Q1) Asymptotic Notations: Are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value

OR

These notations are used to tell the complexity of an algorithm when input is very large.

① Big O Notation: We compute the big O of an algorithm by counting the number of iterations the algorithm always takes the input of n.

② Big O Notation: The big O notation describes the worst-case running time of a program. We compute the big-O of an algorithm by counting how many iterations an algorithm will take in the worst case scenario with an input of N. For example, O(log n) describes the Big-O of a binary search algorithm.

③ Big Ω Notation: Big Ω (omega) notation describes the best case running time of a program. We compute the big-Ω by counting how many iterations an algorithm will take in the best-case scenario based on an input of N. For example, a Bubble sort algorithm has a running time of Ω(N) because in the best case scenario the list is already sorted, & the bubble sort will terminate after the first iteration.

DATE / /
PAGE NO.

(5) Small O Notation: It is used to describe an upper bound that cannot be tight. In other words, loose upper bound of $f(n)$

(6) Small Omega Notation: Commonly written as ω, is an asymptotic notation to denote the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

(12) for (i=1 to N) { i = i * 2; }

$i = 1, 2, 4 \ldots$

$i = \underbrace{2^0, 2^1, 2^2 \ldots 2^k}_{K \text{ terms}}$

This forms a GP, where

$a = 1, \quad r = \frac{a_2}{a_1} = \frac{2}{1} = 2$

$t_k = ar^{k-1}$

$t_k = 1 \times 2^{k-1}$

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

$2^n = 2^k$

Taking log on both side

$\log_2 2 + \log_2 n = k \log_2 2$

$k = 1 + \log_2 n$

Time Complexity $O(\log_2 n)$

$$\sum_{i=1 (i=i*2)}^{N} (1 + 1 + 1 \ldots \log_2 n)$$

$= O(\log_2 n)$

(3) $T(n) = 3T(n-1)$ — (1)

put $n = n-1$ in (1)

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

put $n = n-2$ in (1)

$T(n-2) = 3T(n-3)$ — (3)

$$Tn = \begin{cases} 1 & n=0 \\ 3T(n-1)-n & >0 \end{cases}$$

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

$T(n) = 9T(n-2)$ —

Substitute (3) in (2)

$T(n) = 27\,T(n-3)$

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

put assume $n-k = 0$

$k = n-0$

$T(n) = 3^{n-0}\,T(0) = \frac{3^n}{3^0}$

$O(3^n)$

(4)

$$T_n \begin{cases} 1 & n=0 \\ 2T(n-1)-1 & n>0 \end{cases}$$

$T(n) = 2T(n-1)-1$ — (1)

put $n = n-1$ in (1)

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

$T(n) = 4T(n-2)-2-1$ — (2)

put $n = n-2$ in (1)

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

Substitute in (2)

$T(n) = 4(2T(n-3)-1)-2-1$

$T(n) = 8\,T(n-3) - 4-2-1$

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

$n - k = 0$

$n = k$

$= 2^n T(0) - \frac{1 \times (2^k - 1)}{2-1}$

$= 2^n - 2^k + 1$

$T(n) = 2^n - 2^n + 1$ $\quad O(1)$

(5) int i=1, s=1;
while (s <= n)
{ i++; —— O(1)
st = s+i; —— O(1)
print ("#"); —— O(1)
}

s = 1, 3, 6, 10, 15 . . .

First difference is in AP

$T_k = Ak^2 + Bk + C$

putting $k = 1$

$A + B + C = 1$ — (1)

putting $k = 2$

$4A + 2B + C = 3$ — (2)

putting $k = 3$

$9A + 3B + C = 6$ — (3)

Solving (1) (2) & (3)

$A = \frac{1}{2}$ $\quad B = \frac{1}{2}$ $\quad C = 0$

$$T(n) = \frac{k^2}{2} + \frac{k}{2} = \frac{k(k+1)}{2}$$

$$n < \frac{k(k+1)}{2}$$

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

(6)

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

```
for (i=1 ; i*i <= n; i++)
     O(1)
```

values of i

$\underbrace{1, 4, 9, 16 \ldots\ldots (\sqrt{n})^2}_{k}$

First difference of this series forms AP

$$Ak^2 + Bk + C = t_k$$

put $k=1$ $\quad A + B + C = 1$ — (1)

put $k=2$ $\quad 4A + 2B + C = 4$ — (2)

put $k=3$ $\quad 9A + 3B + C = 9$ — (3)

Solving ①, ② & ③

$A = 1, B = 0$ & $C = 0$

$n = Ak^2 + Bk + C$

$n = Ak^2 + 0 + 0$

$n = k^2$

$k = \sqrt{n}$

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

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

i	j	k
n/2	logn	logn x logn
n/2 + 1	logn	logn x logn
.		
.		
.		
n	logn	logn x logn

$O(n \times (\log_2 n)^2)$

⑧
```
function (int  n)
{
  if (n == 1) return;
  for (i = 1 to n)
  {
```

```
for (j=1 to n)
{
    printf ("* ");
}
} function (n-3);
}
```

$\underbrace{(n-3), (n-6), (n-9) \ldots}_{k} \quad (1)$

$a = n-3 \quad , \quad d = n-6-n+3 = -3$

$l = (n-3) + (k-1)(-3)$

$l = (n-3) - 3k + 3$

$3k = n-l \qquad O(n^3)$

$k = \frac{n-l}{3} = O(n * n^2)$

(4) void function (int n)

```
for (i=1 to n)
{
for (j=1 ; j<=n; j+=i)
{
    printf ("* ");
}
}
```

For $i=1$, $\int = n$ times

$i=2$ $\int = n/2$ times

$i=3$ $\int = n/3$ times

$\vdots$

$i = k$ $\int = n/k$ times

$i = n$, $j = n/n$ times

Total Time Complexity: $n + n/2 + n/3 + \cdots n/n$

$$n = \underbrace{(1 + 1/2 + 1/3 + \cdots 1/n)}_{\log(n)}$$

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}$$

$$= \sum_{k=1}^{n} \frac{1}{k} \qquad O(n \log(n))$$

$$= \log(n) + O(1)$$

10 $f(n) = n^k$ $g(n) = c^n$

= where $k >= 1$ & $c > 1$

Let $k = 1$ & $c = 2$

$f(1) = (1)^1$ $g(1) = (2)^1$

$f(1) < g(1)$

$f(2) = 2^1$ $g(2) = (2)^2 = 4$

$f(2) < g(2)$

satisfies O notation $f(n) \leq cg(n)$

$f(n_0) = c_0 g(n_0)$

$n_0^k = c_0 \cdot c^{n_0}$

$k = 1$, $c = 2$

$n_0^1 = c_0 \cdot 2^{n_0}$

$\left(\frac{n_0}{c_0}\right)^1 = (2)^{n_0}$

Comparing

$$n_0 = 1$$

$$\frac{n_0}{c_0} = 2$$

$$\frac{1}{2} = c_0$$

$$f(n) \leq 0.5\, g(n)$$

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