Asymtotic Notation

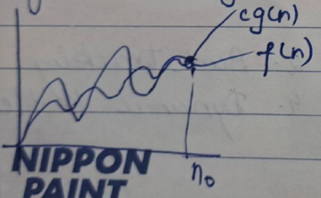
↓
tending to ∞

→ These notation are used to tell the complexity of algorithm when the input is very large.

1) Big O Notation (O)

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

g(n) is 'tight' upper bond of f(n)



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

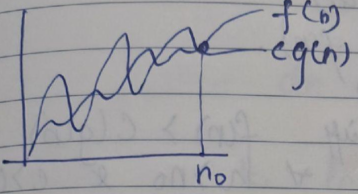
iff $f(n) \leq c\,g(n)$

$\forall n \geq n_0$ & some const. $c > 0$

2) Big Omega Notation (Ω)

$$f(n) = \Omega\, g(n)$$

g(n) is 'tight' lower bond of f(n).



$f(n) = \Omega\, g(n)$

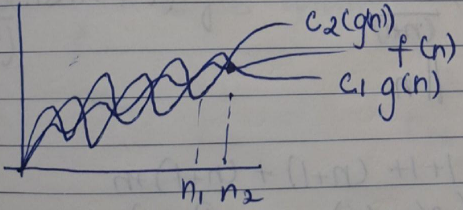
iff $f(n) \geq c\,g(n)$

$\forall n \geq n_0$ & some const $c > 0$.

3) Theta Notation (Θ)

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

It gives both 'tight' upper & 'tight' lower bound.



$f(n) = O(g(n))$ &
$f(n) = \Omega(g(n))$

iff $c_1 g(n) \leq f(n) < c_2 g(n)$

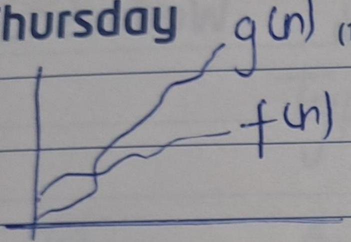
lower upper

$\forall n > \max(n_1, n_2)$ & const. $c_1, c_2 > 0$.

4) Small o Notation (o)

$$f(n) = o(g(n))$$

g(n) is upper bound of f(n)
not asymtotically tight.



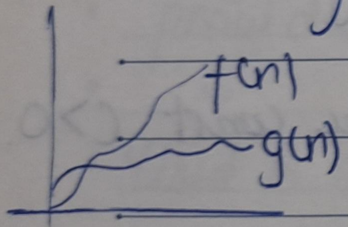
$$0 \leq f(n) \leq c\,g(n)$$

$\forall n > n_0$.

& const $c > 0$.

5) Small Omega Notation (ω)

g(n) is lower bound of f(n).



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

iff $f(n) > c(g(n))$

$\forall n > n_0$ & $c > 0$.

Date: / /

Q2 for (i=1, i<=n; i=i*2)

$n = 1 * 2^{k-1}$ $\qquad t_k = a r^{k-1}$

$n = 2^{k-1}$

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

$\Rightarrow 2n = 2^k$

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

$k = \log_2 2n$

$= \log_2 2 + \log_2 n$

$k = 1 + \log_2 n$

$\Rightarrow O(\log_2 n)$

Q3 $T(n) = \{3T(n-1)$ if $n>0$ otherwise $1\}$

$T(0) = 1$

$n=1 \Rightarrow T(1) = 3T(0) = 3$

$n=2 \Rightarrow T(2) = 3T(1) = 3^2$

$n=3 \Rightarrow T(3) = 3T(2) = 3 \cdot 3^2 = 3^3$

$n=k \Rightarrow T(k) = 3^k$

$T(n) = 3^n$.

Time complexity $= O(3^n)$

Q4

Q4. $T(n) = \{ 2T(n-1) - 1$, if $n > 0$, otherwise $1 \}$

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

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

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

where $n - k = 0$

$k = n$

Best case: $T(0) = 1$

$\because n - k = 0 \Rightarrow n = k$

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

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

$= 1$

Time comp. : $O(1)$

Q5
```
int i=1, s=1
while (s<=n) {
  i++;
  s = s+i;
  printf("#");
}
```

$i = 1, 2, 3 \ldots n$

$s = 1, 3, 6, 10, 15 \ldots n$

$s = \frac{i(i+1)}{2}$

$\frac{i(i+1)}{2} \leq n \Rightarrow i(i+1) \leq 2n$

$i^2 + i - 2n \leq 0$

$\therefore i < \frac{-1 + \sqrt{1+8n}}{2}$

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

Q6
```
void f^n (int n) {
  int i, count = 0;
  for (i = 1; i * i <= n; i++)
      count ++;
}
```

$\Rightarrow$ $i = 1, 2, 3, 4 \text{ — } n$
$i^2 = 1, 4, 9, 16 \text{ — } n^2$

time comp. $\rightarrow$ $O\sqrt{n}$

Q7
```
void f^n (int n) {
  int i, j, k, c = 0;
  for (i = n/2 ; i <= n ; i++)
  for (j = 1 ; j < n ; j = j + 2)
  for (k = 1; k <= n ; k = k * 2)
      c++;
}
```

$T(n) = n/2 \cdot \log 2(n) + \log 2(n)$

$\Rightarrow O(n^* \log^2(n))$

Q8
```
f^n (int n) {                    - T(n)
  if (n == 1) return;
  for (i = 1 to n)               - n
  {
    for (j = 1 to n)             - n^2
    {
      print ("*");               - n^2
    }
  }
  f^n (n-3);                     - T(n-3)
}
```

$T(n) = O(n^2) + T(n-3)$ $\Rightarrow O(n^3)$

Date: /

Q9 void f^n (int n)

{ for (i = 1 to n)

{ for (j = 1 ; j <= n ; j = j + i)

{ print ("*")

} } }

$i = 1, 2, 3, 4$ — $j = 1, 3, 6, 10$ —

$i = 1$ n times

$i = 2$ $n/2$ times

$i = 3$ $n/3$ times

$$n + \frac{n}{2} + \frac{n}{3} + \text{———} + 1.$$

time comp. = $O(n \log n)$