UNIT 4 SOLVING RECURRENCES

Structure

4.0 INTRODUCTION

Complexity analysis of iteration algorithms is much easier as compared to recursive algorithms. But, once the recurrence relation/equation is defined for a recursive algorithm, which is not difficult task, then it becomes easier task to obtain the asymptotic bounds (θ, O) for the recursive solution. In this unit we focus on recursive algorithms exclusively. Three techniques for solving recurrence equation are discussed: (i) Substitution method (ii) Recursion Tree Method and Master Method. In the substitution method, we first guess an asymptotic bound and then we prove whether our guess is correct or not. In the recursion tree method a recurrences equation is converted into a recursion tree comprising several levels. Calculating time complexity requires taking a total sum of the cost of all the levels. The master method requires memorization of three different types of **cases** which help to obtain asymptotic bounds of many simple recurrence relations.

4.1 OBJECTIVES

After going through this unit you will be able to

- Define recurrence relation
- Construct recurrence relation of simple recursive algorithms
- List the techniques used to solve recurrence relation
- Solve the recurrence relation through , Substitution, Recurrence tree & Master methods.

4.2 RECURRENCE RELATION

We often use a *recurrence relation* to describe the running time of a recursive algorithm. A *recursive algorithm* can be defined as an algorithm which makes a recursive call to itself with smaller data size. Many problems are solved

recursively, especially those problems which are solved through Divide and Conquer technique. In this case, the main problem is divided into smaller sub-problems which are solved recursively. Quick Sort, Merge Sort, Binary search, Strassen's multiplication algorithm are formulated as recursive algorithms .These problems will be taken up separately in the next block.

Like all recursive functions, a recurrence relation also consists of two steps: (i) one or more initial conditions and (ii) recursive definition of a problem

Example 1: A Fibonacci sequence $f_0, f_1, f_2, \ldots..$ can be defined by the recurrence relation as:

$$f_n = \begin{cases} 0 & if\ n = 0 \\ 0 & if\ n = 1 \\ f_{n-1} + f_{n-2} & if\ n \geq 2 \end{cases}$$

1. (**Basic Step**)The given recurrence says that if n=0 then $f_0 = 0$ and if n=1 then $f_1 = 1$. These two conditions (or values) where recursion does not call itself is called an **initial condition** (or **Base conditions**).
2. (**Recursive step**): This step is used to find new terms $f_2, f_3, \ldots.,$ from the existing (preceding) terms, by using theformula $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

This formula says that "by adding two previous sequence (or term) we can get the next term".

For example $\boldsymbol{f_2 = f_1 + f_0 = 1 + 0 = 1}$;
$f_3 = f_2 + f_1 = 1 + 1 = 2$; $f_4 = f_3 + f_2 = 2 + 1 = 3$ and so on

Example 2 Find out the value of n! = n (n-1) (n-2)……. (3) (2) (1) for n ≥ 1

Factorial function is defined as:

$$n! = \begin{cases} 1 & if\ n = 1 \\ n.(n-1)! & if\ n > 1 \end{cases}$$

Let us write an algorithm for factorial function:

int fact(intn)

1: $if\ n == 0\ then$
2: $return\ 1$
3: $else$
4: $return\ n * fact(n-1)$
5: $endif$

Let us try to understand the efficiency of the algorithm in terms of the number of multiplications operations required for each value of n

Let $T(n)$ denoted the number of multiplication required to execute the n!.,

that is $T(n)$ denotes the number of times the line 4 is executed in factorial algorithm.

- We have the initial condition T(0) = 1; since when n = 0, fact simply returns (i.e. Number of multiplication is0).

- When $n > 1$, the line 4 performs 1 multiplication plus fact is recursively called with input (n -1). It means, by the definition of $T(n)$, additional $T(n-1)$ number of multiplications arerequired.

We can write a recurrence relation for the factorial as:

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

Algorithm 3: The following algorithm calculates x^n:

Algorithm 3: $\boldsymbol{Power\ (x, n)}$

1: **if**$(n == 0)$

2: $return\ 1$

3: $if(n == 1)$

4: $return\ x$

5: ***else***

6: $return\ x * Power(x, n-1);$

7: $endif$

The base case is reached when n = = 0 or n= = 1.

The algorithm 3 performs one comparison and one return statement. Therefore,

$T(0) and\ T(1) = O(1) = a$

When n > 1; the **algorithm3** performs one recursive call with input parameter (n – 1) at line 6, and some constant number of basic operations. Thus we obtain the recurrence relationas:

$$T(n) = \begin{cases} T(1) = a & (base\ case) \\ T(n-1) + b & (Recursive\ step) \end{cases}$$

Example 4: A customer makes an investment of Rs. 5000 at 15 percent annual compound interest. If T_n denotes the amount earned at the end of n years, define a recurrence relation and initial conditions

Ans- At the end of n-1 years, the amount is T_{n-1}. After one more year , the amount will be T_{n-1} + the interest amount. Therefore

$T_n = T_{n-1} + (15\%)T_{n-1} = (0.15)T_{n-1} = (1.15)T_{n-1}$; n≥ 1

Solving Recurrence

To find out the recurrence relation when n=1 (base value) we have to find the value of T_0.

Since T_0 refers to the initial amount, T_0 = 5000

With the above definitions we can calculate the value of T_n for any value of n. For example:

T_3= (1.15)T_2=(1.15)(1.15)T_1 = (1.15)(1.15)(1.15)T_0 = $(1.15)^3$(5000)

The above computation can be extended to any arbitrary value of n.

T_n = (1.15)T_{n-1}

.. ..

= $((1.15)^n$(5000)

4.3 METHODS FOR SOLVING RECURRENCE RELATIONS

Three methods are discussed here to solve recurrence relations: Substitution method, Recursion Tree method and Mater method. .We start with Substitution method

4.3.1 Substitution Method

Substitution is opposite of induction .We start at n and move backward. A substitution method is one, in which we guess a bound and then use mathematical induction to prove whether our guess is correct or not?. It comprises two steps:

Step1: Guess the asymptotic bound of the Solution.

Step2: Prove the correctness of the guess using Mathematical Induction.

Example 1. Solve the following recurrence by using substitution method.

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

Solution*: step1:* The given recurrence is quite similar to that of Merge Sort algorithm, Therefore, our guess to the solution is $T(n) = O\ (nlogn)$

Or $T(n) \leq c.nlogn$

Step2: Now we use mathematical Induction.

Here our guess does not hold for n=1because $T(1) \leq c.1log1$
$i.e. T(n) \leq 0\ which\ is\ contradiction\ with\ T(1) = 1$

Now for n=2
$T(2) \leq c.2log2$
$2T\left(\frac{2}{2}\right) + 2\ \leq c.2$
$2t(1) + 2\ \leq c.2$
$0 + 2 \leq c.2$

$2 \leq c.2$ which is true. So $T(2) \leq c.nlogn$ is True for n = 2
So
$T(2) \leq c.nlogn$ is True form = 2

(i) **Induction step:** Now assume it is true for n = n/2

$i.e. T\left(\frac{n}{2}\right) \leq c.\left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) is\ true.$

Now we have to show that it is true for the value of n

$i.e. T(2) \leq c.nlogn$

We known that $T(n) \leq 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$

$\leq 2(c\left\lfloor\frac{n}{2}\right\rfloor log\left\lfloor\frac{n}{2}\right\rfloor + n$

$\leq cnlog\left\lfloor\frac{n}{2}\right\rfloor + n \leq cnlogn - cnlog2 + n$

$\leq cnlogn - cn + n$

$\leq cnlogn\ \forall c \geq 1$

Thus $T(n) = O(nlogn)$

Remark: Making a good guess, which can be a solution of a given recurrence, requires experiences. So, in general, we are often not using this method to get a solution of the given recurrence.

4.3.2 RECURSION TREEMETHOD

A recursion tree is a convenient way to visualize what happens when a recurrence is iterated. It is a pictorial representation of a given recurrence relation, which shows how Recurrence is divided till Boundary conditions.

Recursion tree method is especially used to solve a recurrence of the form:

$T(n) = aT\left(\frac{n}{b}\right) + f(n) \ldots\ldots\ldots (1) \qquad where\ a > 1, b \geq 1$

This recurrence (1) describe the running time of any divide-and-conquer algorithm.

Method (steps) for solving a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ using recursion tree:

We make a recursion tree for a given recurrence as follows:

a) To make a recursion tree of a given recurrence (1), First put the value of $f(n)$ at root node of a tree and make a ***a number*** of child nodes of this root value $f(n)$ Now tree will be looks like as:

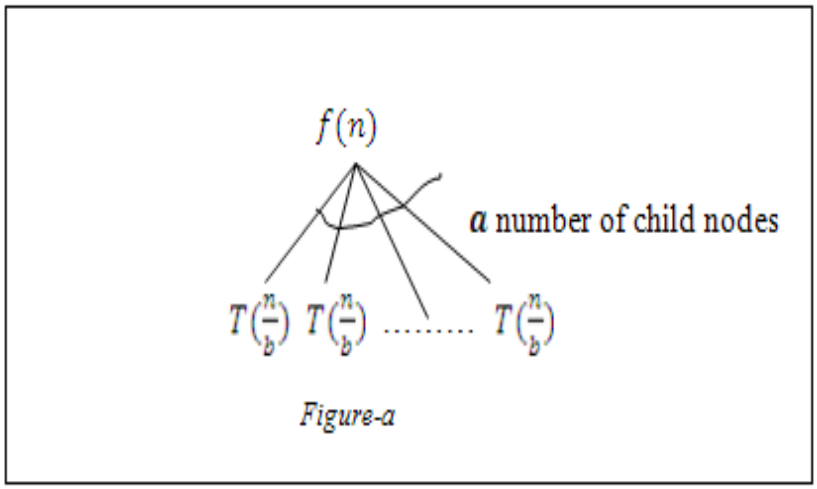

Figure-a

(b) Now we have to find the value of $\boldsymbol{T\left(\frac{n}{b}\right)}$ by putting (n/b) in place of n in equation (1).That is

$$T\left(\frac{n}{b}\right) = aT\left(\frac{\frac{n}{b}}{b}\right) + f(n/b) = aT + f\left(\frac{n}{b^2}\right) + f(n/b) \ldots (2)$$

From equation (2), now $f(n/b)$ will be the value of node having a branch (child nodes) each of size T(n/b). Now each $T\left(\frac{n}{b}\right)$in **figure-a** will be replaced as follows:

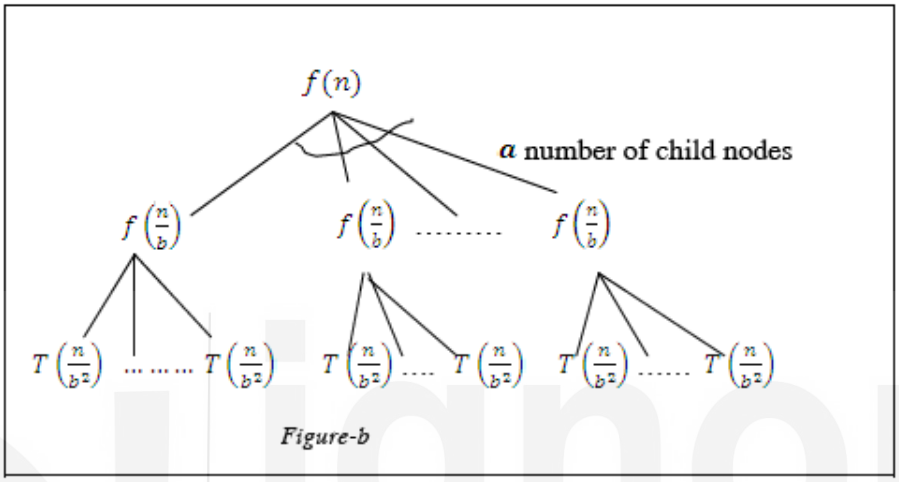

Figure-b

c) In this way you can expend a tree one more level (i.e. up to (at least) 2 levels).

Step2: (a) Now you have to find per level cost of a tree. Per level cost is the sum of the cost of each node at that level. For example per level cost at level1 is

$\left(\frac{n}{b}\right) + f\left(\frac{n}{b}\right) + \cdots f\left(\frac{n}{b}\right)(a\ times)$. This is also called**Row-Sum.**

(b) Now the total (final) cost of the tree can be obtained by taking the sum of costs of all these levels.

$i.e. Total\ cost = sum of\ costs\ of\ (l_0 + l_1 + \cdots \ldots \ldots . + l_k)$.

This is also called **Column-Sum.**

Let us take one example to understand the concept to solve a recurrence using recursion tree method:

Example1: Solve the recurrence $\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n}$ using recursion tree method.

Solution: **Step1:** First you make a recursion tree of a given recurrence.

1. To make a recursion tree, you have to write the value of $f(n)$ at root node. And
2. The number of child of a Root Node is equal to the value of a. (Here the value of a = 2).So recursion tree be looks like as:

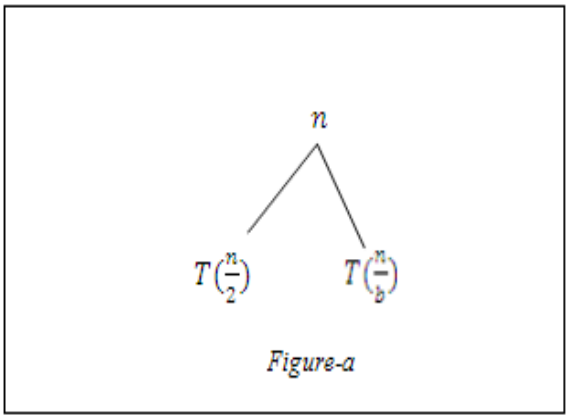

Figure-a

b) Now we have to find the value of $T\left(\frac{n}{2}\right)$ in figure (a) by putting (n/2) in place of n in equation (1).That is

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

From equation (2), now $\left(\frac{n}{2}\right)$ will be the value of node having 2 branch (child nodes) each of size T(n/2). Now each $T\left(\frac{n}{2}\right)$ in **figure-a** will be replaced as follows:

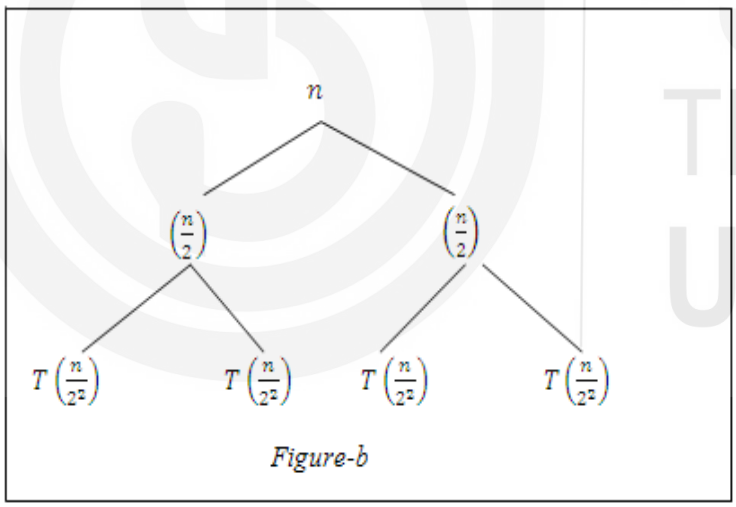

Figure-b

c) **In this way, you can extend a** tree up to Boundary condition (when problem size becomes 1). So the final tree will be looks like:

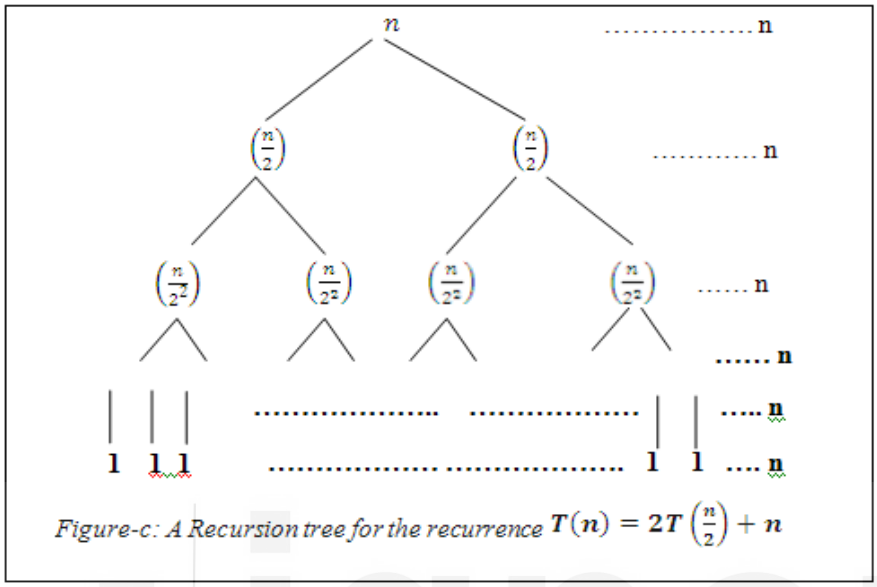

Figure-c: A Recursion tree for the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$

Now we find the per level cost of a tree, Per-level cost is the sum of the costs within each level (called row sum). Here per level cost is For example: per level cost at depth 2 in **figure-c** can be obtained as:

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

Then total cost is the sum of the costs of all levels (called column sum), which gives the solution of a given Recurrence. The height of the tree is

Total cost= $n + n + n + \cdots \ldots \ldots + n$……. (3)

To find the sum of this series you have to find the total number of terms in this series. To find a total number of terms, you have to find a height of a tree.

Height of tree can be obtained as follow (see recursion tree of figure c): you start a problem of size n, then problem size reduces to$\left(\frac{n}{2}\right)$, then$\left(\frac{n}{2^2}\right)$, and so on till boundary condition (problem size 1) is not reached. That is

$$n \to \left(\frac{n}{2}\right) \to \left(\frac{n}{2^2}\right) \to \cdots \ldots \ldots \ldots \to \left(\frac{n}{2^k}\right)$$

At last level problem size will be equal to 1 if

$$\left(\frac{n}{2^k}\right) = 1 \Rightarrow n = 2^k \Rightarrow k = log_2 n.$$

This *k* represent the height of the tree, hence height $= k = log_2 n$.

Hence total cost in equation (3) is

$$n + n + n + \cdots \ldots \ldots + n(log_2 n\ terms) = nlog_2 n \implies O(nlog_2 n).$$

Example2: Solve the recurrence $T(n) = T\left(\left\lfloor\frac{n}{3}\right\rfloor\right) + T\left(\left\lfloor\frac{2n}{3}\right\rfloor\right) + n$

using recursion tree method.

Solution: We always omit floor & ceiling function while solving recurrence. Thus given recurrence can be written as:

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

Figure-a to figure-c shows a step-by-step derivation of a recursion tree for the given recurrence (1).

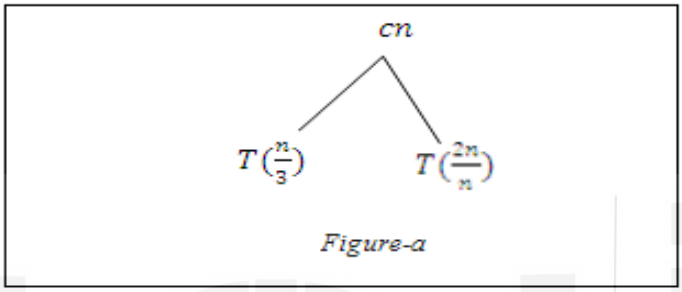

Figure-a

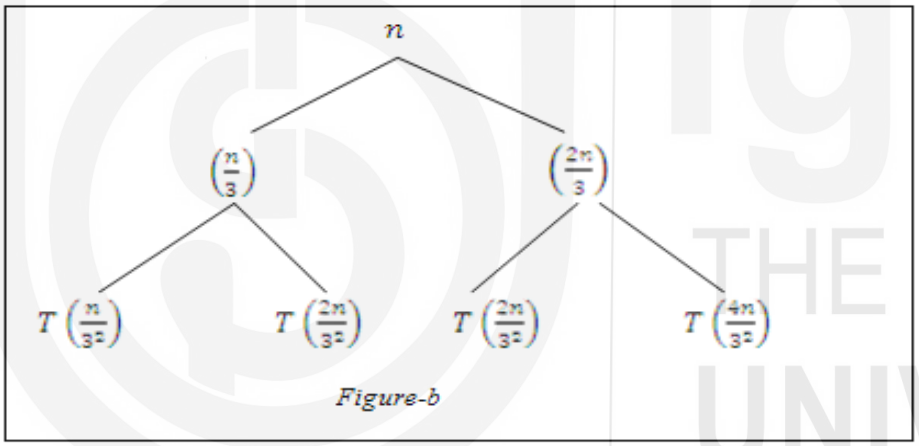

Figure-b

c) **In this way, you can extend a** tree up to Boundary condition (when problem size becomes 1). So the final tree will be looks like:

Here the smallest path from root to the leaf is:

$$n \to \left(\frac{2}{3}\right)n \to \left(\frac{2}{3}\right)^2 n \to \cdots 1$$

$$\left(\frac{2}{3}\right)^k = 1 \Longrightarrow k = log_{3/2}n \implies Height\ of\ the\ tree.$$

$$n + n + \cdots\ldots\ldots\ldots + n\left(log_{3/2}n\ times\right)$$

$$\Longrightarrow nlog_{3/2}n = \frac{nlog_2 n}{log_2 \frac{3}{2}} = O(nlog_2 n) - - - - - (*)$$

Here the smallest path from root to the leaf is:

Solving Recurrence

$n \to \left(\frac{n}{3}\right) \to \left(\frac{n}{3^2}\right) \to \cdots \ldots \ldots \ldots \to \left(\frac{n}{3^k}\right)$

$(n/3)^k = 1 \Longrightarrow k = log_3 n \implies Height\ of\ the\ tree.$

$n + n + \cdots \ldots \ldots \ldots + n(log_3 n\ times)$

$\Longrightarrow nlog_3 n = \frac{nlog_2 n}{log_2 3} = \Omega(nlog_2 n) - - - - - (**)$

Form equation (*) and
(**), $Since\ T(n) = O(nlog_2 n) and\ T(n) = \Omega(nlog_2 n),$ thus we write:

$T(n) = \Theta(nlog_2 n)$

Remark: If
$\boldsymbol{f(n) = O(g(n)) and\ f(n) = \Omega(g(n)) then\ fIn) = \Theta(g(n))}$

Example3: A recurrence relation for Tower of Hanoi (TOH) problem is $T(n) = 2T(n-1) + 1\ with\ T(1) = 1\ and\ T(n) = 3$. Solve this recurrence to find the solution of TOH problem.

Solution:

Figure-a to figure-c shows a step-by-step derivation of a recursion tree for the given recurrence $T(n) = 2T(n-1) + 1$

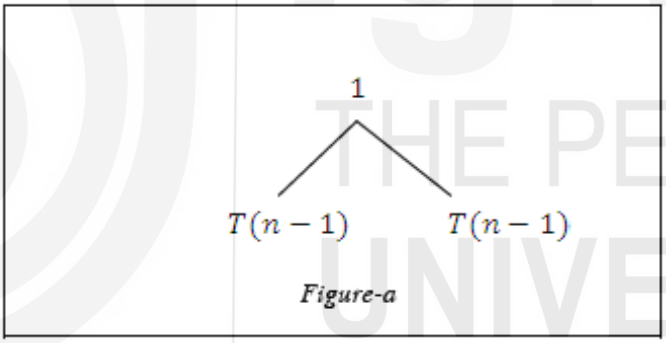

Figure-a

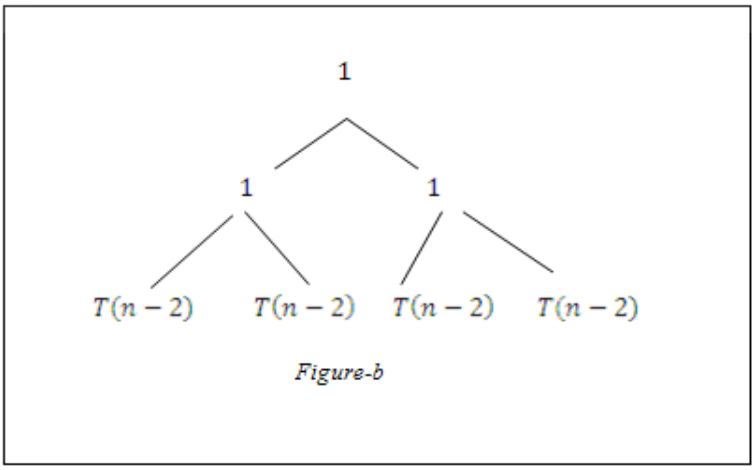

Figure-b

c) In this way, you can extend a tree up to Boundary condition (when problem size becomes 1). That is

$n \to (n-1) \to (n-2) \to \cdots \ldots \ldots \ldots \to \big(n - (n-1)\big)$

$\Rightarrow n \rightarrow (n-1) \rightarrow (n-2) \rightarrow \cdots \ldots \ldots \rightarrow 2 \rightarrow 1$

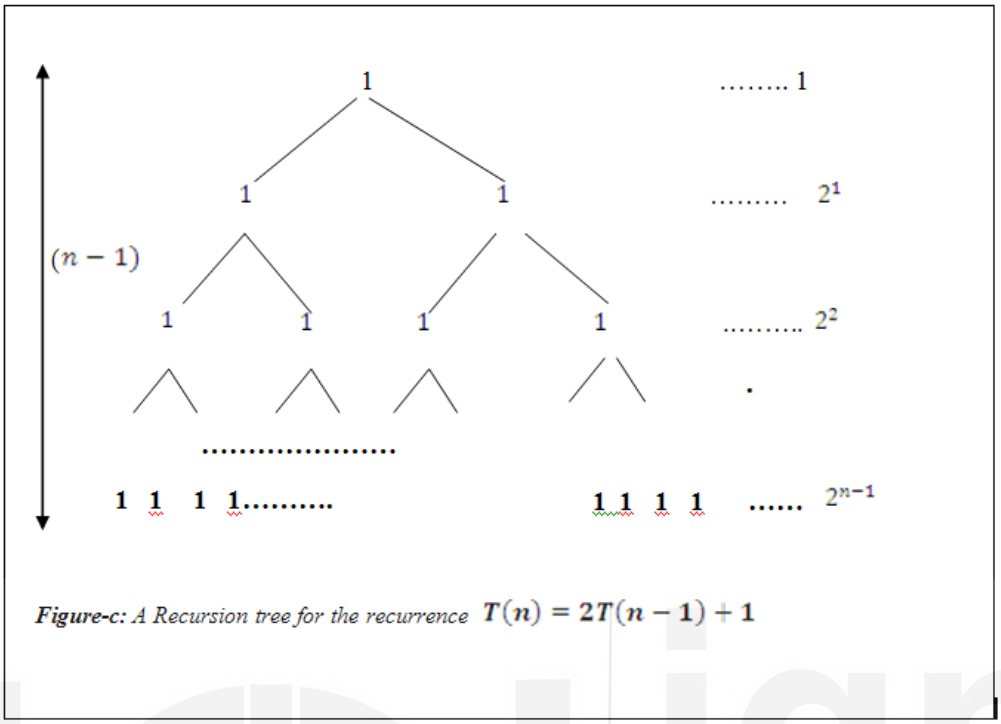

Figure-c: *A Recursion tree for the recurrence* $T(n) = 2T(n-1) + 1$

So the final tree will be looks like:

At last level problem size will be equal to 1 if
$(n-(n-1)) = 1 \Rightarrow Hight\ of\ the\ tree\ \Rightarrow (n-1)$.

Hence Total Cost of the tree in figure (c) can be obtained by taking column sum upto the height of the tree.

$$T(n) = 1 + 2^1 + 2^2 + \cdots \ldots + 2^{n-1} = \frac{1(2^n - 1)}{2 - 1} = 2^n - 1.$$

Hence the solution of TOH problem is $T(n) = (2^n - 1)$

☞**Check Your Progress 1**

Q1: write a recurrence relation for the following recursive functions:

a) $Fast\_Power(x, n)$

$\{if\ (n == 0)$

$return\ 1;$

$elseif\ (n == 1)$

$return\ x;$

$\boldsymbol{elseif}\ ((n\%2) == 0)$ $\quad //if\ n\ is\ even$
$return\ Fast\_power\left(x, \frac{n}{2}\right) * Fast\_power\left(x, \frac{n}{2}\right);$

$\boldsymbol{else}$

$return\ x * Fast\_power\left(x, \frac{n}{2}\right) * Fast\_power\left(x, \frac{n}{2}\right)$

}

b)

Fibnacci (n)

{ if (*n = = 0*)

return 0;

if (*n = = 1*)

return 1;

return fibnacci (*n*-1) + *fibnacci* (*n*-2); }

Q.2: Solve the following recurrence Using Recursion tree method

$a. T(n) = 4T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$

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

.....

.....

.....

.....

4.3.3 MASTERMETHOD

Definition 1: A function$f(n)$ is ***asymptotically positive*** if and only if there exists a real number *n* such that $f(x) > 0 \; for \; all \; x > n$.

The master method provides us a straight forward method for solving recurrences of the form
$T(n) = aT\left(\frac{n}{b}\right) + f(n), where \; a \geq 1 \; and \; b > 1$ are constants and $f(n)$ is an asymptotically positive function. This recurrence gives us the running time of an algorithm that divides a problem of size *n* into *a* **subproblems** of size $\left(\frac{n}{b}\right)$.

The *a* **subproblems** are solved recursively, each in time $T\left(\frac{n}{b}\right)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$This recurrence is technically correct only
when $\left(\frac{n}{b}\right)$ is an integer, so the assumption will be made that $\left(\frac{n}{b}\right)$ is either $\left\lfloor\frac{n}{b}\right\rfloor$
or $\left\lceil\frac{n}{b}\right\rceil$ since such a replacement does not affect the asymptotic behavior of the recurrence. The value of *a and b* is a positive integer since one can have only a whole number of subproblems.

Theorem1: Master Theorem

The Master Method requires memorization of the following 3 cases; then the solution of many recurrences can be determined quite easily, often without using pencil & paper.

Let $T(n)$ be defined on the non negative integers by:
$T(n) = aT\left(\frac{n}{b}\right) + f(n), where \; a \geq 1, b > 1 \; and \; \frac{n}{b}$ is treated as above------(1)

Then $T(n)$ can be bounded asymptotically asfollows:

Case1: If $f(n) = O\left(n^{log_b a - \epsilon}\right)$ for some $\epsilon > 0, then \; T(n) = \Theta\left(n^{log ba}\right)$

Case2: If$(n) = \Theta\left(n^{log_b a}\right)$, then $T(n) = \Theta\left(n^{log_b a}.logn\right)$.

Case3: If $f(n) = O\left(n^{log_b a + \epsilon}\right)$for some $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for

some constant c $< 1 \; and \; all \; sufficiently \; large \; n. T(n) = \theta\left(f(n)\right)$

Remark: To apply Master method you always compare $n^{log_b a}$and $f(n)$The larger of the two functions determines solution to the recurrence problems.If the growth rate of these two functions then it belongs to **case 2**.In this case we multiply by a logarithmic factor to get the run time solution (T(n)) of recurrence relation.

If $f(n)$ is polynomially smaller than $n^{log_b a}$ (by a factor of n^{ϵ}then **case 1** will be applicable to find $T(n)$.

If $f(n)$ is polynomially larger than $n^{log_b a}$ (by a factor of$1/n^{\epsilon}$ then $T(n) = \theta(f(n)\ which\ is\ in\ \boldsymbol{case\ 3}$.

Examples of Master Theorem

Example1: Consider the recurrence of $T(n) = 9T\left(\frac{n}{3}\right) + n$, in which a = 9, b = 3, $f(n) = n$, $n^{log_b a} = n^2$ and$f(n) = O\ (n^{log_b a - \epsilon})$, where $\epsilon = 1$. The growth rate of f(n) is slower, we will apply the case 1 of Master Theorem and we get $T(n) = \Theta\left(n^{log b^{a-\epsilon}}\right) = \Theta(n^2\ )$

Example2: Consider the recurrence of $T(n) = T\left(\frac{2n}{3}\right) + 1$, in which $a = 1$, $b = \frac{3}{2}, f(n) = n^{log_3 2}$=1. Since $f(n) = \Theta(\ n^{log_3 2})$. By Master Theorem (case2), we get $T(n) = \Theta\left(\ n^{log_b a}.logn\right) = \Theta(logn)$.

☞ **Check Your Progress 2**

Q.1: Write the first two cases (Case 1 and Case 2) of Master method to solve a recurrence relation of the form:

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

.....

.....

.....

.....

Q.2: Use Master Theorem to give the tight asymptotic bounds of the following recurrences:

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

.....

.....

.....

.....

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

.....

.....

.....

.....

4.4 SUMMARY

The following points were discussed in the unit:

1. Two important ways to characterize the effectiveness of an algorithm are its *space complexity* and *time complexity.*

2. *Space complexity* of and algorithm is the number of elementary objects that this algorithm needs to store during its execution. The space occupied by an algorithm is determined by the number and sizes of the variables and data structures used by the algorithm.

3. Number of machine instructions which a program executes during its running time is called *time complexity.*

4. There are 3 cases, in general, to find the time complexity of an algorithm:
   Best case: The minimum value of $f(n)$ for any possibleinput.

   Worst case: The maximum value of $f(n)$ for any possibleinput.

Average case: The value of $f(n)$ which is in between maximumandminimum for any possible input. Generally the Average case implies the ***expected value*** of $f(n)$

5. *Asymptotic analysis* compares relative performance of algorithms

6. There are 3 Asymptotic notations used to express the time complexity of an algorithm *O,* Ω *and* Θ notations.

7. ***O*-notation:** Given functions *f(n)* and *g(n),* we say that *f(n)* is *O(g(n))* if there exist two positive constants c > 0 and $n_0 \geq 1$ such that $f(n) \leq cg(n)$for all n, $n \geq n_0$ Big-Oh notation gives an upper bound on the growth rate of a function.

8. **Ω-notation**: Given functions *f(n)* and *g(n),* we say that *f(n)*is *Ω(g(n))* if there exist two positive constants c > 0 and $n_0 \geq 1$ such that *f(n) cg(n)* for all n, $n \geq n_0$ Big-Omega notation gives a lower bound on the growth rate of a function.

9. **Θ-notation**: Let *f(n)* and *g(n)* be two asymptotically positive real-valued functions. We say that *f(n)* is *Θ(g(n))* if there is an integer n_0 and two positive real constants c_1 and c_2 such that $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all n ≥ n_0

10. When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A *recurrence relation* is an equation or inequality that describes a function in terms of its value on smaller inputs.

Solving Recurrence

11. There are three basic methods of solving the recurrence relation:

    1. The Substitution Method
    2. The Recursion-tree Method
    3. The Master Theorem

12. *Master method* provides a "cookbook" method for solving recurrences of the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n) where\ a \geq 1\ and\ b > 1$ are constants

13. In master method you have to always compare the value of $f(n)$ with $n^{log_b a}$ to decide which case is applicable. If $f(n)$ is asymptotically smaller than $n^{log_b a}$, then case 1 is applied. If $f(n)$ is asymptotically same as $n^{log_b a}$, then case 2 is applied. If $f(n)$ is asymptotically larger than $n^{log_b a}$, and if $af\left(\frac{n}{b}\right) \leq c.f(n) for\ some\ c < 1,$ then case3 isapplied.

4.5 SOLUTIONS to Check Your Progress

Check Your Progress 1:

Q 1: a)

At every step the problem size reduces to half the size. When the power is an odd number, the additional multiplication is involved. To find a time complexity of this algorithm, let us consider the ***worst case,*** that is we assume that at every step additional multiplication is needed. Thus total number of operations T(n) will reduce to number of operations for n/2, that is T(n/2) with three additional arithmetic operations(In odd power case: 2 multiplication and one division). Now we can write:

$T(n) = 1\ if\ n = 0\ or\ 1$

$T(n) = T\left(\frac{n}{2}\right) + 3\ if\ n \geq 2$

Instead of writing exact number of operations needed by the algorithm, we can use some constants. The reason for writing this constant is that we are always interested to find "asymptotic complexity" instead of finding exact number of operations needed by algorithm, and also it would not affect our complexity also.

$$T(n) = \begin{cases} T(1) = a & if\ n = 0\ or\ n = 1 \quad (base\ case) \\ T\left(\frac{n}{2}\right) + b & if\ n \geq 2 \quad (Recursive\ step) \end{cases}$$

b) $$T(n) = \begin{cases} a & if\ n = 0\ or\ n = 1 \quad (base\ case) \\ T(n-1) + T(n-2) + b & if\ n \geq 2 \quad (Recursive\ step) \end{cases}$$

Q2 *(a)* The recursion tree for the given recurrence relation is:

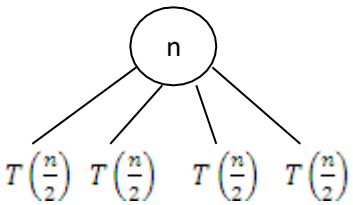

Figure a

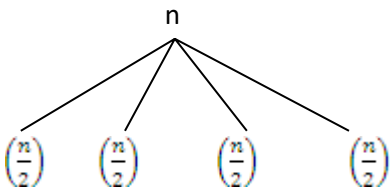

Figure b

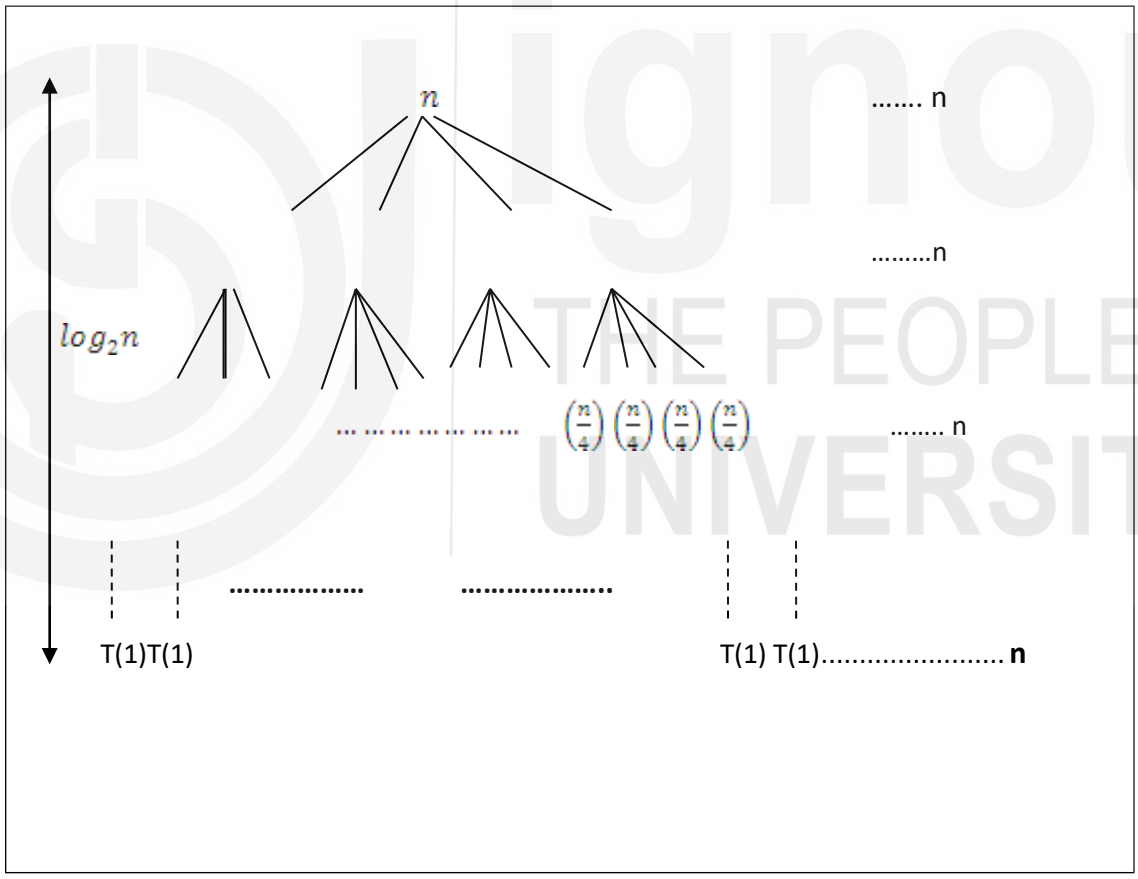

Figure c: A Recurrence Tree for T(n) = 4T($\frac{n}{2}$) + n

We have$Total = n + 2n + 4n + \cdots log_2 n\ times$

= n (1 + 2 + 4 +… $log_2 n\ times$)

$$= n\ \frac{\left(2^{log_2 n} - 1\right)}{2 - 1} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

$\therefore T(n) = \theta(n^2)$

Q2(d)

Check Your Progress 2:

Q1: The following 3 cases are used to solve a recurrence

Case1: If for some $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$,then $T(n) = \Theta(n^{log_b a})$

Case2: If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}.logn)$..

Solution2:

a) In a recurrence $T(n) = 4T\left(\frac{n}{2}\right) + n, a = 4, b = 2,$
$f(n) = n^{log_b a} = n^2. Now\ compare\ f(n) with\ n^{log_b a}.$
$Since\ f(n) = O\left(n^{log_b a - \epsilon}\right).$
$where\ \epsilon = 1. By\ Master\ Theorem\ case\ 1\ we\ get\ T(n) = \Theta(n^2).$

b) $T(n) = 4T\left(\frac{n}{2}\right) + n^2; in\ which\ a = 4, b = 2, f(n) = n^2\ and\ n^{log_b a} = n^2.$
$Now\ compare\ f(n) wth\ n^{log_b a};$
$since\ f(n) = n^2 = \Theta(n^{log_b a}).$
$Thus\ By\ Master\ Theorem\ (case 2),$
$we\ get\ T(n) = \Theta(n^{log_b a}.logn) = \Theta(n^2 logn).$