example 1

```
for (i=0; i<N; i++)
    for (j= 0; j<N; j++)
        k++;
```

$O(N^2)$

example-2

```
for (i=0; i<N; i++)
    A[i] = 0;
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        A[i] += A[j] + i + j;
```

$O(N)$

$O(N^2)$

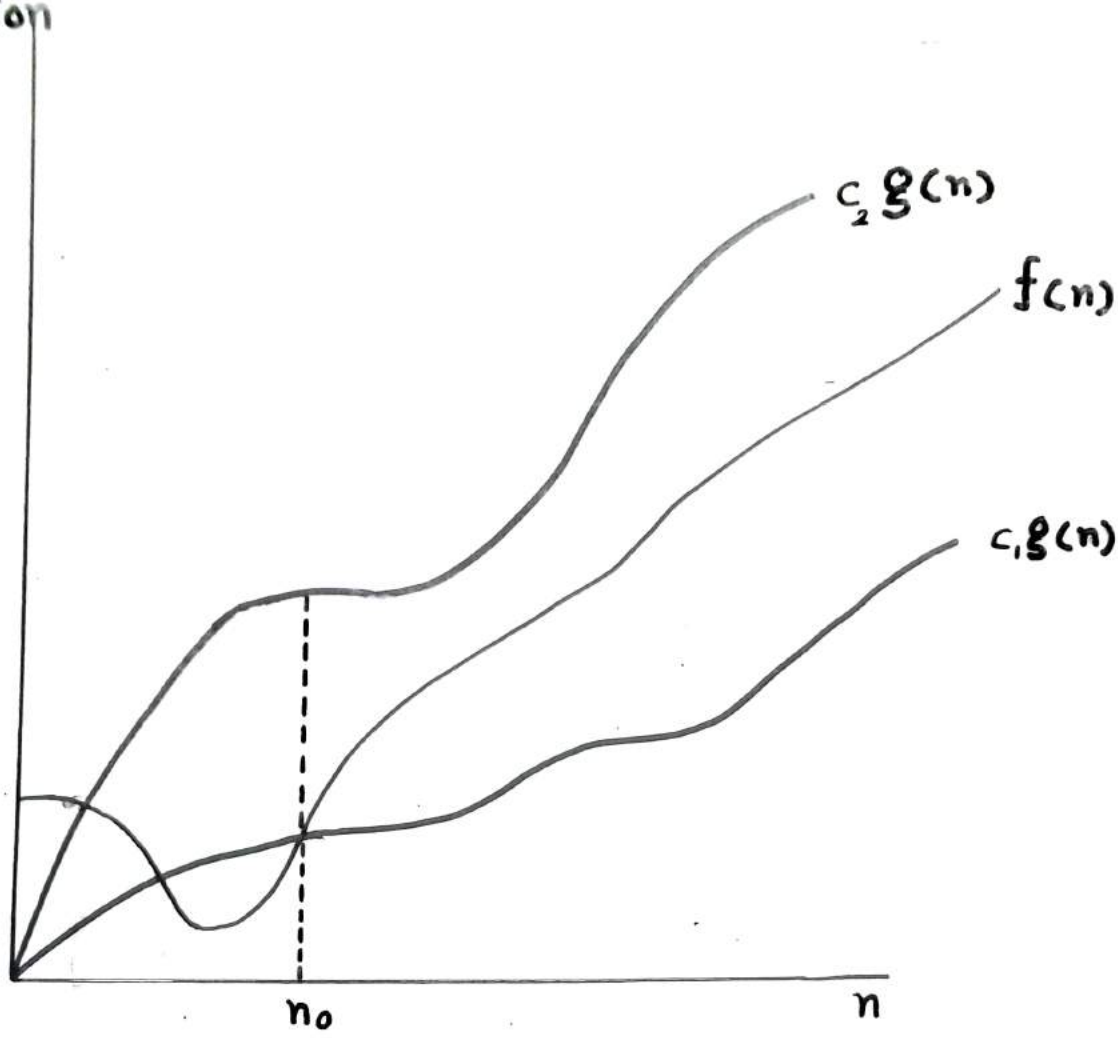
example: 3

```
long int
Factorial (int N)
{
    if N <= 1
        return 1;
    else
        return N * Factorial (N-1);
}
```

example: 4.

```
int MaxSubSqnSum ( const int A[], int N)
{
    int ThisSum, MaxSum, i, j, k;
    MaxSum = 0;
    for ( i=0; i<N; i++)
        for ( j= i; j<N; j++)
        {
            ThisSum =0;
            for ( k = i; k <=j ; k++)
                ThisSum += A[k];
            If ThisSum > MaxSum
                MaxSum = ThisSum;
        }
    return MaxSum;
}
```

Θ notation



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

$\Theta(g(n))$ the set of Functions

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \; \forall \; n \geq n_0 \}$

Θ notation bounds a function to within constant factors

The Theta Notation

$t(n)$ is in Theta of $f(n)$, or equivalently that $t(n)$ is in the exact order of $f(n)$, denoted as $t(n) \in \Theta(f(n))$ if $t(n)$ belongs to both $O(f(n))$ and $\Omega(f(n))$

i.e. $$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

OR

It can be stated as $\Theta(f(n))$ is

$$\{t: N \to R^{\geq 0} \mid (\exists c, d \in R^+)(\overset{\infty}{\forall} n \in N) \; [df(n) \leq t(n) \leq cf(n)]\}$$

Limit Rule for Θ notation

Let f and $g: N \longrightarrow R^{\geq 0}$ then

1. if $\lim_{n \to \infty} f(n)/g(n) \in R^+$ then $f(n) \in \Theta(g(n))$

2. if $\lim_{n \to \infty} f(n)/g(n) = 0$ then $f(n) \in O(g(n))$
   but $f(n) \notin \Theta(g(n))$

3. if $\lim_{n \to \infty} f(n)/g(n) = +\infty$ then $f(n) \notin \Omega(g(n))$

Asymptotic notation in equations

CASE:1

Example: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

Stands for some anonymous function that cannot be named in particular (not care to be named)

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that

$2n^2 + 3n + 1 = 2n^2 + f(n)$

Where $f(n)$ is some function in the set $\Theta(n)$

so $f(n) = 3n + 1$ which is in $\Theta(n)$

Note:

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears

$\sum_{i=1}^{n} O(i)$ $\Rightarrow$ There is a single anonymous function (a function of i)

CASE:2

$2n^2 + \Theta(n) = \Theta(n^2)$

Rules to interpret above equation

* NO matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.

For any function $f(n) \in \Theta(n)$ there is some function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n.

OR

The right hand side of an equation provides coarser level of detail than the left-hand side.

Rules for Running Time Calculation

FOR LOOPS:

Rule-1: The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

NESTED FOR LOOPS:

Rule-2: The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the size of all the for loops.

CONSECUTIVE STATEMENTS:

Rule-3: These just add (This means that the maximum is the one that counts.

IF/ELSE:

Rule-4: For the fragment of the type

```
if (condition)
    S1
else
    S2.
```

The running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S_1 and S_2

Rule 1. If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ then

(a) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$

(b) $T_1(N) * T_2(N) = O(f(N) * g(N))$

Rule 2. If $T(N)$ is a polynomial of degree k then $T(N) = \Theta(N^k)$

Rule 3. $\log^k N = O(N)$ for any constant k. This tells us that logarithms grow very slowly.

Function	Name
C	Constant
log N	Logarithmic
$\log^2 N$	Log squared
N	Linear
N log N	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Typical growth rate.

Running Time Calculations

Program fragment to calculate $\sum_{i=1}^{N} i^3$

```
int
Sum ( int N)
{
   int i, partialsum;          <- no time
   Partialsum = 0;             <- one unit
   (for i=1; i <= N; i++)      (2N+2)
   Partialsum += i * i * i;    <- 4 unit per time executed
   return partialsum;          <- one unit
}
```

The cost of calling the function & returning

$$6N + 4$$

$\Rightarrow$ The function / Algorithm is $O(N)$

Asymptotic Notation (O, Ω, θ)

These notations are used to make meaningful (but inexact) statements about the time complexity and space complexity.

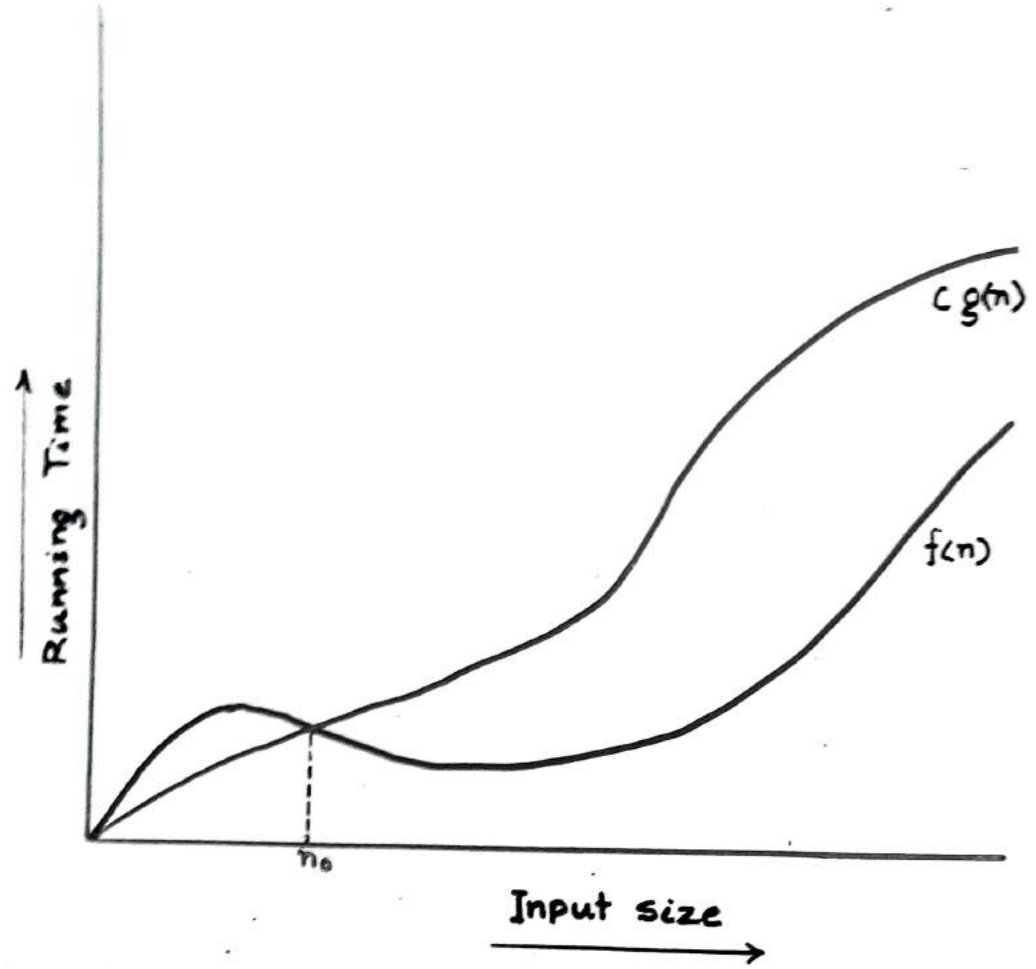
Definition: Big "Oh": Let f and g are non-negative functions. Then the function $f(n) = O(g(n))$ iff there exist positive constant c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$

* The statement $f(n) = O\ g(n)$ states that $g(n)$ is an upper bound on the value of $f(n)$ $\forall$ n; $n \geq n_0$
* $f(n) = O\ g(n)$ is not same as $O(g(n)) = f(n)$
* The "Big Oh" notation does not say anything about how good this bound is.

Definition: [Omega]: Ω The function $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$.

* The function $g(n)$ is only a lower bound on $f(n)$
* $g(n)$ should be as large a function of n as possible for which $f(n) = \Omega(g(n))$

The Big-Oh Notation



Theorem: Let $d(n)$, $e(n)$, $f(n)$ and $g(n)$ be functions mapping nonnegative integers to non-negative reals. Then

1. If $d(n)$ is $O(f(n))$
   then
   $ad(n)$ is $O(f(n))$ for any constant $a > 0$
2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$
   then $d(n) + e(n)$ is $O(f(n) + g(n))$
3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$
   then $d(n)\, e(n)$ is $O(f(n)\, g(n))$

Big "Oh"

* $3n + 2 = O(n)$

  as $3n + 2 \leq 4n$

  for all $n \geq 2$

* $3n + 3 = O(n)$

  as $3n + 3 \leq 4n$

  for all $n \geq 3$

* $100n + 6 = O(n)$

  as $100n + 6 \leq 101n$

  $\forall\ n \geq 6$

* $10n^2 + 4n + 2 = O(n^2)$

  $10n^2 + 4n + 2 \leq 11n^2$

  $\forall\ n \geq 5$

* $1000n^2 + 100n - 6 = O(n^2)$

  $1000n^2 + 100n - 6 \leq 1001n^2$

  $\forall\ n \geq 100$

* $6 * 2^n + n^2 = O(2^n)$

  $6 * 2^n + n^2 \leq 7 * 2^n \quad \forall\ n \geq 4$

* $10n^2 + 4n + 2 = O(n^4)$

  $10n^2 + 4n + 2 \leq 10n^4$

  for $n \geq 2$

Order of Magnitude

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

$\Rightarrow$ $f(n)$ does not grow more rapidly than $g(n)$ as n grows

$f(n)$ is bounded above by some constant times $g(n)$; so $g(n)$ can be used as crude estimate of $f(n)$.

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

is equivalent to $\lim_{n \to \infty} \left| f(n) \right| / \left| g(n) \right| = c$

Where c is the constant

NOTE:

An algorithm A is said to be of polynomial complexity if $f(n)$ of A does not grow more rapidly than a polynomial of n.

If this is not the case, the algorithm is said to be of exponential complexity

Seven orders of magnitude are generally used for time complexity

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

$O(1)$ means that the algorithm is a constant time algorithm and the computation time does not depend on n.

Most powerful and versatile tool both to prove some functions are in the order of others

Limit Rule:

f and g: $\mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$

Natural numbers to non-negative reals

1. If $\lim_{n\to\infty} f(n)/g(n) \in \mathbb{R}^+$ then

$f(n) \in O(g(n))$ and

$g(n) \in O(f(n))$

2. if $\lim_{n\to\infty} f(n)/g(n) = 0$ then

$f(n) \in O(g(n))$

but $g(n) \notin O(f(n))$

3. if $\lim_{n\to\infty} f(n)/g(n) = +\infty$ then

$f(n) \notin O(g(n))$

but $g(n) \in O(f(n))$

Example: $f(n) = \log n$ $\quad g(n) = \sqrt{n}$

$\lim_{n\to\infty} f(n) \to \infty$ $\quad \lim_{n\to\infty} g(n) \to \infty$? $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$

using L'Hopital's Rule $= \lim_{n\to\infty} 1/n \Big/ 1/(2\sqrt{n}) = \lim_{n\to\infty} 2/\sqrt{n} = 0$

Notation "the order of"

n^2 is in the order of n^3 $\quad n^2 \in O(n^3)$

$n^2 = O(n^3)$ ✓

$O(n^3) = n^2$ ✗

Threshold rule:

Let $f, g : N \longrightarrow R^+$

Two functions from the natural numbers to strictly positive real numbers

Threshold rule states that

$f(n) \in O(g(n))$ if and only if there exists a positive real constant c such that $f(n) \leq c\, g(n)$ for each natural number

Maximum Rule:

Useful tool for proving that one function in the order of another

Let f and $g : N \longrightarrow R^{\geq 0}$ be two arbitrary functions from natural numbers to nonnegative reals.

By maximum Rule $\quad O(f(n) + g(n))$

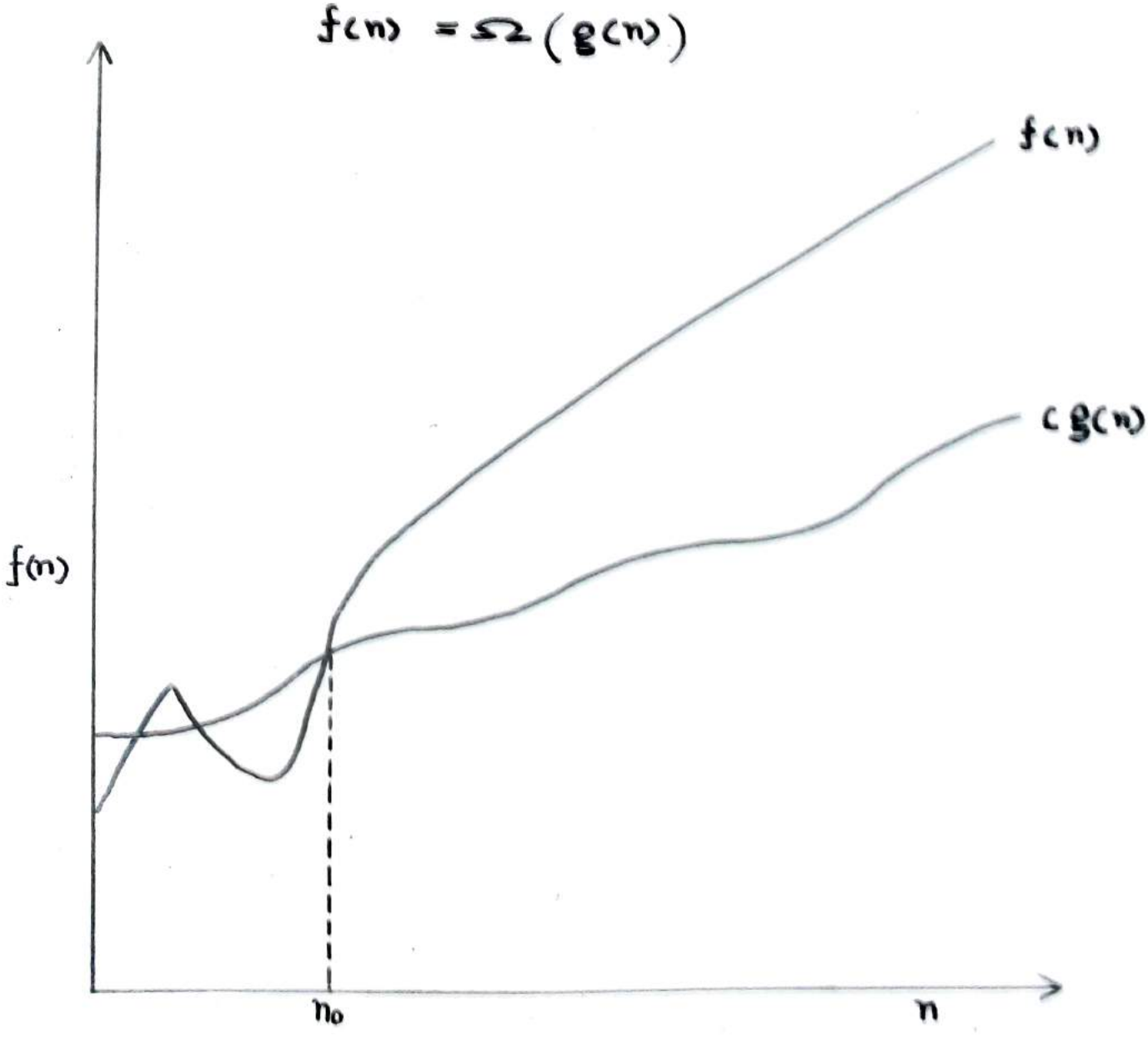
$$= O(\max(f(n), g(n)))$$

Even though the time taken by an algorithm is logically the sum of time taken by its disjoint parts, it is the order of the time taken by its most time consuming part, provided number of parts is constant, independent of input size.

$O(n^2 + n^3 + n \log n) = O(\max(n^2, n^3, n \log n)) = O(n^3)$

Asymptotic Lower Bound:

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0 \}$



Example: Omega

$3n+2 = \Omega(n)$

$3n+2 \geq 3n$ for $n \geq 1$

$3n+3 = \Omega(n)$

as $3n+3 \geq 3n$ for $n \geq 1$

$100n+6 = \Omega(n)$

as $100n+6 \geq 100n$ for $n \geq 1$

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

as $10n^2+4n+2 \geq n^2$ for $n \geq 1$

$6 * 2^n + n^2 = \Omega(2^n)$

as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$.

$3n+3 = \Omega(1)$

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

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

$6 * 2^n + n^2 = \Omega(n^{100})$

$6 * 2^n + n^2 = \Omega(n^2)$

$6 * 2^n + n^2 = \Omega(n)$