DAA

Assignment - 01

(1.) Asymptotic Notation :- Asymptotic notations are the methods used to define the running time of the algorithm based on input size.

These are of following types :-

1. Big-O :- $f(n) = O(g(n))$ → Tight upper bound

The Complexity of function $f(n)$ doesn't go beyound the growth of the asymptotic notation.

Ex :- $f(n) = 3\log n + 100$

$g(n) = \log n$

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

Is $3\log n + 100 = O(\log n)$

$3\log n + 100 <= C \cdot \log n$

let $C = 200$

$3\log n + 100 <= 200 * \log n \quad \forall\ n > 2$

The defination of Big-O has been met therefor $f(n)$ is $O(g(n))$

$f(n) \leq C \cdot g(n)$

$\forall\ n \geq n_0$

for $C > 0$

2. Small-o : gives upper bound

$f(n) < g(n) \quad \forall n > n_0 \in \forall\ C > 0$

3. Big omega (Ω) :- $f(n) = \Omega(g(n))$ → Tight lower bound of $f(n)$

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

$f(n) \geq C \cdot g(n)$

$\forall\ n \geq n_0 \cdot C > 0$

4. Small ω : given lower bound

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

$f(n) > c \cdot g(n)$

5. Theta (Θ): gives both tight upper and lower bound.

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

(2)

```
for (i=1 to n)
{
    i = i * 2 ; // This instruction will be executed n times
}
```

$T(n) = O(n)$

(3) $T(n) = 3T(n-1)$ ① if $n>0$

else $T(n) = 1$

Put $n = n-1$

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

from eqn ①

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

Put $n = n-2$

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

Put $T(n-2) = \frac{T(n)}{3^2}$

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

we can say that $T(n) = 3^n T(n-n)$

$= 3^n T(0)$

$T(n) = 3^n$

(4) $T(n) = 2T(n-1) - ①$ if $n>0$
else $= 1$

Put $n = n-1$

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

using eqn ①

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

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

$T(n) = 4T(n-2) - 2 - 1 - ②$

Put $n = n-2$

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

using eqn ②

$\frac{(T(n)+2+1)}{4} = 2T(n-3) - 1 \Rightarrow T(n) = 8T(n-3) - 4 - 2 - 1$ ← ③

from here we can see

$T(n) = 2^n T(n-n) - 2^{n-1} - 2^{n-2} - \cdots 1$

$= 2^n - 2^{n-1} - 2^{n-2} - \cdots 1$

$= 2^n - [2^{n-1} + 2^{n-2} + \cdots + 1]$

$= 2^n - [2^n - 1]$

$= 1$

$T(n) = O(1)$

(5)

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

$i = 1, 2, 3, 4, 5, 6$

$s \Rightarrow 1, 3, 6, 10, 15, 21 \ldots < n$

$s_i = s_{i-1} + i$

$i = 1+2+3+4+5+6+\ldots K$

$$\frac{K(K+1)}{2} > n$$

$$\frac{K^2+K}{2} > n$$

$$K > \sqrt{n}$$

$$T(n) = O(\sqrt{n})$$

(6)

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

$i = 1$ to $i^2 <= n$

$i^2 = n$

$i = \sqrt{n}$

$T(n) = O(\sqrt{n})$

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
1	1	$\frac{n}{2}$
2	2	$\frac{n}{2}+1$
4	4	$\frac{n}{2}+3$
8	8	1
1	1	1
n	n	n

$2^k <= n$

$k = \log_2 n$

Here $T(n) = O(n * \log n * \log n)$

$= O(n \log^2 n)$

⑧

```
function (int n)
{
    if (n == 1)
      return n;  → (1)
    for (i = 1 to n) → (n)
    {
        for (j = 1 to n) → (n)
        {
            printf ("*");
        }
    }
    function (n-3); → (1)
}
```

$n^2 + 2$

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

(11)

```
void fun (int n)
{
    int j=1, i=0;
    while (i<n)
    {
      i = i+j;
      j++;
    }
}
```

i=0
i = 1, 3, 6, 10, 15
here it depends on j
$j = 1+2+3+4 \cdots k$
$= \frac{k(k+1)}{2} = \frac{k^2+k}{2}$
$k^2 = n$
$k = \sqrt{n}$

$T(n) = O(\sqrt{n})$

(12) Recursive Binary Search:

```
int fibonacci (int n)
{
    if (n<=1)          -(1)
      return 1;
    return fibonacci (n-1) + (n-2);   -T(n-2)
                      T(n-1)
}
```

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

$\rightarrow n = n-1$

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

characteristic eqn

$x^2 = x+1$
$x^2 - x - 1 = 0$
$x = \frac{(1+\sqrt{5})}{2}$ $\quad x = \frac{(1-\sqrt{5})}{2}$

$f(n)(k_1)^n + (k_2)^n$

$f(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$

$T(n) = O((1+\sqrt{5})/2)^n$
$= O(1.6180)^n$. Space comp: $O(n)$
↳ Golden Ratio

(13) (i) $T(n) \rightarrow n(\log n)$

```
for(int i=0 ; i<n ; i++) → n
{
    int s=0 ; e = n-1 ;
    while (s<=e)
    {
        if (arr[mid] == key)
            return;
        if(arr[mid] < key)
            s = mid+1;
    } else   e = mid-1;
}
```

} $\log(n)$

$$T(n) = O(n \cdot \log(n))$$

ii) n^3

```
for(int i=0 ; i<n ; i++) → n
{
    for(int j=0; j<n; j++) → n
    {
        for(int k=0; k<n; k++) → n
            count++;
    }
}
```

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

(14) $T(n) = T(n/4) + T(n/2) + c \cdot n^2$

(19)

```
search ( arr , n , x )
    iff arr [n-1] == x
        return " true "
    backup = arr [n-1]
    arr[n-1] = x
    for i=0 ; i++
        if arr [i] == x
            arr [n-1] = backup
            return ( i < n-1 )
```

(20) iterative recursion :-

```
insertion sort ( arr , n )
{
    for ( i = 1 to n )
    key = arr [i]
    j = i-1
    while ( j >= 0 && arr [j] > key )
    {
        arr [j+1] = arr [j]
        j = j-1
    }
    arr [j+1] = key
}
}
```

Recursive Insertion

```
insertion Sort (arr, n)
{
    if (n<=1)
    return;
    insertion Sort (arr, n-1)
      last = arr[n-1]
       j = n-2
    while (j>=0 && arr[j] > (ar)
      arr[j+1] = arr[j]
        j--
    }
    arr[j+1] = last
}
```

(21)

	Best	Avg	Worst
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$

(23) iterative binary search

```
binarySearch(arr, l, r, x)
{
    while (l <= r)
    {
        if (arr[mid] == x;
        return mid;
        if (arr[mid] > x)
            r = m-1;
        else l = m+1;
    }
```

recursive binary search

```
binarySearch (arr, r, e, x)
  if (e >= l)
  {
      mid = r + (e - r) /2;
      if (arr[mid] == x)
          return mid;
      if (arr[mid] > x)
          return binarySearch (arr, r, m-1, x);
          return binarySearch (arr, mid+1, e, x);
  }
      return -1;
}
```

(24) using Previous Pseudo Code;

$$T(n) = T\left(\frac{n}{2}\right) + 1$$