UNIT 3 RECURSIVE FUNCTION THEORY

Structure **Page Nos.**

3.0 INTRODUCTION

Let us stop for a moment and know that there really is another way (rather, more ways) of looking formally at the notion of computation.

Here is Edward Bear, coming downstairs now, bump, bump, bump, on the back of his head, behind Christopher Robin. It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way, if only he could stop bumping for a moment and think of it.

Winnie-the-Pooh, 1926 A.A.Milne

In the previous units, we have discussed the automata or machine models of the computational phenomenon. The automata approach to computation is *Operational* in nature, i.e., automata approach is concerned with the computational aspect of '**how** the computation is to be performed'.

In this unit, we will be concerned with *Recursive Function Theory*, which is a **functional** or **declarative** approach to computation. Under this approach, *computation* is described in terms of '**what** is to be accomplished' in stead of '**how** to accomplish'.

Each computational theory (rather each theory about any other phenomenon also) starts with some *assumptions*, for example, *about basic (undefined) concepts, operational capabilities and a set of statements, called axioms and postulates,which are assumed to be fundamentally true* (i.e, assumed to be true without any argument). In Automata Theory, the concepts like 'state' 'initial state', 'final state' and 'input' etc are assumed to be understood, without any elaboration. Further, *the capabilities of an automata* to accept an input from the environment; to change its state on some, or even on no input; to give signal about acceptability/unacceptability of a string; *are assumed.*

In Recursive Function Theory, *to begin with*, it is **assumed** that **three types of functions** (viz ξ, σ and ^k_i which are called *initial functions* and are described under **Notations** below) and **three structuring rules** (viz combination, composition and primitive recursion) for constructing more complex functions out of the already constructed or assumed to be constructible functions **are so** simple that ***our ability*** to construct machines to realize these functions and the structuring rules is taken as *acceptable without any argument*. The functions, obtained by applying a finite sequence of the structuring rules to the initial functions, are called **Primitive Recursive functions.** However, with these simple functions and elementary

structuring rules, though it is possible to construct very complex functions yet, even some simple functions like **division** are not constructible by the above mechanism.

Two other well-known formalisms are (i) Church's -Calculus and (ii) Curry's Combinatory Logic

Therefore, another structuring rule, viz **unbounded minimalization** is added which leads to the concepts of **μ-recursion and partial recursion**.

Constructibility/Computability has been a pursuit of the mathematicians, since at least the peak of Greek civilization in third/fourth century B.C. The intellectual concern was about the **constructibility of real numbers**, i.e, for a given real number α, to attempt to draw a line of length α, *with the help of only an unmarked straight edge and a compass*, provided fundamental unit length is given. These attempts at constructibility of real numbers, lead to some *famous problems* including the problems of

(i) Trisecting an angle, (ii) Duplicating a cube and (iii) squaring a circle.

In this unit, the concept of *constructible or computable*, the latter being the more often used term in Computer Science, is based only on our ***intuitive*** understanding of the concept. Discussion of *computable* in the ***formal*** sense based on *Church-Turing hypothesis*, is taken up in other units.

To some of the learners, the treatment of some of the topics may appear to be undesirably too detailed. However, the details are justified in view of the fact that the subject matter is presented from the point of view of the undergraduate students, many of whom may not have studied Mathematics even at 10+2 level.

In order to facilitate faster coverage of the material by advanced learners, some of the contents are placed in boxes which, without any loss of continuity, can be skipped after first reading or even after a cursory glance.

Note: *Exercises in the Block are numbered in one sequence; all other numbered items like theorems, examples, lemmas, statements are taken together for another numbering sequence.*

Key words: recursive definition, partial function, total function, initial functions, structuring rules, primitive recursion, bounded minimalization unbounded minimalisation, partial recursion, -recursion.

Notations:

N	:	the set of natural numbers including 0
I	:	the set of integers
ξ	:	the zero function which maps every element of the domain to 0.
σ	:	the successor function, which maps each natural number n to n + 1
$^{k}_{i}$	:	the projection function which maps the k-tuple (m_1, …, m_i, .., m_k) to the ith component m_i, for 1 i k.
┐	:	negation
	:	there exists

3.1 OBJECTIVES

At the end of this unit, you should be able to

For more details refer pp 297-299, *A First Course in Abstract Algebra* by J.B. Fraleigh, VII edition, Pearson Education, 2003.

To explain the concepts of primitive recursion,   -recursion and partial recursion alongwith other auxiliary concepts

to tell the hierarchy between the classes of primitive recursive functions, total computable functions,   -recursive functions and partial recursive functions.

use these concepts and techniques for generating functions of these classes

3.2 SOME RECURSIVE DEFINITIONS

We are familiar with the concept of factorial of a natural number **n**, denoted as **n!**, with one of the ways of defining it as:

$$n! = n.\ (n - 1)\ \ldots\ldots\ldots\ldots 1 \qquad (1)$$

This is an *explicit* definition of n!.

However, the following is an *implicit* definition, called **recursive definition,** of *factorial.*

$$0! = 1 \quad \text{and}$$
$$n! = n\ .\ (n - 1)! \qquad \text{for } n \quad 1. \qquad (2)$$

The definition (2) above of the factorial is *recursive* in the sense that in order to find the value of factorial at an argument n, we need to find the value of factorial at some simpler argument, in this case (n-1), alongwith possibly some other calculations. In both the explicit and implicit definitions (1) and (2) above of n!,our approach is *functional or declarative* in nature, where computation is described in terms of '**what** is to be accomplished' instead of '**how** to accomplish'.

Similarly, for a natural number n or a real number (or even a complex number) x, the **exponential x^n is explicitly defined as**

$$x^n = \underbrace{x.x\ldots\ldots x}_{ntimes} \qquad (3)$$

Also, the exponential x^n is recursively defined as:

$$x^0 = 1$$
$$x^n = x \quad x^{n-1}, \text{ for a natural number } n \quad 1. \qquad (4)$$

Remark 1

(i) We may observe that non-recursive definitions (1) and (3) given above respectively for n! and x^n use the *imprecise* notation '…..' . On the other hand, the corresponding recursive definitions (2) and (4) use only *precise* notations.

(ii) In (2) and (4), the definitions are given in terms of their own partial definitions viz. n! in terms of (n – 1)! and x^n in terms of x^{n-1}. In this way, the problem of evaluating n! is *converted* to the problem of evaluation of (n – 1)!. This *conversion* of a problem to a less complex version of the problem may be called *reduction* in case we are able to show that calculating (n – 1)! is relatively less complex than calculating n!. If we look back on definition (2) of n!, we observe that 0! is given as a definite number requiring no more applications of the definition of factorial to another number. And reaching 0! from (n —1)! takes lesser number of applications of (2) than reaching 0! from n!. Thus, we can see that the problem of calculating n! is *reduced* through successive applications of the definition of factorial as given by (2) and is *terminated* when 0! is replaced by 1. Exactly on the similar lines, the problem of calculating x^n is gradually *reduced* by the application of definition (4) and is terminated when x^0 is replaced by 1.

3.3 PARTIAL,TOTALANDCONSTANT FUNCTIONS

As mentioned under Remarks (ii) above, the factorial of n is defined in terms of *only* the factorial of another, but smaller, number. However, **this idea of defining a function in terms of *only* itself may be further generalized when a function f may be defined, in addition to in terms of f itself, possibly in terms of some other functions also. Another way in which the idea of recursion as explained above is generalized, is through extending the scope of recursive definitions to *partial* functions (*to be defined*). Various generalizations, including the one given below, lead to the definitions of *primitive recursion and partial recursion.***

The idea of functions *from N to N,* can be generalized to functions from *N^k to N^p*
where
k = 0, l, 2, …….
p = 0, l, 2, ……

Example 2: of Functions from N^k N^p where k > 1

Plus: N N N, with
plus (n, m) = n + m, (5)

Mapping **every pair** of integers of N to integers in N.
E.g., *Plus* takes the ordered pair (3, 2) and returns 5. Similarly, *Plus* takes the ordered pair (4, 0) and returns 4.

Similarly, we may define
PROD: N N **N,** with
PROD (m, n) = m n for m, n N. (6)
And we may define
Exp (m, n) = m^n for all m , n N (7)

Example 3: of a function from N^k to N^p where k > 1 and p > 1:

Plus-Prod: N^2 N^2, such that
Plus-Prod (m, n) = (m + n, m n) = (*Plus* (m, n), *Prod* (m, n)) (8)

In other words, the function **Plus-Prod** takes a pair of elements m and n of N and maps this pair (m, n) to a pair of integers, viz, (m + n) and (m **.** n)

Also, we may define the function
Plus-Prod-Exp: N^2 N^3 with
Plus-Prod-Exp (m, n) = (m + n, m n, m^n)
= (*Plus* (m, n), *Prod* (m, n), *Exp* (m, n)) (9)

Here the ordered pair (m, n) is mapped to the ordered triple of three integers, viz,
(m + n), (m **.** n) and m^n

Example 4: of function from N^k N^q N^p where k, q, p N
A somewhat similar but distinct function say
New-Plus-Prod-Exp: N^2 N^2 N
may be defined as

New-PIus-Prod-Exp (m, n) = ((m+n, m n), m^n))
= (*Plus-Prod* (m, n), *Exp* (m, n)) (10)

Please note the minute difference between Plus-Prod-Exp and New-Plus-Prod-Exp

Remarks 5

In the definitions under (5) to (10) above, among other facts, we may observe that earlier defined functions may be used in defining more complex functions. Our ability to define more and more complex functions in terms of earlier defined functions, plays a very important role in the study of primitive recursion and partial recursion etc, which are generalizations of the concept of recursion discussed in defining n! and x^n etc.

The recursive definitions of Plus, Prod etc. will be discussed later.

The constant Functions:

Though it is not intuitive, yet we may have functions on N which do not require any argument.

Consider the function

C_5 : N   N   such that
C_5 (n) = 5,   for all n   N (11)

In view of the fact that the value 5 is independent of n in (11), we can very well write (11) as

C_5 ( ) = 5, (12)

Given the fact that we are considering domains of functions as N^k for k   N, we extend our notation for functions from N^k N to include functions from N^0 N, and rewrite (11) as

C_5 : N^0 N   such that
C_5 ( ) = 5.

Also, in order to include in the notation itself the fact that the function takes zero number of arguments, we may use the notation C^0 instead of C , i.e.,

C^0_5 = 5 (13)

Generalizing the constant function C^0_5 we may define

C^0_q : N^0 N   such that
C^0_q () = q,   for some fixed integer q in N.

Further, we can extend the set of constant functions to include the functions

C^k_q : N^k N   such that
C^k_q (n_1, n_2, …, n_k) = q,
for n_1, n_2, …, n_k N   and for some integers k and q in N. (14)

Partial Function

We are already familiar with the concept of *function* in the mathematical sense. Informally, for two given sets X and Y a **function**
f : X   Y
is a **rule** that associates to **each** element x of X a **unique** element y of Y. Here X is called the *domain* of the function f and Y the *codomain* of f. (15)

However, in order to extend the class of computable functions beyond the class of primitive recursive functions (*to be defined*), to parial-recursive functions (*to be defined*), we relax the condition *'for each element x of X'* in the definition of function leading to the following definition.

Partial Function: A partial function is a **rule**

f: X   Y

that associates elements of Y to elements of X in such a way that, for y_1 Y **if there exists** an element x_1 of X s.t. $f(x_1) = y_1$, then there is **no element** y_2 of Y, with y_1 y_2, s.t. $f(x_1) = y_2$. (16)

However, there *may be some elements* x of X for which there *may not* be any y such that f(x) = y. In other words, in the definition of a *partial* function, it is *not necessary* that *for each* element x of X, there *must* be an element y of Y that corresponds to x under f. However, for an element x of X, *if* there is an element y_1 of Y that corresponds to x under f, then there *can not* be a y_2 in Y with y_1 y_2 such that y_2 also corresponds to x under the partial function under consideration.

Example 6: We consider a rule of correspondance *Quot:* N   N   N that takes a pair (m, n) of integers and associates an integer q, if it exists, s.t. m = nq + r with 0   r < n. Now if n =0 then no r with 0   r < 0 exists implying *Quot* (m, n) is not defined for n = 0. Thus *Quot* is a *partial* function, but not a *function* or a *total* function as is going to be defined below.

Total Function: If a partial function satisfies the condition given in (15), i.e., it is a function in the conventional sense, then it will be called **Total Function**. The adjective *total* is added to a function in conventional sense in order to differentiate the function in conventional sense from the *partial* function which satisfy condition (16) but do not satisfy the condition (15) above.

Remarks 7

In this block, **unless it is mentioned otherwise** we will be dealing with functions (partial or total), the *domains of which are only of the form* N^k = N x … x N for k   N.

Functions with domain N^k are called ***k-place functions***.

Also, unless it is mentioned otherwise, the functions under consideration are restricted to the ones that have N as their **codomain.** Our consideration of only the functions of the form f : N^k N, is not a major restriction, because using some encoding techniques like *Gödel Numbering*, any domain can be expressed as a subset of the set N^k.

Why we need partial functions?

We know there are infinitely many possible sets which can be represented by finite means. For example the infinite set **N is finitely representable by the following two statements:**

(i) 0 is a member of N and
(ii) if n is a member of N then so is   (n) (i.e, (n + 1)).

And for each such non-empty set X, at least one function, say the identity function I : X   X, can be defined. Also, for such sets X, Y, Z etc., we can think of new sets which may be the product sets, for example X   Y, Y   Z   X, just to name a few. Each of these product sets itself can be the domain (or even the range) of some functions.

Thus, unless we use an ingenious method of the type described below, the general discussion of functions would involve consideration of *infinitely many types* of domains and codomains, even if, each of these may be finitely representable.

By an appropriate encoding, it can be easily seen that each countable set can be thought of as either N^k or as a proper subset of N^k, for some
$k = 0, 1, 2,$ ----

For example

The set X={a, b, c, …., z}can easily be thought of as a subset of N, by using the following encoding

a   1
b   2
.
.
.
.
z   26

Thus, functions, in stead of being considered between arbitrary but countable domains and codomains, may be considered as functions of the form P N^m, where P either equals an N^k or is a proper subset of N^k, for suitable integers k and m.
However, any function f: X N^m for X = {a, b, ..., z} above when considered after an encoding as a function f: N N^m, **cannot be total**, because f (m) for m   27, is not defined. In general, any function with a *finite* domain when considered as a function between the encoded sets N^K and N^m must be *strictly partial.*
Also, for large number of functions involved in the solution of everyday problems, each has a finite domain.

Thus, in order to simplify the discussion of functions with arbitrary but countable domains and ranges, it is possible, through appropriate encoding, to consider such a function as a function of the form N^k N^m provided functions are allowed to be *partially defined* **on the domain N^k.**

Examples 8: ***of total and Partial Functions***

(i) **The successor function S**: N   N, s.t., S(x) = x + 1 for all x ∈ N. The function **S is a total function** ***from[†] N to N.*** Successor function plays an important role in the recursion theory. Therefore, it is useful to know that even the **notation is used** to denote *the successor function*.

(ii) The function
Plus: N^2 N   such that for all x, y   N.
Plus (x, y) = x + y,
is also a *total* function

(iii) However, the function
Minus: N^2 N   such that
Minus (x, y) = x – y   for x   y in N,
is only a *partial* function, which is *not* a total function.
In other words, *Minus* is a strictly *partial* function.

[†] While talking of total or partial functions, it is understood that the domain is of the form N^k.

However, a slightly different function say Minus_Int becomes total, if we allow
Minus_Int: N^2 I,

with I, the set of integers as codomain and the rule of correspondance given by
Minus_Int (x, y) = x – y for x, y N. *(in stead of, for just x y)*
However, as mentioned earlier, we are restricting to only functions of the form *f*: N^k N.

Therefore, if required, we discuss only the *strictly partial* function *Minus*.

However, we will discuss a scheme of discussing Minus-Int as a function from N^2 to N^2.

(iv) corresponding to the partial function *minus*, there is a well-known function

monus, also denoted as $\dot{-}$ and defined as

$$\textbf{Monus}\ (x, y) = \begin{array}{ll} x - y & \text{if } x \quad y \\ 0, & \text{if } x \quad y \end{array}$$

Monus (x, y) may also be written as x $\dot{-}$ y.

(v) We define the function **div** from N^2 to N with *div* (x,y) = z, only for those pairs (x,y) of elements of N for which x = y.z for some z from N. Then *div* is strictly partial.

(vi) The **Square-Root function**, named as say SQRT,
(also denoted by $\sqrt{\ }$) and given by
SQRT: N N
such that for x, y N,
SQRT (x) = y if y^2 = x.

Again SQRT is a *strictly partial function.*

After having provided the necessary background, we explain the concept of **primitive recursive** functions the set of which forms a proper subset of the set of total functions. As the discussion of general **partial recursive** functions requires introduction of some more background material, the general partial recursive functions will be discussed later.

We mentioned earlier **that, each of the approaches to computation starts** with some **elementary entities** of some domain and some **structuring rules**, where the rules are easily applicable to form more and more complex entities of the domain.

3.4 PRIMITIVE RECURSIVE FUNCTIONS

The set of primitive recursive functions is obtained by three types of **initial functions** (which are *elementary* primitive functions) and three **structuring rules** for constructing more complex functions from already constructed functions.

In the literature, two of the three structuring rules are combined in one rule and hence, in most of the literature, number of structuring rules is mentioned as TWO and not THREE. However, then presentation of the subject matter becomes too complex from the point of view of undergraduate students.

Three types of initial functions are

(i) The 0-Place *zero function ξ* from N^0 to N such that ξ( ) = 0 (19)

(ii) The *successor function*

  : N N

such that

  (n) = n + 1   for all n N. (20)

(iii) *The Projections:* We know that for k 1, N^k is the set of all k-tuples of the form $\overline{n}$ = (n_1, n_2, …, n_i, …, n_k) for 1 i k.

For each fixed i, with 1 i k, we may define a function, denoted by ${}^{k}_{i}$, with

${}^{k}_{i}$: N^k N such that for $\overline{n}$ = (n_1, n_2, …, n_i, …, n_k)

${}^{k}_{i}\overline{n}$ = ${}^{k}_{i}$ (n_1, n_2, … n_i…., n_k)

= ith component of (n_1, n_2, … $n_{i…..}$, n_k) = n_i (21)

Thus, we have defined *k projection functions*, each with domain N^k, viz.

${}^{k}_{1}$, ${}^{k}_{2}$, …, ${}^{k}_{i}$,…., ${}^{k}_{k}$ and each of which maps to N.

For the sake of explanation, we have ${}^{5}_{3}$ (2, -7, 12, 4, 3) = 12

Finally, the zero-place **zero function ξ**, the **successor function** and the **projection** function ${}^{k}_{i}$ for k N, and i N, with 1 i k, **are the *only* initial functions,** which are also called ***elementary* primitive functions.**

Ex.1) Prove that each of the elementary primitive function viz zero function ξ, successor function σ and each of the projection functions ${}^{k}_{i}$, 1 i k, is a ***total*** function.

In the very beginning itself, it was mentioned that *computability of functions* is a major concern in the Theory of Computation.

In this context, consider the following

Statement 9: The initial functions ξ, and ${}^{k}_{i}$ are all computable .

The statement is not a *theorem*, the truth of which can be established through a *proof*. The statement is axiomatic in the sense that it should not only be intuitively correct but should be fundamentally true in the sense that it can not be otherwise. In spite of the above, we give an informal argument in support of the apparent truth of the statement. The *computability of the initial function ξ* is about our capability of constructing a machine to perform the activity of writing the symbol 0. This capability can be assumed without any doubt. Similarly, our capability of constructing a machine which returns (n +1) for each input n, can also be assumed without any doubt. Hence, we may assume that the successor function σ is computable.

Finally, computability of a projection function say ${}^{k}_{i}$ *,* is about the designing of a machine capable of scanning a k-type say $\overline{m}$ = (m_1, m_2, ……….m_i,……….m_k) starting with the first component m_1 and go on moving to the right till ith component

 Computability in the formal sense of Church-Turing Thesis has been discussed in other unit. Here computability is taken as an intuitive informal notion.

m_i is scanned and then writing m_i as output. In view of the type of machines available, it can be safely assumed that we can construct a machine to execute these activities required for a projection function.

Next, we define the three structuring rules which are known as

(i) combination
(ii) composition and
(iii) primitive recursion.

Remarks 10

Before providing the definitions for the above–mentioned structuring rules, it may be stated that by applying these structuring rules, to begin with, to the initial functions and then by successive applications of these structuring rules to the functions already obtained by previous applications, we can construct quite complex functions

(I) Combination as a structuring rule

The combination of two functions
$g: N^k \quad N$
$h: N^k \quad N$
is a function
$f: N^k \quad N x N$
such that for $(n_1, \ldots . n_k) = \bar{n} \in N^k$,
$f(\bar{n}) = (g(\bar{n}), h(\bar{n}))$. (22)
Then, the function f is denoted by g x h and is called combination of g and h.

Remarks 11

In stead of g: $N^k \quad N$ and h: $N^k \quad N$, we may take g: $N^k \quad N^m$ and h: $N^k \quad N^n$, and ***then*** *we get a function f: $N^k \quad N^{m+n}$ from the definition of f given by (22).*

(II) Composition as a structuring Rule

Let
$g: N^k \quad N^p$ and
$h: N^p \quad N^q$ for $k, p, q \quad N$
be two given functions.
Then we define a function
f : $N^k \quad N^q$
as follows:
if $(n_1, n_2, \ldots n_k) \quad N^k$ and
$g(n_1, n_2, \ldots n_k) = (m_1, m_2, \ldots m_p) \quad N^p$
Further, if
$h(m_1, m_2, \ldots m_p) = (t_1, t_2, \ldots t_q) \in \mathbf{N^q}$
then
$f: N^k \quad N^q$ is such that
$f(n_1, n_2, \ldots n_k)$
$= h(g(n_1, n_2, \ldots n_k))$
$= h(m_1, m_2, \ldots m_p) = (t_1, t_2, \ldots t_q)$ (23)
The function f, so defined, is called the **composition** of g and h and **is denoted by h.g** *(Some authors use the notation g·h instead)*

Examples 12:

(i) If $g: N \quad N$ is given by

$g(n) = n^2$ and
$h : N \quad N$ is given by
$h(n) = (n + 3)$

Then we define a function
$f : N \quad N$ such that
$f = h\cdot g$, then
$f(n) = h(g(n)) = h(n^2) = n^2 + 3$ for all $n \in \mathbf{N}$

(ii) Functions g and h are as given in Example 12 (i) above. Let us define a function
$k: N \quad N$
such that $k = g\cdot h$, then
$k(n) = g(h(n))$
$= g(n+3) = (n + 3)^2$
for all $n \in N$

Ex. 2) What is the result of applying the function $^{2}_{2}\quad ^{2}_{1}\quad ^{4}_{1}\quad ^{4}_{4}$ to four tuple (8,7,4,2)?, where $(f \times g)(x) = (f(x), g(x))$

Ex. 3) A function $f: N \quad N$ is defined as
$f(0) = \quad (\,)$ and
$f(y) = \quad f(y - 1)$
What is the value of f(4)?

Remarks 13

As mentioned earlier the two structuring rules discussed so far, viz, combination rule and composition rule, are presented in the literature as a single rule, and is generally called as composition rule, which is actually a generalization of our composition and combination rules. We call this rule as ***Generalised Composition Rule*** *and discuss it below:*

Let the function

$g: N^k \quad N^p$ and
$h_1, h_2, \ldots, h_k$ are k functions with
$h_i : N^m \quad N$, for $i = 1,2,\ldots, k$

Then we define a function

$f: N^m \quad N^p$ such that for $\bar{n} = (n_1, \ldots n_m) \in N^m$
$$f(\bar{n}) = g(h_1(\bar{n}), h_2(\bar{n}), \ldots, h_k(\bar{n})). \qquad (24)$$

Then f is said to be obtained from g, $h_1, h_2, \ldots, h_k$ by *generalized composition* or if there is no confusion just by *composition*.

Ex.4) Show that the *combination* rule given by (22) and *composition* rule given by (23) are special cases of the *generalized composition* rule given by (24).

Next, we consider the *third structuring rule, viz Primitive Recursion*. As the rule requires comprehensive discussion, we discuss it in an independent section below.

3.5 INTUITIVE INTRODUCTION TO PRIMITIVE RECURSION

Earlier, we considered the recursive definition of n! for n N as:
0! = 1 and
n! = n ((n –l)!) for n l.

Also, we considered for recursive definition of the exponential x^n of x, a real number and n N as:

$x^0 = 1$ and
$x^n = x \; x^{n-1}$ for n 1.

In order to understand the **generalization of the recursive definitions** considered above, let us consider the following example.

Example 14: As an Intuitive Introduction to Primitive Recursion

Let us consider a special kind of tree that initially has only one branch, which is treated as a *new* branch. At the end of each year, out of each new branch, m new branches grow out (**we call m as the branching factor of the tree**). And the branch, out of which branches grew out once, is no more a new branch. We define a *function say* f which gives the total number of branches in the tree after *n years*, assuming the branching out process continues for ever (or at least for more than n years) at the rate of *m branches* per year.

It is clear that the function f depends on both m and n. In order to facilitate the understanding of the process of getting f as a function of m and n, *let us initially consider m as a constant.* Also to begin with, we consider only *the* ***function b(n)*** that returns the number of *new* branches that are generated at the end of the nth year for n = 1, 2, ……. Subsequently, *b (n) shall be used in computing f (m, n),* where

b(1) = After one year, number of **new** branches = m
b(2) = After two years, number of **new** branches. = m **.** b (1) = m **.** m = m^2
b(3) = After three years, number of **new** branches = m (number of new branches after 2 years) = m^3.

Continuing like this, we get,

b(n) = After n years, number of new branches
= m(number of new branches after (n – 1) years)
= m b (n –1) = mm^{n-1} = m^n

Next, we consider **f(m, n)**, the number of **all** branches at the end of n years.

f (m, 1) = **Total** Number of branches after one year
= old branches at the end of one year + new branches generated at the end of the first year, i.e.,
f(m, l) = l + m
f (m, 2) = Total number of branches after two years
= Old branches at completion of two years + New branches generated at the end of second year, i.e.,
f (m, 2) = f(m, 1) + m^2
f (m, n) = Continuing like this, total number of branches after n years
= Total number of branches after (n – 1) year + New branches at the end of the nth year, i.e,
= f(m, n –1) + m b(n –1), i.e,
f(m, n) = f(m, n – 1) + m^n (25)

For a short while, if we denote f(m, n – 1) as t
Then (25) becomes

f(m, n) = $t + m^n$, a function of m, n and t, say
= h (m, n, t), for some function $h : N^3 \quad N$.

Summerising, we get

f(m, n) = h (m, n, t), for some function $h: N^3 \quad N$.
Replacing t by f (m, n-1), we get
f(m, n) = h(m, n, f(m, n – 1))

Thus f(m, n), the number of all branches immediately on completion of n years can be defined as:

f(m, 0) $=1 = m^0$
f (m, n) = h (m, n, f (m, n – 1)) (26)
where
h (m, n, t) = $m^n + t$

In order to further generalize the concept of recursion, *we consider the same problem with a little difference* by taking a new *starting time* (i.e., zeroth year) for the problem of counting of the number of branches when the tree already has, say, $1+ m + m^2 + m^3$ branches. (i. e., the initial branch is already 3 years old) and let **J (m, n)** *denote the number of total branches after n years* of the new starting time, where m is the branching factor.

Then J(m, n) is defined as

J (m, 0) = $1 + m + m^2 + m^3$, which is some function say g(m) of m, i.e.,
J(m, 0) = g(m) and
J(m, n +1) = J(m, n)+m^{n+4},
which is some function say L of m, n and J (m, n), i.e,
J (m, n + 1) = L (m, n, J (m, n)).
Summarizing, the function J (m, n) may be defined as
J (m, 0) = g(m) and (27)
J (m, n +1) = L (m, n, J(m, n)), (28)

for some function g of m and another function L of m, n and J (m, n).
*The above discussion can be generalized still further when instead of one tree, initially, we have **k** trees $T_1, T_2, \ldots T_k$, s.t. for the ith tree T_i the branching factor is m_i for i = 1, 2, ..., k.*

Also, we assume that different trees started growing (i.e., having their first branches) in different calendar years. After all these trees have started growing, some calendar year is taken as *starting or zeroth* year for the purpose of counting the number of branches in all the trees taken together.

Then (27) and (28) can be rewritten as

J_i (m_i, 0) = g_i (m_i)
J_i (m_i, n + 1) = L_i (m_i, n, J_i (m_i, n)) for i = 1, 2, ..., k. (29)
(where g_i (m_i) denote the number of branches, in the zeroth year, of the ith tree whose branching factor is m_i)

Then total number of branches on all the k trees, after completion of n years after the starting year, may be defined *by a function F of $m_1, m_2, \ldots m_k$, and n* as follows:

F($m_1, m_2, \ldots m_k$, 0) = $g_1(m_1) + g_2(m_2) + \ldots + g_k(m_k)$ = G ($m_1, m_2 \ldots.., m_k$),
for some function G of $m_1, m_2, \ldots m_k$, and
F($m_1, m_2, \ldots m_k$, n + 1) = J_1 (m_1, n + 1) + J_2 (m_2, n + 1) + …. + J (m_k, n+1) (30)
But by (29), each J_i (m_i, n +1) is a function of J_i (m_i, n)
If $\overline{m}$ = ($m_1, m_2, \ldots., m_i, \ldots., m_k$)

Then in view of these facts, i.e,

(i) $F(\overline{m}, n+1)$ is a sum of $J_i(m_i, n+1)$ by (30)

(ii) each $J_i(m_i, n+1)$ is a function of $J_i(m_i, n)$ in addition to being a function of m_i and n by (29)

(iii) the sum of $J_i(m_i, n)$, 1   i   k, is an expression which can be obtained by replacing n + 1 by n in the R.H.S of (30) and hence, by replacing (n + 1) by n in the L.H.S of the equality (30),

this sum of $J_i(m_i, n)$'s must be of the form $F(\overline{m}, n)$

In view of (i), (ii) & (iii) above, $F(\overline{m}, n+1)$ is some function H of $F(\overline{m}, n)$ in addition to being function of m_i' s and n

Then the above definition of F may be rewritten as

$$F(\overline{m}, 0) = G(\overline{m}) \quad \text{and}$$
$$F(\overline{m}, n+1) = H(\overline{m}, n, F(\overline{m}, n)) \qquad (31)$$

This is exactly what, in formal sense, we say that F is obtained from G and H by primitive recursion. Restating the above, we get the formal
Definition: Primitive Recursion
For k   0, a function
f: N^{k+1} N^m
is said to be constructed *using primitive recursion from the functions*
g : N^k N^m and
h : N^{k+m+1} N^m,
if, for $\overline{x} \in N^k$ and $y \in N$,

$$f(\overline{x}, 0) = g(\overline{x}) \quad \text{and}$$
$$f(\overline{x}, y+1) = h(\overline{x}, y, f(\overline{x}, y)), \qquad (32)$$

The above discussion about three *initial functions* viz. the zero-place *zero function* ( ), the *successor function* σ and the projections $^{k}_{i}$ and about the three *structuring* rules viz. *combination*, *composition* given by the discussion preceding and including (22), (23) and *primitive recursion* given by (31), leads to **the following definition**:

Primitive Recursive Function: A function f is **primitive recursive function** if (and only if) either

(i) it is one of the initial functions viz. ξ ( ), or one of the projections $^{k}_{i}$, i   k, or

(ii) it is obtained by application of some finite sequence of structuring rules viz combination, composition, and primitive recursion to the initial functions.

Remark 15

It is implied in the above definition that if a function f is obtained by a finite sequence of application of structuring rules including primitive recursion to some functions (*not necessarily initial functions*) say $g_1, g_2, \ldots g_k$, each of which has been obtained earlier by application of some finite sequence of combination, composition and *primitive recursion* to some of the *initial functions*, then F must be *primitive recursive*. The implication follows from the fact that F can be obtained from initial functions by first applying sequences of combination, composition and primitive recursion to obtain each of $g_1, g_2, \ldots g_k$, and then applying a sequence of combination, composition and primitive recursion to get f from $g_1, g_2, \ldots g_k$.

Examples 16: All functions considered below are from N^k to N, for some k N.

Example 16(i): *The well-known binary function plus (m, n) = m + n is primitive recursive*, because

$$Plus\ (m, 0) \quad = \quad {}^{1}_{1}\ (m)$$

$$Plus\ (m, n + 1) = \quad {}^{3}_{3}\ (m, n, plus\ (m, n)),\ for\ n \quad 0 \qquad (33)$$

Example 16 (ii): *The well-known binary function Product (written here as prod) and given by*
prod (m, n) = m n is primitive recursive, because

$$\text{prod (m, 0)} \quad = \xi\ (\ )$$

$$\text{prod (m, n + 1) = plus (} \ {}^{3}_{1}\ \text{(m, n, prod (m, n)),} \quad {}^{3}_{3}\ \text{(m, n, prod(m, n)))} \qquad (34)$$

As plus has already been shown to be primitive recursive, therefore, prod is also primitive recursive.

Example 16 (iii): In this example we consider simple funtion say *int-plus whose domain and codomain are not N but I, the set of all integers* including negative integers and zero. We show that the function

int-plus : I x I → I

is primitive-recursive.

The proof is based on the fact that *Each integer can be thought of as a member of N^2*, for example, 5 may be thought of as (6, 1) and –5 as (1, 6). In general, (m, n) ∈ N^2 denotes the integer m — n.

Then the function int-plus : I x I I can be thought of as

int-plus : (N x N) x (N x N) N x N
Such that, if (m_1, n_1) and (m_2, n_2) ∈ N x N then
int-plus $((m_1, n_1), (m_2, n_2)) = (m_1 + m_2, n_1, n_2)$
$= (\text{plus}\ (m_1, m_2), \text{plus}\ (n_1, n_2))$ (35)

Plus is already shown to be primitive recursive and combination of two primitive recursive functions (viz plus and plus) is primitive recursive. Hence the above equation (35) shows that *int-plus* is a primitive recursive function.

Example 16(iv): *The factorial function*

f(n) = n! *for n∈N,*
is primitive recursive.

The proof follows from the following argument based on Principle of Mathematical Induction:

Base Case:
for n = 0
f(0) = 0! = 0 = ξ ( )

Induction Hypothesis:
Let f(p) be primitive recursive

Induction step:
$f(p + 1) = (p + 1)! = (p!)\cdot (p+1) = f(p)\cdot (p + 1) = prod\ (f (p), p + 1).$

Using Induction Hypothesis and the fact that product is primitive recursive, f (p +1) is primitive recursive.

Generalizing the above example, we get the

Theorem 17:
Let

$$g: N^{k+1} \quad N$$

be primitive recursive.
Then, for an m ∈ N, the function
$f: N^{k+1} \quad N$
given by

$$f(\bar{n}, m) = {}^{m}_{i\;0}\quad g(\bar{n}, i) = g(\bar{n},0)\cdot g(\bar{n},1)\cdots g(\bar{n},m) \qquad (36)$$

with $\bar{n} = (n_1,n_2,\ldots n_k) \in N^k$,
is primitive recursive

Proof:

We prove the result by Principle of Mathematical Induction on m

Base case:
When m = 0, by (36), we get

$$f(\bar{n}, 0) = g(\bar{n},0).$$

As g is given to be primitive recursive, the base case follows

Induction Hypothesis:
for m = p we assume

$$f(\bar{n}, p) = {}^{p}_{i\;0}\quad (g(\bar{n}, i))$$

is primitive recursive

Induction Step:
Consider

$$f(\bar{n}, p +1) = {}^{p\;1}_{i\;0}\quad (g(\bar{n}, i)) = {}^{p}_{i\;0}\quad (g(\bar{n}, i))\cdot g(\bar{n}, p + 1)$$

$$= f(\bar{n}, p) \cdot g(\bar{n}, p + 1) = prod\ (f (\bar{n}, p), g (\bar{n}, p +1)).$$

In view of the Induction Step and the fact that both g and product are primitive recursive, we get $f(\bar{n}, p + 1)$ is primitive recursive.

Definition: The function f, given by (36) above, is said to be obtained from g by Bounded Product. Thus the above theorem may be restated as

Theorem 17:
Bounded Product of a primitive function is primitive recursive

---

Ex.5) Show that each of the following, earlier defined, functions is primitive recursive:
(i) Plus-Prod (*given by equation(8)*)
(ii) Exp
(iii) New-Plus-Prod-Exp (*given by equation (10)*)

Ex. 6) Show that the predecessor function pred: N N defined as

$$\text{pred}(n) = \begin{matrix} 0 & \text{if n } \ 0 \\ n \quad 1 & \text{if n } \ 1 \end{matrix}$$

is primitive recursive.

We recall the definition of constant functions:

For each k 0 and each j 0, a *constant function* C_j^k maps each k-tuple (m$_l$, m$_2$, ... m$_k$) to the fixed integer j, i.e,

C_j^k : N^k N such that C_j^k (m$_l$, m$_2$, ..., m$_k$) = j, for all (m$_1$, m$_2$, …, m$_k$) ∈ N^k.

We show that C_j^k are all primitive recursive functions. To begin with, consider the

Lemma 18: Each of the functions C_j^0 for j 0 is primitive recursive.

Proof: The proof is presented in two parts:

Case (i) C_0^0 is the function which maps a zero-tuple to the constant 0. It is primitive recursive, because

C_0^0 =

Case (ii) Each of the functions C_j^0 for j 1 is primitive recursive

As C_1^0 = 1 = σ . ξ,

therefore, C_1^0 is primitive recursive

Again as C_2^0 = · C_1^0

and C_1^0 is already shown to primitive recursive, therefore, C_2^0 is primitive recursive.

We use mathematical induction on j to show C_j^0 is primitive recursive for all j.

Base case. For j = 0, we have already shown, that C_0^0 is primitive recursive.

Induction Hypothesis: Let C_m^0 is primitive recursive, for any integer m.

Induction step

$C_{m\ 1}^0$ = C_m^0 By induction hypothesis, C_m^0 is assumed to be primitive recursive and is primitive recursive and composition of two recursive functions is primitive recursive, therefore, $C_{m\ 1}^0$ is primitive recursive.

Hence by Principle of mathematical induction, C_j^0 is primitive recursive, for all j N.

The lemma proves C_j^k as primitive recursive *only for k = 0.*

The proof for the general integer k follows from the

Theorem 19: The constant function C_j^k**, for k 0 and j 0, is primitive recursive.**

Proof: We prove the theorem by induction on k.

Base case: When k =0, the proof follows from the lemma, in which we proved that C_j^o is primitive recursive *for all j*.

Induction Hypothesis: Assume C_j^i is primitive recursive, for all integers j and all integers i   p.

Induction step:
Let $\overline{m}$ = (m_1, m_2, ….., m_p) ∊ N^p
Now $C_j^{p\ \ 1}$ ($\overline{m}$, 0) = C_j^p ($\overline{m}$), each of the two sides of the equality, is equal to j
$C_j^{p\ \ 1}$ ($\overline{m}$, n + 1) = $^{p\ \ 2}_{p\ \ 2}$ ($\overline{m}$, n, $C_j^{p\ \ 1}$ ($\overline{m}$, n))
Hence, the theorem is proved.

Let us try the following exercises

Ex. 7) The monus function defined earlier as

monus (m, n) = m - n, if m n
0, otherwise,

is primitive recursive.

Ex. 8) Show that following function

eq (m, n) = 1, if m n and
0, else

is primitive recursive.

Ex. 9) Show that the function minus: I x I I, with
Minus (m, n) = m-n for all m, n ∊ I,
where I is the set of all integers, is primitive recursive

Ex. 10) Show that the function

┐eq (m, n) = 1, if m n
0, if m n

is primitive recursive

Ex. 11) Show that for i ∊N, characteristic functions

K_i (n) = 1, if n i
0, otherwise

is primitive recursive

Ex. 12) Show that the function
f: N N given by

f(n) = 7 when n 0
12 when n 5
8 when otherwise

is primitive recursive.

Statement 20: The structuring rules viz *combination, composition* and *primitive recursion* produce computable functions from computable functions.

Like Statement 9 earlier, no formal proof of the statement is possible. As earlier, we present an informal/intuitive argument in support the apparent truth of the Statement (20).

(i) Composition Rule produces computable functions from computable functions

Let

$g : \quad N^k \quad N^n$ and

$h : \quad N^k \quad N^m$

be computable functions.

Then value $f(\overline{n})$ of $\overline{n} = (n_1, n_2, \ldots.., n_k) \in N_k$

under the function f which is the combination function of g and h, is given by

$(g(\overline{n}),\ h(\overline{n}))$

In other words, if the values $g(\overline{n})$ and $h(\overline{n})$ are computable then the additional computational effort required is that for putting these values between a pair of parentheses separated by a comma. However, a machine having these additional capabilities, in addition to the capabilities of the already existing machines for computing $g(\overline{n})$ and $h(\overline{n})$, can easily be constructed. The above informal argument supports the claim that *combination rule* produces computable functions from computable functions.

(ii) Composition Rule produces computable functions from computable functions.

Let

$g: \quad N^k \quad N^m$ and

$h: \quad N^m \quad N^p$

be computable and

$\overline{x} = (x_1, x_2, \ldots.., x_k) \in N^k$,

then g being computable produces through a computational process, some m-tuple say

$\overline{y} = (y_1, y_2, .., y_m) \in N^m$,

such that $g(\overline{x}) = \overline{y}$

Next, h is computable function with domain N^m and $\overline{y} \in N^m$.

Therefore, the process of getting $h(\overline{y})$ from $\overline{y}$ is computable.

Thus, if we assume computational capabilities already exist for computing $g(\overline{x})$ and $h(\overline{y})$, then for computing the value $h(g(\overline{x}))$ under the composition function of g and h, the only additional computational capability required is that of passing the value $g(\overline{x})$ as an argument to h. This capability can reasonably be assumed.

Thus, the computability of the *composition structuring rule* is justified.

Next, we present an argument for the claim that ***the structuring rule primitive recursion gives computable functions from computable functions****.*

Let us recall that a ***function***

$f : N^{k+1} \quad N^m$

is said to be constructed using primitive recursion

from the functions

$g : N^k \quad N^m$ and

$h : N^{k+m+1} \quad N^m$,

if, for $\overline{x} \in N^k$ and $y \in N$,

$f(\overline{x}, 0) = g(\overline{x})$ and

$$f(\overline{x}, y+1) = h(\overline{x}, y, f(\overline{x}, y)), \quad (32)$$

((32) was used to denote this equation once earlier also).

The claim about computability of f as defined above, is justified by using the Priniciple of Mathematical Induction on the argument y of f ($\overline{x}$, y).

Base case, For y = 0, as g is computable, therefore, for $\overline{x} \in N^k$, g ($\overline{x}$) and hence f ($\overline{x}$, 0) is computable.

Induction Hypothesis: Let us assume that for $\overline{x} \in N^k$, and for some y ∈ N, f ($\overline{x}$, y) is computable.

Induction Step: In view of the assumption under Induction Hypothesis and the fact that h is given to be computable; for h ($\overline{x}$, y, f ($\overline{x}$, y)) and therefore, for f ($\overline{x}$, y + 1) to be computable, the only additional computational capability required is that of passing the value of f($\overline{x}$,y) as an argument to h. This capability can be reasonably assumed.

Thus, we have *informally* argued in favour of the truth of statement.

Theorem 21: Each *primitive* recursive function is a *total* function.

Proof: We know primitive recursive functions are, by definition

(a) either initial functions
(b) or the functions obtained by some finite number of applications of the three structuring rules to initial functions.

First, we show initial functions are total:
By definition

(i) *The Zero function*: : N^0 N, is such that ( ) = 0
Thus is defined *for all elements* of its domain N^0, which is by definition, empty. Thus, is a total function.

(ii) the *Successor function* : N N is such that
(x) = x + 1, *for all x ∈ N, the domain.*
Thus, successor function is also *defined for all elements of its domain* N. Thus is a total function.

(iii) *the projection* ^k_i *with i, k N, and i k.*
is s.t. if $\overline{x} = (x_1, x_2, \ldots. x_i, \ldots, x_k)$ N^k
then $^k_i(\overline{x}) = x_i$ *for all* $\overline{x}$ *N^k, the domain.*

Thus each of the initial functions, is a total function.
Next, we establish that **the structuring rules lead from total functions to total functions.**

(i) *The Structuring Rule: Combination*
Let g: N^k N^m
h: N^k N^n
. be two *total* functions, for which f: N^k N^{m+n} is such that
f = g X h
Then by definition of *total*, *for each* $\overline{x} = (x_1, x_2, \ldots x_i, \ldots x_k)$ N^k
$\overline{y} = (y_1, y_2, \ldots y_i, \ldots y_k)$ N^m such that

$g(\bar{x}) = \bar{y}$ and

$\bar{z} = (z_1, z_2, \ldots z_i, \ldots, z_n) \in N^n$ such that

$h(\bar{x}) = \bar{z}$

Thus *for each* $\bar{x} \in N^k$

$(g(\bar{x}), h(\bar{x})) \in N^{m+n}$

Therefore f = g x h is total, and hence, *combination* of two total functions is total.

(ii) ***The Structuring Rule: Composition***

Let functions

$g: N^k \quad N^m$ and

$h: N^m \quad N^n$

be *total* and $f = h \cdot g$

Then by definition of *total*, *for each* $\bar{x} = (x_1, x_2, \ldots x_i, \ldots x_k) \quad N^k$

$\bar{y} = (y_1, y_2, \ldots y_i, \ldots y_k) \quad N^m$ such that

$g(\bar{x}) = \bar{y}$ and

further, as h from N^m to N^n is a total function, therefore, for each $\bar{y}$ in N^m, there is a $\bar{z}$ in N^n such that $h(\bar{y}) = \bar{z}$.

But then for each $\bar{x} \quad N^K, \quad \bar{z} \quad N^n$

such that $(h \cdot g)(\bar{x}) = h(g(\bar{x})) = h(\bar{y}) = \bar{z} \quad N^n$

Therefore, $f = h \cdot g: N^k \quad N^n$ *is a total function* if *g* and h are total functions.

(iii) *The structuring rule: primitive recursion*

Let $f: N^{k+1} \quad N^m$ be a primitive recursive function which is obtained from the two already defined *total* functions viz

$g : N^k \quad N^m$ and

$h : N^{k+m+1} \quad N$,

as follows:

$$f(\bar{x}, 0) = g(\bar{x}) \text{ and} \tag{37}$$

$$f(\bar{x}, y + 1) = h(\bar{x}, y, f(\bar{x}, y)) \quad \text{for } \bar{x} \quad \mathbf{N^k} \tag{38}$$

Let $\bar{z} = (x_1, x_2, \ldots x_k, x_{k+1})$ be an arbitrary element of N^{k+1}. We *show by induction* on the (k + 1) th component of $\bar{z}$ *that f is total*, given that both g and h are *total.*

Base Case: When $x_{k+1} = 0$

Then from (37), using the fact that *g is total* we get that f is *defined for all* $(x_1, x_2, \ldots x_k, 0)$ with $(x_1, \ldots x_k) \quad N^k$

Induction Hypothesis*:* Let us assume that for all $\bar{x} = (x_1, \ldots, x_k) \quad N^k$ and for $y \in N$, f is defined for $(x_1, \ldots x_k, y)$.

Induction step*:* Using the above induction hypothesis and the ***fact that h is total***, the R.H.S of (38) above is defined for all $\bar{x}$ and y. Hence the L.H.S. of (38), i.e, $f(\bar{x}, y+1)$ is total on N^{k+1}.

Hence, primitive recursion leads from total functions to total functions.
Thus, we see that all *primitive recursive functions* must be **total** and, as mentioned earlier, **computable** also.

3.6 PRIMITIVE RECURSION IS WEAK TECHNIQUE

It is natural to ask whether class of all primitive recursive functions cover all computable functions or not? Or in other words, every function, which can be accepted as computable, is also primitive recursive?

The answer to the above is *no*, which is substantiated by the following

Theorem 22: (i) There are *computable* functions which are *not primitive recursive*, and even,

(ii) there are *total computable* functions which are *not primitive recursive*.

Proof: In order to establish the above, it is sufficient to give an appropriate example for each of the two results.
Example for Theorem part (i) We have established that a primitive recursive function is *necessarily* total. Hence a function which is **not total can not be primitive recursive**.

Consider the following function, which has been discussed earlier.
Quot : N N N
s.t.

Quot (x, y) = z if y 0 and x y . z k for 0 k y
undefined if y 0

is not total, i.e., is strictly partial. Hence **Quot** can not be *primitive recursive function*.
Example for Theorem part (ii)

The Ackermann's function A : N N N defined below is *total and computable* function but *not primitive recursive*.

A (0, y) = y + 1
A (x + 1, 0) = A(x, 1)
A(x + 1,y + l) = A(x, A(x +1, y))

The proof, that A is total and computable but not primitive recursive, is beyond the scope of the course.

Existence Theorems & Their Constructive/Nonconstructive Proofs

Many a theorem is an assertion about the existence of objects(s) of a particular type. For example, the assertion, CUBE_SUM: *'There is a positive integer, which can be written as the sum of two cubes of positive integers in two different ways', if proved true is an example of an existence theorem*. There are two ways of proving an existence theorem viz through

(i) a constructive proof
(ii) a non-constructive proof.

A *constructive proof* of an existence theorem is *actually about showing* an object of the required type. E.g, writing

$1729 = 10^3 + 9^3 = 12^3 + 1^3$,

provides a *constructive proof* of the CUBE –SUM assertion.

In many cases of existence theorems, either it is quite difficult to produce a constructive proof or the constructive proof is not known, then we use a *nonconstructive* method to prove an existence theorem.

Non–Constructive Proof: Sometimes, we do not (rather we are unable to) show the existence of an object of the required type. In such cases, we prove an existence theorem by some non-constructive method of establishing the truth of the existence theorem. A non-constructive method shows that *some* element of the required type must exist, but the method is not able to tell exactly which is the element of the required type. We give below two examples of non-constructive proofs of existence theorems.

The first non-constructive existence proof is about the claim: The polynomial equation

$5x^{1001} + 23x^{93} + 37x^{17} + 52x - 88 = 0$
has a real root.

The truth of the claim is based on the following well-known result:

A polynomial equation p(x) = 0 of degree n and having real coefficients, has n complex roots (not necessarily all distinct) and for each complex root a + ib with b 0, a – ib is also a root of P (x) = 0.

As a consequence, if P(x) is odd degree, it must have a real root. However, it is quite difficult to find out the real number, which is a real root of the given polynomial equation given above.

The next non-constructive proof is about a well-known result: These exist irrational numbers x and y such that x^y is a rational number.

The following argument establishes the truth of the above result: We know $\sqrt{2}$ is an irrational number, but we do not know whether $(\sqrt{2})^{\sqrt{2}}$ is irrational OR not. **If** $(\sqrt{2})^{\sqrt{2}}$ **is rational then** x and y each equal to $\sqrt{2}$ are the required rational numbers. **However, if** $(\sqrt{2})^{\sqrt{2}}$ **is irrational then** $x = (\sqrt{2})^{\sqrt{2}}$ and $y = \sqrt{2}$ are two irrational numbers such that $((\sqrt{2})^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^2 = 2$ is rational.

However, in the argument above, we exactly do not know whether the required pair is $(\sqrt{2})^{\sqrt{2}}$ *and* $\sqrt{2}$ ***or*** $\sqrt{2}$ *and* $\sqrt{2}$.

We give below a non–constructive proof of the theorem***: There is a total computable function which is not primitive recursive.***

Second Proof of Theorem 21 (ii)

All the functions in the following argument are assumed to be of the form
f: N N. only. Let us assume that the above statement is false, i.e, we *assume that every total computable function is primitive recursive.* Then, *we use Cantors'* Diagonalization Method, (as is used in showing the existence of a non-rational real number) to arrive at a contradiction.

The representation of a primitive recursive function is obtained by applying finite number of times the structuring rules to the initial functions ξ, σ, ${}^{k}_{i}$. The *representation* of a function which is obtained by an application of a structuring rule to initial functions gives the function as a finite sequence of symbols, e.g, σ . ${}^{k}_{i}$ *(m_1, ..., m_k)* uses only finitely many symbol. Each structuring rule adds only finitely many additional symbols to get the representation of a new function from that of already defined function. Thus each primitive recursive function must be representable as a finite sequence of symbols. We arrange the primitive recursive functions according to the number of symbols in the sequence representing the functions, *starting with the* one with least number of symbols in it, *followed by* the one having least number of symbols among the remaining. Among function represented by equal number of symbols, we use dictionary type of ordering. Thus, all the sequences of symbols representing the primitive recursive functions can be written in the form of an ordered table starting at the top with a function having least number of symbols in its representation. According to the order of the function in the table we name the functions, with the top one named as f_1, next as f_2 and in general nth function in the table being called f_n,

Next, we construct a new function

g: N N such that

$g(n) = f_n(n) + 1$ (39)

In other words, the value under function g of the argument n ∈ N, is obtained by taking value under the nth function f_n of n and then adding 1 to it.
As f_n is primitive recursive for each n, the value $f_n(n)$ exists and is obtainable in finite number of steps. Also, adding 1 is only one additional step to get g(n) from $f_n(n)$. Also as f_n is total, therefore for each n∈N, $f_n(n)$ exists and hence $f_n(n)+1$ exists and hence for each n∈N, g (n), being equal to $f_n(n)+1$, exists. *Thus g(n) is also total and computable and its value at n differs from the value of f_n, because*

$g(n) = f_n(n) + 1 \quad f_n(n)$,
for each f_n in the table.

Thus g is not in the table of *all* the primitive recursive functions, i.e., g is *not* primitive recursive.

The last statement contradicts the assumption that every total computable function is primitive. Hence the assumption is wrong, thereby proving the theorem.

Thus, we have proved that the class of primitive recursive functions is a proper subclass of the class of total computable functions.
Thus primitive recursion as a technique for constructing computable functions is **weak** in the sense that it is not able to construct even such simple functions as *Quot*. *The above discussion suggests that the formal technique of primitive recursion should be further strengthened, so that, the enhanced formal technique captures all such functions which are otherwise, easily seen to be computable. One such technique, called* ***unbounded*** *minimalisation, is discussed in the next section.*

3.7 THE TECHNIQUES OF UNBOUNDED MINIMALISATION, PARTIAL RECURSION AND μ-RECURSION

In order to achieve the goal mentioned in the previous paragraph, we first consider the

Definition: Unbounded Minimalisation
For a *given* function

$g : N^{k+1} \quad N$
we *define* a function
$f: N^k \quad N$ such that
for $\bar{x} = (x_1, x_2, \ldots, x_k) \quad N^k$ and for some $y \quad N$,
$f(\bar{x})$ equals y

if (and only if) the following conditions are satisfied:

(i) $g(\bar{x}, y) = 0$ and

(ii) if $g(\bar{x}, z) = 0$ then $y \quad z$
(*i.e., y is the smallest among all the values z N for which $g(\bar{x}, z) = 0$*)

(iii) $g(\bar{x}, u)$ is defined for all $u \quad y$, with $u \quad N$. (40)

Further, *if*, for some $\bar{x} \quad N^k$, such a y does not exist, *then* $f(\bar{x})$ = *undefined*.
Such a function f is said to be obtained from g through unbounded minimalization and is denoted as

$$\mathbf{f(\bar{x}) = \quad y\, [g(\bar{x}, y) = 0]}$$

Example 22: Let $g : N \quad N \quad N$ be defined by the following table.

g(0, 0) = 5	g(l, 0) = 5	g(2, 0) = 8	g(3, 0) = 1
g(0, 1) = 4	g(l, 1) = 6	g (2,3) = 0	g(3, 1) = 2
g(0, 2) = 6	g(l, 2) = 0	g(2, l) = 5	g(3, 2) = 0
g(0, 3) = 0	g(l, 3) = 3	g(2, 2) = *undefined*	g(3, 3) = 4
g(0, 4) = 1	g(l, 4) = 0	g(2, 4) = 7	g(3, 4) = undefinied

Then
f(0) = 3
f(1) = 2 *(though g(1, 4) = 0 also, but 2 is the minimum k such that f (1, 2) = 0)*
f(2) = ***is not defined**, because g (2,3) =0, yet g (2,n) is undefined for n = 2 which is less than 3.*
f(3) = 2 *(though g(3, 4) is undefined for y = 4 but then 4 > 2 and g(3, 2)=0)*

*As can be seen from the above example, minimalisation can be **defined** for functions that are **undefined** for some values of the domains. Also, minimalisation **may produce** functions that are **undefined** for some values of the domain.*

Also, unbounded minimalization may lead from total functions g: $N^{k+1} \quad N$ to partial functions f: $N^k \quad N$.

For Example 23:

$$g(n, m) = \begin{cases} m \quad 1, & \text{for all } m \quad n \quad 10 \\ 0, & \text{for } m \quad n \quad 10 \\ n, & \text{otherwise} \end{cases}$$

Then obviously g: NxN N is total, but, f: N N is such that f (n) is not defined for $n \quad 11$.

Also the converse may happen, i.e., unbounded minimalization may lead from some partial functions g: $N^{k+1} \quad N$ to total functions f: $N^k \quad N$. For example

Let g(n, i)= $\begin{array}{ll} n \quad i & \textit{f}\text{or i} \quad \text{n} \\ \text{undefined} & \text{if } i \quad \text{n} \end{array}$

Then we can see that f (n) = n   for all n

Thus unbounded minimalisation leads from a strictly partial function g to a total function f.

Using the technique of unbounded minimalization, we extend the set of computable functions to the class of µ-recursive Functions, also called *Partial Recursive Functions*. The new class *includes* the class of Primitive Recursive Functions as its proper subclass.

Remarks24

The reason for the use of the adjective **unbounded** before minimalization lies in the fact that, there is **no** bound, on the argument, upto which we are required to try to find a y, which satisfies (i) and (iii) under (40).

Remark 25

Problems with unrestricted application of unbounded minimalisation to a primitive recursive function.

If g is an arbitrary primitive recursive function, then there is no general method of telling whether a y that satisfies all the three conditions of unbounded minimalisation given by (40), exists. In other words, unbounded minimalisation applied to an arbitrary primitive recursive function, *may* not yield a function which may be computable in any intuitive sense (the proof of the claim is beyond the scope of the course).

Remark 26

Bounded Minimalisation: **On the lines of the definition of unbounded minimalisation, we can define bounded minimalisation, for a given integer, m, of a partial function**

g: N^{k+1} N

as a function

f: N^{k+1} N   such that

for $\bar{x}$ = (x_1, x_2, …., x_k) ∊ N^k, m ∊ N

and y ∊N with y   m

f($\bar{x}$, m) equals y

if (and only if) the following conditions are satisfied:

(i) g ($\bar{x}$, y) = 0

(ii) if g ($\bar{x}$, z) = 0 then y   z

(i.e, y is the smallest among all values z ∊ N for which g ($\bar{x}$, z) = 0)

(iii) g ($\bar{x}$, u) is defined for all u   y with u ∊ N

Further, if, for some $\bar{x}$ ∊ N^k, such a y (m), does not exist, then f($\bar{x}$) = undefined.

Such a function f is said to be obtained from g through *bounded* minimalisation and is denoted as

$$\mathbf{f(\bar{x}, m) = \mu\, y \quad m\, [g(\bar{x}, y)]}$$

However, through the following theorem, we show that bounded minimalization is not a powerful technique to extend the class of primitive recursive functions to more general class of computable functions.

Theorem 27: If the function $f(\bar{x}, m)$ is obtained by bounded minimalization for a given integer m and *when applied to only primitive recursive function* $g(\bar{x}, y)$, then $f(\bar{x}, m)$ must be primitive recursive (*however, only the last value may be 'undefined'*).

Proof: Define the functions

$$h_i(\bar{x}) \;:\; N^k \qquad N^{i+1}$$

as follows:

$h_0(\bar{x}) \;=\; g(\bar{x}, 0)$
$h_1(\bar{x}) \;=\; (g(\bar{x}, 0),\;\; g_2(\bar{x}, 1))$

As $h_1(\bar{x})$ is obtained by combination rule applied to values of a primitive recursive function $g(\bar{x}, y)$, therefore, $h_1(\bar{x})$ is a primitive recursive function.
Next, consider

$h_2(\bar{x}) = (h_1(\bar{x}),\;\; g(\bar{x}, 2))$

Again as $h_2(\bar{x})$ is obtained by applying combination rule to two primitve recursive functions $h_1(\bar{x})$ and $g(\bar{x}, 2)$ therefore, $h_2(\bar{x})$ is primitive recursive. Continuing like this, the function $h_1(\bar{x})$ ….. $h_m(\bar{x})$ are all primitive recursive functions. But
$h_m(\bar{x}) = (\ldots(f(\bar{x}, 0), g(\bar{x}, 1)), g(\bar{x}, 2))\ldots, g(\bar{x}, m))$

Thus for any $\bar{x}$, we can compute the row of values $g(\bar{x}, 0)$, …… $g(\bar{x}, m)$ can find the minmum $i \leq m$, if it exists, with $g(\bar{x}, i) = 0$

However, if such an i does not exist then also we able to **determine** that $f(\bar{x}, m)$ is '*undefined*'. **Thus, we can say that bounded minimalizations of a primitive recursive function is primitive recursive**

Remarks 28

At this stage, it is important to note that in **unbounded** minimalization, the number m is not given and hence, in the case of $f(\bar{x})$ the unbounded minimalization of a given primitive function $g(\bar{x}, y)$, **we can not know when to stop finding values** $g(\bar{x}, 0)$, $g(\bar{x}, 1)$………., if all these values happen to be non-zero, before **declaring f $(\bar{x})$ as undefined.**

Therefore, we can not claim that the unbounded minimalization of a primitive recursive function is primitive recursive.

Remarks 29

We already know that primitive recursion is a weak computational technique in the sense that it is not able to show even *div* as computable function. Further, through

Theorem 30: we show that *Bounded Minimalisation* produces only primitive recursive functions from primitive recursive functions. Thus **Bounded**

Minimalisation can not be used as a technique to extend primitive recursion to more powerful computational technique.

Also, under Remarks25, we mentioned that **Unbounded Minimalization** though is more powerful technique, yet, its *unrestricted application* may lead to functions which may not be computable *in any intuitive sense*. Thus we have to find a technique which is a restriction of unbounded minimalisation but is an extension of Bounded Minimalisation. The technique to be described is called μ-recursion *or* partial recursion. The discussion of partial recursion requires introduction of a number of concepts including the

Definition: Regular Function

A function

$f: N^{k+1} \quad N$

is said to be *regular* if (and only if)

for each $\bar{n} \in N^k$, *there is an m* such that

$f(\bar{n}, m) = 0$

In view of the fact that unbounded minimalization may lead from total functions to strictly partial functions, therefore, we need to generalize our definitions of combinations, composition of functions and that of primitive recursion so as to be applicable to strictly partial functions also.

Generalized/New) Combination Rule

Let

$g: N^k \quad N^m$ and

$h: N^k \quad N^n$

be two *partial* functions. Then the composition *partial* function

$f: N^k \quad N^{m+n}$

is defined as follows:

If $\bar{x} = (x_1, \ldots., x_k) \in N^k$

$f(\bar{x}) = (g(\bar{x}), h(\bar{x}))$,

and both the values $g(\bar{x})$ and then $h(\bar{x})$ are defined; **else** $f(\bar{x})$ is undefined.

(Generalized/New) Composition Rule

Let $f: N^k \quad N^p$ and

$g: N^m \quad N^p$

be *partial* functions then

$g \cdot f : N^k \quad N^m$

is given as follows:

Let $\bar{x} = (x_1, x_2, \ldots x_k,) \quad N^k$

Then $(g \cdot f)(\bar{x}) = (g(f(\bar{x})))$, if *both* $f(\bar{x})$ and $g(f(\bar{x}))$ are defined, else g·f is undefined.

Similarly, we have the

(Generalized/New) Primitive Recursion:

Given the *partial* functions

$g : N^k \quad N$ and

$h : N^{k+2} \quad N$

then a (partial) function
$f: N^{k+1} \quad N$
is said to be obtained through partial recursion from g and h, if
$f(\bar{x}, 0) = g(\bar{x})$,
including '*undefined*' as a possible value for g as well as f *and*
$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y))$,
which will have the value '*undefined*' if either $f(\bar{x}, y)$ is '*undefined*' or if $f(\bar{x}, y)$ is defined but $h(\bar{x}, y, if(\bar{x}, y))$ is '*undefined*'.
Now, we define below the concept of μ-recursion, which as a technique for constructing more complex computable functions, subsumes partial recursion and is more powerful a technique than primitive recursion.

Definition: A μ-recursive function is a partial function (including a total function) that can be constructed from the initial functions by a finite number of applications of the (i) combinations, (ii) compositions, (iii) primitive recursions and (iv) unbounded minimalization *to (only) regular functions*.
Remarks 31

The fact of primitive recursion technique is a special case of μ-recursion technique, easily follows from the fact that any primitive recursive function f is obtained by finite numbers of applications of (i) combination (ii) composition and (iii) primitive recursion to initial functions. But then by definition of μ-recursion, f must be μ-recursive function (ii) However, μ-recursion is strictly more powerful a technique than primitive recursive from the facts that *div* is not primitive recursive but is μ-recursive as follows from

Example 32: Show the function **quot**: $N^2 \quad N$ defined earlier as
div (x, y) = {integer portion of x/y if y 0, undefined if y =0},
is μ-recursive, but not primitive recursive.

Hint: *quot is μ-recursive, as*
quot (m, n) = μ t [((m + 1) $\dot{-}$ (prod (t, n) + n)) = 0]
Futher div is a partial function, therefore, it can not be primitive recursive.

Ex. 13) Show that the function SQRT: N N such that SORT (x) = y if and only if $x = y^2$**,** is μ-recursive but not primitive recursive.

Finally we come to the end of this unit with **Church's Thesis** which states: The class of μ-recursive functions contains all computable functions. Church's thesis about μ-recursive functions is parallel of Turing thesis about Turing Machines. Church's thesis claims that the μ-recursion technique is ultimate in constructing computable function in the sense that if a function is not μ-recursive then it can not be computable by any formal technique. As mentioned in the previous unit, similar claim is made by Turing Thesis about Turing Machine Model. **We repeat the claim of Turing Thesis:** Turing Machines possess the power of solving any problem that can solved by any computational means. In the next unit, we discuss the equivalence of the two theses giving rise to what is commonly known as Church-Turing Thesis.

3.8 SUMMARY

In this unit, we introduced the *Theory of Recursive Functions*, which is a *declarative* approach to the study of computational phenomenon. We started with some examples of *recursive definitions* of some functions. Then we introduced the concepts of *initial functions* and *primitive recursion followed by the concept of primitive recursive* *function*. An example to motivate the student for the understanding of the concept of primitive recursion, was given before the introduction of the concept of primitive recursion. Next, we exhibited that *primitive recursion is not strong enough a technique* to capture the computational phenomenon, in the sense that some of even elementary functions, though easily seen to be computable, are not primitive-recursive. Then the notion of total computable functions which subsumes the concept of primitive recursive function was introduced, that captures more functions which are intuitively computable. But again it was shown that even the concept of total computable function is not satisfactory in capturing a number of functions which are, intuitively and even formally, computable. Finally, we discussed *μ-recursion* using *unbounded minimalization* technique to capture essentially all the functions which can be shown to be computable by any formal means.

Also, we established the following inclusion relation ( ) between various classes as: set of **Initial** Functions set of **Primitive Recursive** Functions set of Total **computable** Functions set of **μ-Recursive** Function ≤ set of **partial recursive** function ≤ set of all (partial) functions.

3.9 SOLUTIONS/ANSWERS

Exercise 1 For a function f: X Y to be *total*, we need to show that for *each* element x of the domain X, there is an element y of the codomain Y such that f(x) = y

***ξ is total* :** ξ : N N is such that for *each* n ∈ N, the domain there exists 0 ∈ N, the codomain, such that ξ (n) = 0. Hence ξ is total

***σ is total*:**

σ : N N is such that for *each* n ∈ N, the domain, there exists n + 1 ∈ N, the codomain, such that
σ (n) = n + 1.
Therefore σ is total

$^{k}_{i}$ ***1 ≤ i ≤ k*, is total :** By definition

$^{r}_{i}$: N^k N

is such that
for *an arbitrary element* ($n_1, n_2, \ldots\ldots n_k$) of the domain N^k,

$^{k}_{i}$ ($n_1, n_2, .., n_i, \ldots, n_k$) = n_i

Hence $^{k}_{i}$ is total.

Exercise 2 Consider

$= \quad {}^{2}_{2} \quad {}^{2}_{1} \qquad {}^{4}_{1} \quad {}^{4}_{4} \ (8, 7, 4, 2)$

$= \quad {}^{2}_{2} \quad {}^{2}_{1} \qquad {}^{4}_{1}(8, 7, 4, 2) \qquad {}^{4}_{4}(8, 7, 4, 2)$

$= \quad {}^{2}_{2} \quad {}^{2}_{1} \ (8,2)$

$= \quad {}^{2}_{2}(8,2) \qquad {}^{2}_{1}(8,2)$

= (2,8)

Exercise 3 f(4) = ^{3} f(3)

= ^{3} (^{3} f(2))

= (^{3} ^{3}) (f(2))

$= {}^{6} ({}^{3} f(1))$
$= {}^{9} ({}^{3} f(0))$
$= {}^{12} f(0) = 12$

Exercise 4 **Hint:** n(24)**, take k = 2**, and g as identity function g: NxN NxN i.e, $g(n_1, n_2) = (n_1, n_2)$ for all $n_1, n_2 \in N$.
Then (24) takes the form $f(\bar{n}) = g(h_1(\bar{n}), h_2(\bar{n})) = (h_1(\bar{n}), h_2(\bar{n}))$ for all $\bar{n} \in N^m$,
Which gives f as combination of h_1 and h_2.
Again take k=1 and we get f as the Composition of *h_1 and g.*

Exercise 5 (i) For m, n ∈ N
Plus-Prod (m,n) = (Plus (m,n), Prod (m,n))
Plus and Prod are already shown to be Primitive recursive. And combination of primitive recursive functions gives a primitive recursive function. Hence the proof

(ii) Exp is primitive recursive follows from the following
Exp (m,0) = σ . ξ ( ) and
Exp (m, n + 1) = prod ($^{3}_{1}$ (m, n, Exp (m, n)), $^{3}_{3}$ (m, n, Exp (m, n)

(iii) **Hint:** on the line of Exercise 5 (i)

Exercise 6 Pred (0) = ξ ( )
Pred (1) = ξ
Pred (n + 1) = σ $^{2}_{2}$ (n, pred (n))

Exercise 7 monus (m, 0) = m
monus (m, n + 1) = pred (monus (m, n))

Exercise 8 It can be easily verified that

$Eq (m, n) = 1 \dot{-} ((m \dot{-} n) + (n \dot{-} m))$,
which in formal notation turns out to be
Eq (m, n) = monus (σ ξ ( ), plus (monus (m, n), monus (n, m))
= (monus $^{2}_{2}$ $^{2}_{1}$ monus $^{2}_{1}$ $^{2}_{2}$))) (m, n)

For example

$Eq (4, 1) = 1 \dot{-} ((4 \dot{-} 1) + (1 \dot{-} 4))$
$= 1 \dot{-} (3 + 0)$
$= 0$
Again

$Eq. (4, 4) = 1 \dot{-} ((4 \dot{-} 4) + (4 \dot{-} 4))$
$= 1 \dot{-} (0 + 0) = 1$

Exercise 9 Each *integer* can be thought of as a member of N^2, for example, 5 may be thought of as
(6, 1) and –5 as (1, 6). In general, $(m, n) \in N^2$ denotes the integer m - n.

Then the function *minus*: I x I I can be sought of as
minus: (N x N) x (N x N) N x N such that if (m_1, n_1) & $(m_2, n_2) \in$ N x N then

minus $((m_1, n_1)$ & $(m_2, n_2))$

$= (m_1 + n_2, n_1 + m_2)$
$= (plus\,(m_1, n_2), plus\,(n_1, m_2))$ (39)

Plus is already shown to be primitive recursive and combination of two primitive recursive functions (viz plus and plus) is primitive recursive. Hence the above equation (39) shows that *minus* is a primitive recursive function.

Exercise 10 Hint

$\neg\, eq = monus.\,(C_1^2 \times eq)$

Exercise 11 Hint

$Ki = monus\,(I_i, I_{i-1})$
Where
$I_j\,(m) = eq\,(m \dot{-} j, o)$

Exercise 12 Hint

$f = mult\,(7, k_o) + mult\,(12, k_5) + mult\,(8, mult\,(\neg K_o, \neg k_5))$

Exercise 13 As SQRT is a strictly partial function, therefore, SQRT is not primitive **recursive.**

Further as SQRT (x) = μ t[(1$\dot{-}$Eq (x, prod (t,t)))=0], therefore, SQRT is μ-recursive.

3.10 FURTHER READINGS

Lewis H.R and **Papadimitriou** C.H., *Elements of the Theory of Computation* PHI (1981),

Peter R., *Recursive Functions*, Academic Press (1967)

Epstein Richard L. and **Carmilli** W.A., *Computability, Computable Functions, Logic and the Foundations of Mathematics* (II Edition) Wadsworth & Brooks (2000).

(*A highly readable book presenting advanced level topics from elementary point of view)*

Goodstein R.L., *Recursive Analysis* North-Holland (1961)

Rogers H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill (1967)