

Analysis of Algorithms

SY Computer

Even 2022-23

Introduction

Algorithm:

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

****We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.**

Every algorithm must satisfy the following criteria:

- **Input:** there are zero or more quantities, which are externally supplied;
- **Output:** at least one quantity is produced
- **Definiteness:** each instruction must be clear and unambiguous;
- **Finiteness:** if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
- **Effectiveness:** every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

Performance Analysis

- The performance of a program is the amount of computer memory and time needed to run a program.
 1. Time Complexity
 2. Space Complexity
- How to compare Algorithms?
 1. Execution time
 2. Number of statements executed
 3. Running time Analysis

Time Complexity

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm.

The time complexity of a program is the amount of computer time it needs to run to completion.

Time Complexity is mainly of 3 Types:

1. Best Case
2. Worst Case
3. Average Case

Space Complexity

- The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:
- Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.
- Data space: Data space is the space needed to store all constant and variable values.
- Environment stack space: used to save information needed to resume execution of partially completed functions.
- The space requirement $S(P)$ of any algorithm P may therefore be written as,

$$S(P) = c + S_p(\text{Instance characteristics})$$

where “c” is a constant.

Complexity of Algorithms

- The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size “ n ” of the input data.
- Approaches to calculate Time/Space Complexity:
 1. Frequency count/Step count Method
 2. Asymptotic Notations – (Order of)

Frequency count/Step count Method

Rules:

1. For comments, declaration
count = 0
2. return and assignment statement
count = 1
3. Ignore lower order exponents when higher order exponents are present

Ex. Complexity of following algo is as follows:

$$f(n) = 6n^3 + 10n^2 + 15n + 3 \Rightarrow 6n^3$$

4. Ignore constant multipliers

$$6n^3 \Rightarrow n^3$$

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

Example 1: sum of n values of an array

```
Algorithm sum (int a[], int n
```

```
    s = 0;
```

```
    for(i=0; i<n; i++)
```

```
    {
```

```
        s=s + a[i];
```

```
    }
```

```
    return s;
```

Time Complexity

→ 1

→ n+1

→ n

→ 1

→ 2n+3

$f(n) = O(n)$

Space Complexity

a[] = n words

n = 1 word

s = 1 word

i = 1 word

n+3

Space complexity =

$O(n)$

Example 2: Addition of two square Matrices of dimension $n \times n$

```
Algorithm addMat (int a[][], int b[][])
```

```
{ int c[][];
```

```
  for(i=0; i<n; i++) {
```

```
    for(j=0; j<n; j++) {
```

```
      c[i][j] = a[i][j] + b[i][j]
```

```
    }
```

```
  }
```

```
}
```

Time Complexity

→ $n+1$

→ $n \times (n+1)$

→ $n \times n$

→ $n+1+n^2+n+n^2$

→ $2n^2 + 2n + 1$

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

Space Complexity

$a[][] = n^2$ words

$b[][] = n^2$ words

$c[][] = n^2$ words

$i = 1$ word

$j = 1$ word

$n = 1$ word

→ $3n^2 + 3$

Space complexity =
 $O(n^2)$

Example 3: Multiplication of two Matrices of dimension $n \times n$

```
Algorithm matMul (int a[][], int b[][])
```

```
{ int c[][];
```

```
  for(i=0; i<n; i++) {
```

```
    for(j=0; j<n; j++) {
```

```
      c[i][j] = 0;
```

```
      for(k=0; k<n; k++){
```

```
        c[i][j] = a[i][j] * b[i][j]
```

```
      }
```

```
    }
```

```
  }
```

```
}
```

Time Complexity

$\rightarrow n+1$

$\rightarrow n \times (n+1)$

$\rightarrow n \times n$

$\rightarrow n \times n \times (n+1)$

$n \times n \times n$

$\rightarrow n+1+n^2+n+n^2$
 $+n^3+1+n^3+n^2$

$\rightarrow 2n^3 + 3n^2 + 2n + 1$

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

Space Complexity

$a[][] = n^2$ words

$b[][] = n^2$ words

$c[][] = n^2$ words

$i = 1$ word

$j = 1$ word

$k = 1$ word

$n = 1$ word

$\rightarrow 3n^2 + 4$

Space complexity =

$O(n^2)$

Example: loops

1.

for(i=0; i<n; i++) {
statements;
}

Time Complexity
→ n+1
→ n
$f(n) = 2n+1$ $f(n) = O(n)$

2.

for(i=n; i>0; i--) {
statements;
}

Time Complexity
→ n+1
→ n
$f(n) = 2n+1$ $f(n) = O(n)$

Example: loops

3.

for(i=1; i<n; i=i+2) {
statements;
}

Time Complexity

→ $n+1$

→ $n/2$

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

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

4.

for(i=0; i<n; i++) {
for(j=0; j<n; j++) {
statements;
}
}

Time Complexity

→ $n+1$

→ $n(n+1)$

→ $n \times n$

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

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

Example: loops (By tracing)

```
5.  for(i=0; i<n; i++) {  
    for(j=0; j<i; j++) {  
        statements;  
    }  
}
```

$$1 + 2 + 3 + 4 + \dots + n = n(n + 1)/2$$

$$T(n)=1 + 2 + 3 + 4 + \dots + n - 1 = \frac{(n-1)(n)}{2} = O(n^2)$$

Time Complexity		
i	j	statements
0	0	0
1	0	1
	1	
2	0	2
	1	
	2	
3	0	3
	1	
	2	
	3	
...
N	0 to n-1	n

Example 5: loops (By tracing)

```
6.  p=0 ;  
    for(i=1; p<=n; i++) {  
        p=p+i;  
    }  
}
```

$$= 1+2+3+4+\dots+k > n$$

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

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

$$\cong k^2 > n$$

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

Time Complexity		
i	p	statements
1	0+1	1
2	1+2	1
3	1+2+3	1
4	1+2+3+4	1
5	1+2+3+4+5	1
6	1+2+3+4+5+6	1
k	1+2+3+4+...+k	???

Example: loops (By tracing)

6.

```
for(i=1; i<n; i=i*2) {  
    statements;  
}  
}
```

$$\begin{aligned}i &\geq n \\ i &= 2^k \\ 2^k &\geq n \\ 2^k &= n\end{aligned}$$

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

Time Complexity	
i	statements
$1 \cdot 2^0$	1
$1 \cdot 2$	1
$1 \cdot 2 \cdot 2$	1
$1 \cdot 2 \cdot 2 \cdot 2$	1
...	...
...	...
2^k	1

Example: loops (By tracing)

7.

```
for(i=n; i>=1; i=i/2) {  
    statements;  
}  
}
```

$$\begin{aligned}i &< 1 \\ n/2^k &= 1 \\ n &= 2^k \\ \mathbf{k} &= \mathbf{\log_2 n} = \mathbf{O(\log_2 n)}\end{aligned}$$

Time Complexity
i
n
n/2
$n/2^2$
$n/2^3$
$n/2^4$
...
$n/2^k$

Example: loops (By tracing)

8.

```
for(i=0; i*i<n; i++) {  
    statements;  
}
```

$$k * k \geq n$$

$$k^2 = n$$

$$k = \sqrt{n}$$


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

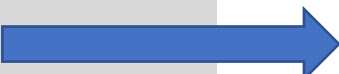
Time Complexity	
i	statements
1	1
2	2^2
3	3^2
4	4^2
5	5^2
...	...
k	k^2

Example: loops (By tracing)

9.

```
for(i=1; i<n; i=i*2) {  
    p++;  
}  
for(j=1; j<p; j=j*2){  
    statements;  
}
```

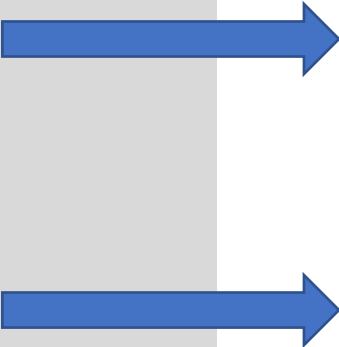

$$p = \log_2 n$$


$$T(n) = \log_2 p$$
$$T(n) = \log_2 \log_2 n$$

Example: While loops (By tracing)

10.

```
i=0;  
while(i<n){  
    Statements;  
    i++;  
}
```



Time Complexity
→ 1
→ n+1
→ n
→ n
f(n) = 3n+2 f(n) = O(n)

Example: While loops (By tracing)

11.

```
a=1;
while(a<b){
Statements;
a=a*2;
}
```

$$a \geq b$$

$$a = 2^k$$

$$2^k \geq b$$

$$2^k = b$$

$$k = \log_2 b = O(\log_2 b)$$

Time Complexity
a
$1 * 2 = 2$
$2 * 2 = 2^2$
$2^2 * 2 = 2^3$
$2^3 * 2 = 2^4$
$n / 2^4$
...
2^k

Example: While loops (By tracing)

12.

```
i=1;
k=1;
while(k<n){
Statements;
k=k+1;
i++;
}
```

$$= 1+2+3+4+\dots+m > n$$

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

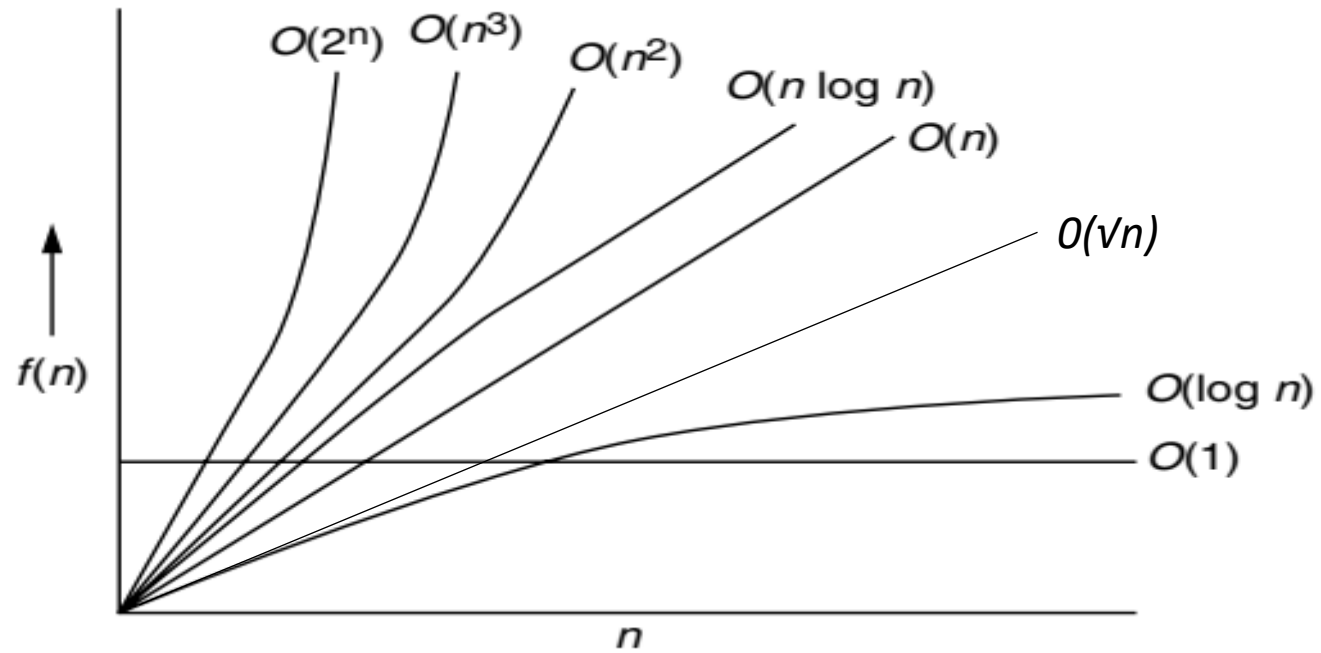
$$= \frac{m^2}{2} + \frac{m}{2} > n$$

$$\cong \frac{m^2}{2} > n$$

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

Time Complexity		
i	k	statements
1	1	1
2	1+1	1
3	2+2	1
4	2+2+3	1
5	1+2+3+4	1
	1
	2+2+3+4+...+m	1

Rate of Growth



Numerical Comparison of Different Algorithms

n	log₂ n	n*log₂n	n²	n³	2ⁿ
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Asymptotic Notations:

- Asymptotic notations have been developed for analysis of algorithms.
- By the word asymptotic means “for large values of n ”
- The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:
 1. Big-OH(O)
 2. Big-OMEGA(Ω),
 3. Big-THETA (Θ)

Big O notation:

- This notation gives the tight upper bound of the given function

- Represented as:

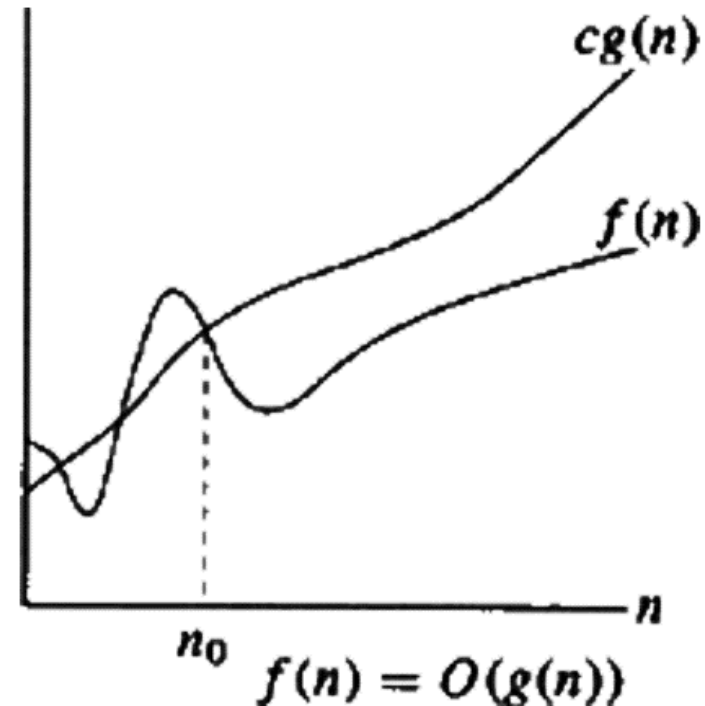
$$f(n) = O(g(n))$$

that means, at larger values of n , upper bound of $f(n)$ is $g(n)$.

Definition:

Big O notation defined as $O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n > n_0\}$$



Big Omega (Ω) notation:

- This notation gives the tight lower bound of the given function

- Represented as:

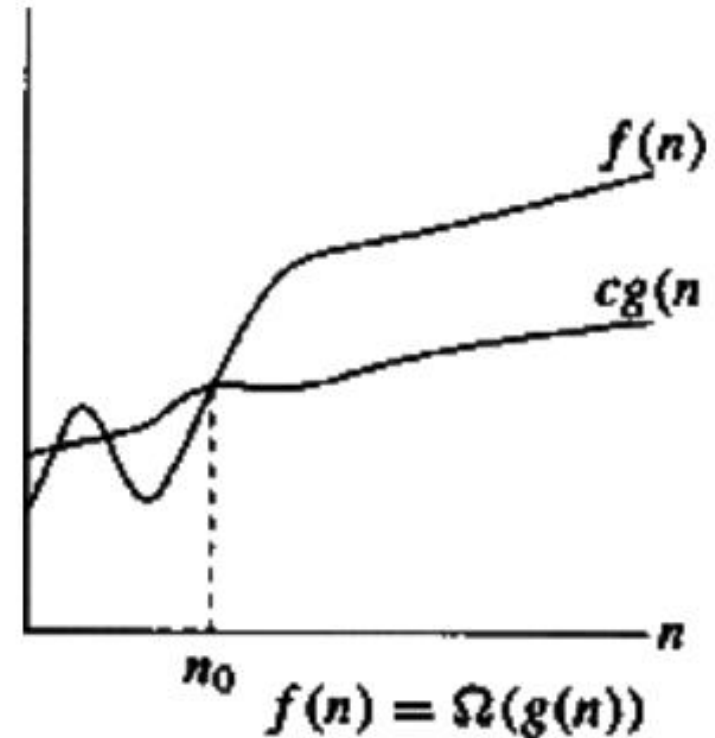
$$f(n) = \Omega(g(n))$$

that means, at larger values of n , lower bound of $f(n)$ is $g(n)$.

Definition:

Big Ω notation defined as $\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

$$0 <= c \cdot g(n) \leq f(n) \text{ for all } n > n_0\}$$



Big Theta (θ) Notation:

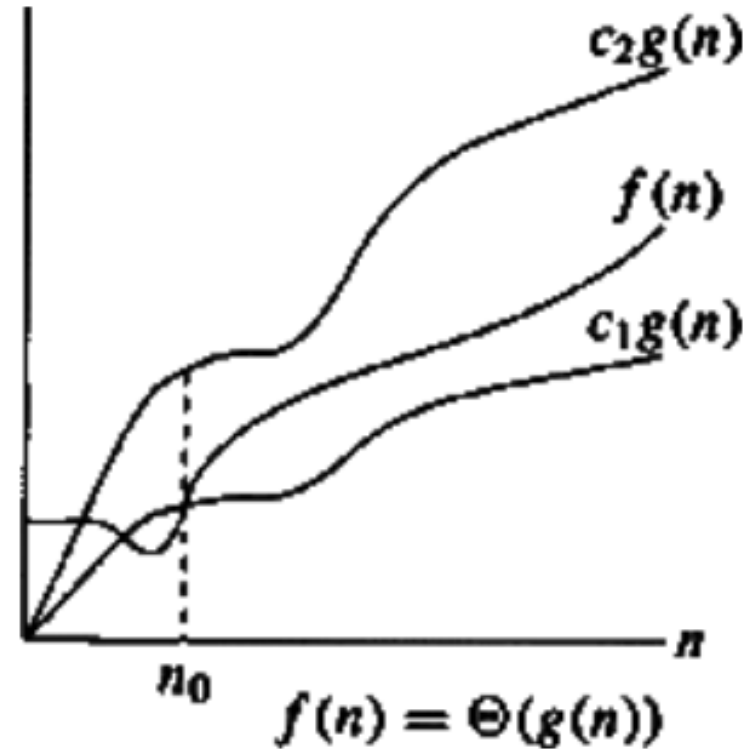
- Average running time of an algorithm is always between lower bound and upper Bound
- Represented as:

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

that means, at larger values of n , lower bound of $f(n)$ is $g(n)$.

Definition:

Big θ notation defined as $\theta(g(n)) = \{f(n): \text{there exist positive constants } c_1 \text{ and } c_2 \text{ and } n_0 \text{ such that}$
$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$
$$\text{for all } n > n_0\}$$



Properties of Asymptotic Notations:

1. Transitivity:

$$f(n) = O(g(n)) \text{ \& } g(n) = O(h(n))$$

$$\rightarrow f(n) = O(h(n))$$

Valid for θ and Ω as well.

2. Reflexivity:

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

Valid for θ and Ω as well.

3. Symmetry:

$$f(n) = \theta(g(n)), \text{ then } g(n) = \theta(f(n))$$

Valid for θ only.

4. Transpose Symmetry:

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

Valid for O and Ω only.

Examples:

$$1. f(n) = n \text{ \& } g(n) = n^2 \text{ \& } h(n) = n^3$$

$$n = O(n^2) ; n^2 = O(n^3), \\ \text{then } n = O(n^3)$$

$$2. f(n) = n^3 = O(n^3) = \theta(n^3) = \Omega(n^3)$$

$$3. f(n) = n^2 \text{ \& } g(n) = n^2 \\ \text{then, } f(n) = \theta(n^2)$$

$$4. f(n) = n \text{ \& } g(n) = n^2 \\ \text{then } n = O(n^2) \text{ \& } n^2 = \Omega(n)$$

Properties of Asymptotic Notations:

Observations:

1. If $f(n) = O(g(n))$ then $a * f(n)$ is $O(g(n))$
2. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) f_2(n) = O(g_1(n) \cdot g_2(n))$
4. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \theta(g(n))$

Recursion:

- Recursion is an ability of an algorithm to repeatedly call itself until a certain condition is met.
- Such condition is called the **base condition**.
- The algorithm which calls itself is called a **recursive algorithm**.
- The recursive algorithms must satisfy the following two conditions:
 1. It must have the **base case**: The value of which algorithm does not call itself and can be evaluated without recursion.
 2. Each recursive call must be to a case that eventually leads toward a base case.

Recursion:

Recurrence Relation:

- An algorithm is said to be recursive if it can be defined in terms of itself.
- The running time of recursive algorithm is expressed by means of recurrence relations.
- A recurrence relation is an equation of inequality that describes a function in terms of its value on smaller inputs.
- It is generally denoted by $T(n)$ where n is the size of the input data of the problem.
- The recurrence relation satisfies both the conditions of recursion, that is, it has both the base case as well as the recursive case.
 - The portion of the recurrence relation that does not contain T is called the base case of the recurrence relation and
 - The portion of the recurrence relation that contains T is called the recursive case of the recurrence relation.

$$T(n) = \begin{cases} d & ; n = 1 \\ T(n-1) + c & ; n > 1 \end{cases}$$

Recursion:

Recurrence Relation:

There are various methods to solve recurrence:

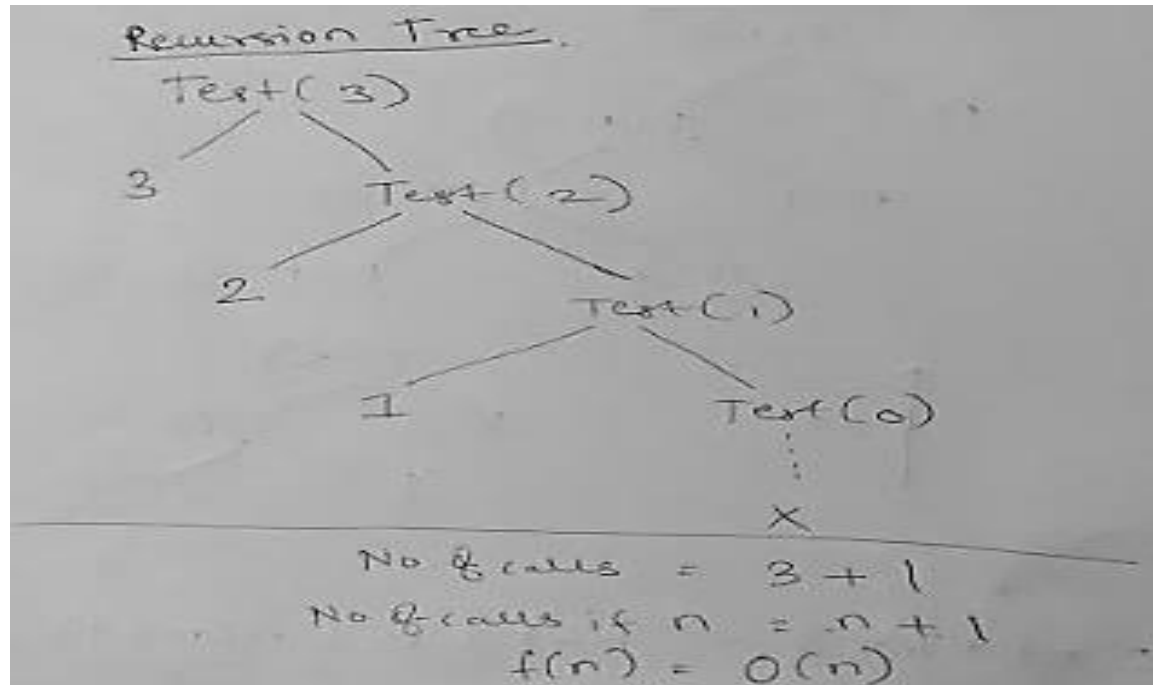
1. Substitution Method
2. Recurrence Tree
3. Master Method/ Master's Theorem

Recurrence Relation for Decreasing Function:(Recursion Tree)

```
Void Test(int n)  →  $T(n)$ 
{
  If(n > 0) → 1
  {
    printf ("%d",n); → 1
    Test(n - 1); →  $T(n-1)$ 
  }
}
```

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

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + 1 & ; n > 0 \end{cases}$$



Recurrence Relation for Decreasing Function: (substitution Method)

$$T(n) = T(n-1) + 1 \dots\dots \text{Eq 1}$$

If $T(n) = T(n-1) + 1$, Then $T(n-1) = T(n-2) + 1$

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

Substituting $T(n-1)$ in Eq 1

$$T(n) = [T(n-2) + 1] + 1$$

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

Substituting $T(n-2)$ in above eq

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

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

.....

$$T(n) = T(n-k) + k \text{ (continue for } k \text{ times)}$$

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + 1 & ; n > 0 \end{cases}$$

$$T(n) = T(n-k) + k$$

Assume $n-k = 0$ (base condition)

Therefore $k = n$

Substituting k with n in above eq

$$T(n) = T(n-n) + n$$

$$T(n) = T(0) + n$$

$$T(n) = 1 + n = O(n)$$

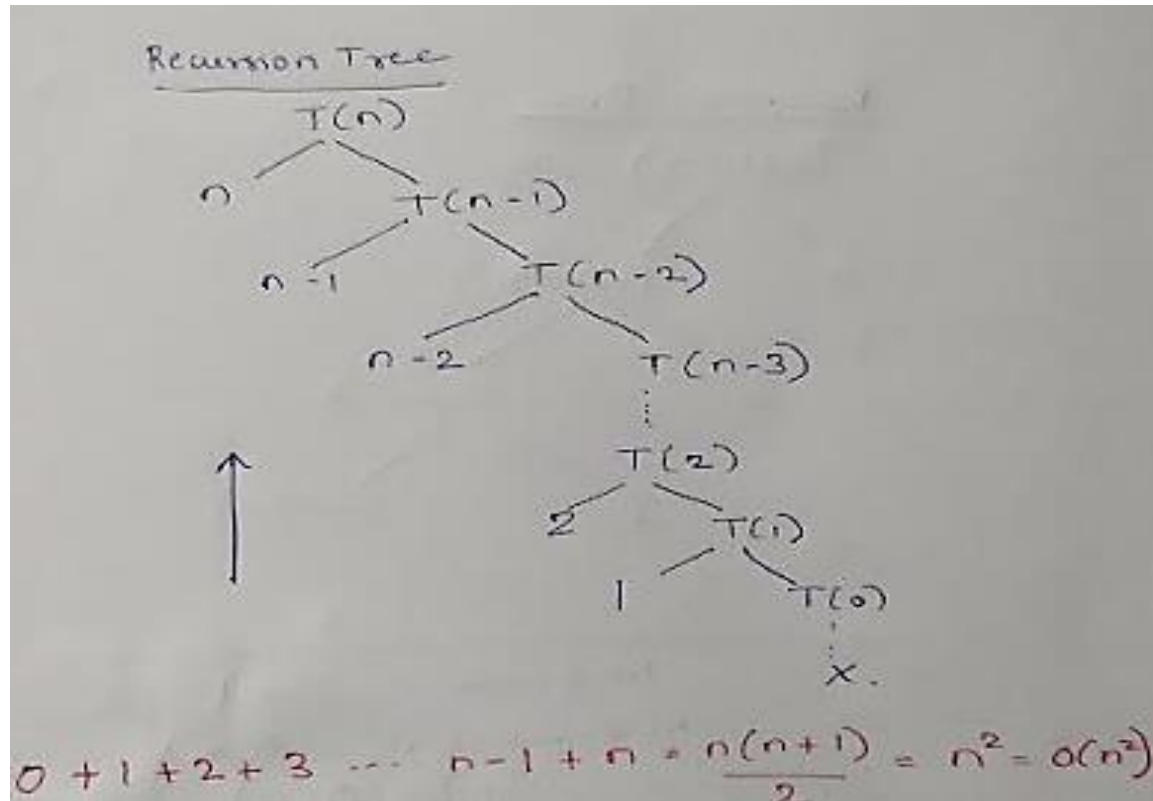
Recurrence Relation for Decreasing Function: (Recursion Tree)

```
Void Test(int n)  —————→  $T(n)$ 
{
    If(n > 0)  —————→ 1
    {
        for(i=0; i<n; i++) {  —————→  $n+1$ 
            printf ("%d",n);  —————→  $n$ 
        }
        Test(n - 1);  —————→  $T(n-1)$ 
    }
}
```

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

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

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + n & ; n > 0 \end{cases}$$



Recurrence Relation for Decreasing Function: (substitution Method)

$$T(n) = T(n-1) + n \dots \text{Eq 1}$$

If $T(n) = T(n-1) + n$, Then $T(n-1) = T(n-2) + n-1$

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

Substituting $T(n-1)$ in Eq 1

$$T(n) = [T(n-2) + n-1] + n$$

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

Substituting $T(n-2)$ in above eq

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

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

.....

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$$

(continue for k times)

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + n & ; n > 0 \end{cases}$$

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$$

Assume $n-k = 0$ (base condition)

Therefore $k = n$

Substituting k with n in above eq

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

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

$$T(n) = T(0) + 1 + 2 + 3 + \dots + (n-1) + n$$

$$T(n) = 1 + n(n+1)/2$$

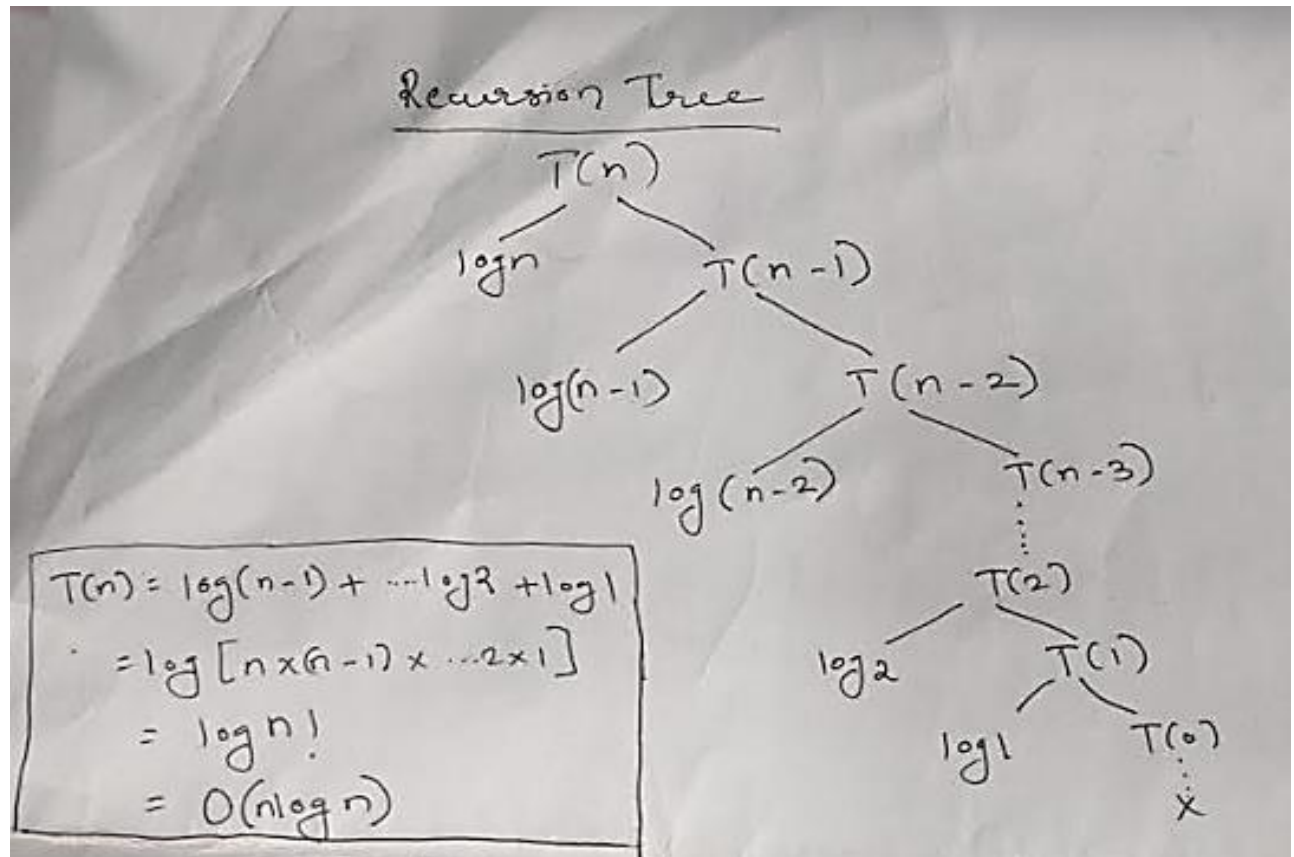
$$T(n) = O(n^2)$$

Recurrence Relation for Decreasing Function:(Recursion Tree)

```
Void Test(int n)  —————→  $T(n)$ 
{
    If(n > 0)
    {
        for(i=0; i<n; i=i*2) {
            printf ("%d",i);  —————→  $\log n$ 
        }
        Test(n - 1);  —————→  $T(n-1)$ 
    }
}
```

$T(n) = T(n-1) + \log n$

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + \log n & ; n > 0 \end{cases}$$



Recurrence Relation for Decreasing Function: (substitution Method)

$$T(n) = T(n-1) + \log n \dots \text{Eq 1}$$

If $T(n) = T(n-1) + \log n$, Then $T(n-1) = T(n-2) + \log(n-1)$

And $T(n-2) = T(n-3) + \log(n-2)$

Substituting $T(n-1)$ in Eq 1

$$T(n) = [T(n-2) + \log(n-1)] + \log n$$

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

Substituting $T(n-2)$ in above eq

$$T(n) = [T(n-3) + \log(n-2)] + \log(n-1) + \log n$$

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

.....

$$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \dots + \log(n-1) + \log n \text{ (continue for } k \text{ times)}$$

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + \log n & ; n > 0 \end{cases}$$

$$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \dots + \log(n-1) + \log n$$

Assume $n-k = 0$ (base condition)

Therefore $k = n$

Substituting k with n in above eq

$$T(n) = T(n-n) + \log(n-(n-1)) + \log(n-(n-2)) + \dots + \log(n-1) + \log n$$

$$T(n) = T(n-n) + \log(n-n+1) + \log(n-n+2) + \dots + \log(n-1) + \log n$$

$$T(n) = T(0) + \log[1 \cdot 2 \cdot 3 \dots (n-1) \cdot n]$$

$$T(n) = 1 + \log n! \text{ (order of } n! = O(n^n) \text{)}$$

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

Recurrence Relation for Decreasing Function(Observations for no coefficients)

$$T(n) = T(n-1) + 1 \text{ — } O(n)$$

$$T(n) = T(n-1) + n \text{ — } O(n^2)$$

$$T(n) = T(n-1) + \log n \text{ — } O(n \log n)$$

$$T(n) = T(n-1) + n^2 \text{ — } O(n^3)$$

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

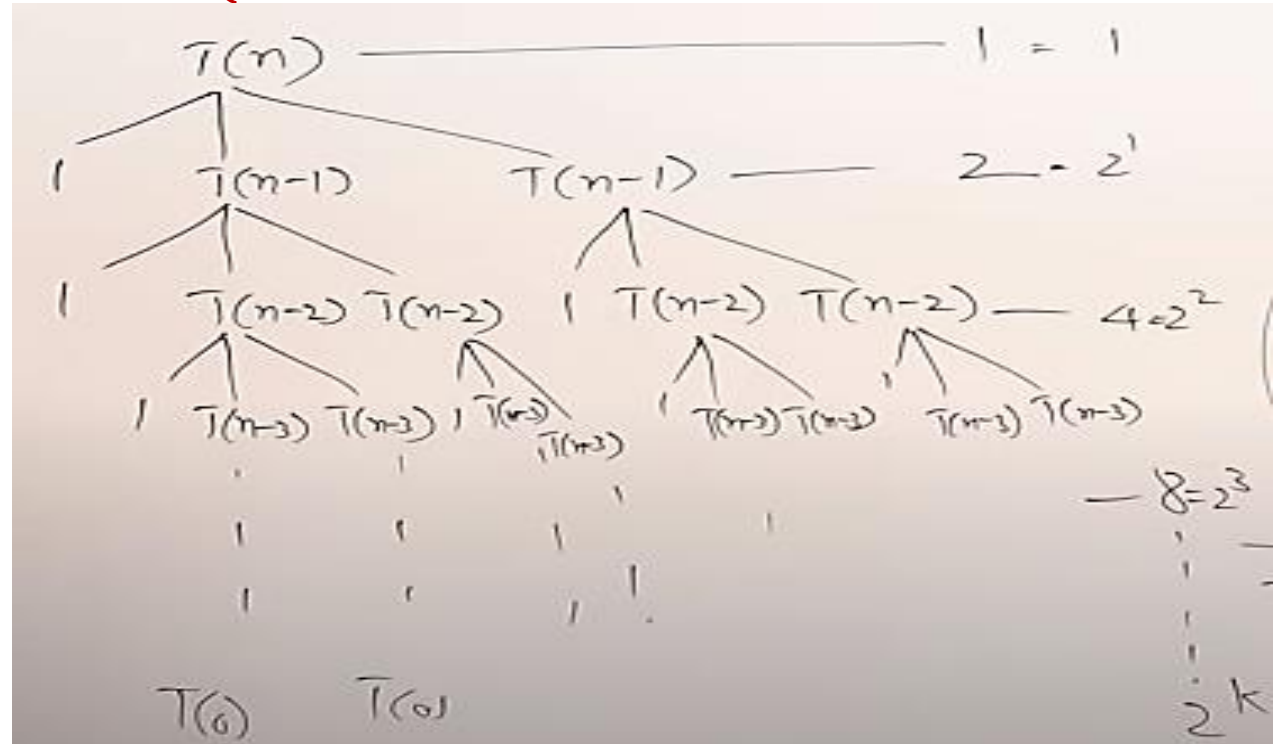
$$T(n) = T(n-100) + n \text{ — } O(n^2)$$

Recurrence Relation for Decreasing Function with coefficient:(Recursion Tree)

```
Void Test(int n)  —————→  $T(n)$ 
{
    If(n > 0)
    {
        printf ("%d",n);  —————→ 1
        Test(n - 1);      —————→  $T(n-1)$ 
        Test(n - 1);      —————→  $T(n-1)$ 
    }
}
```

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

$$T(n) = \begin{cases} 1 & ; n = 0 \\ 2T(n-1) + 1 & ; n > 0 \end{cases}$$



$$\begin{aligned} &= 1 + 2 + 2^2 + 2^3 + \dots + 2^k \\ &= 2^{k+1} - 1 \\ &= O(2^n) \end{aligned}$$

Recurrence Relation for Decreasing Function with coefficient:(substitution method)

$$T(n) = 2T(n-1) + 1 \dots \text{Eq 1}$$

$$T(n) = 2[T(n-2) + 1] + 1 \quad (T(n-1) \text{ substitution})$$

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

$$T(n) = 2^2[2T(n-3) + 1] + 2 + 1 \quad (T(n-1) \text{ substitution})$$

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

$$T(n) = 2^kT(n-k) + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2 + 1$$

Assume $n-k = 0$ (base condition)

Therefore $k = n$

Substituting k with n in above eq

$$T(n) = 2^nT(0) + 1 + 2 + \dots + 2^{n-1}$$

$$T(n) = 2^n * 1 + 2^n - 1$$

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

$$T(n) = \begin{cases} 1 & ; n = 0 \\ 2T(n-1) + 1 & ; n > 0 \end{cases}$$

Recurrence Relation for Decreasing Function(Observations for with coefficients)

- $T(n)=T(n-1)+1$ $O(n)$
- $T(n)=T(n-1)+n$ $O(n^2)$
- $T(n)=T(n-1)+\log n$ $O(n\log n)$
- $T(n)=2T(n-1)+1$ $O(2^n)$
- $T(n)=3T(n-1)+1$ $O(3^n)$.
- $T(n)=2T(n-1)+n$ $O(n2^n)$
- $T(n)=2T(n-2)+1$ $O(2^{n/2})$

Master's Theorem (for Decreasing Function)

Let $T(n)$ be a function defined on positive n

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ aT(n-b) + f(n) & \text{if } n > 1 \end{cases}$$

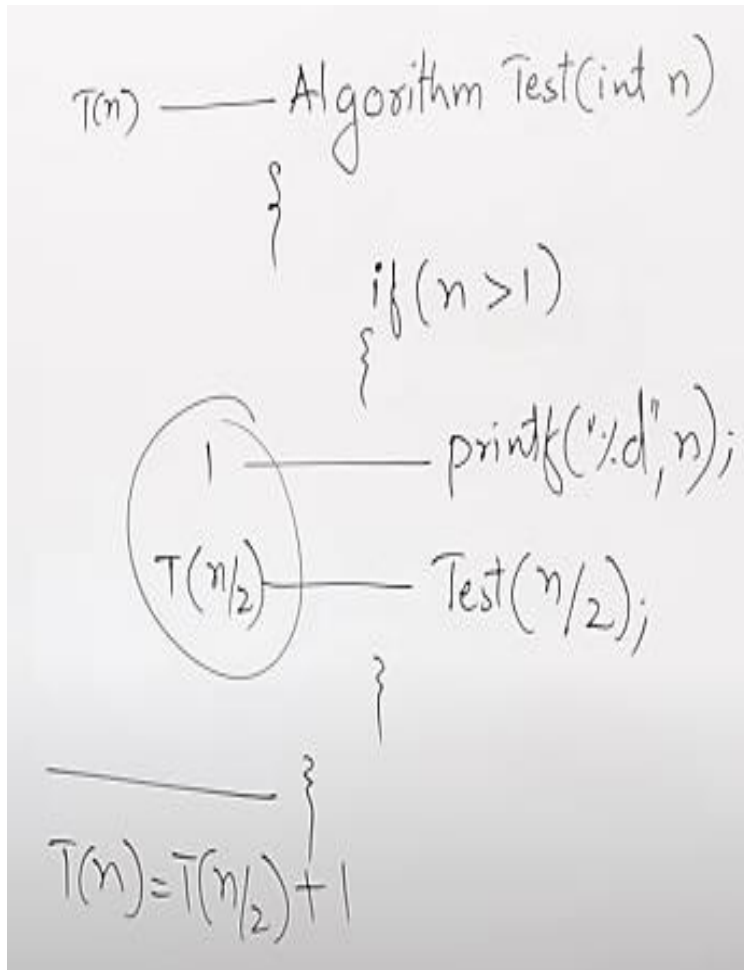
for some constants $c, a > 0, b > 0$,
and $f(n) = O(n^k)$, where $k \geq 0$

$$T(n) = O(n^k) \quad \text{if } a < 1$$

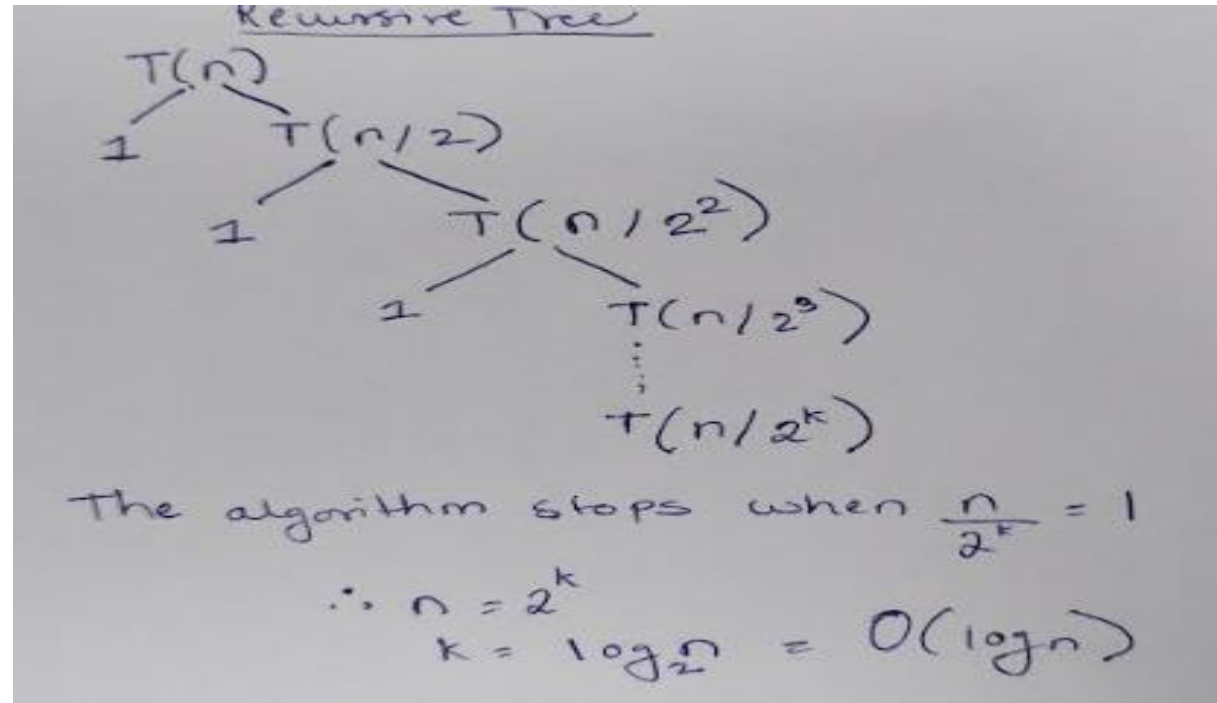
$$= O(n^{k+1}) \quad \text{if } a = 1$$

$$= O\left(n^k \cdot a^{\frac{n}{b}}\right) \quad \text{if } a > 1$$

Recurrence Relation for Dividing Function:(Recursion Tree)



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



Recurrence Relation for Dividing Function: (substitution Method)

$$T(n) = 2T(n/2) + 1 \dots \text{Eq 1}$$

If $T(n) = 2T(n/2) + 1$, Then $T(n/2) = T(n/2^2) + 1$

And $T(n/2^2) = T(n/2^3) + 1$

Substituting $T(n/2)$ in Eq 1

$$T(n) = [T(n/2^2) + 1] + 1$$

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

Substituting $T(n/2^2)$ in above eq

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

.....

$$T(n) = T(n/2^k) + k$$

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

$$T(n) = T(n/2^k) + k$$

Assume $n/2^k = 1$ (base condition)

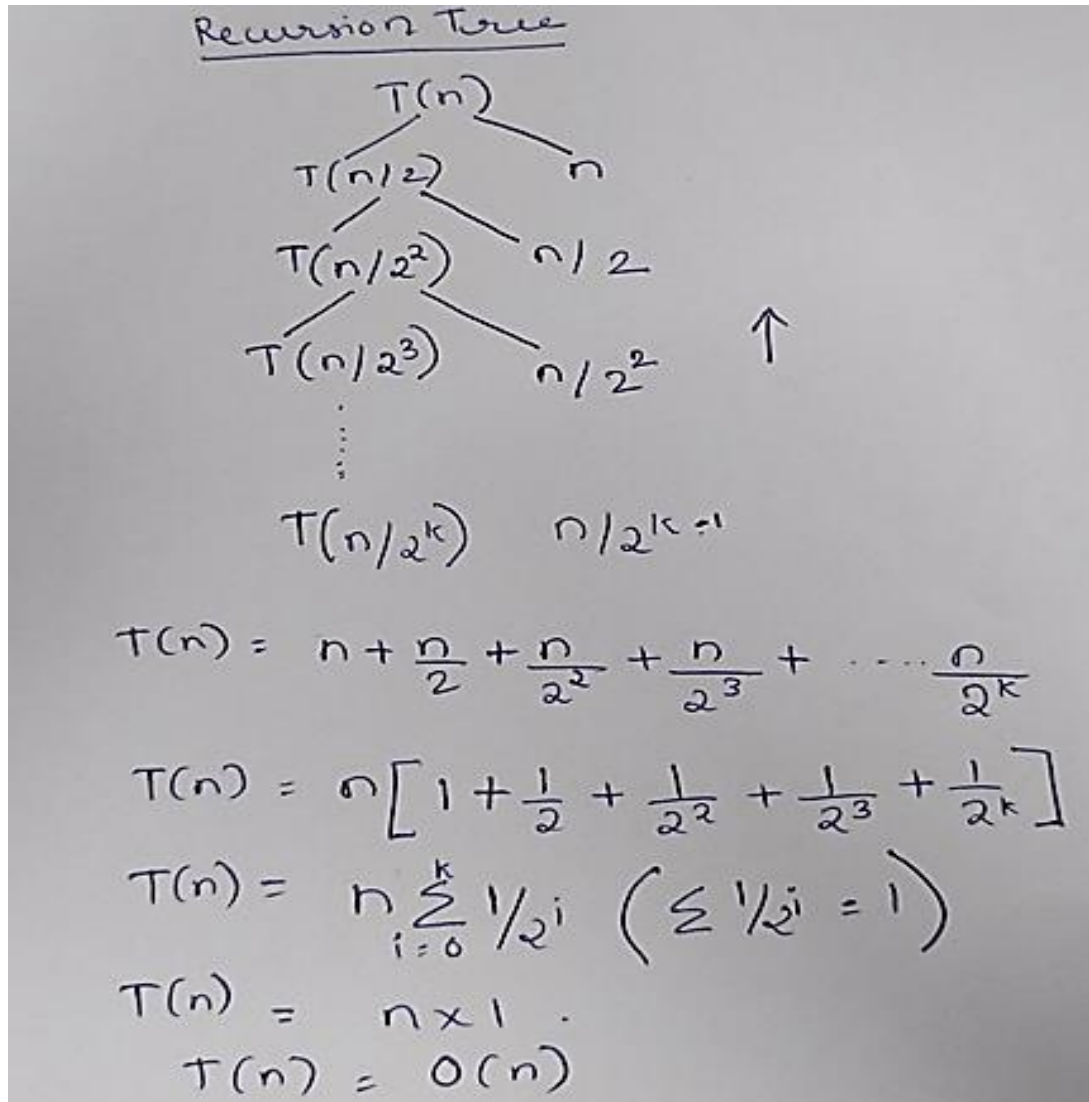
Therefore $n = 2^k$, $k = \log n$

$$T(n) = T(1) + \log n$$

$$T(n) = T(1) + \log n$$

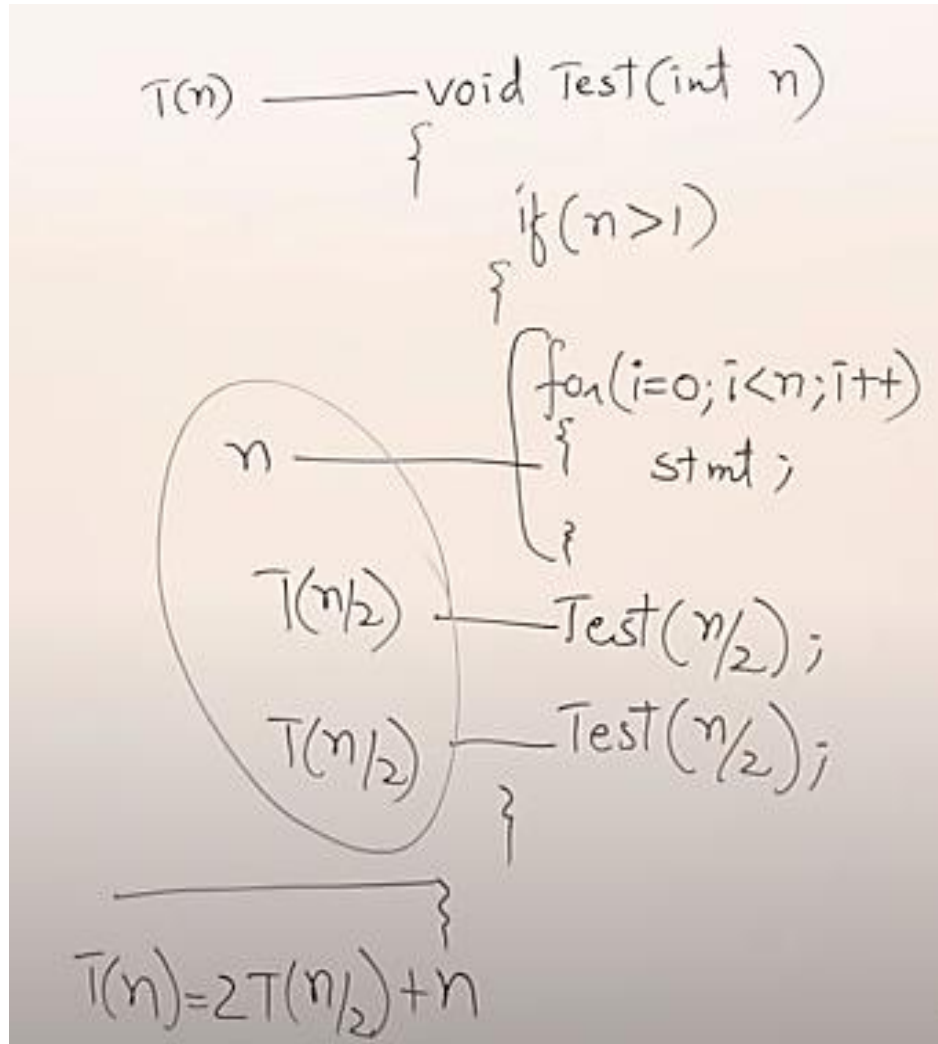
$$O(\log n)$$

Recurrence Relation for Dividing Function: (Recursion Tree)

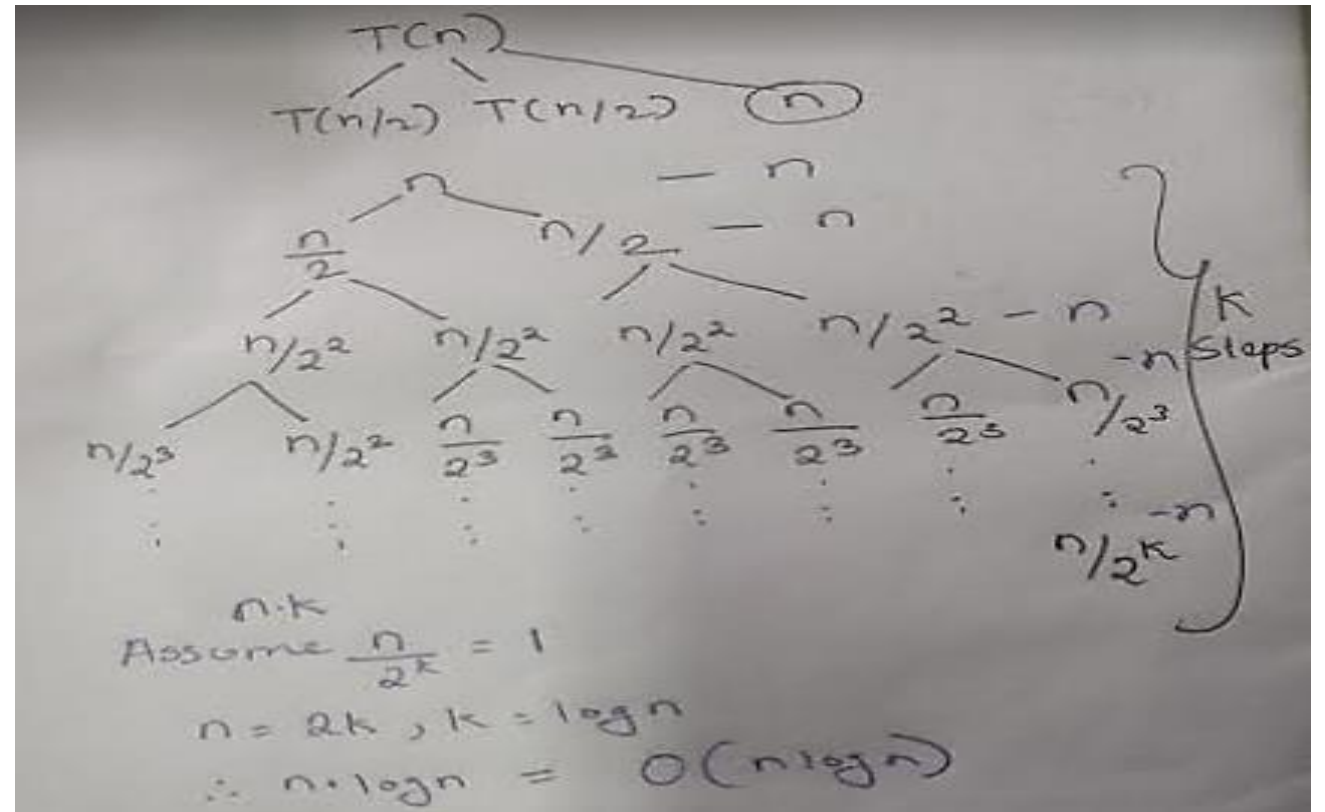


$$T(n) = \begin{cases} 1 & ; n = 1 \\ T(n/2) + n & ; n > 1 \end{cases}$$

Recurrence Relation for Dividing Function: (Recursion Tree)



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



Master's Theorem (for Dividing Function)

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ \theta(1) & n = 1 \end{cases}$$

Where,

$$f(n) = \Theta(n^k \log^p n)$$

and

$$a \geq 1 ; b > 1 ; k \geq 0$$

and **p** is a real number

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ \theta(1) & n = 1 \end{cases}$$

Case 1: If $a > b^k$ or $\log_b a > k$

then,

$$T(n) = \Theta(n^{\log_b a})$$

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

Case 2: If $a = b^k$ or $\log_b a = k$

then,

A.] If $p > -1$, then

$$T(n) = \Theta(n^{\log_b a} \log^{p+1} n) \Rightarrow \theta(n^k \log^{p+1} n)$$

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

Case 2: If $a = b^k$ or $\log_b a = k$

then,

B.]. If $p = -1$, then

$$T(n) = \Theta(n^{\log_b a} \log \log n) \Rightarrow \theta(n^k \log \log n)$$

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

Case 2: If $a = b^k$ or $\log_b a = k$

then,

C.] If $p < -1$, then

$$T(n) = \Theta(n^{\log_b a}) \Rightarrow \theta(n^k)$$

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

Case 3: If $a < b^k$ or $\log_b a < k$

A.] If $p \geq 0$ then

$$T(n) = \Theta(n^{\log_b a} \log^p n) \Rightarrow \theta(n^k \log^p n)$$

B.] If $p < 0$ then

$$T(n) = \Theta(n^{\log_b a}) \Rightarrow \theta(n^k)$$

Master's Theorem:

1. $T(n) = 2T(\sqrt{n}) + \log n$

Exercise:

1. $T(n) = 2T\left(\frac{n}{2}\right) + n^3 \log n$

2. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n^3}{\log^2 n}$

Recurrence Relation for Root Function: (Recursion Tree)

$$T(n) = \begin{cases} 1 & ; n = 2 \\ T(\sqrt{n}) + 1 & ; n > 2 \end{cases}$$

```

T(n) — void Test(int n)
        {
            if (n > 2)
            {
                stmt;
                T(√n) — Test(√n);
            }
        }
    
```

$$T(n) = T(\sqrt{n}) + 1$$

Root Function

$$T(n) = \begin{cases} 1 & n = 2 \\ T(\sqrt{n}) + 1 & n > 2 \end{cases}$$

$$T(n) = T(\sqrt{n}) + 1$$

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

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

$$T(n) = T(n^{\frac{1}{2^3}}) + 3 \quad \text{--- (3)}$$

$$\vdots$$

$$T(n) = T(n^{\frac{1}{2^k}}) + k \quad \text{--- (4)}$$

Assume $n = 2^m$

$$T(2^m) = T(2^{\frac{m}{2^k}}) + k$$

Assume $T(2^{\frac{m}{2^k}}) = T(2^1)$

$$\therefore \frac{m}{2^k} = 1$$

$$m = 2^k \text{ and } k = \log_2 m$$

$$\therefore n = 2^m \quad m = \log_2 n$$

$$k = \log \log n$$

$$O(\log \log n)$$