties of Big-O notation analysis are as follow:

Constant Multiplication:
If f(n) = c.g(n), then O(f(n)) = O(g(n)) ; where c is a nonzero constant.

Polynomial Function:
If $f(n) = a_0 + a_1.n + a_2.n^2 + \text{—-} + a_m.n^m$, then $O(f(n)) = O(n^m)$.

Summation Function:
If $f(n) = f_1(n) + f_2(n) + \text{—-} + f_m(n)$ and $f_i(n) \leq f_{i+1}(n)$ $\forall i=1, 2, \text{—-}, m$,
then $O(f(n)) = O(max(f_1(n), f_2(n), \text{—-}, f_m(n)))$.

Logarithmic Function:
If $f(n) = \log_a n$ and $g(n)=\log_b n$, then $O(f(n))=O(g(n))$
; all log functions grow in the same manner in terms of Big-O.

Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically. For any algorithm, the Big-O analysis should be straightforward as long as we correctly identify the operations that are dependent on n, the input size.

RUNTIME ANALYSIS OF ALGORITHMS

In general cases, we mainly used to measure and compare the worst-case theoretical running time complexities of algorithms for the performance analysis. The fastest possible running time for any algorithm is O(1), commonly referred to as *Constant Running Time*. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it's rarely achievable.

In actual cases, the performance (Runtime) of an algorithm depends on n, that is the size of the input or the number of operations is required for each input item. The algorithms can be classified as follows from the best-to-worst performance (Running Time Complexity):

A logarithmic algorithm – O(logn)
Runtime grows logarithmically in proportion to n.

A linear algorithm – O(n)
Runtime grows directly in proportion to n.

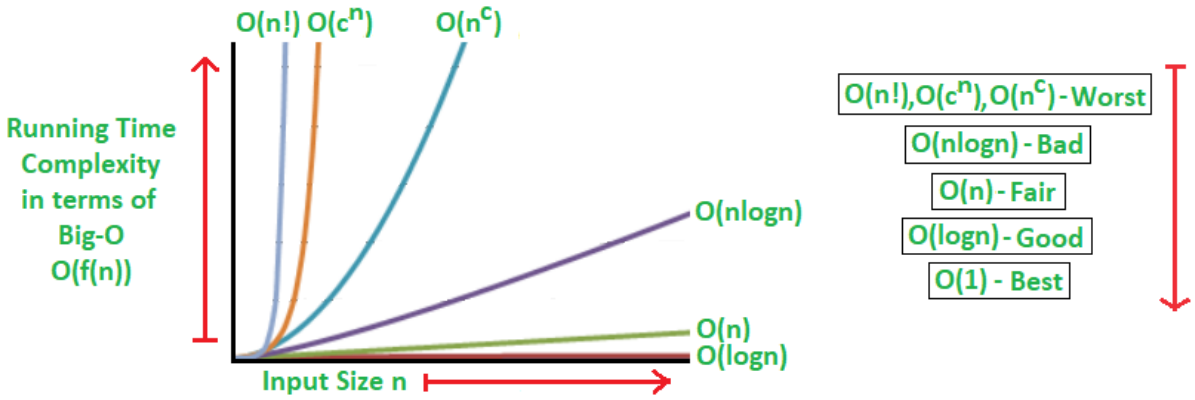
A superlinear algorithm – O(nlogn)
Runtime grows in proportion to n.

A polynomial algorithm – $O(n^c)$
Runtime grows quicker than previous all based on n.

A exponential algorithm – $O(c^n)$
Runtime grows even faster than polynomial algorithm based on n.

A factorial algorithm – O(n!)
Runtime grows the fastest and becomes quickly unusable for even small values of n.

Where, n is the input size and c is a positive constant.



Algorithmic Examples of Runtime Analysis:
Some of the examples of all those types of algorithms (in worst-case scenarios) are mentioned below:

Logarithmic algorithm – O(logn) – Binary Search.

Linear algorithm – O(n) – Linear Search.

Superlinear algorithm – O(nlogn) – Heap Sort, Merge Sort.

Polynomial algorithm – O(n^c) – Strassen's Matrix Multiplication, Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort.

Exponential algorithm – O(c^n) – Tower of Hanoi.

Factorial algorithm – O(n!) – Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem.

Mathematical Examples of Runtime Analysis:
The performances (Runtimes) of different orders of algorithms separate rapidly as n (the input size) gets larger. Let's consider the mathematical example:
If n = 10, If n=20,

log(10) = 1; log(20) = 2.996;

10 = 10; 20 = 20;

10log(10)=10; 20log(20)=59.9;

10^2=100; 20^2=400;
2^{10}=1024; 2^{20}=1048576;
10!=3628800; 20!=2.432902e+18^{18};

MEMORY FOOTPRINT ANALYSIS OF ALGORITHMS

For performance analysis of an algorithm, runtime measurement is not only relevant metric but also we need to consider the memory usage amount of the program. This is referred to as the Memory Footprint of the algorithm, shortly known as Space Complexity.
Here also, we need to measure and compare the worst case theoretical space complexities of algorithms for the performance analysis.
It basically depends on two major aspects described below:

- Firstly, the implementation of the program is responsible for memory usage. For example, we can assume that recursive implementation always reserves more memory than the corresponding iterative implementation of a particular problem.
- And the other one is n, the input size or the amount of storage required for each item. For example, a simple algorithm with a high amount of input size can consume more memory than a complex algorithm with less amount of input size.

Algorithmic Examples of Memory Footprint Analysis: The algorithms with examples are classified from the best-to-worst performance (Space Complexity) based on the worst-case scenarios are mentioned below:

Ideal algorithm - O(1) - Linear Search, Binary Search,

Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Shell Sort.

Logarithmic algorithm - O(log n) - Merge Sort.

Linear algorithm - O(n) - Quick Sort.

Sub-linear algorithm - O(n+k) - Radix Sort.

SPACE-TIME TRADEOFF AND EFFICIENCY

There is usually a trade-off between optimal memory use and runtime performance.

In general for an algorithm, space efficiency and time efficiency reach at two opposite ends and each point in between them has a certain time and space efficiency. So, the more time efficiency you have, the less space efficiency you have and vice versa.

For example, Mergesort algorithm is exceedingly fast but requires a lot of space to do the operations. On the other side, Bubble Sort is exceedingly slow but requires the minimum space.

We can conclude that finding an algorithm that works in less running time and also having less requirement of memory space, can make a huge difference in how well an algorithm performs.

Example of Big-oh noatation:

```
// C++ program to findtime complexity for single for loop
#include <bits/stdc++.h>
using namespace std;
// main Code
int main()
{
        //declare variable
   int a = 0, b = 0;
        //declare size
```

```
    int N = 5, M = 5;
    // This loop runs for N time
    for (int i = 0; i < N; i++) {
        a = a + 5;
    }
    // This loop runs for M time
    for (int i = 0; i < M; i++) {
        b = b + 10;
    }
   //print value of a and b
    cout << a << ' ' << b;
    return 0;
}
```

Output
25 50

Explanation :
First Loop runs *N* Time whereas Second Loop runs *M* Time. The calculation takes *O(1)times*.
So by adding them the time complexity will be O (N + M + 1) = O(N + M).
Time Complexity : O(N + M)

PROPERTIES OF ASYMPTOTIC NOTATIONS :

Some important properties of those notations.

1. General Properties :

If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant.

Example: f(n) = $2n^2+5$ is $O(n^2)$
then 7*f(n) = $7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

We can say
If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)) ; where a is a constant.
If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)) ; where a is a constant.

2. Transitive Properties :

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if $f(n) = n$, $g(n) = n^2$ and $h(n)=n^3$
n is $O(n^2)$ and n^2 is $O(n^3)$
then n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say
If $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n))$ then $f(n) = \Theta(h(n))$.
If $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n))$ then $f(n) = \Omega(h(n))$

3. Reflexive Properties :

Reflexive properties are always easy to understand after transitive.

If $f(n)$ is given then $f(n)$ is $O(f(n))$. Since *MAXIMUM VALUE OF f(n) will be f(n) ITSELF !*

Hence $x = f(n)$ and $y = O(f(n))$ tie themselves in reflexive relation always.

Example: $f(n) = n^2$; $O(n^2)$ i.e $O(f(n))$

Similarly, this property satisfies both Θ and Ω notation.

We can say that:
If $f(n)$ is given then $f(n)$ is $\Theta(f(n))$.

If $f(n)$ is given then $f(n)$ is $\Omega(f(n))$.

4. Symmetric Properties :

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$.

Example: $f(n) = n^2$ and $g(n) = n^2$
then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

5. Transpose Symmetric Properties :

If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$.

Example: $f(n) = n$, $g(n) = n^2$
then n is $O(n^2)$ and n^2 is $\Omega(n)$

This property only satisfies O and Ω notations.

6. Some More Properties :

1.) If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$

2.) If f(n) = O(g(n)) and d(n)=O(e(n))
then f(n) + d(n) = O(max(g(n), e(n)))
Example: f(n) = n i.e O(n)
d(n) = n^2 i.e O(n^2)
then f(n) + d(n) = n + n^2 i.e O(n^2)

3.) If f(n)=O(g(n)) and d(n)=O(e(n))
then f(n) * d(n) = O(g(n) * e(n))
Example: f(n) = n i.e O(n)
d(n) = n^2 i.e O(n^2)
then f(n) * d(n) = n * n^2 = n^3 i.e O(n^3)

Note :
If f(n) = O(g(n)) then g(n) = Ω(f(n))
Exercise:
Which of the following statements is/are valid?

1. Time Complexity of QuickSort is Θ(n^2)
2. Time Complexity of QuickSort is O(n^2)
3. For any two functions f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
4. Time complexity of all computer algorithms can be written as Ω(1)