21F-9516
Suresh Kumar
BS(SE) 6A

Design & Analysis of Algorithm

Q:1 :~

Sol

$\because f(n) = 3n^2 + 5n + 9$

$\Rightarrow f(n) = 3n^2$ → This is covering all values.

Hence:

$\Rightarrow \boxed{f(n) = O(n^2)}$ is the Big O notation.

Q:2 :~ Show $2n^2 + n + 1$ is $\Omega(n^2)$

Sol

$\Rightarrow f(n) = 2n^2 + n + 1$

$\because f(n) \geq c.g(n)$

$\Rightarrow 2n^2 + n + 1 \geq n^2$

$\Rightarrow 2n^2 + n + 1 - n^2 \geq 0$

$\Rightarrow n^2 + n + 1 \geq 0$

Hence ~~Proved~~ Proved

Now.

$\Rightarrow n = 5 \quad \because c2$

$\Rightarrow f(5) = 2(5)^2 + 5 + 1 \geq 2.g(5^2)$

$\Rightarrow f(5) = 2(25) + 5 + 1 \geq 2.25$

$\Rightarrow f(5) = 56 \geq$ ~~50~~ 50

So f(n) is must greater or equal to c.g(c) Hence Proved.

Q: 3:~

Sol

A) Asymptotic notation, such Big O, Omega and theta, are used to characterize how function behave as their input size increase towards infinity or in certain situations, towards negative infinity. They adhere to the transitive characteristic If f(n) is related to g(n) and g(n) is related to h(n), then f(n) is related to h(n), according to the transitive Property. If f(n) is O(g(n)) & g(n) is O(h(n)) then f(n) grows no faster than g(n) for sufficiently big n, & g(n) grows no faster than h(n), in the context of aysmptotic notations therefore it logically follows that f(n) also grows no faster than h(n).

B).

a) Assume that g(n) = n, h(n) = n^3 and f(n) = n^2.
Given that f(n) is O(g(n)) allow us to decide that, for sufficiently big n, f(n) grows no faster than g(n). meaning that f(n) is bounded above by a constant multiple of g(n). which is proved by the reason that n^2 grows slower than n as n approaches infinity. Similarly applies for (g(n)) when n grows slower than n^3 as n approaches infinity, we can say that g(n) grows no faster than h(n) for sufficiently Big n. in simpler term g(n) has limitations above by a constant multiple of h(n). we might decide that f(n)=n^2 is similarly O(h(n)) = n^3.

Q:4 :~

Sol // $f(n) = 2n^2$ is $\Omega(g(n)) = n^3$

A)

$\because f(n) \geq c \cdot g(n)$ for all $n \geq n_0$

$\Rightarrow f(n) \geq c \cdot g(n)$

$\Rightarrow 2n^2 \geq c \cdot n^3$

Divide (n^2) on both side;

$\Rightarrow \frac{2n^2}{n^2} \geq c \cdot \frac{n^2}{n^2}$

$\Rightarrow 2 \geq c \cdot n \quad \because c = 1$

$\Rightarrow 2 \geq n$

Hence, $f(n)$ is $\Omega(g(n))$ when $(n \geq 2)$ TRUE for all $n \geq 2$. So $f(n) = 2n^2$ is indeed $\Omega(g(n)) = n^3$.

B): $n/100 = \Omega(n)$

$\Rightarrow n/100 \geq c \cdot n \rightarrow$ for all $n \geq n_0$

Divide (n) on both side,

$\Rightarrow \frac{n}{n \times 100} \geq c \frac{n}{n}$

$\Rightarrow 1/100 \geq c \quad \because c = 1/100$

$\Rightarrow \frac{1}{100} \geq \frac{1}{100}$

$\Rightarrow \frac{1}{100} - \frac{1}{100} \geq 0$

$\Rightarrow 0 \geq 0 \quad \because n \geq 1$

Therefore, $n/100 = \Omega(n)$ with $c = \frac{1}{100}$ and $n = 1$ is true.

Q. 5 :~

Sol

A)

Time complexity

- The outermost runs from i=1 to n, → $O(n)$.
- The second loop runs from j=1 to i*i, → $O(i^2)$
- The innermost loop runs from k=j to j+j, → $O(1)$

overall time complexity is;

$$\Rightarrow O(n) \cdot O(i^2) \cdot O(1)$$

$$\Rightarrow \boxed{\Theta(n \cdot i^2)} \rightarrow \boxed{O(n^3)}$$

B).

↳ To simplify the time complexity, we need the maximum value of i in n terms. The value of i ranges from 1 to n.

- worst case is: $\Rightarrow O(n \cdot n^2)$

$$\Rightarrow \boxed{O(n^3)}$$

Hence, $O(n^3)$ is simplified time complexity.

Q. 6 :~

Sol

Time complexity.

- The outermost loop runs n time → $O(n)$
- The middle loop runs n time → $O(n)$
- The innermost loop runs n time → $O(n)$
- Bitwise AND operator (x & y & z) & the print statement have constant time complexity & do not affect the overall time complexity.

overall time complexity :

$$\Rightarrow O(n) \cdot O(n) \cdot O(n) \Rightarrow \boxed{O(n^3)}$$

Hence.

$O(n^3)$ is the time complexity of given code.

Q.7 :~

Sol

Time complexity.

- Initialized variables → O(1) constant.
- loop through the Array This loop iterates from 1 to n, and doubling the value of i in each iterations. the no. of iterations can be calculated as $(\log_2 n)$
- Return the result → $O(\log n)$
- Assignmenting Array → o(1) constant.
- → o(1) constant.

overall time complexity:

=> $o(1) * O(\log n) * o(1) * o(1)$

=> $O(\log n)$

Hence, $o(\log n)$ is the time complexity of Given code snippet.