

DAA Assignment - 1

① Asymptotic notation describe the growth rate of algorithms as their input size approaches infinity.

3 commonly used notations are

(i) Big O (O) :- Represents the upper bound on the growth rate.

Example :- If an algorithm has a time complexity of $O(n)$, it means the worst case running time grows linearly with the input size (n).

(ii) Omega (Ω) :- Represents a lower bound on the growth rate. If an algorithm has a time complexity of $\Omega(n^2)$, it means the worst-case time running grows at least quadratically with the input size n .

(iii) Theta (Θ) :- Represents both upper and lower bounds, indicating a tight bound on the growth rate.

If an algorithm has a time complexity of $\Theta(n)$, it means the worst case running time grows linearly, and there is well defined constant factor.

Ans 2

for (int i=0 ; i<n ; i+=2)

{
=
}

$$x_k = a x^{k-1}$$

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

Taking log both sides

$$\log_2 n = (k-1) \log_2 2 \Rightarrow \log_2 n = k-1$$

$$k = \log_2 n + 1$$

Time complexity $\Rightarrow O(\log_2 n)$

Ans ③ $T(n) = \begin{cases} 3T(n-1) & \text{if } n > 0, \\ \text{otherwise } 1 \end{cases}$

$$T(n) = 3T(n-1) \quad \text{put } n=n-1$$

$$T(n-1) = 3T(n-2) \quad \text{put } n=n-2$$

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

And so on

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

$$\text{put } n-k=0, \Rightarrow n=k$$

$$T(n) = 3^n T(0)$$

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

④ $T(n) = \begin{cases} 2T(n-1) - 1 & \text{if } n > 0, \\ \text{otherwise } 1 \end{cases}$

$$T(n) = 2T(n-1) - 1 \quad \text{--- (1)}$$

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

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

⋮

If we expand this, we get

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

we will continue this until $n-k = 0$, $k = n$

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

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

time complexity $\Rightarrow 2^n$

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

i	=	1	2	3	4	5
s	=	1	3	6	10	15

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

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

$$i(i+1) \leq 2n$$

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

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

$$\text{complexity} \Rightarrow O(\sqrt{n})$$

⑥ void function (int n)

{ int i, count = 0;

for (int i=1 ; i*i <= n ; i++)

{ count ++;

}

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

$$i^2 = 1 \quad 4 \quad 9 \quad 16$$

$$i \times i \leq n$$

$$i \leq \sqrt{n}$$

$$\text{Time complexity} \Rightarrow O(\sqrt{n})$$

⑦ void function (int n)

{
int i, j, k, count = 0;

n/2 → for (i = n/2; i ≤ n; i++)

log₂ n → for (j = 1; j ≤ n; j++)

log₂ n → for (k = 1; k ≤ n; k = k * 2)
count++;

}

$$\frac{n}{2} \times \log_2 n \times \log_2 n$$

$$\text{complexity} = \cancel{O(n \log_2 n)} O(n \log^2 n)$$

⑧ function (int n)

— T(n)

{

if (n == 1) return;

for (i = 1 to n) {

— n

for (j = 1 to n) {

— n²

printf(" * ");

}

}

function (n-3);

— T(n-3)

}

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

$$T.C. = O(n^3)$$

⑨ void function (int n)

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

i = 1, 2, 3, 4, ...
j = 1, 3, 6, 10, ...

i=1 n times

i=2 n/2 times

i=3 n/3 times

$$n + \frac{n}{2} + \frac{n}{3} + \dots + 1$$

complexity $\Rightarrow O(n \log n)$

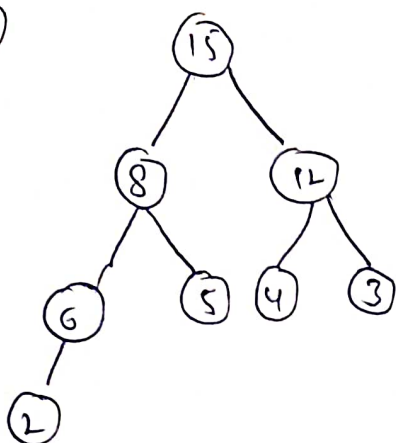
⑩ n^k ($k > 1$) c^n ($c > 1$)

c^n grows faster than n^k .

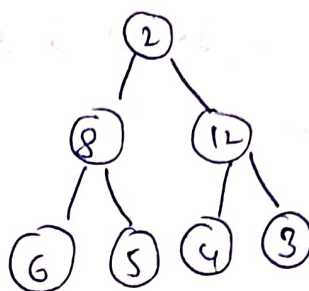
⑪ The time complexity for extractmin() given a min heap of n nodes is $O(\log n)$. This is because extractmin() removes the root node, which is the minimum element, and then calls heapify() to store the heap property. Heapify() takes $O(\log n)$ times as it traverse

the height of the heap, which is $\log n$.

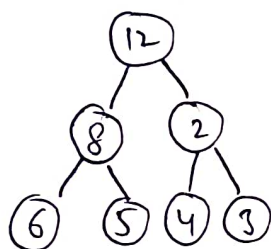
(12)



\Rightarrow



\Rightarrow



\Rightarrow

