Analysis of Algorithms

SY Computer

Even 2022-23

Module1:Introduction to analysis of algorithm

- Performance analysis
- space and time complexity
- Growth of function-BigOh, Omega, Theta Notation.
- Solving Recurrence Problems by Substitution Method, Recursion Tree Method, Masters Method.

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

<u>Rules:</u>

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:

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

4. Ignore constant multipliers

   $6n^3 \Longrightarrow n^3$

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

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[][])	Time Complexity	Space Complexity
`{ int c[][];`		$a[][] = n^2$ words
`for(i=0; i<n; i++) {`		$b[][] = n^2$ words
`for(j=0; j<n; j++) {`	→ $n+1$	$c[][] = n^2$ words
`c[i][j] = a[i][j] + b[i][j]`	→ $n \times (n+1)$	$i = 1$ word
`}`	→ $n \times n$	$j = 1$ word
`}`		$n = 1$ word
`}`		→ $3n^2 + 3$ Space complexity $= O(n^2)$
	→ $n+1+n^2+n+n^2$ → $2n^2 + 2n + 1$ $f(n) = O(n^2)$	

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

Algorithm matMul (int a[][], int b[][])	Time Complexity	Space Complexity
`{ int c[][];`		$a[][] = n^2$ words
`for(i=0; i<n; i++) {`	$\rightarrow n+1$	$b[][] = n^2$ words
`for(j=0; j<n; j++) {`	$\rightarrow n\times(n+1)$	$c[][] = n^2$ words
`c[i][j] = 0;`	$\rightarrow n\times n$	$i = 1$ word
`for(k=0; k<n; k++){`	$\rightarrow n\times n\times(n+1)$	$j = 1$ word
`c[i][j] = a[i][j] * b[i][j]`	$n\times n\times n$	$k=1$ word
`}`		$n = 1$ word
`}`	$\rightarrow n+1+n^2+n+n^2+n^3+1+n^3+n^2$ $\rightarrow 2n^3+3n^2+2n+1$ $f(n) = O(n^3)$	$\rightarrow 3n^2+4$ 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;
    }
}
```

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

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

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;
  }
}
```

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	???

=1+2+3+4+…+k>n

$= \ k(k+1)/2 > n$

$= \ k^2 + k/2 \ > n$

$\cong \ k^2 > n$

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

Example: loops (By tracing)

6.

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

$$i >= n$$
$$i = 2^k$$
$$2^k >= n$$
$$2^k = n$$

$$\boldsymbol{k = log_2 n \;=\; O(log_2 n)}$$

Time Complexity	
i	**statements**
$1*2^0$	1
$1*2$	1
$1*2*2$	1
$1*2*2*2$	1
…	…
…	…
2^k	1

Example: loops (By tracing)

7.

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

$$i<1$$

$$n/2^k = 1$$

$$n=2^k$$

$$\boldsymbol{k = log_2 n \ = \ O(log_2 n)}$$

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;
  }
}
```

$$k * k >= n$$

$$k^2 = n$$

$$k=\sqrt{n}$$

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

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;
}
```

$$a >= b$$

$$a = 2^k$$

$$2^k \geq b$$

$$2^k = b$$

$$\boldsymbol{k = log_2 b \; = \; O(log_2 b)}$$

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++;
}
```

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

=1+2+3+4+…+m>n

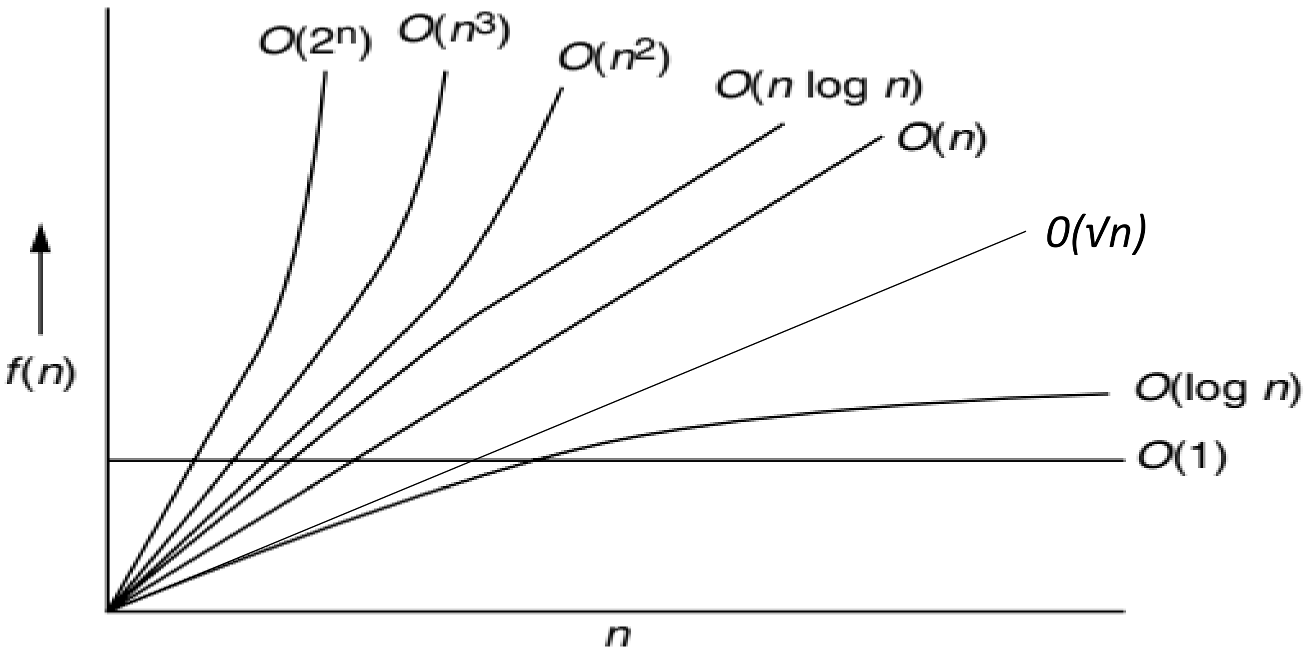
$= \ m(m+1)/2 > n$

$= \ m^2 + m/2 \ > n$

$\cong \ m^2 > n$

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

Rate of Growth



Numerical Comparison of Different Algorithms

n	log2 n	n*log2n	n^2	n^3	2^n
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:

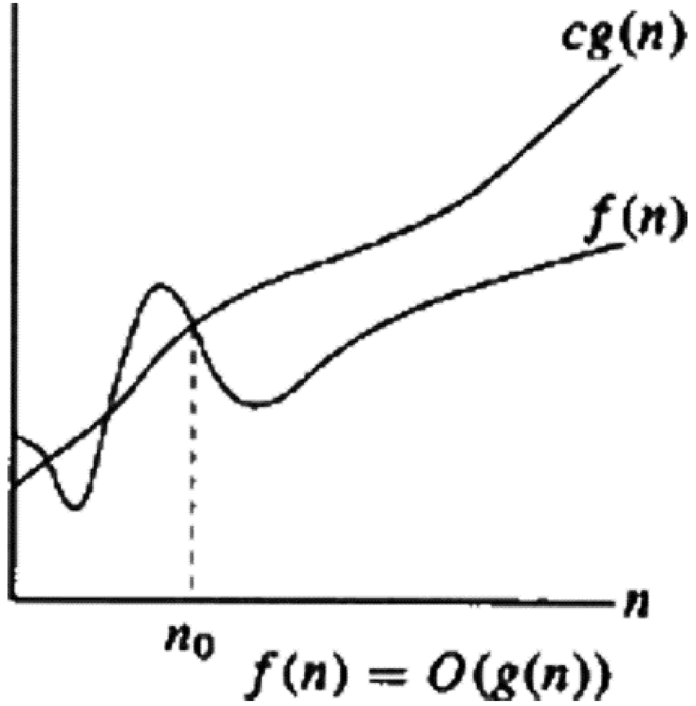
1. Big–OH(O)
2. Big–OMEGA(Ω),
3. Big–THETA (Θ)

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:

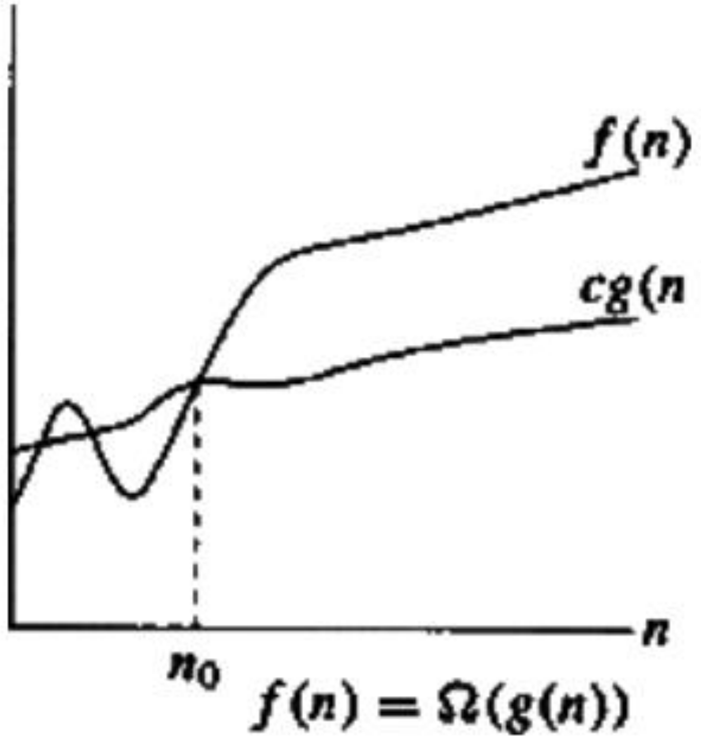
Big O notation defined as $O(g(n)) =$ $\{f(n)$: there exist positive constants c and no such that
$0 <= f(n) <= c.g(n) for\ all\ n > n_0\}$

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 $\Omega(g(n))$ = $\{f(n)$: there exist positive constants c and no such that

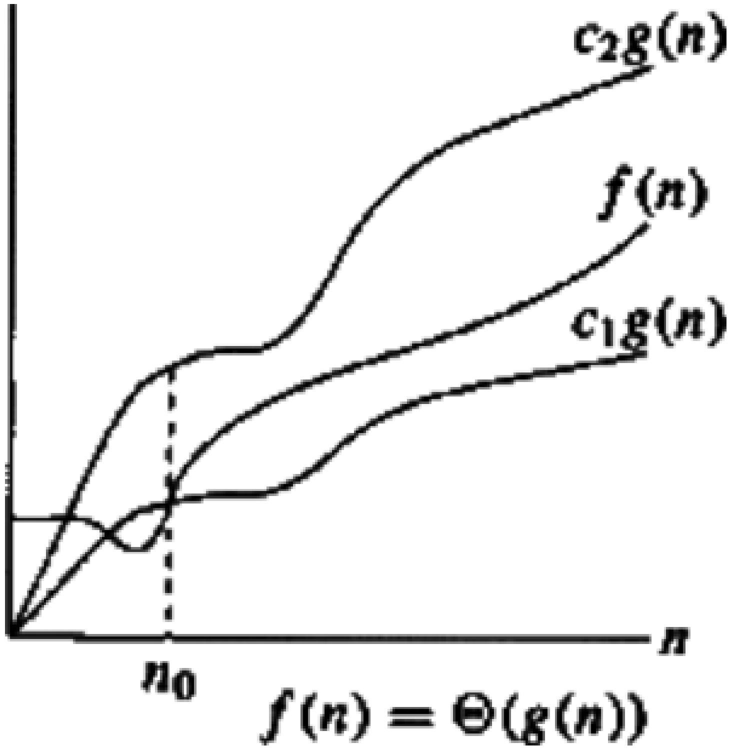
$$0 <= c.g(n) \leq= f(n) \quad for\ all\ n > n_0\}$$

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:

Bigθ notation defined as $\theta(g(n))$ = $\{f(n)$: there exist positive constants c_1 and c_2 and n_o such that

$$0 \leq c_1.\, g(n) \leq f(n) \leq c_2.\, g(n) \quad for\ all\ n > n_0\}$$

Properties of Asymptotic Notations:

1. Transitivity:

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

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

Valid for θ and Ω as well.

2. Reflexivity:

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

Valid for θ and Ω as well.

3. Symmetry:

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

Valid for θ only.

4. Transpose Symmetry:

$f(n) = O\big(g(n)\big)$ then $g(n) = \Omega(f(n))$

Valid for O and Ω only.

Examples:

----

1. $f(n) = n$ & $g(n) = n^2$ & $h(n)=n^3$

   $n = O(n^2)$; $n2=O(n^3)$,

   then $n = O(n^3)$

----

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

----

3. $f(n) = n^2$ & $g(n) = n^2$

   then, $f(n) = \theta(n^2)$

----

4. $f(n) = n$ & $g(n) = n^2$

   then $n = O(n^2)$ & $n^2= \Omega(n)$

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 $f_1(n)\, f_2(n) = O(g_1(n).\, g_2(n))$

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 $\boldsymbol{T(n)}$ where $\boldsymbol{n}$ 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.

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

Recursion:

Recurrence Relation:

There are various methods to solve recurrence:

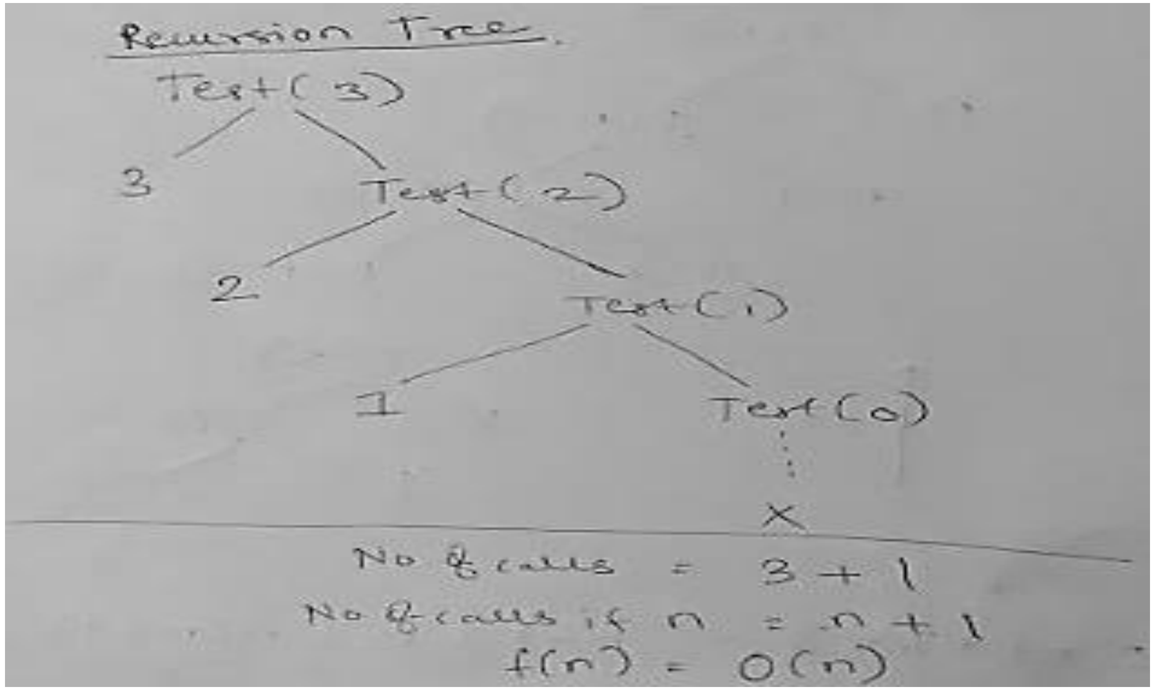
1. Substitution Method
2. Recurrence Tree
3. Master Method/ Master's Theorem

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: (Backward substitution Method)

T(n) = T(n-1) + 1 ….. 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 *(continue for k 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)

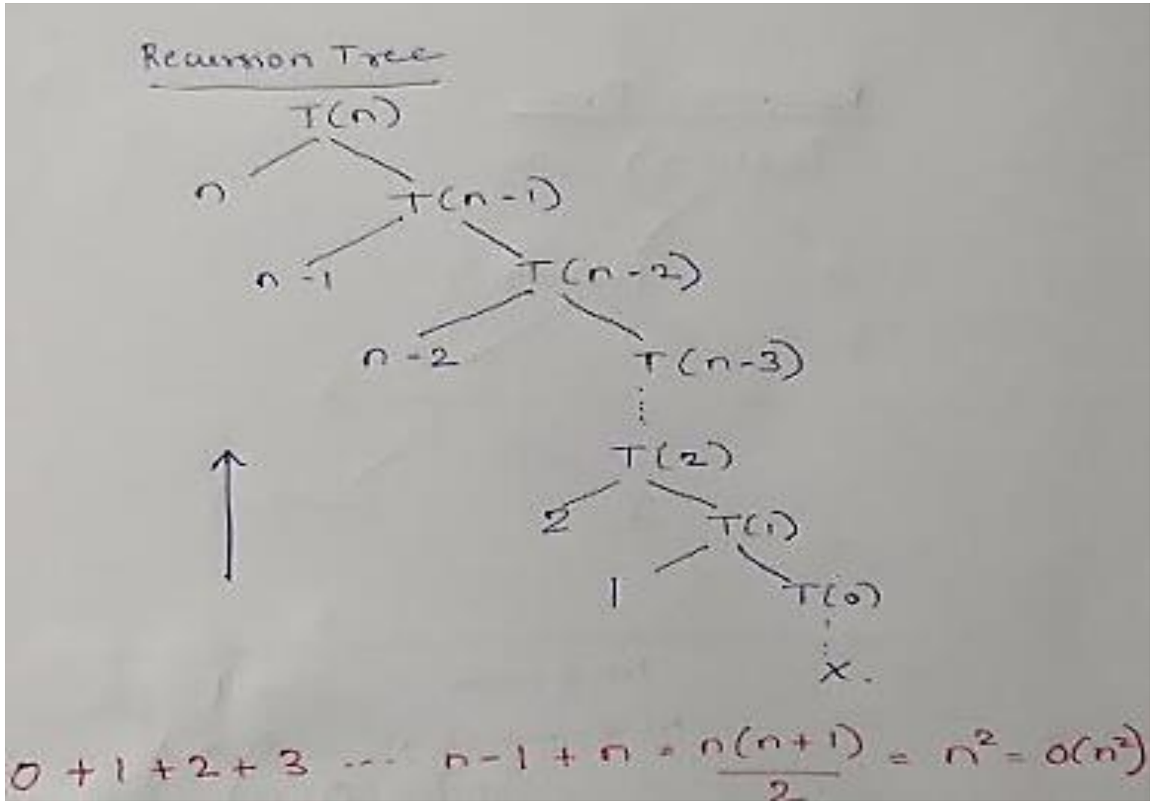
Recurrence Relation for Decreasing Function: (Recursion Tree)

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: (Backward substitution Method)

T(n) = T(n-1) + n ….. 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)) +…..(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)) +…..(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)) +…..(n-1)+n

T(n) = T(n-n) + (n-n+1)) +(n-n+2) +…..(n-1)+n

T(n) = T(0) + 1+2+3…(n-1)+n

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

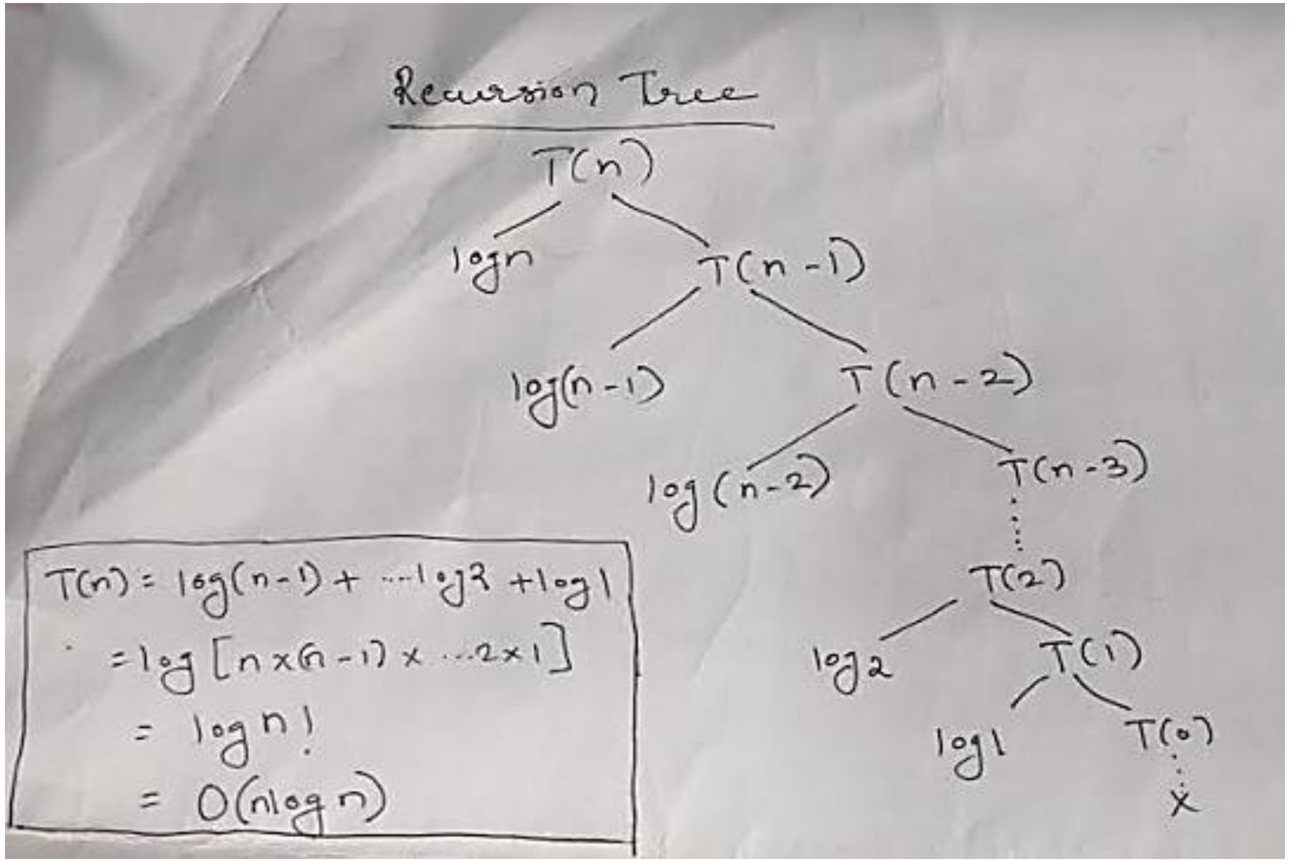
T(n) = O(n^2)

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: (Backward substitution Method)

$T(n) = T(n-1) + \log n$ ….. 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

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

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

…………………………….

$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \ldots \log(n-1) + \log n$ *(continue for k times)*

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

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

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

Therefore $k = n$

Substituting k with n in above eq

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

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

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

$T(n) = 1 + \log n!$ *(order of $n! = O(n^n)$)*

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

Recurrence Relation for Decreasing Function(Observations for no coefficients)

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

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

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

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

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

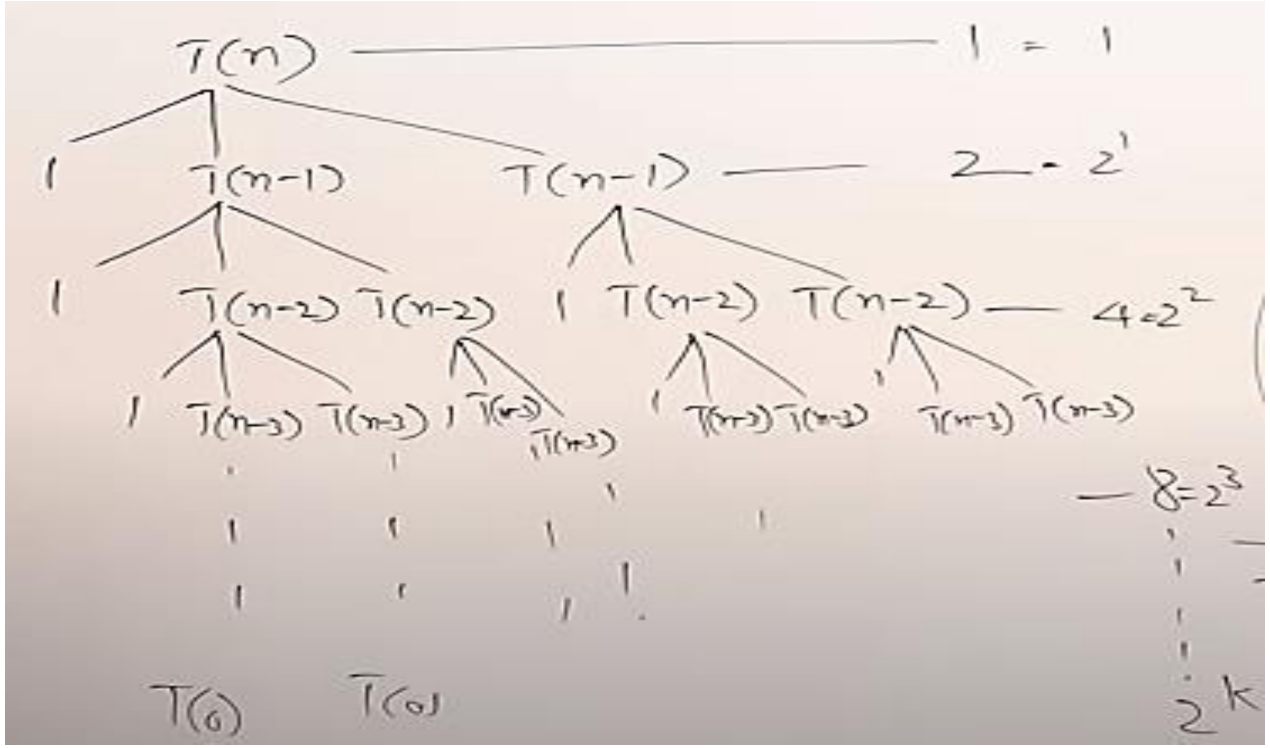
$$T(n) = T(n-100) + n \text{ --- } O(n^2)$$

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

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



$=1+2+2^2+2^3+....+2^k$

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

$= O(2^n)$

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

T(n) = 2T(n-1) + 1 ….. Eq 1

T(n) = 2[T(n-2) + 1]+1   *(T(n-1) substitution)*

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

T(n) = 2^2[2T(n-3) +1]+2+1   *(T(n-1) substitution)*

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

T(n) = 2^kT(n-k)+ 2^{k-1} + 2^{k-2} +………+ 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+…. 2^k-1

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

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

for some constants c, a>0, b>0,

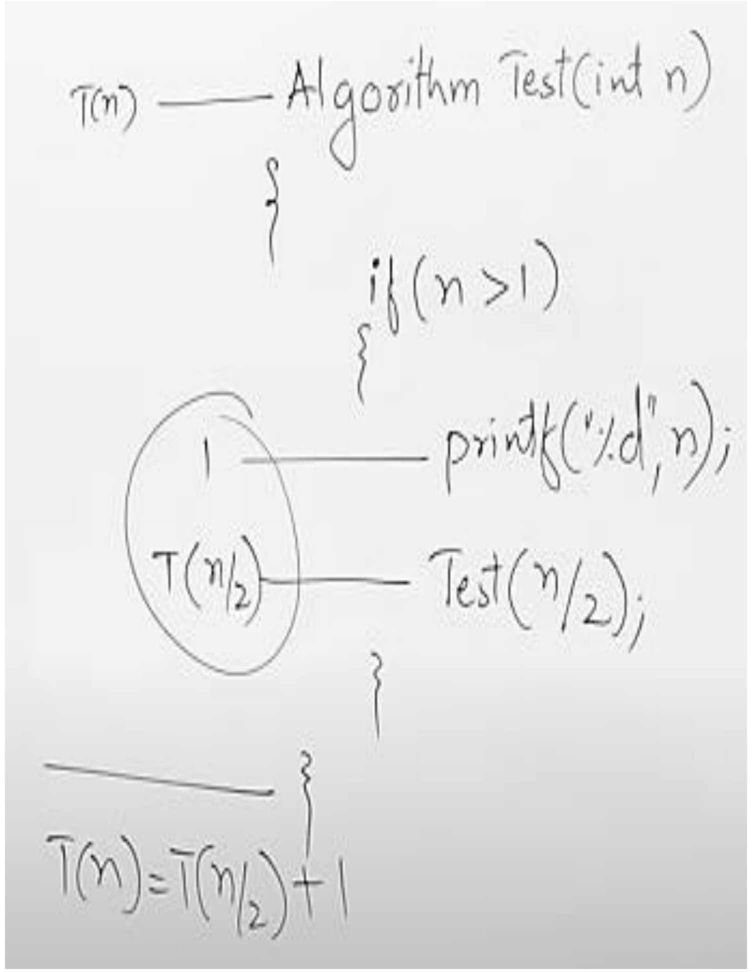
and $f(n) = O(n^k)$,where k>=0

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

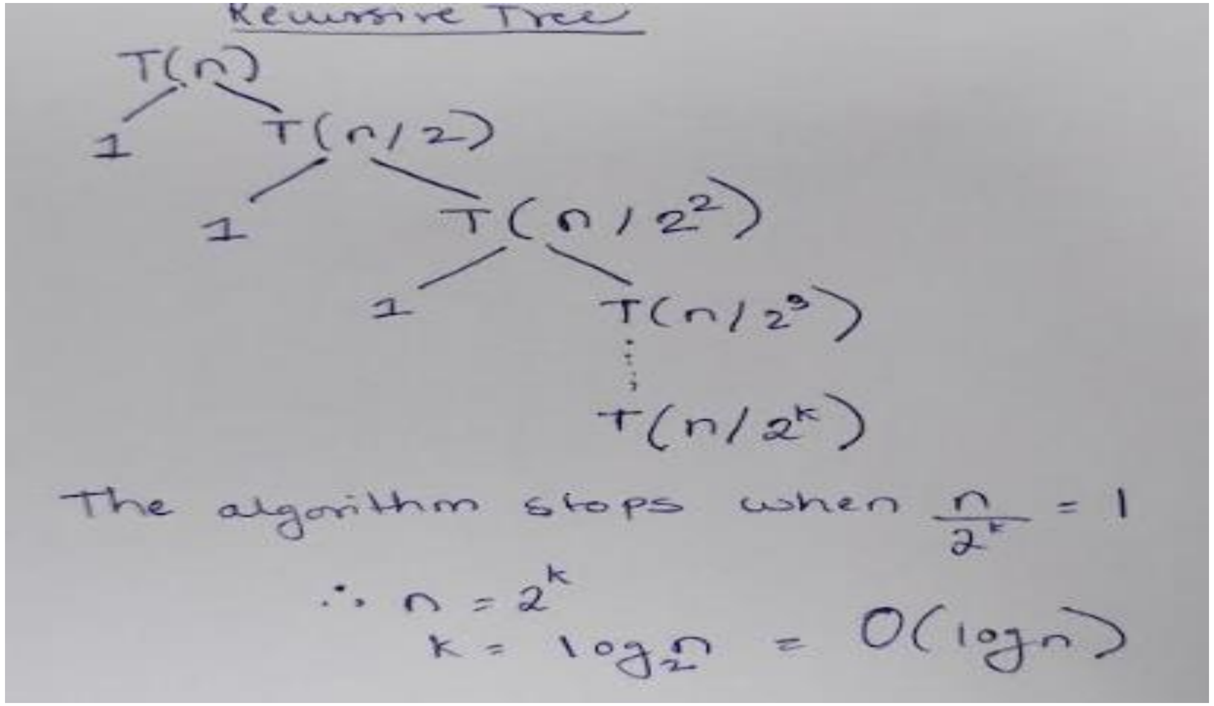
$$= O(n^{k+1}) \quad if\ a = 1$$

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

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: (Backward substitution Method)

$T(n) = T(n/2) + 1$ ….. 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)

Recursion Tree

$T(n)$

$T(n/2)$ n

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

$T(n/2^3)$ $n/2^2$ $\uparrow$

$\vdots$

$T(n/2^k)$ $n/2^k = 1$

$$T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots \frac{n}{2^k}$$

$$T(n) = n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^k}\right]$$

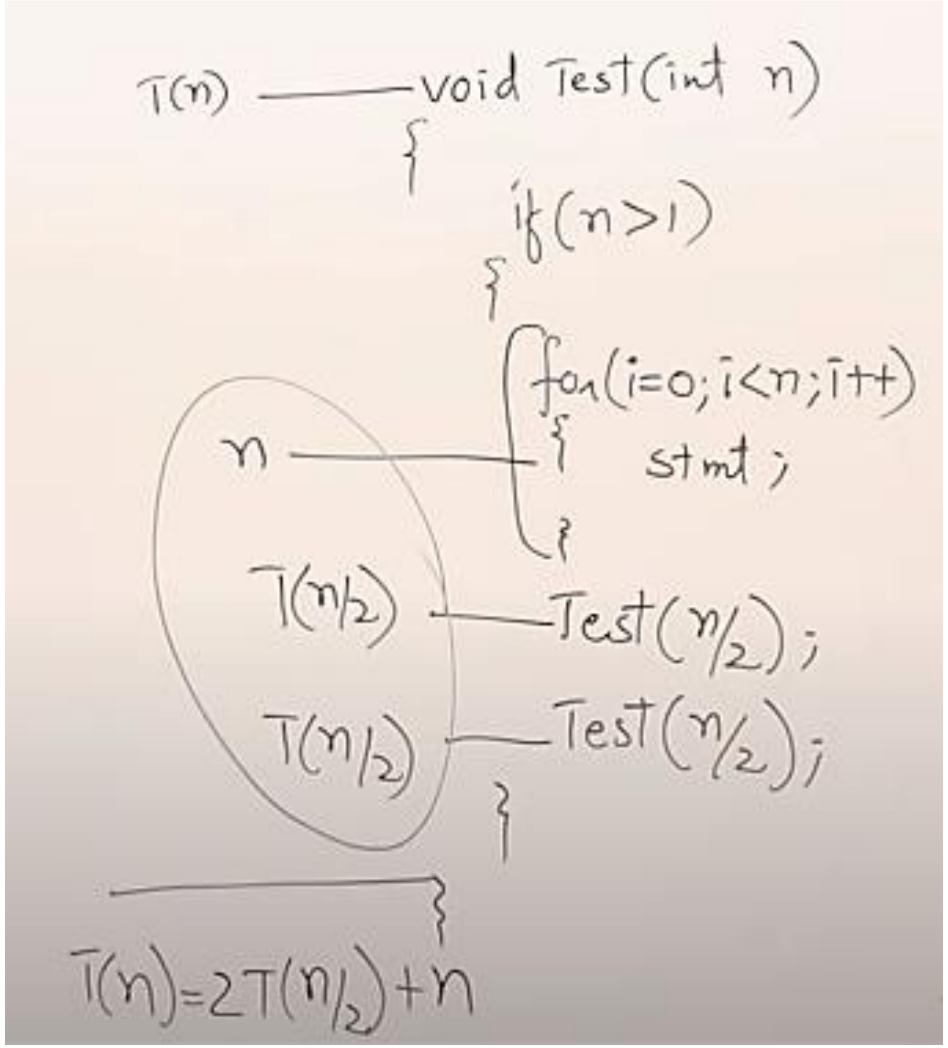
$$T(n) = n \sum_{i=0}^{k} 1/2^i \quad \left(\sum 1/2^i = 1\right)$$

$$T(n) = n \times 1.$$

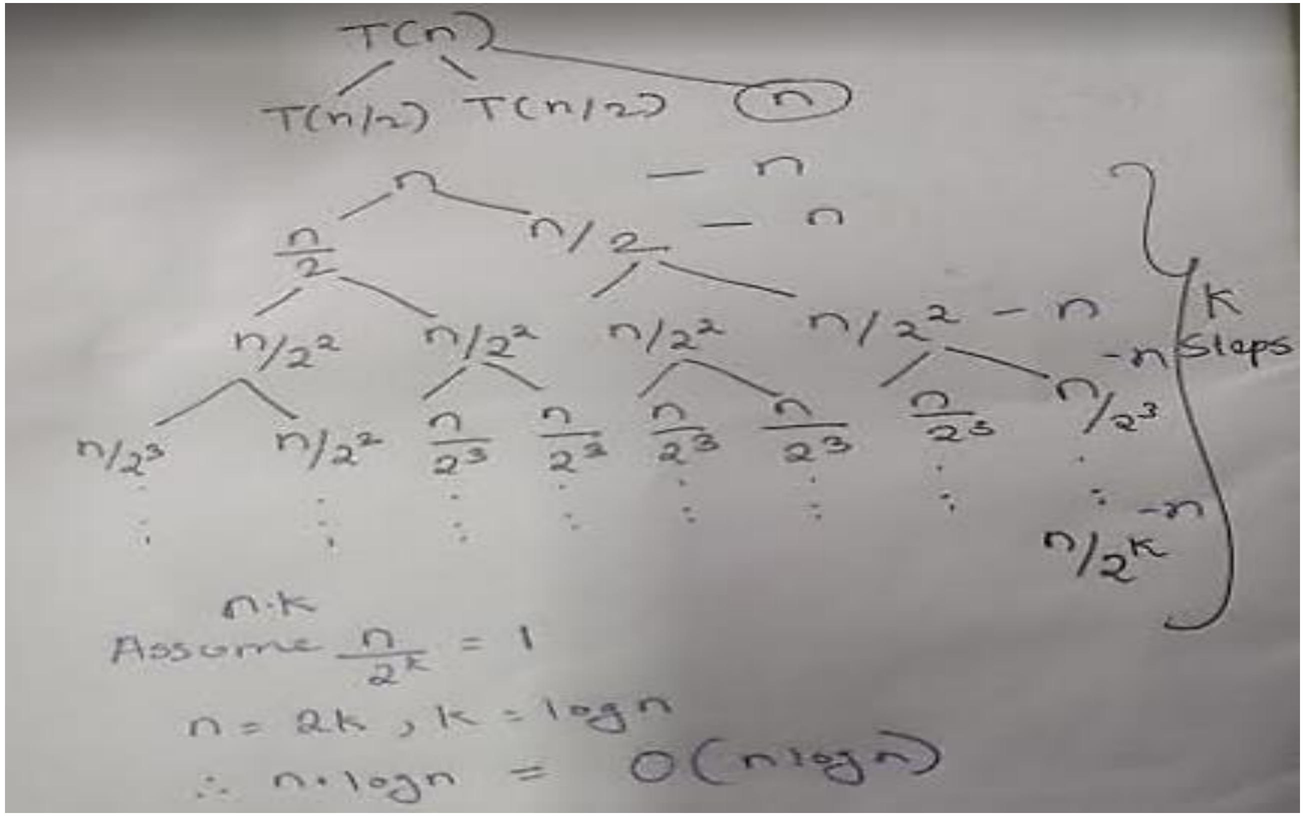
$$T(n) = O(n)$$

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

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,

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

and

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

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

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

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

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

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

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

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

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

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

B.] If $p < 0$ then

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

Recurrence Relation for Dividing Function(Observations)

$$T(n) = T(n/2) + n^1 \text{ --- } \Theta(n)$$

$$T(n) = 2T(n/2) + n^2 \text{ --- } \Theta(n^2)$$

$$T(n) = 2T(n/2) + n^2 \log n \text{ --- } \Theta(n^2 \log n)$$

$$T(n) = 4T(n/2) + n^3 \log^2 n \text{ --- } \Theta(n^3 \log^2 n)$$

$$T(n) = 2T(n/2) + \frac{n^2}{\log n} \text{ --- } \Theta(n^2)$$

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)
            {
   1 ——         stmt;
   T(√n) ——     Test(√n);
            }
        }
```

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

Root Function

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

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

$T(n) = T(n^{\frac{1}{2}}) + 1$ —— ①

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

$T(n) = T(n^{\frac{1}{2^3}}) + 3$ —— ③

$\vdots$

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

Assume $n = 2^m$

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

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

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

$m = 2^k$ and $k = \log_2 m$

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

$k = \log\log_2 n$

$O(\log\log_2 n)$

Substitution method: Forward Substitution method

- 1) Take the Recurrence Equation and Initial Condition
- 2) Put the initial condition in equation and look for the pattern
- 3) Guess the pattern
- 4) Prove that Guess pattern is correct using Induction.

Substitution method: Forward Substitution method

EXAMPLE: $T(N) = T(N - 1) + N$

(1) Take the Equation and Initial Condition

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

(Assume initial condition)

(2) Look for the Pattern

$$
\begin{aligned}
T(1) &= 1 \\
T(2) &= T(2-1)+2 = T(1) + 2 = 1 + 2 \\
T(3) &= T(3-1)+3 = T(2) + 3 = 1 + 2 + 3 \\
T(4) &= T(4-1)+4 = T(3) + 4 = 1 + 2 + 3 + 4 \\
T(5) &= T(5-1)+5 = T(4) + 5 = 1 + 2 + 3 + 4 + 5
\end{aligned}
$$

...

$T(n) = n\,(n+1) / 2$ (it is summation of n numbers)

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

(3) Guess the Pattern as per above step

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

Substitution method: Forward Substitution method

(4) Prove $T(n) = n(n+1)/2$ using Induction

As per Induction for $T(n) = n(n+1)/2$ we can write

1) Let's prove that $T(1) = 1$

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

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

$= 1(2)/2$

$= 1$

$T(1) = 1$ is proved.

2) Assume $T(n-1)$ is true, means

$T(n-1) = (n-1)(n-1+1)/2$ is true

3) Now we will prove that $T(n)$ is also true

Proof:

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= (n-1)(n-1+1)/2 + n \quad \text{(rule 2)} \\
&= (n-1)(n)/2 + n \\
&= n(n-1)/2 + n \\
&= n^2/2 - n/2 + n \\
&= n^2/2 + n/2 \\
T(n) &= n(n+1)/2
\end{aligned}
$$

Hence, it is proved that $T(n)$ is true.

So, as per the Induction, $T(n) = n(n+1)/2$ is true and it is our solution

Recursion Tree

If recurrence relation is

$$T(n) = aT(n/b) + f(n)$$

- then, $f(n)$ is the root of the tree and each node should have a children.
- size of each child node is $\frac{1}{b}$ of parent node n

$$T(n) = aT(n/b) + c(n) \rightarrow \text{cost incurred for dividing and combining.}$$

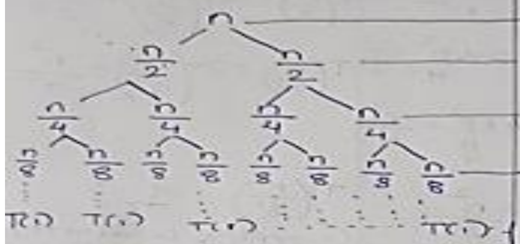
a → No of subproblems.

n/b → size of subproblems.

- Recursion tree is generally used when the recurrence relation is defined for divide and conquer technique. (mostly used in dividing functions).
- It generates a good guess.

Example 1

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



Total nodes at each level.	Cost at each level.
1	n
2^1	$2 \times \frac{n}{2}$
2^2	$4 \times \frac{n}{4}$
2^3	$8 \times \frac{n}{8}$
2^k	$\cancel{2^k} \times \frac{n}{\cancel{2^k}}$

subtree   tree nodes

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

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

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{4}$

$T\left(\frac{n}{8}\right) = 2T\left(\frac{n}{2^4}\right) + \frac{n}{8}$

$\vdots$

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

Total level are k, each level has cost n

$\therefore$ cost of k levels $= \underline{n \cdot k}$

$T\left(\frac{n}{2^k}\right) = T(1)$
$\therefore n = 2^k, \; k = \log_2 n$ — eq 1 } Finding k value

Total cost $=$ cost of leaf nodes $+$ cost of internal nodes
$= L_C + I_C$

* cost of individual leaf node is 1, so how many leaf nodes at k^{th} level.

$2^k = 2^{\log n}$ (substitute k value from eq 1)

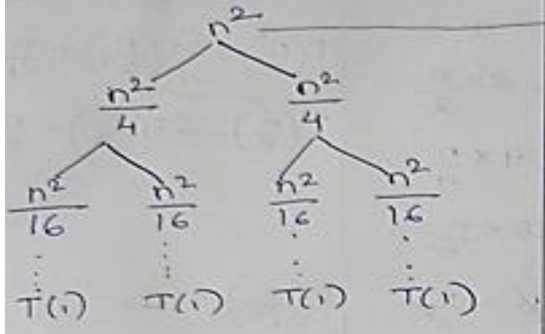
Expressing in terms of n

$n^{\log_2 2} = n^1 = O(n)$ * $\left[x^{\log_y n} \Rightarrow n^{\log_y x} \right]$

$\therefore$ Total cost $= L_C + I_C$
$= n + n \cdot k$ (substituting k)
$= n + n\log n$
$= O(n\log n)$

Example 2

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



Total nodes at each level.	Cost at each Internal level.
1	n^2
2^1	$2 \times \frac{n^2}{4^2} = \frac{n^2}{2}$
2^2	$4 \times \frac{n^2}{16} = \frac{n^2}{4}$
⋮	⋮
2^k	$\frac{n^2}{2^k}$

↑ add all terms

I_c

Sub functions

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

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

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n^2}{4^2}$

⋮

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

(Stopping condition)

Finding out k

$\frac{n}{2^k} = 1, \quad n = 2^k$

$k = \log n.$

$l_c = 2^k = 2^{\log n} = n^{\log 2} = n$

$I_c = n^2\left[\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3 + \cdots \left(\frac{1}{2}\right)^{k-1}\right]$

$= n^2\left[\frac{1}{1-\frac{1}{2}}\right] \quad \left(\begin{array}{c} x < 1 \\ x^0 + x^1 + x^2 \cdots x^k = \frac{1}{1-x} \end{array}\right)$

$= 2n^2$

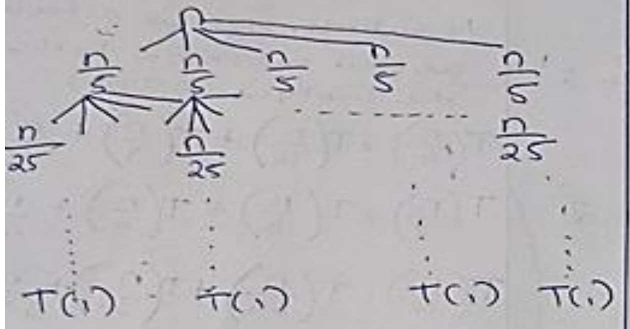
Total cost $= l_c + I_c$

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

Rec T

Example 3

$T(n) = 5T\left(\frac{n}{5}\right) + n$



Total nodes at each level	cost at each level.
1	n
5^1	$5 \times \frac{n}{5}$
5^2	$5^2 \times \frac{n}{5^2}$
$\vdots$	
5^k.	n

$T(n) = 5T\left(\frac{n}{5}\right) + n$

$T\left(\frac{n}{5}\right) = 5T\left(\frac{n}{5^2}\right) + \frac{n}{5}$

$T\left(\frac{n}{5^2}\right) = 5T\left(\frac{n}{5^3}\right) + \frac{n}{5^2}$

$\vdots$

$T\left(\frac{n}{5^k}\right) = 1$

lets find k

$\frac{n}{5^k} = 1$, $n = 5^k$

$k = \log_5 n$

cost at leaf nodes.

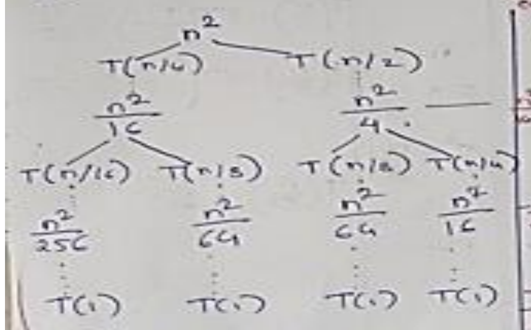
$5^k = 5^{\log_5 n} = n^{\log_5 5} = n$

cost of internal nodes.

$n.k = n.\log n$

Total cost $= n + n\log n = n\log n$

(E4) $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$



cost $\frac{n^2}{256} + \frac{n^2}{64} + \frac{n^2}{64} + \frac{n^2}{16} = \frac{25n^2}{256}$

cost at each level	no of node
n^2	1
$\frac{n^2}{16} + \frac{n^2}{4} = \frac{5n^2}{16}$	2^1
$\frac{25n^2}{256}$	2^2
$\frac{5^{k-1}}{16^{k-1}} n^2$	2^k

$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$

As subtree size is different we will substitute to the the subtrees in $T(n)$

$T\left(\frac{n}{4}\right) = T\left(\frac{n}{16}\right) + T\left(\frac{n}{8}\right) + \frac{n^2}{16}$

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{8}\right) + T\left(\frac{n}{4}\right) + \frac{n^2}{4}$

$T\left(\frac{n}{16}\right) = T\left(\frac{n}{64}\right) + T\left(\frac{n}{32}\right) + \frac{n^2}{256}$

$T\left(\frac{n}{8}\right) = T\left(\frac{n}{32}\right) + T\left(\frac{n}{16}\right) + \frac{n^2}{64}$

$T\left(\frac{n}{4}\right) = \frac{n^2}{16}$

Finding k value

Now here n is divided into 2 different functions or subtrees

Hence tree will not be complete binary tree, will have subtrees of 2 different sizes.

$T\left(\frac{n}{4}\right)$ and $T\left(\frac{n}{2}\right)$

We will select the tree which will have more number of levels

$T(n/4)$ - less no of levels.

$T(n/2)$ - more no of levels.

Hence $T(1) = T\left(\frac{n}{2^k}\right)$

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

cost of ~~leaf~~ internal nodes $= n^2\left[\left(\frac{5}{16}\right)^0 + \left(\frac{5}{16}\right)^1 + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 \cdots \left(\frac{5}{16}\right)^{k-1}\right]$

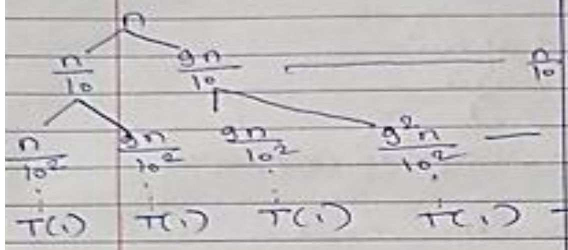
$= n^2\left[\frac{1}{1-\frac{5}{16}}\right] = \frac{16}{11}n^2 = n^2$

cost of leaf nodes $= 2^k = 2^{\log n} = n^{\log 2} = n$

Total cost $= n^2 + n = O(n^2)$

Example 5 :-

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$



cost	no of nodes
n	1
$\frac{n}{10} + \frac{9n}{10} = n$	2^1
n	2^2
n	2^k

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$

$$T\left(\frac{n}{10}\right) = \frac{n}{10^2} + \frac{9n}{10^2} + \frac{n}{10}$$

$$T\left(\frac{9n}{10}\right) = \frac{9n}{10^2} + \frac{9^2 n}{10^2} + \frac{9n}{10}$$

$$T\left(\frac{n}{10^2}\right) = T\left(\frac{n}{10^3}\right) + \frac{9n}{10^3} + \frac{n}{10^2}$$

$$T\left(\frac{9n}{10^2}\right) = \frac{9n}{10^3} + \frac{9^2 n}{10^3} + \frac{9n}{10^2}$$

$$T\left(\frac{9^2 n}{10^2}\right) = \frac{9^2 n}{10^3} + \frac{9^3 n}{10^3} + \frac{9^2 n}{10^2}$$

Finding k value

2 subtrees $\frac{n}{10}$, $\frac{9n}{10}$

$\frac{9n}{10}$ will have more levels.

$$\therefore \frac{9^k n}{10^k} = T(1)$$

$$\therefore \frac{9^k n}{10^k} = 1, \quad k = \log^{10/9} n$$

* cost of internal nodes $= n \cdot k$
  $= n \cdot \log^{10/9} n$

* cost of leaf nodes $= 2^k = 2^{\log^{10/9} n}$

Total cost $= n \log^{10/9} n + 2^{\log^{10/9} n}$

$= O(n \log n)$

Solved in class.

① $2T\left(\frac{n}{2}\right) + n^2$

② $2T\left(\frac{n}{2}\right) + n$

③ $3T\left(\frac{n}{4}\right) + n^2$

④ $T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$

⑤ $T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

⑥ $5T\left(\frac{n}{5}\right) + n$

⑦ $T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$

⑧ $T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + n$