Name → Kavya Bansal

Sec → DS 1

Roll No. → 63

Date.

Page No.

1 → Asymptotic notations are mathematical notations used to describe the limiting behaviour of function as the input size approaches infinity.

The three main asymptotic notations are :-

(i) Big O Notation ⇒ Describes the upper bound or worst-case scenario of an algorithm's runtime or space complexity.

example ⇒ An algorithm with a time complexity of $O(n^2)$ means that its runtime grows quadratically as the input size n increases.

(ii) Omega notation (Ω) ⇒ Describes the lower bound or best-case scenario of an algorithm's runtime or space complexity. It represents minimum growth rate of a function.

example ⇒ An algorithm with time complexity of $\Omega(n)$ means that the algorithm's runtime grows linearly at least as the input size (n) increases.

(iii) Theta notation (Θ) ⇒ Describes both the upper and lower bounds of an algorithm's runtime. Represents the tightest possible growth rate of the function.

example ⇒ An algorithm has time complexity of $\Theta(n)$ means that the algorithm's runtime grows linearly as the input size (n) increases and it is bounded both from above and below by a linear function.

Date.
Page No.

2 → for (int i=0 ; i<n; i*=2) $t_K = a r^{K-1}$
{ $n = (2)^{K-1}$

———

}

Taking $\log_2$ both sides
$\log_2 n = (K-1)\log_2 2 \Rightarrow \log_2 n = (K-1)$
$K = \log_2 n + 1$

Time complexity ⇒ $O(\log_2 n)$

3 ⇒ $T(n) = \{3T(n-1)$ if $n>0$, otherwise $1\}$
$T(0) = 1$
Substituting $n=1 \Rightarrow T(1) = 3T(0) = 3$
substituting $n=2 \Rightarrow T(2) = 3T(1) = 3^2$
Substituting $n=3 \Rightarrow T(3) = 3T(2) = 3\cdot 3^2 = 3^3$
substituting $n=K \Rightarrow T(K) = 3^K$
$T(n) = 3^n$
Time complexity ⇒ $O(3^K)$.

4 ⇒ $T(n) = \{2T(n-1)-1$ if $n>0$, otherwise $1\}$
$T(0) = 1$
substituting $n=1 \Rightarrow T(1) = 2T(0)-1 = 1-1 = 0$
substituting $n=2 \Rightarrow T(2) = 2T(1)-1 = -1$
Substituting $n=3 \Rightarrow T(3) = 2T(2)-1 = -3$
$n=4 \Rightarrow T(4) = 2T(3)-1 =$
$T(n) = 2T(n-1)-1 = 2(2T(n-2)-1)-1 = 2^2 T(n-2)-2-1$
$= 2^3 T(n-3) - 2^2 - 2 - 1$ — — —
$= 2^n T(n-n) - (2^{n-1} + 2^{n-2} + \text{——} 2^0)$
$= 2^n - n - 1$
Time complexity ⇒ $O(2^n)$

Date.
Page No.

5⇒

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

$i = 1 \quad 2 \quad 3 \quad 4 \quad 5$

$s = 1 \quad 3 \quad 6 \quad 10 \quad 15$ $\Rightarrow$ $s = \frac{i(i+1)}{2}$

$t_k = \frac{k(k+1)}{2} \Rightarrow n = \frac{k(k+1)}{2} \Rightarrow k^2 + k - 2n = 0$

$k = \frac{-1 \pm \sqrt{1+8n}}{2} = \frac{-1 + \sqrt{1+8n}}{2}$

Time complexity ⇒ $O(\sqrt{n})$

6⇒

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

$i = 1 \quad 2 \quad 3 \quad 4$ —

$i^2 = 1 \quad 4 \quad 9 \quad 16$ —

$t_k = k^2$

$n = k^2 \Rightarrow k = \sqrt{n}$

Time complexity → $O(\sqrt{n})$

Date.
Page No.

7→ void function (int n)
{
int i, j, k, count = 0;
for (i = n/2; i <= n; i++) → O(n/2)
{
for (j = 1; j <= n; j = j * 2) → O(log n)
{
for (k = 1; k <= n; k = k * 2) } → $O(\log_2 n)$
{
count ++
}
}
}
}

~~O(n/2) x O(log n)~~ Time complexity = $O\left(\frac{n}{2} \times \log n \times \log n\right) = O\left(\frac{n}{2}(\log n)^2\right)$

Time complexity = $O\left(n(\log_2 n)^2\right)$

8 → function (int n)
{
if (n==1) return;
for (int i=1; i<n; i++) → O(n)
{
for (j=1; j<n; j++) → O(n)
{
printf(" ");
}
}
function (n-3); → T(n-3)
}

$T(n) = O(n \times n) * T(n-3)$
$T(n) = O(n^3)$

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

i	1	2	3	4	5	→ n times
j	1	3	6	10	15	→ $\left(n + \frac{n}{2} + \frac{n}{3} + \text{——} 1\right)$ times
	n times	n/2	n/3	n/4		

Date. ____________

Page No. ____________

Time complexity = $O(n \log n)$

10 ⇒ n ^ k   k >= 1   c ^ n   c > 1

c^n grows faster than n^k.

11 → The time complexity for extractMin(), given a min heap of n nodes is $O(\log n)$. This is because extractMin() removes the root node which is the minimum element and then calls heapify() to restore the heap property. Heapify() takes $O(\log n)$ time as it traverses the height of the heap which is logn.

12 ⇒