ASYMPTOTIC NOTATION

➔ Formal way notation to speak about functions and classify them

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O) ,
2. Big–OMEGA (Ω),
3. Big–THETA (Θ) and
4. Little–OH (o)

Asymptotic Analysis of Algorithms:

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

· It is a way to describe the characteristics of a function in the limit.
· It describes the rate of growth of functions.
· Focus on what's important by abstracting away low-order terms and constant factors.
· It is a way to compare "sizes" of functions:

$O \approx \leq$

$\Omega \approx \geq$
$\Theta \approx =$
$o \approx <$
$\omega \approx >$

Time complexity	Name	Example
O(1)	Constant	Adding an element to the front of a linked list
O(logn)	Logarithmic	Finding an element in a sorted array
O (n)	Linear	Finding an element in an unsorted array
O(nlog n)	Linear	Logarithmic Sorting n items by 'divide-and-conquer'-Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

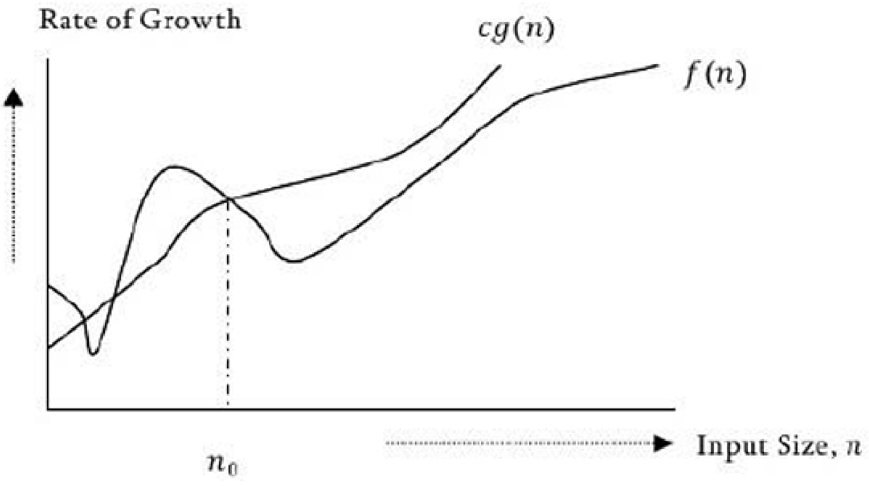
Big 'oh': the function f(n)=O(g(n)) iff there exist positive constants c and no such that f(n)<=c*g(n) for all n, n>= no.
Omega: the function f(n)=(g(n)) iff there exist positive constants c and no such that f(n) >= c*g(n) for all n, n >= no.
Theta: the function f(n)=(g(n)) iff there exist positive constants c1,c2 and no such that c1 g(n) <= f(n) <= c2 g(n) for all n, n >= no

Big-O Notation

This notation gives the tight upper bound of the given function. Generally we represent it as f(n) = O(g (11)). That means, at larger values of n, the upper bound off(n) is g(n). For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is g(n). That means g(n) gives the maximum rate of growth for f(n) at larger values of n.

O —notation defined as O(g(n)) = {f(n): there exist positive constants c and n_o such that $0 <= f(n) <= cg(n)$ for all $n >= n_o$}. g(n) is an asymptotic tight upper bound for f(n). Our objective is to give some rate of growth g(n) which is greater than given algorithms rate of growth f(n).

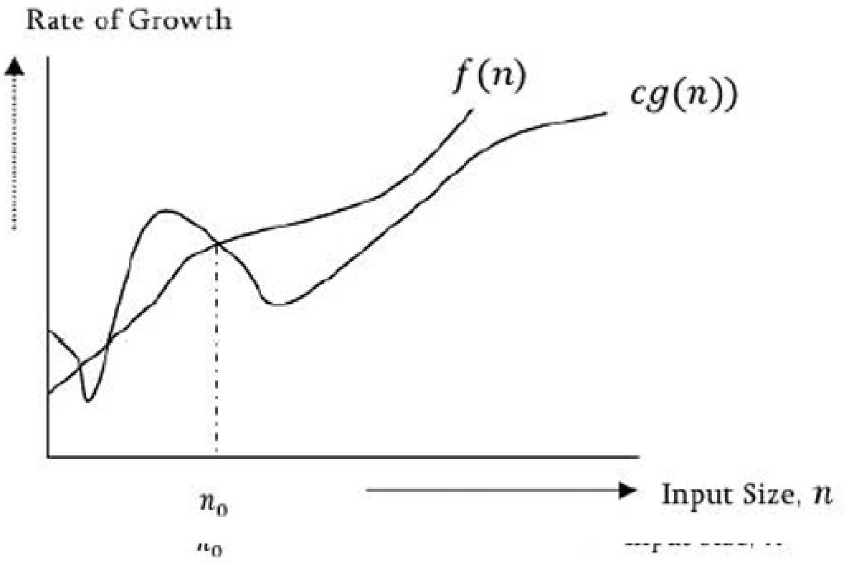
In general, we do not consider lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_o is the point from which we consider the rate of growths for a given algorithm. Below n_o the rate of growths may be different.



Note Analyze the algorithms at larger values of n only What this means is, below no we do not care for rates of growth.

Omega— Ω notation

Similar to above discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as f(n) = Ω (g(n)). That means, at larger values of n, the tighter lower bound of f(n) is g
For example, if f(n) = $100n^2 + 10n + 50$, g(n) is Ω (n^2).
The . Ω. notation as be defined as Ω (g (n)) = {f(n): there exist positive constants c and n_o such that 0 <= cg (n) <= f(n) for all n >= n_o}. g(n) is an asymptotic lower bound for f(n). Ω (g (n)) is the set of functions with smaller or same order of growth as f(n).

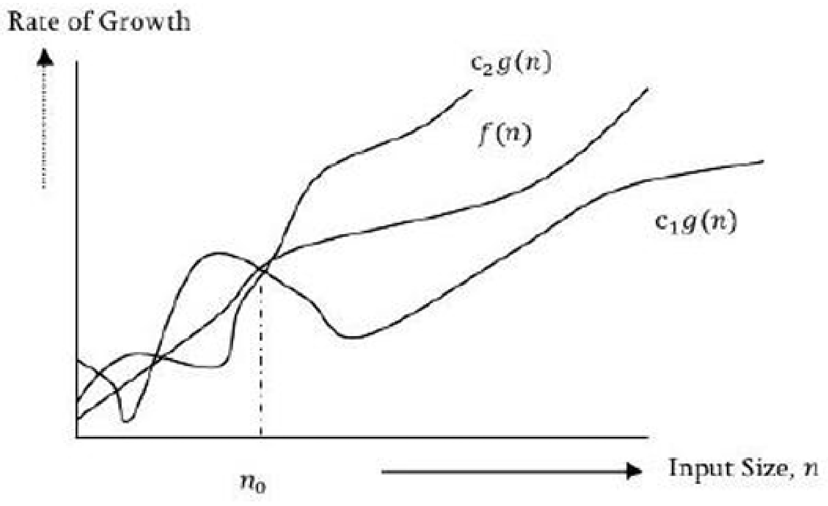


Theta- Θ notation

This notation decides whether the upper and lower bounds of a given function are same or not. The average running time of algorithm is always between lower bound and upper bound.

If the upper bound (O) and lower bound (Ω) gives the same result then Θ notation will also have the same rate of growth. As an example, let us assume that f(n) = 10n + n is the expression. Then, its tight upper bound g(n) is O(n). The rate of growth in best case is g (n) = 0(n). In this case, rate of growths in best case and worst are same. As a result, the average case will also be same.

None: For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth Θ case may not be same.



Now consider the definition of Θ notation It is defined as Θ (g(n)) = {f(71): there exist positive constants C1, C2 and no such that O<=5 c_1g(n) <= f(n) <= c_2g(n) for all n >= n_o}. g(n) is an asymptotic tight bound for f(n). Θ (g(n)) is the set of functions with the same order of growth as g(n).

Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (Θ) may not be possible always.
For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (Θ).
In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance and we use 9 notation if upper bound (O) and lower bound (Ω) are same.

Little Oh Notation

The little Oh is denoted as o. It is defined as : Let, f(n} and g(n} be the non negative functions then

$$\lim_{n \to \propto} \frac{f(n)}{g(n)} = 0$$

such that f(n}= o(g{n)} i.e f of n is little Oh of g of n.

f(n) = o(g(n)) if and only if f'(n) = o(g(n)) and f(n) != Θ {g(n))

PERFORMANCE ANALYSIS:

- What are the Criteria for judging algorithms that have a more direct relationship to performance?
- computing time and storage requirements.

- **Performance evaluation** can be loosely divided into two major phases:
- a priori estimates and
- a posteriori testing.
- ➔refer as *performance analysis* and *performance measurement* respectively

- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Space Complexity:

- Space Complexity Example:
- Algorithm abc(a,b,c)

```
{
    return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

→ The Space needed by each of these algorithms is seen to be the sum of the following component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.
The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

The space requirement s(p) of any algorithm p may therefore be written as,
S(P) = c+ Sp(Instance characteristics)
Where ‘c’ is a constant.

Example 2:

```
Algorithm sum(a,n)
{
s=0.0;
for I=1 to n do
s= s+a[I];
return s;
}
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
- The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
- This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
- So,we obtain Ssum(n)>=(n+s)
- [n for a[],one each for n,I a& s]

Time Complexity:

- The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

- The number of steps any problem statement is assigned depends on the kind of statement.

- For example, comments à 0 steps.
Assignment statements is 1 steps.

[Which does not involve any calls to other algorithms]
Interactive statement such as for, while & repeat-untilà Control part of the statement.

We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm:

```
Algorithm sum(a,n)
{
s= 0.0;
count = count+1;
for I=1 to n do
{
count =count+1;
s=s+a[I];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}
```

→ If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

→By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	*Steps per execution*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Complexity of Algorithms

The complexity of an algorithm M 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. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'.

Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. **Best Case** : The minimum possible value of f(n) is called the best case.

2. **Average Case** : The expected value of f(n).

3. **Worst Case** : The maximum value of f(n) for any key possible input.