Types of algorithm

1) Iterative algorithm: ⇒ Loop : for, while, do-while

2) Recursive algorithm: It is a technique in which function calls itself, inside its body.

Eg:
```
A()
{ A();
}
```

we use Divide & Conquer technique.

Time analysis of Iterative Algorithms

Step1 Identify the executable statements used in the algorithm.

Step2 Calculate number of times (freq Count) the statement will execute.

Step3 Add all the frequency Count to find frequency Count of the entire algorithm.

Step4 Convert the frequency count of algorithm into time Complexity using asymptotic notations.

```
for(i=1 ; i<5; i++)
{
}
```

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

total 6 times (n+1)

Example:

Algorithm: Sum (a,n)

```
{
    S=0 ;                    ------ 1
for i= 1 to n do             ------ n+1
{
    S = S + a[i]             ------ n
}
    return S;                ------ 1
}
```

$f(n) = 1 + n+1 + n + 1$

$f(n) = 2n + 3$

Time Complexity = $O(n)$

Example: Sum of 2 matrices

```
for i=1 to n do              ------ n+1
{ for j=1 to n do            ------ n(n+1)
  { c[i][j] = A[i][j] + B[i][j]  ------ n^2
  }
}
```

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

$= n+1+n^2+n+n^2$

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

Time Complexity = $O(n^2)$

Example Multiplication of 2 matrices

```
S = 0;                                  ... 1
for (i = 1 to n do                      ... (n+1)
{ for j = 1 to n do                     ... n(n+1)
  { for = K to n do                     ... n^2(n+1)
    {
      S = S + A[i][K] * [K][j]          ... n^3
    }
    C[i][j] = S                         ... n^2
    S = 0                               ... n^2
  }
}
```

$f(n) = 1 + n^3 + n^2 + n^2$

$f(n) = n^3 + 2n^2 + 1$

Time Complexity $\Rightarrow O(n^3)$

Example:

```
x = 0 .... 1 ... 1
for i = 1 to n do
  { for j = 1 to i do
  { {
    x = x+1 .... 1 + 2 + 3 + ... + n
    }
}
```

$f(n) = 1 + (1 + 2 + 3 + \dots + n)$

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

Time Complexity $O(n^2)$

Example

```
i = 1 ---------- 1
n = 0 ---------- 1
while (i < n)
{
    n = n + 1 ------ k (let)
    i = i * 2 ------ k (let)
}
```

loop	i
1	2
2	$4 \text{ or } 2^2$
3	$8 \text{ or } 2^3$
4	$32 \text{ or } 2^4$
:	
k	2^k

$f(n) = 2k + 2$

$f(n) = 2 \log_2 n + 2$

Time Complexity = $O(\log_2 n)$

Let $n = 2^k$

Take log on both sides

$\Rightarrow \log n = \log_2 2^k$

$\Rightarrow \log_2 n = k \log_2 2$

$\Rightarrow \log_2 n = k \cdot 1$

$\Rightarrow k = \log_2 n$

Example:

```
for i = 1 to n
{     i = n  - - - - - - - - n
      n = 0  - - - - - - - - n
  while (i > 1)
  {
    n = n + 1  - - - - - - - - n/k
    i = i/2    - - - - - - - - n/k
```

loop	i
1	$n/2$
2	$n/2^2$
3	$n/2^3$
.	
.	
.	
k	$n/2^k$

$f(n) = 2n + 2nk$

$f(n) = 2n + 2n \log_2 n$

$\therefore$ time complexity $= O(n \log_2 n)$

Let $\frac{n}{2^k} = 1$

$n = 2^k$

take log on both sides

$\log_2 n = \log_2 2^k$

$\boxed{k = \log_2 n}$

Example

```
x = 0          - - - - 1
y = 0          - - - - 1
while (i <= n)
{
  x = x+1      - - - - n
  i = i+1      - - - - n
}
for (j = 1 to n)
{  for (k = 1 to n)
   {  y = y+1      - - - - n^2
   }
}
```

$f(n) = n^2 + 2n + 2$

$\therefore$ time complexity $O(n^2)$

Time analysis of recursive algorithm

Step 1 Create a recurrance relation for the algorithm

Step 2 Convert recurrance relation into a closed form using Substitution method.

```
Algorithm Rsum (a, n)
{
    if (n<=0) then ------ 1
      return 0;  ------ 1
    else
      return Rsum (a, n-1) + a(n) ....
                                1 + t(n-1)
```

$n \leq 0$ Sum $\rightarrow 0$

$n > 0$

Let $n=4$ Rsum(a,4) return Rsum(a,3) + a(4) = a(1) + a(2) + a(3) + a(4)

$n=3$ Rsum(a,3) return Rsum(a,2) + a(3) = a(1) + a(2) + a(3)

$n=2$ Rsum(a,2) return Rsum(a,1) + a(2) = a(1) + a(2)

$n=1$ Rsum(a,1) return Rsum(a,0) + a(1) = 0 + a(1)

Recurrence Relation

$$t(n) = \begin{cases} 2 & \text{if } (n \leq 0) \\ 2 + t(n-1) & \text{if } n > 0 \end{cases}$$

Solving recurrence relation using Substitution method:

$t(n) = 2 + t(n-1)$ — ①

$= 2+2 + t(n-2)$

$= 4 + t(n-2)$

$= 2(2) + t(n-2)$ — ②

$= 2(2) + [2 + t(n-3)]$

$= 2+2+2 + t(n-3)$

$= 2(3) + t(n-3)$ — ③

⋮

$t(n) = 2(n) + t(n-n)$

$t(n) = 2n + t(0)$

$= 2n + 2$

Substitute $t(n)$ by $t(n-1)$

$\Rightarrow t(n-1) = 2 + t(n-1-1)$

$= 2 + t(n-2)$

Now substitute $t(n-1)$ value in eq ①

Substitute $t(n)$ by $t(n-2)$

$t(n-2) = 2 + t(n-2-1)$

$= 2 + t(n-3)$

Substitute value in eq ②

∴ time complexity = $O(n)$

Q. Solve the recurrence relation:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + C & \text{if } n > 1 \end{cases}$$

Solving using Substitution method:

1 - $T(n) = T(n/2) + C$ — ①

$= [T(n/4) + C] + C$

$= T(n/4) + 2C$

2 - $= T(n/2^2) + 2C$

3 - $= T(n/2^3) + 3C$

k - $= T(n/2^k) + kC$ — ②

Let $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$ — ③

take log on both sides

$\log_2 n = \log_2 2^k$

$\boxed{k = \log_2 n}$ — ④

Substitute ③ & ④ in eq ②

Substitute the value of $T(n)$ in $T(n/2)$

$\therefore T(n/2) = T\left[\frac{(n/2)}{2}\right] + C$

$= T(n/4) + C$

Substitute in eq ①

$$T(n) = T\left(\frac{n}{n}\right) + C \log_2 n$$

$$= T(1) + C \log_2 n$$

$$= 1 + C \log_2 n$$

∴ time complexity is $O(\log_2 n)$