Priyank Gupta
Section-SE
49

Design and Analysis of Algorithm (TCS 505)

Assignment-1

Solution-1 :- These notations are used to tell the Complexity of an algorithm when the input is very large

Asymptotic ⟶ towards infinity.

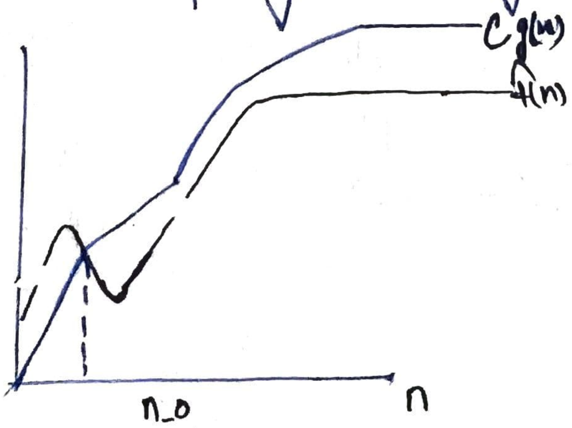
Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic Analysis.

3-types of Asymptotic Notation :-

1) Big O Notation :- This defines an upper bound of an algorithm, it bounds a functions only from above.

For Example :- in Case of Insertion Sort, it takes linear time in best Case and quadratic time in worst Case. So, we can say that Time Complexity of Insertion Sort is $(O(n^2))$.

It is useful only when we have upper bound on time time complexity of an algorithm.



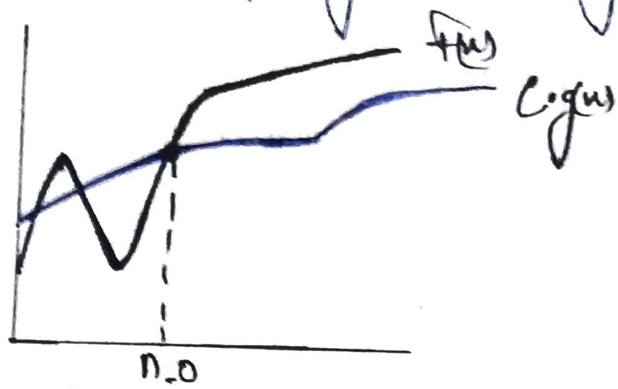
$O(g(n)) = \{ f(n)$: there Exist positive Constant c and n0 such that $0 <= f(n) <= c^* g(n)$ for all $n >= n0)$

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

2) **Big Omega (Ω) Notation** :- This defines lower bound of an algorithm. This Notation is the least used Notation among all.

Example:- Time Complexity of Insertion Sort can be written as $\Omega(n)$

This Notation can be useful when we have lower bound on time complexity of an algorithm.



$\Omega(g(n)) = \{f(n)$: there exist positive constants c and $n0$ such that $0 <= c^* g(n) <= f(n)$ for all $n >= n0\}$

$f(n) = \text{Omega}(g(n))$

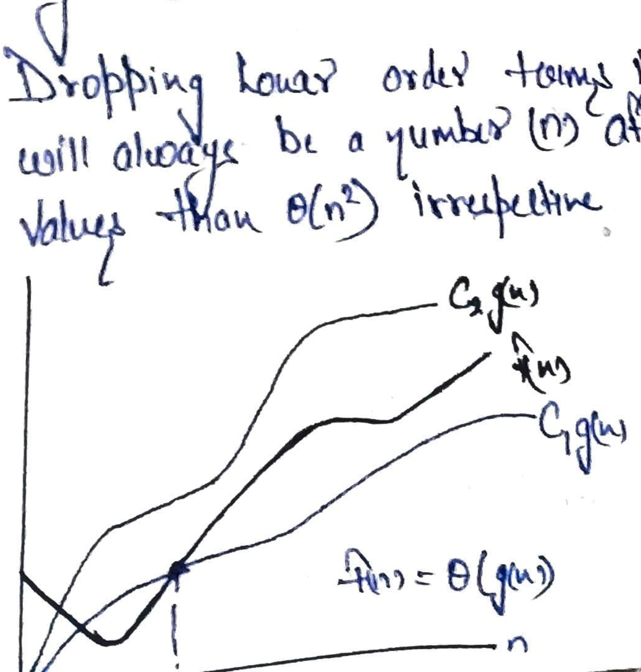
3) **Theta (θ) Notation** :- This Notation bounds a function a function from above and below, so it defines exact asymptotic behaviour.

A simple way to get θ Notation of an expression is to drop low order terms and ignore leading constants.

Eg - $3n^3 + 6n^2 + 6000 = \theta(n^3)$.

Dropping lower order terms is always fine because there will always be a number (n) after which $\theta(n^3)$ has higher values than $\theta(n^2)$ irrespective of the constant involved.

$\theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 & $n0$ such that $0 <= c_1^* g(n) <= f(n) <= c_2^* g(n)$ for all $n >= n0\}$



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

Solution 2:- for (i=1 to n) { i=i*2 }

1, 2, 4, 8, 16 ... n

it's a G.P. So,

$a=1,\ r=2,$

So,

$$S_n = \frac{a(r^{\text{terms}}-1)}{(r-1)} \quad \& \quad T_k = ar^{k-1}$$

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

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

$$2^k = 2n$$

taking log both side

$$k = \log_2(2n)$$

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

$$k = \log_2(n) + 1$$

So,

Time Complexity $= O(\log_2(n)+1) = O(\log_2 n)$

Solution 3:-

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

$$T(n) = 3T(n-1) \quad -(1)$$

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

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

put $T(n-1)$ Value to in eq-(1)

So,

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

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

$$T(n-2) = 3T(n-3) \quad -(4)$$

put the value of $T(n-2)$ in eq-③

$T(n) = 9(3T(n-3)$ - ⑤

$\vdots$

$T(n) = 3^k T(n-k)$ - ⑥

Now,

$T(1) = 1$

$n-k = 1$

$k = n-1$

$T(n) = 3^{n-1} T(n-(n-1))$ - ⑦

$T(n) = 3^{n-1} T(1)$

$T(n) = 3^{n-1}$

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

$\boxed{T(n) = O(3^n)}$

Solution:-4

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

$T(1) = 1$

$T(n) = 2T(n-1) - 1$ — ①

put $n = n-1$ in eq-①

$T(n-1) = 2T(n-2) - 1$ — ②

put $T(n-1)$ in eq ①

$T(n) = 2(2T(n-2) - 1) - 1$ - ③

put $n = n-2$ in eq-①

$T(n-2) = 2T(n-3) - 1$ - ④

put the value of $T(n-2)$ in eq-③

$T(n) = 4(2T(n-3) - 1) - 2 - 1$ - ⑤

$$T(n) = 8T(n-3) - 4 - 2 - 1 \quad - (6)$$

$$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} \cdots 2^2 - 2^1 - 2^0 \quad - (6)$$

Now

$$n-k = 1$$

$$k = n-1$$

put the value of k in eq-6

$$T(n) = 2^{n-1} T(1) - [2^0 + 2^1 + 2^2 + \cdots + 2^{n-3} + 2^{n-2}]$$

$$T(n) = 2^{n-1} \times 1 - \left[\frac{1(2^{n-1} - 1)}{2-1} \right] \qquad \left\{ \begin{array}{l} S = \frac{a(r^{terms} - 1)}{r-1} \\ \text{G.P. formula} \end{array} \right\}$$

$$T(n) = 2^{n-1} - [2^{n-1} - 1]$$

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

$$\boxed{T(n) = 1}$$

Solution:-5

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

$i = 1, 2, 3, 4 \cdots + k$

$\frac{k(k+1)}{2} \Rightarrow k^{th}$ term formula.

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

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

$$\boxed{T.C. = O(\sqrt{n})}$$

Solution:- 6

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

$1, 2, 4, 8, 16 \ldots n$

it's a G.P, $a=1, r=2$

$t_k = a(r^{term-1}) = 1(2^{k-1})$

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

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

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

$\boxed{T.C. = O(\log_2 n)}$

Solution :- 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++;
            }
        }
    }
}
```

for the first loop Time Complexity will be $O(n)$

* for the second loop it will be $O(\log n)$
* for the last loop it will be $O(\log n)$

So,

Time Complexity $= O(n) * O(\log n) * O(\log n)$

$\boxed{T.C. = O(n \log^2 n)}$

Solution :- 8

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

for the first loop time Complexity will be $O(n)$.

for the second loop time Complexity will be $O(n)$.

Total Complexity = $O(n^2)$

Solution :- 9

```
Void function (int n) {
    for (i=1 to n) {
        for (J=1; J<=n; J=J+i)
            print("*");
    }
}
```

Time Complexity = $O(n)$

Solution :- 10

$f(n) = n^k$
$g(n) = a^n$

$g(n)$ is tight upper bound of $f(n)$

$f(n) = O(g(n))$
$n^k = O(a^n)$

iff

$f(n) \le c \cdot g(n)$
$n^k \le c \cdot a^n \quad \forall \ n \ge n_0$

and $\underline{\underline{c > 0}}$.

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

$$\boxed{n^k = O(a^n)}$$

Solution :- 11

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

1, 2, 3, 4 - - - - n

$T_k \ge \frac{K(K+1)}{2}$ ⇒ The value of i after k iterations.

$n \ge \frac{k(K+1)}{2}$ ⇒ So, if loop runs k times, then this

$2n \ge k^2 + k$

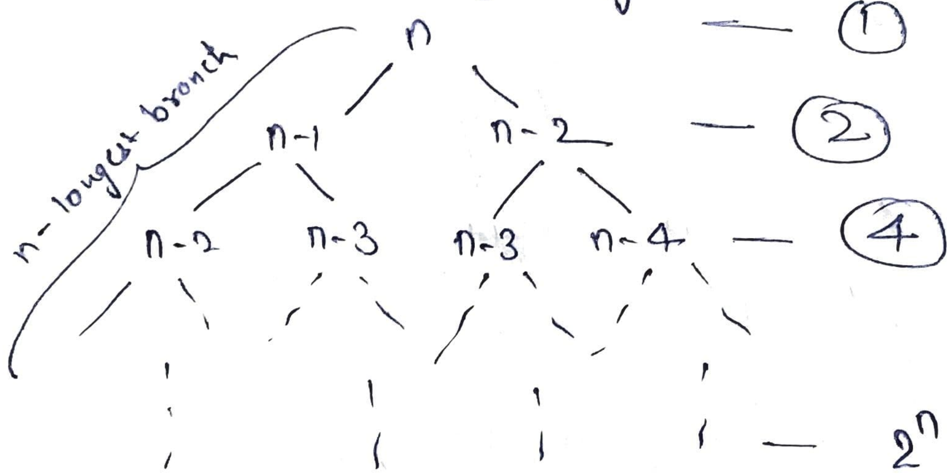
$\boxed{T.C. = O(\sqrt{n})}$ ⇒ Therefore time Complexity.

Solution :- 12

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

Recursive Relation :- $T(n) = T(n-1) + T(n-2) + 1$

We solve this using tree Method.



$$T = 1 + 2 + 4 + \cdots + 2^n$$

$$a = 1, \ r = 2$$

$$= \frac{a(r^{\text{terms}} - 1)}{r - 1}$$

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

$$T.C. = O(2^{n+1})$$

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

$$\boxed{T.C. = O(2^n)}$$

The Max Depth is proportional to the n, hence the space complexity of Fibonacci Series is $O(n)$.

Solution :- 13 For nlogn time Complexity.

```
For (int i=1; i<=n ; i*=2)   //O(logn)
{
  for(int J=1; J<=n ;J+=2) // O(n)
  {
    Sum +=J;
  }
}.
```

$$T.C. = O(n\log n)$$

For n^3 time Complexity

```
for (int i=1 ; i<=n; i++)
{
  for (int J=i+1 ; J<=n; J++)
  {
    for (int i=k+1 ; k<=n ; K++)
    {
      Sum += k;
    }
  }
}
```

$$\boxed{T.C. = O(n^3)}$$

For O(log(logn)) time Complexity.

```
for(int i=2; i<=n; i=pow(i,c))
{
for (int i=n; i>1; i=fun(i))
{
  // O(1) Expression
}
}.
```

$$\boxed{T.C. = O(\log(\log n))}$$

Solution:- 14 $T(n) = T(\frac{n}{4}) + T(\frac{n}{2}) + cn^2$

Using Master's Method.

Let us assume that

$T(\frac{n}{2}) \geq T(\frac{n}{4})$

then our recurrence relation will be

$T(n) = 2T(\frac{n}{2}) + cn^2$ | $T(n) = aT(\frac{n}{b}) + f(n)$

$a = 2, \ b = 2$

$c = \log_b a$

$c = \log_2 2 = 1$

$n^c = n^1 = n$

$f(n) = n^2$

$f(n) > n^c$

$n^2 > n$

So, Time Complexity will be

$T(n) = \Theta(f(n))$

$$\boxed{T(n) = \Theta(n^2)}$$

Solution:- 15

```
int fun (int n) {
  for (int i=1; i<=n; i++)
  {
    for (int j=i; j<=n; j+=i)
    {
      //some O(1) task
    }
  }
```

for the first loop time complexity will be $O(n)$
for the second loop time complexity will be $O(\log n)$ as in second loop formation of G.P. is there

So,

$$\boxed{T.C. = O(n\log n)}$$

Solution:-16

```
for(int i=2; i<=n; i=pow(i,k))
{
    // O(1) expression
}
```

In this case, i takes values $2, 2^k, (2^k)^k = 2^{k^2}, (2^{k^2})^k = 2^{k^3}, \ldots, 2^{k^{\log_k(\log(n))}}$. The last term must be less than or equal to n, and we have $2^{k^{\log_k(\log(n))}} = 2^{\log(n)} = n$, which completely agrees with the value of our last term. So there are in total $\log_k(\log(n))$ many iterations, and each iteration takes a constant amount of time to run, therefore the total time complexity is $O(\log(\log(n)))$.

$$\boxed{T.C. = O(\log(\log(n)))}$$

Solution:-17

Recurrence Relation:-

$$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{10}\right) + n$$

$$T(1) = 1$$

Suppose Using Master's Method :-

$T\left(\frac{99n}{100}\right) \geq T\left(\frac{n}{10}\right)$

then our recurrence relation will be

$T(n) = 2T\left(\frac{99n}{100}\right) + n$

$a = 2,\ b = \frac{100}{99},\ f(n) = n$

$c = \log_b a$

$c = \log_{\frac{100}{99}} 2$

$n^c = n^{\log_{\frac{100}{99}} 2}$

$f(n) < n^c$

So,

$\cancel{f(n) \neq n}$

$$\boxed{T.C. = \Theta\left(n^{\log_{\frac{100}{99}} 2}\right)}$$

Solution:-18 a) $100 < \text{root}(n) < \log\log(n) < \log(n) < n\log(n) < n < n^2 <$
$\log(n!) < 2^n < 2^{2n} < 4^n$

b) $1 < \log(\log(n)) < \sqrt{\log n} < \log(n) < \log(2n) < n\log(n)$
$< 2\log(n) < n < \log n! < 2n < 4n < n^2 < n! < 2(2^n)$

c) $96 < \log_8(n) < \log(n) < n\log_6(n) < n\log_2(n) < 5n$
$< 8n^2 < \log(n!) < 7n^3 < n! < 8^{2n}$

Solution:-19

```
int LinearSearch (int *arr, int n, int key)
{
   for i<-0 to n-1
      if (arr[i]= key)
         return i.
   return -1
}
```

Solution:-20

Iterative Insertion Sort:-

```
Void insertionSort (int arr[], int n)
{
   for i=1 to n
      int Value ← arr[i];
      int j ← i;
      while j>0 and arr [J-1] < value
      {
         arr[i] ← arr [J-1];
         J--;
      }
      arr [J] ← Value;
}
```

Recursive Insertion Sort:-

```
Void InsertionSort (int arr[], int i, int n)
{
   int Value ← arr[i];
   int J ← i;
   while J>0 and arr [J-1] > Value
   {
      arr [J] ← arr[J-1];
      J--;
   }
   arr[J] ← Value;
   if (i<=n) {
      InsertionSort (arr, i+1 , n);
   }
}
```

Insertion sort is an online sorting algorithm since it can sort a list as it receives it. In all other algorithms, we needs all elements to be provided to the algorithms before applying it.

Solution:- 21

Insertion Sort : $O(n^2)$

```
Void insertionSort (int arr[], int n)
{
    int i, temp, J;
    for i <- 1 to n
    {
        temp <- arr[i];
        J <- i-1;
        while (J>=0 AND arr[J] > temp)
        {
            arr[J+1] <- arr[J];
            J <- J-1;
        }
        arr[J+1] <- temp;
    }
}
```

Bubble Sort : $O(n^2)$

```
Void BubbleSort(int arr[], int n)
{
    int i, J;
    for i <- 0 to n-1
        for J <- 0 to n-i-1
            if (arr[J] > arr[J+1])
                swap (arr[J], arr[J+1]);
}.
```

Selection sort : $O(n^2)$

```
Void selectionSort (int arr[], int n)
{
    int i, J, min_idx;
    for i <- 0 to n-1;
    {
        min_idx <- i;
```

```
for j <- i+1 to n
    if (arr[j] < arr[min_idx])
        min_idx <- j;
  swap (arr [min_idx] ,arr[i]);
  }
}
```

Solution:-22

	Inplace	Stable	Online
Insertion	✓	✗	✓
Selection	✓	✗	✗
Bubble	✓	✓	✗
Quick	✓	✗	✗
Merge	✗	✓	✗

Solution:-23

```
int binarysearch(int arr[], int n, int key)
{
    int l <- arr[0];
    int r <- arr [n-1];
    while (l<=r)
    {
        mid = ((l+(r-l))/2);
        if (mid==key)
            return mid;
        Else if (mid>key)
            r <- mid-1;
        Else
            l <- mid+1;
    }
    return -1;
}
```

→ Time Complexity linear search (Iterative) : $O(n)$

Space Complexity : $O(1)$

→ Time Complexity Linear Search (Recursive) : $O(n)$

Space Complexity : $O(n)$

→ Time Complexity Binary Search (Iterative) : $O(\log n)$

Space Complexity : $O(1)$

→ Time Complexity Binary Search (Recursive) : $O(\log n)$

Space Complexity : $O(\log n)$

Solution :- 24 Recursive Relation :-

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