DATE _____

Tutorial-1

Asymptotic Notation:- It is used to describe the running time of an algorithm - how much time can algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω).

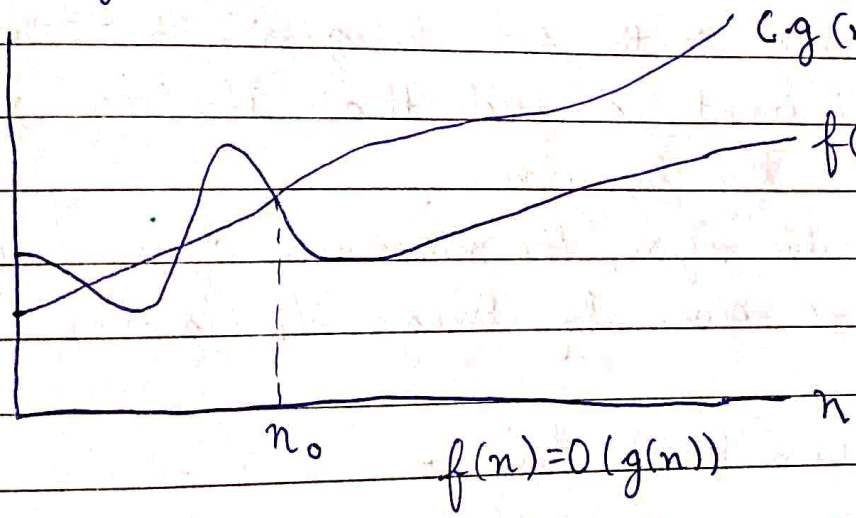
Big-Θ is used when the running time is the same for all cases.

Big-O is used for the worst case running time.

Big-Ω for the best case running time.

Big-O Notation (O-Notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



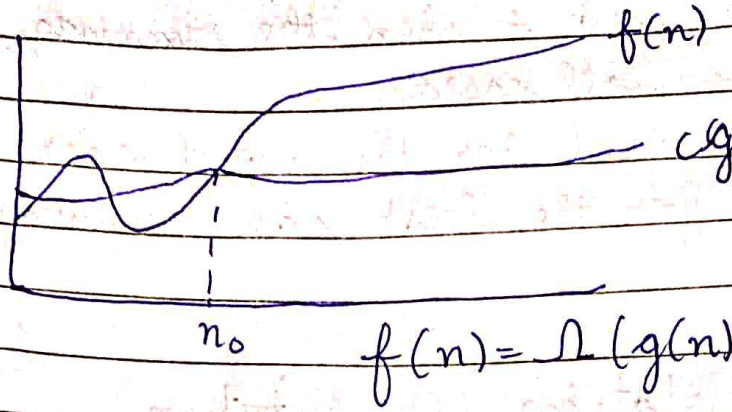
$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0$ 3

The above expression can be described as a function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies

between 0 and $c(g(n))$ for sufficiently large n.

Omega Notation (Ω - Notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Omega gives the lower bound of a function.

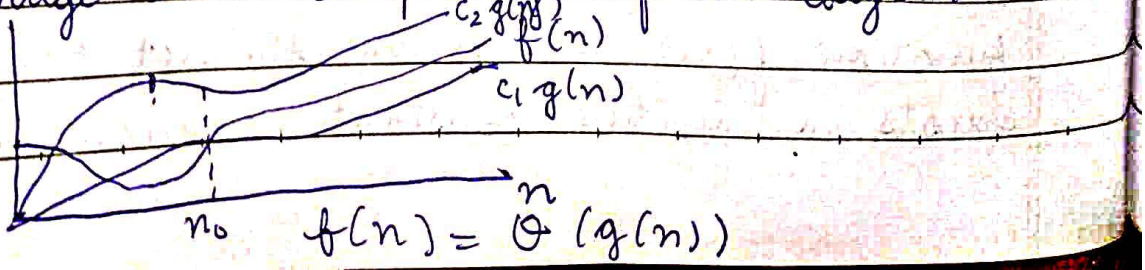
$\Omega(g(n)) = \{f(n)$: there exist positive constants c & n_0 such that $0 \leq c\,g(n) \leq f(n)$ for all $n \geq n_0\}$

The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $c\,g(n)$, for sufficiently large n.

⇒ For any value of n, the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

Theta Notation (Θ - Notation) -

Theta notation encloses the function from above & below. Since it represents the upper & the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



PAGE _____
DATE _____

2. for (i=1 to n)
{
i = i * 2;
}

for(i=1 to n) // i = 1, 2, 4, 8 ... n
{ i = i * 2 } // O(1).

$\Rightarrow \sum_{i=1} 1+2+4+8+\ldots+n$

K^{th} term of GP $\Rightarrow T_k = ar^{k-1}$

$n = 1 * 2^{k-1}$

$n = 2^{k-1}$

$n = \frac{2^k}{2}$

$2n = 2^k$

$\log_2(2n) = k(\log_2 2)$

$k = \log_2(2n)$

$k = \log_2 2 + \log_2 n$

$k = 1 + \log_2 n$

$O(\log_2 n)$

~~$O(n)$~~

3. $T(n) = \{3T(n-1)$ if $n > 0$, otherwise 1 $\}$

Solve using substitution

$T(n) = 3T(n-1)$

$= 3(3T(n-2))$

$= 3^2 T(n-2)$

$= 3^3 T(n-3)$

...

...

$3^n T(n-n)$

$3^n T(0)$

3^n

This clearly shows that the complexity of this function is $O(3^n)$

4. $T(n) = \{2T(n-1)-1$ if $n > 0$, otherwise 1 $\}$

Solving using substitution

$T(n) = 2T(n-1) - 1$

$= 2(2T(n-2)-1)-1$

$= 2^2(T(n-2))-2-1$

$= 2^2(2T(n-3)-1)-2-1$

$= 2^3 T(n-3) - 2^2 - 2^1 - 2^0$

...

...

$= 2^n T(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^2 - 2^1 - 2^0$

$= 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^2 - 2^1 - 2^0$

$= 2^n - (2^n - 1)$

[Note: $2^{n-1} + 2^{n-2} + \ldots + 2^0 = 2^n - 1$]

$T(n) = 1$

Time complexity is $O(1)$.

5.
```
int i=1, s=1;
while (s<=n){
i++;
s = s+i;
printf("#");
}
```

$s_i = s_{i-1} + i$

value of 'i' increases by 1 for each iteration.

Value contained in 's' at i^{th} iteration

$1+2+3 \ldots + k = \frac{k(k+1)}{2} > n$

$$k = O(\sqrt{n})$$

Time complexity: $O(\sqrt{n})$

6.
```
void function (int n) {
int i, count=0;
for (i=1; i*i<=n; i++)
count++;
}
```

As $i^2 <= n$

$i <= \sqrt{n}$

$i = 1, 2, 3, 4, \ldots \sqrt{n}$

$\sum_{i=1}^{n} 1+2+3+\ldots+\sqrt{n}$

PAGE

DATE

7. void function(int n){
int i, j, k, count = 0;
for(i = n/2; i<=n; i++) — n
for(j = 1; j<=n; j = j*2) — log n
for(k = 1; k<=n; k = k*2) — log n
count++
}

$O(n \log^2 n)$

8. function(int n) {
if(n == 1) return;
for(i = 1 to n) { — n
for(j = 1 to n) { — n
printf("*");
} }
function(n-3); — $T(2^n)$
}

$O(n^2 2^n)$.

9. void function(int n){
for(i = 1 to n) {— n
for(j = 1; j<=n; j = j+i) — n
printf("*")
}
}

$O(n^2)$

DATE _____

10. For the functions, n^k and c^n, what is the asymptotic relationship between these functions? Assume that $k >= 1$ and $c > 1$ are constants. Find out the value of c and n0 for which relation holds.

n^k is $O(c^n)$