3.3.1 Time complexity: The time complexity of an algorithm is the amount of computer time it need to run to complete.

In simple language, every algorithm requires some amount of computer time to execute its instructions to perform specific task. This time is called as time complexity.

Generally, the running time of an algorithm depends upon following:

i. Whether it is running on single processor system or multiprocessor system.
ii. Whether it is a 32-bit system or 64-bit system.
iii. Read and write speed of the system.
iv. The amount of time required by an algorithm to perform arithmetic operations, logical operations, assignment operations and return values.
v. Input data.

Time complexity considers how many number of times each statement executes.

Is time complexity of an algorithm same as running time time/execution time of the algorithm?

Answer: time complexity of an algorithm is not equal to the actual time required to execute a particular algorithm, but number of times a statement executes.

T(P) = compile time + execution time
T(P) = tp (execution time)

Step count:

i. For algorithm heading = 0 time
ii. For braces = 0
iii. For expressions = 1 unit of time
iv. For any looping statement = number of times the loop is repeating

Example:

i.
```
Algorithm abc(d, e, f)...............................................0 time
{.....................................................................0 time
    Return d + e * f + (d + e + f)/(d + e) + 4.0.............1 unit of time
} ....................................................................0 time
                                                       Total time = 1 unit
```

ii. Algorithm sum(a,n)…………………………………… ..0 time
{……………………………… ………………………0 time
S = 0.0; ……………………………………………1
For i=1 to n ………………n + 1 time // n = true cases, 1 = false case
S = s + a[i]; ………………………………………..n
Return s; ………………………………………………..1
} ……………………………………………………….0
Total time = 2n + 3

iii. Algorithm rsum(a, n) //recursive algorithm

```
{
    If (n<=0) then
            Return 0.0;
    Else
            Return rsum(a, n-1) + a[n];
}
```

Assume n = 0
Then T (0) = 2 // 1unit time for if statement and 1 for return statement

Assume n > 0
Then T(n) = 2+ T(n-1)
T(n) = 2+(2+T(n-2))
= 2*2+T(n-2)
= 2* 2+(2+T(n-3))
= 2*3+T(n-3)
.
.
.
= 2 * n + T(n-n)
= 2n+T(0) //T (0)=2; from above
T(n) = 2n+2

iv. Write a C/C++ code to find maximum between N numbers, where N varies from 10, 100, 1000, 10000. In linux operating system, if we run the program with the following commands:
gcc program.c -o program
time ./program

result:
for N = 10……………………..0.5 ms time
for N = 10,000 ………………0.2 ms time may require

this example shows that actual time required to execute code is machine dependent.

Consider the following two scenarios to more understand the time complexity:

Suppose you are having one problem and you have 3 algorithms for the same problem. Now among these 3 algorithms you want to choose the best one. How to choose it?

Solution 1: run all the 3 algorithms on 3 different computers. Provide same input and find time taken by all 3 algorithms and choose the one who took less time. But in this solution there is possibility that these systems might be using different processors, so processing speed might vary. So this solution is not efficient.

Solution 2: run all 3 algorithms on same computer and find which algorithm is taking least time. But here also we might get wrong results because at the time of execution of a program, other things are also running simultaneously. So this solution is also not efficient.

So there should be some standard notation to analyze the algorithm. These notations are called as ***"Asymptotic Notations"***.

In asymptotic notation, system configuration is not considered. Rather order of growth of input will be considered.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Will try to find out how time or space taken by the algorithm will increase/decrease after increasing/decreasing input size.

There are 3 asymptotic notations that are used to represent time complexity of an algorithm:

a. Big O Notation
b. Omega Ω Notation
c. Theta Θ Notation

Usually, time required by an algorithm falls under 3 types:

a. **Best case:** minimum time required for program execution.
b. **Average case:** average time required for program execution.
c. **Worst case:** maximum time required for program execution.

Example:

Consider an array of size 5. If we want to find out one particular element from say [1, 2, 3, 4, 5] i.e. consider we want to find 1, then we found it at the very first position. So this is best case. Now suppose array elements are [2, 3, 4, 5, 1] and we want to find 1 again. It is present, but this time at the last position. So this can be considered as average case. Again if the array is [2, 4 ,5, 3, 1] and if we are trying to find element 6 then this can be considered as worst case scenario as 6 is not present in the array.

1. **Big O Notation:**

The Big O notation defines the upper bound of any algorithm i.e. your algorithm cant take more time than this time.
In other words, Big O denotes maximum time taken by an algorithm. Thus it gives worst case complexity of an algorithm.
The Big O notation is the most used notation for time complexity of an algorithm.

A function f(n) is said to be in O(g(n)), if f(n) is bounded above by some constant multiple 'c' of g(n) for all large 'n'.

$$F(n) <= c * g(n) \qquad \text{for every } n >= n_0$$

Note: f (n) = runtime of our algorithm
G (n) = arbitrary time complexity we are trying to relate to our algorithm.

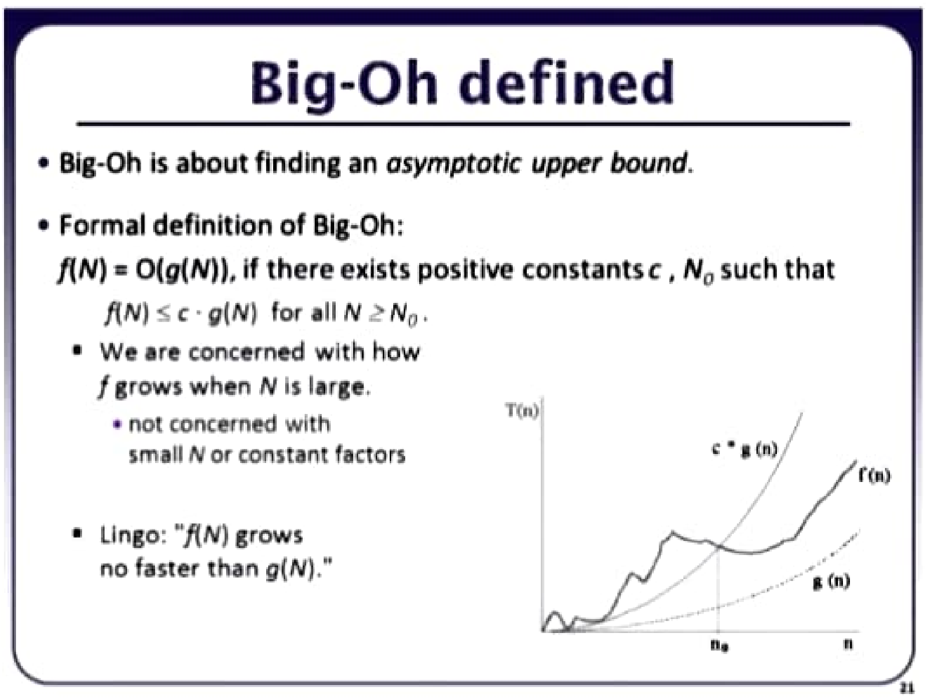

Fig. 1.4 Big O Notation

Example:

1. $F(n) = 2n + 2$, $\qquad g(n) = n^2$
Note: f (n) should always be less than or equal to g(n)

Let n = 1
$F(1) = (2*1) + 2 = 4 \qquad g(1) = 1^2 = 1 \qquad //f(1)>g(1)$

Let n = 2
$F(2) = (2*2) + 2 = 6 \qquad g(2) = 2^2 = 4 \qquad //f(2)>g(2)$

Let n = 3
$F(3) = (2*3) + 2 = 8 \qquad g(3) = 3^2 = 9 \qquad //f(3)<g(3)$

Let n = 4
F(4) = (2*4) + 2 = 10 g(4) = 4^2 = 16 //f(4)<g(4)

Hence, f(n) = O(g(n)), where **n>=3 and** g(n) is bounding above when n>=3

2. **Omega Ω Notation:**
   It denotes the lower bound of an algorithm i.e. the time taken by the algorithm cannot be lower than this.
   In other words, this is the fastest time taken algorithm when provided with best case input.
   A function f(n) is said to be in Ω(g(n)) denoted by f(n) = Ωg(n)), if f(n) is bounded below by some constant multiple of g(n) for all large 'n'.
   i.e. if there exists some positive constant 'c' and some non-negative integer n_0 such that

$$f(n) >= c*g(n) \text{ for all } n>=n_0$$

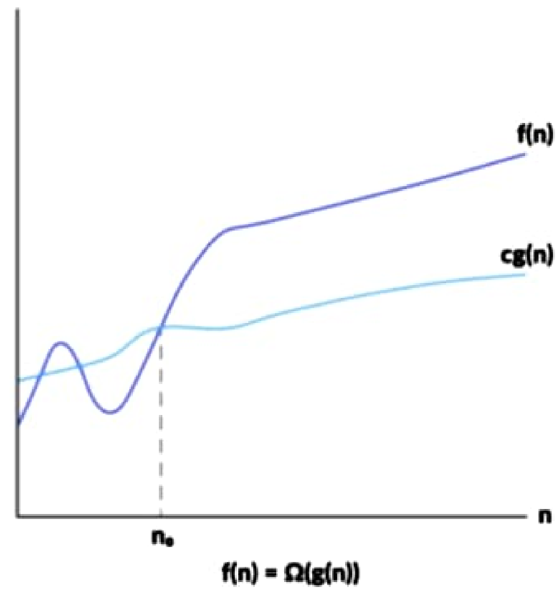

Fig. 1.5 Omega Ω Notation

Example:

F(n) = $2n^2 + 3$ g(n) = 7n
Note: f(n) should always be greater than or equal to g(n)

Let n = 1
F(1) = 2 * (1^2) + 3 = 5 g(1) = 7 * 1 = 7 //f(1)<g(1)

Let n = 2
F(2) = 2 * (2^2) + 3 = 11 g(2) = 7 * 2 = 14 //f(2)<g(2)

Let n = 3
F(3) = 2 * (2^3) + 3 = 19 g(3) = 7 * 3 = 21 //f(3)<g(3)

Let n = 4
F(4) = 2 * (2^4) + 3 = 35 g(4) = 7 * 4 = 28 //f(4)>g(4)

Hence, f(n) = Ω(g(n)) for all n>=3

3. **Theta Θ Notation:**
A function f(n) is said to be Θ(g(n)), denoted as f(n) = Θ(g(n)), if f(n) is bounded both above and below by some positive constant multiples of g(n) for all large 'n'.
Θ is used to find out the average bound of an algorithm i.e. it defines upper bound and lower bound and your algorithm will lie between these levels.

$$C_2 * g(n) <= f(n) <= c_1 * g(n) \text{ for all } n>=n_0$$

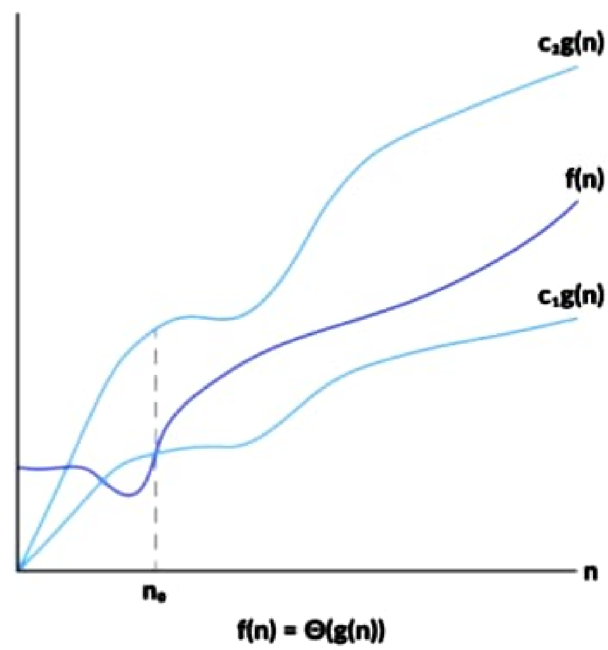

Fig. 1.6 Theta Θ Notation

Example:

F(n) = 2n+8 g(n)=n

Assume c_1 = 8 and c_2 = 2

Let n=1

F(1) = (2 * 1) + 8 = 10 ; g(1) = 1
C_1 * g(1) = 8*1 = 8
C_2 * g(1) = 2*1 = 2

C_2 * g(n) <= f(n) <= c_1 * g(n)
2 <= 10 <=8 //not satisfied

Let n=2

F(2) = (2*2) + 8 = 12 g(2) = 2
C_1 * g(2) = 8*2 = 16
C_2 * g(2) = 2*2 = 4

C_2 * g(n) <= f(n) <= c_1 * g(n)
4 <= 12 <=16 //satisfied

Therefore, f(n) = Θ(g(n)) for n>=2

Let's see all different time complexities in one graph:

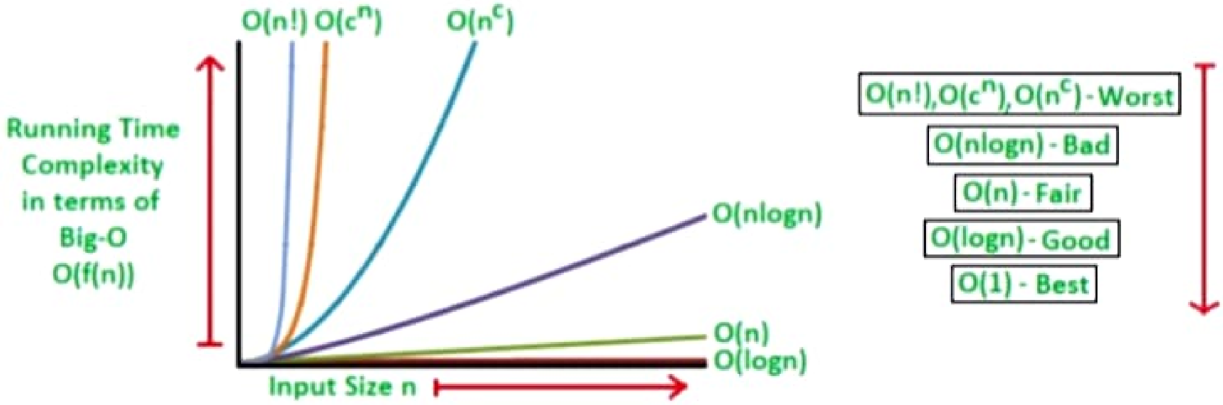

Fig. 1.7 Time complexity graph

Common Asymptotic Notations:

Following is the list of some asymptotic notations:

Sr. No	Time Complexity	Notation	Examples
1.	Constant	O(1)	Accessing a value with an array index int a[]={1,2,3,4,5} printf("%d", a[2]); Output: 3
2.	Logarithmic	O(log n)	Binary Search
3.	Linear	O(n)	Linear Search
4.	Linearithmic	O (n log n)	Merge sort
5.	Quadratic	$O(n^2)$	Bubble sort (2 for loops)
6.	Cubic	$O(n^3)$	Matrix update (3 for loops)
7.	Exponential	$O(2^n)$	Brute Force algorithm
8.	Factorial time	O(n!)	Travelling Salesman Problem (TSP)

3.2.2 Space Complexity:

Space complexity is nothing but the amount of memory space that an algorithm or a problem takes during execution of that particular problem.
Space needed by an algorithm is given by:

$S(P) = C$ (fixed part) + S_p (variable part)

Fixed part: is independent of instance characteristic. i.e. space for simple variables, constants, etc (int a; int b)

Variable part: is space for variables whose size is dependent on particular problem instance.
Example: array

Example:

i. Algorithm max(A, n)

```
{
    Result = A[1];
    For i = 2 to n do
    If A[i]>result then
            Result = A[i];
    Return = A[i];
}
```

Answer:
Variables i, n, result = 1 unit each // 4 bytes each, which is fixed, can't be changed
Variable A = n units // n * 4 bytes, completely depends on 'n'
Total = n + 4
Removing constant value:
Space complexity: O(n)

ii. Algorithm abc(d, e, f)

```
{
    Return d + e * f + (d + e + f)/(d + e) + 4.0
}
```

Answer:
d: 4 bytes, fixed
e: 4 bytes, fixed
f: 4 bytes, fixed
total: 12 bytes at least
removing constant values:
space complexity: O(1), constant space

iii. Algorithm sum(a, n)

```
{
    s=0.0;
for i=1 to n do
    s=s+a[i];
return s;
}
```

Answer:
Variables i, n, s = 4 bytes each i.e. (4 + 4 + 4 = 12)// fixed part
Variable a: n * 4 bytes // variable part
Total =(4*n+12) bytes
Removing constant values:
Total = 4*n
Space complexity: O(n)