UNIT 1 SETS, RELATIONS AND FUNCTIONS

Structure **Page No.**

1.0 INTRODUCTION

In common parlance, we find people using the words given in the title of this unit. Do they have the same meaning in mathematics? You'll find this out by studying this unit. You will also see how basic the concept of 'set' and 'function' or to any area of mathematics and subjects depend on mathematics.

In this unit we will begin by introducing you to various kinds of sets. You will also study operations like, 'union' and 'intersection'. While doing so you will see in what way Venn diagrams are a useful tool for understanding and working with sets.

Next we will discuss what a relation is, and expose you to some important types of relations. You will come across while studying banking, engineering, information technology and computer science, of course mathematics. As you will see in your study of computer science, an extensive use of functions is made in problem-solving.

Finally, we lead you detailed discussion of functions. Over here we particularly focus on various points of functions and fundamental operations on functions.

1.1 OBJECTIVES

After studying this unit, you should be able to:

- explain what a set , a relation or a function is
- give examples and non-examples of sets, relations and functions
- perform different operations on sets
- establish relationships between operations on sets and those on statements in logic
- use Venn diagrams
- explain the difference between a relation and a function.
- describe different types of relations and functions.
- define and perform the four basic operations on functions

1.2 INTRODUCING SETS

In our daily life we encounter collections, like the collection of coins of various countries, a collection of good students in a class, a collection of faculty members of IGNOU, etc. In the first of these examples, it is easy for anybody anywhere to tell

whether an object belongs to this collection or not. If we take the collection of coins of a country, then a coin will be in the collection if it is a coin of that country, not otherwise. The criterion for being a member of the collection is objective and clear. However, if we take the collection of all good students, it is very difficult to say whether a person belongs to this collection or not because the characteristic *good* is not very clearly defined. In this case the collection is not 'well-defined', while the previous collection is 'well-defined'. Similarly, the collection of all the IGNOU students is well-defined.

Definition: A **set** is a well-defined collection. The objects belonging to a set are called **elements** or **members** of that set.

We write the elements of a set within curly brackets. For instance, consider the set A of stationary items used by Nazia. We write this as

A = {pen, pencil, eraser, sharpener, paper}

Another example is the set

B = {Lucknow, Patna, Bhopal, Itanagar, Shillong}

of the capitals of 5 states of India.

Note that A and B are well-defined collections. However, the collection of short people is not well-defined, and therefore, it is not a set.

Also note that **the elements of a set don't have to appear 'similar'**. For example, **{pen,Lucknow,4} is a set** consisting of 3 clearly defined elements.

As you have seen, we usually, denote sets by capital letters of the English alphabet. We usually denote the elements by small letters a,b,x,y …. If x is an element of a set A, we write this as $\mathbf{x \in A}$ (read as 'x belongs to A'). If x is not an element of A, we write this as $\mathbf{x \notin A}$ (read as 'x does not belong to A').

There are three ways of representing sets: 'Set-builder form', 'Tabular form' and the pictorial representation through Venn diagrams.

In the **'Set-builder form',** or **'property method'** of representation of sets, we write between brackets { } a variable x,which stands for each of the elements of the set which have the properties p(x), and separate x and p(x) by a symbol ':' or '|' (read as 'such that'). So the set looks like {x: p(x)} or {x | p(x)}.

For instance, the set {x | x is a white flower} is the set of all white flowers, or {x: x is a natural number and $2<x<11$}is the set of natural numbers lying between 2 and 11.

In **'Tabular form'**, or the **'listing method'**, the elements of a set are listed one by one within the brackets { }, each separated from the other by a comma, as in the examples A and B given above.

The accepted convention for writing a set by the listing method is that elements will not be repeated. For example, in the set A= {4,2,8,2,6}, 2 is repeated, which is not necessary.So we will write A ={4,2,8,6}.

We shall introduce you to Venn diagrams a little later. For now, let us consider a few more sets.

Definition: A set with no element is called the **empty(**or **null**, or **void) set**, and is denoted by ϕ or{}.

For example, A = {x:x is an integer between 13 and 17 which is divisible by 6}, has no element, i.e., A is the **empty set.**

Definition: A set having a finite number of elements is called a **finite set**.

For example, {1,2,4,6} is a finite set because it has four elements, ϕ , the null set, is also a finite set because it has zero number of elements; the set of stars in the sky is also a finite set.

Definition: A set having infinitely many elements is called an **infinite set**.

For example, the set **N** of natural numbers is infinite. Similarly, **Q , Z, R** and **C**, the set of rational numbers, integers, real numbers and complex numbers, respectively, are infinite set.

Now try the following exercises.

E1) How would you represent the set of all students who have offered the IGNOU course?

E2) Explain, with reason, whether or not

i) the collection of all good teachers is a set

ii) the set of points on a line is finite.

E3) Represent the set of all integers by the listing method.

When we deal with several sets, we need to understand the nature of the elements of those sets, whether the elements of two given sets have some elements in common or not, and so on. These questions involve concepts, which we now define.

Definition: A set A is said to be a **subset** of a set B if each element of A is also an element of B. In this case B is called a **superset** of A. If A is a subset of B, we represent this by $\mathbf{A \subseteq B}$.

As a statement in logic we represent this situation as,

$A \subseteq B \Leftrightarrow [x \in A \Rightarrow x \in B]$

'B contains A' or 'B is a superset of A' is represented by $B \supseteq \mathbf{A}$.
If A is not a subset of B, we represent this by $\mathbf{A} \quad \mathbf{B}$.

For example, if A ={4,5,6} and B ={4,5,7,8,6}, then $A \subseteq B$. But if C = {3,4} then $C < B$.

Remark: If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.

Definition: Two sets A and B are **equal** if every element of A belongs to B and every element of B belongs to A. We represent this by **A=B**.

For example, if A= {1,2,3}, B ={2,3,1}, then $A \subseteq B$ and $B \subseteq A$, so that A = B.

Definition: A set A is said to be a **proper subset** of a set B if A is a subset of B and A and B are not equal. We represent this by $\mathbf{A \subset B}$.

For example, if A ={4,5,6} and B = {4,5,7,8,6}, then $A \subset B$; and if A={Java, C, C++, Cobol} and B={Java, C++}, then $A \supset B$.

(1834 -1923)
Fig 1: John Venn

Note: A set can have many subsets and many supersets. For example A={1,2,3,4,5}, B= {2,3,4,5,6,7}, and C= {2,3}, then for C, A and B can be used as supersets. Similarly, if X= {Ram, Rani, Sita, Gita}, Y= {Rani}, and Z= {Sita}, then Y and Z both are subsets of X.

Definition: The **power set** of a set A is the set of all the subsets of A, and is denoted by **P(A)**.

Mathematically, $P(A) = \{ x : x \subseteq A\}$.

Note that $\phi \in P(A)$ and $A \in P(A)$ for all sets A.
For example, if $A=\{1\}$, then $P(A)= \{\phi ,\{1\}\}$ and
if $A= \{1,2\}$, then $P(A)=\{\phi , \{1\}, \{2\}, \{1,2\}\}$
Similarly, if $A = \{1,2,3\}$, then $P(A) = \{\phi ,\{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.

Definition: Any set which is a **superset** of all the sets **under consideration** is known as the **universal set**. This is usually denoted by Ω**, S** or **U.**

For example, if $A = \{1,2,3\}$, $B = \{3,4,6,9\}$ and $C = \{0,1\}$, then we can take $U = \{0,1,2,3,4,5,6,7,8,9\}$ or U= **N**, or U=**Z** as the universal set.

Note that the universal set can be chosen arbitrarily for a given problem. But once chosen, it is fixed for the discussion of that problem.

Theorem 1: If A is a set with n elements, then $|P(A)| = 2^n$.

Proof: We shall prove this by mathematical induction.
For this, we first check if it is true for n= 1.Then assuming that it is true for n = m, we prove it for the case n = m +1. It will, then, follow that the result will be true $\forall$ n $\in$ **N**.

Step I: If A = 1, then $P(A) = 2=2^1$.

Step II: Assume that the theorem holds for all sets A of cardinality k, i.e. if $|A| = k$, then A has 2^k subsets.

Step III: Now consider any set $A = \{x_1, x_2, x_3,\ldots,x_k, x_{k+1}\}$, with k+1 elements. Consider its subset $B= \{x_1, x_2, x_3,\ldots,x_k \}$. Now B has 2^k subsets, each being a subset of A. Now, take any such subset $\{x_{i_1},x_{i_2},\ldots,x_{i_r}\}$of B. Then $\{x_{i_1},x_{i_2},\ldots,x_{i_r},x_{k+1}\}$ is a subset of A that is not a subset of B. So, for each of the 2^k subsets of B, we attach x_{k+1} to it to get 2^k more subsets of A.

You can see that this covers all the subsets of A.

So the number of subsets of $A=2^k+2^k = 2.2^k = 2^{k+1}$.
Hence the theorem.

Now try these exercises.

E4) Give two proper subsets and two supersets of the set of vowels of the English alphabet.

E5) Find the power set of the set $A =\{a,e,i,o,u\}$.

E6) For which set A, is $P(A) = 1$?

E7) If $A \subseteq B$, is $P(A) \subseteq P(B)$? Why?

E8) $P(A) = P(B) \Rightarrow A = B$. True or false? Why?

Let us conclude this section with the **pictorial** representation of sets. You know that the pictorial representation of any object helps in understanding the object. This is why a pictorial representation of sets, known as a **Venn diagram,** helps in understanding and dealing with sets.

The English priest and logician John Venn invented the Venn diagram. Through Venn diagrams we can easily visualize the abstract concept of a set and operations on sets. In this diagram, the universal set is usually represented by a rectangle and its subsets are shown as circles or other closed geometrical figures inside this rectangle.

For example, A = {Lucknow, Patna, Bhopal, Itanagar, Shillong} can be represented using a Venn diagram as in Fig. 2. Here U could be any superset of A.

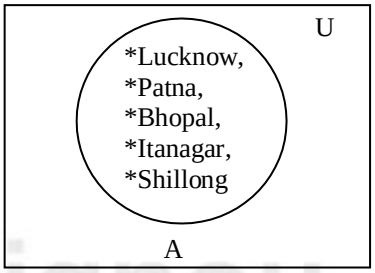


Fig. 2: A Venn diagram

Now that you are familiar with basic definitions related to sets, let us discuss some basic operations that can be performed on sets. This is when we shall appeal to Venn diagrams very often, as you will see.

1.3 OPERATIONS ON SETS

Let us now study sets obtained by applying operations on sets. We will cover four operations here, namely, union of sets, intersection of sets, complement of sets and symmetric difference. While studying them you will see how useful a Venn diagram can be for proving results related to these operations. In this section we will also look at some rules that are common to operations on sets and operations on statements, which you studied in Block 1.

1.3.1 Basic Operations

In this sub-section we shall define each of the operations one by one.

Definition: The **union** of two sets A and B is the set of all those elements which are either in A or in B or in both A and B. This set is denoted by $\mathbf{A \cup B}$, and read as 'A union B'.

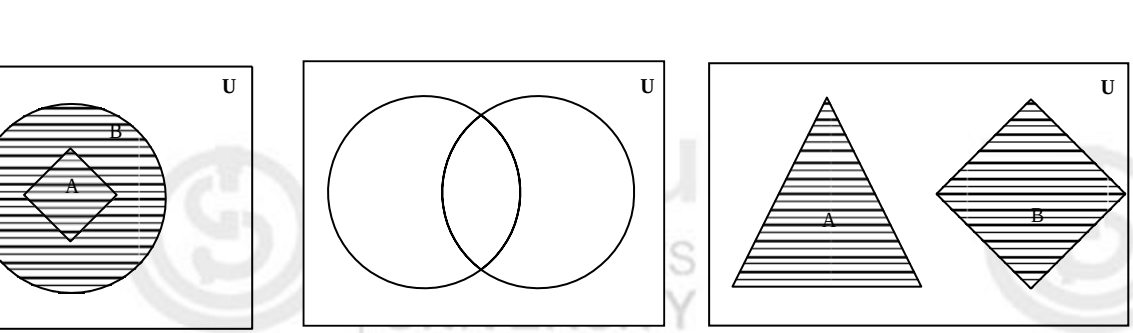
Symbolically, $A \cup B = \{x : x \in A \text{ or } x \in B\}$

For example, if A = {x: x is a stamp} and B = {4,5}, then

$A \cup B$= {x:x is a stamp or a natural number lying between 3 and 6}.

And A={Ram, Mohan, Ravi} and B={Ravi, Rita, Neetu},
then $A \cup B$={Ram, Mohan, Ravi, Rita, Neetu}.

If $A \subseteq B$, then $A \cup B = B$, and vice versa. This can be shown immediately using a Venn diagram, as in Fig.3.(a), where A is shown as the square contained in the circle representing B. In Fig.3(b), $A \cup B$ is shown when A and B have some elements in common, and in Fig.3(c), we depict $A \cup B$ when A and B have no element in common.



(a) (b) (c)

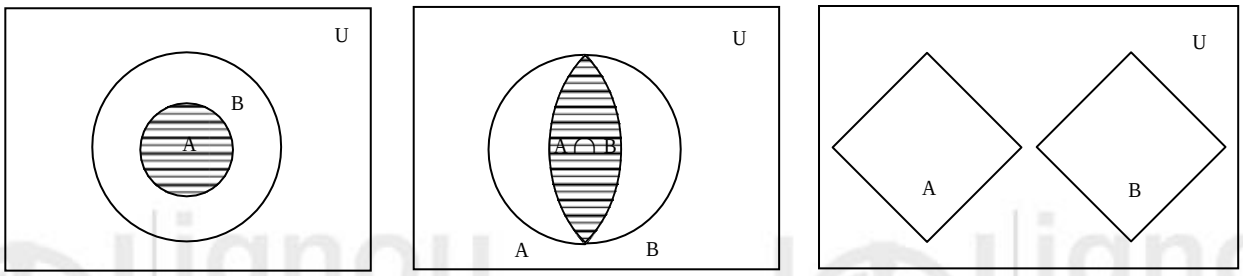
Fig. 3: Venn diagram for union

Definition: The **intersection** of sets A and B is the set of all the elements which are common to both A and B. This set is denoted by **A ∩ B**, and read as 'A intersection B'.

Symbolically, $A \cap B = \{x : x \in A \text{ and } x \in B\}$;
For example A = {1,2,3} and B = {2,1,5,6}, then $A \cap B = \{1,2\}$.
Again if A= {1} and B = {5} then $A \cap B = \{\}$ or ϕ .

Remark: For any two sets A and B, $A \cap B \subseteq A \subseteq A \cup B$ and $A \cap B \subseteq B \subseteq A \cup B$.

What is $A \cap B$ if $A \subseteq B$? Do you agree that it is A? Let us use a Venn diagram to check this (see Fig.4(a)). If A and B have some elements in common, then the Venn diagram for $A \cap B$ looks like Fig 4.(b), and if A and B have no element in common, then the Venn diagram will be as in Fig.4(c).



(a) (b) (c)

Fig. 4: Venn diagram for intersection of sets

Definition: The **difference of two sets** A and B is the set of all those elements of A which are not elements of B. Sometimes, we call this set the **relative component** of B in A. It is denoted by **A~B** or **A\B**, and is read as 'A complement B'.

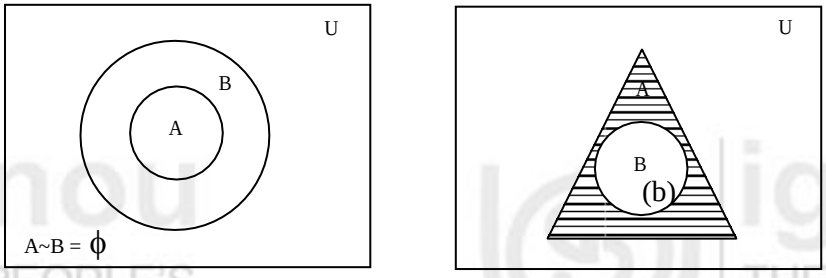
Symbolically, $A \sim B = \{x : x \in A \text{ and } x \notin B\}$ and
$B \sim A = \{x : x \in B \text{ and } x \notin A\}$

For example, if A = {4,5,6,7,8,9} and B = {3,5,2,7}, then $A \sim B = \{4,6,8,9\}$ and $B \sim A = \{3,2\}$. From this example it is clear that $A \sim B \neq B \sim A$. In fact, this is usually the case. So, **the difference of sets is not a commutative operation**.

In Fig.5(a), $A \subseteq B$, so that $A \sim B = \phi$.

In Fig.5(b) we show A~B when $A \supseteq B$, and in Fig.5(c) we show A~B when neither $A \subseteq B$ nor $B \subseteq A$.

In Fig. 5(d), we show A~B when A and B are disjoint.



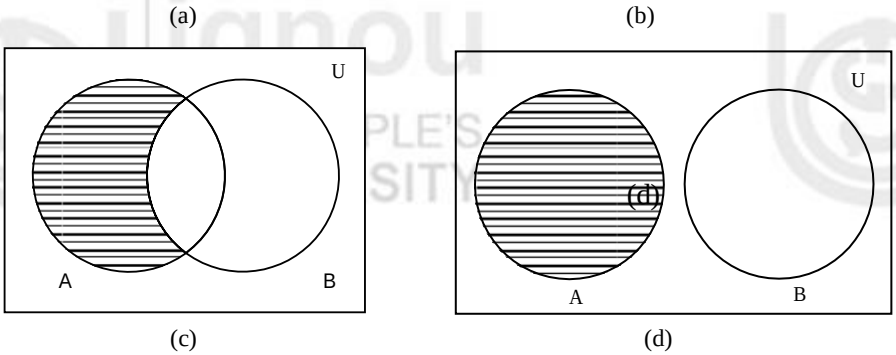


Fig. 5: A ~ B in different situation is the shaded portion.

There is one particular 'difference' that shows up very often, which we now define.

Definition: The **complement of a set** A, is the set U \ A, and is denoted by A' or A^c. For example, U={Physics, Chemistry, Mathematics} and A={Mathematics}, then the complement of A is A'= {Physics, Chemistry}.

The Venn diagram showing the complement of Á is the set of those elements of the universal set U which are outside A (see Fig.6).

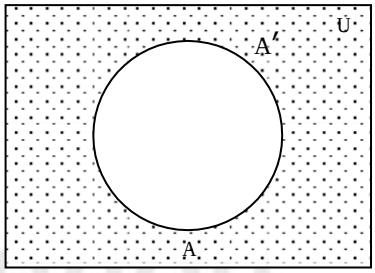


Fig. 6: Venn diagram for A'.

Definition: The **symmetric difference** of two sets A and B is the set of all those elements which are in A or in B, but not in both. It is denoted by $A \Delta B$.

i.e., $A \Delta B = (A \sim B) \cup (B \sim A)$.

Note that $A \Delta B = B \Delta A$, i.e. the symmetric difference is commutative.

For example A= {1,2,3,4,5} and B= {3,5,6,7}, then A~ B= {1,2,4},and B~A = {6,7}
$\therefore$ $A \Delta B = \{A \sim B) \cup (B \sim A) = \{1,2,4,6,7\}$

Now you may try these exercises.

E 9) Make a Venn diagram for $A \Delta B$ for each of the situations i) $A \subseteq B$, ii) $A < B$, iii) $B < A$ and $A \cap B \neq \phi$; iv) $A \cap B = \phi$.

E10) Let A= {Math, Physics, Science}, B={Computer,Math,Chemistry}, C={Math}. Find $A \cup (B \cap C)$.

E11) If A = {1,2,3,4,5,6}, B={4,5,6,7,8,9},find i) A~ B, ii) B~ A, iii) $A \Delta B$.

E12) For which sets A and B would A~B = B~A ?

E13) Write a program in C to perform E 10.

E14) Under what conditions can $A \cap B = A \cup B$?

While discussing these operations, you may be wondering that they seem to satisfy properties very similar to those of propositional logic covered in Block 1 of this course.You are right! Let us look at this aspects now.

1.3.2 Properties Common to Logic and Sets

Before looking into the properties we shall first present a very useful principle to you. This will allow you to see how one property can be proved in several situations simultaneously.

Duality Principle: The 'duality principle' is very convenient for dealing with theorems about sets. Basically if any theorem is given to you, by applying the duality principle you can get another theorem (the dual of the previous one). In any statement involving the union and intersection of sets, we can get from one of the statements to the other by interchanging $\cap$ with $\cup$ andϕ with U.

For example, the dual of $A \cup (B \cap C)$ is $A \cap (B \cup C)$ and the dual of $U \cup \phi = U$ is $U \cap \phi = \phi$. So, for example what is true for $A \cup (B \cap C)$ will be true for $A \cap (B \cup C)$ too. This is why if the first property in each of the pairs below is proved the second one follows immediately.

For any universal set U and subsets A, B and C of U, **the following properties hold**.

i) **Associative properties:**
Union: $A \cup (B \cup C) = (A \cup B) \cup C$
Intersection: $A \cap (B \cap C) = (A \cap B) \cap C$

ii) **Commutative properties:**
Union: $A \cup B = B \cup A$
Intersection: $A \cap B = B \cap A$.

iii) **Identity:**
Union: $A \cup \phi = A$
Intersection: $A \cap U = A$.

iv) **Complement:**
Union: $A \cup A' = U$
Intersection: $A \cap A' = \phi$

Fig. 7: Augustus De Morgan (1806–1871)

v) **Distributive properties:**
Union: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
Intersection: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

De Morgan's Laws:

For any two sets A and B the following laws known as De Morgan's laws, hold

1. $(A \cup B)' = A' \cap B'$, and
2. $(A \cap B)' = A' \cup B'$

De Morgan's laws can also be expressed as

1. $A \sim (B \cup C) = (A \sim B) \cap (A \sim C)$
2. $A \sim (B \cap C) = (A \sim B) \cup (A \sim C)$

Fig. 7: Augustus De Morgan (1806-1871)

Each of the properties above corresponds to a related property for mathematical statements in logic (which we have covered in Unit 2 and Unit 3 of Block 1 of this course).

Now try these exercises.

E15) Find the dual of

i) $A \cap (B \cap C) = (A \cap B) \cap C$, and ii) $(A \cup B) \cap (A \cup C)$.

E16) Draw a Venn diagram to represent $A \cup (B \cap C)$.

E17) Check whether $(A \cup B) \cap C = A \cup (B \cap C)$ or not using a Venn diagram.

Let us now focus on subsets of a particular kind of product of sets.

1.4 RELATIONS

Sometimes we need to establish relations between two or more sets. For example, a software development company has a set of specialists in different technology domains, or a company gets some projects to develop. Here the company needs to establish a relation between professionals and the project in which they will participate. To solve this type of problem the following concepts are required.

1.4.1 Cartesian product

Very often we deal with several sets at a time, and we need to study their combined action. For instance, combinations of a set of teachers and a set of students. In such a situation we can take a product of these sets to handle them simultaneously. To understand this product let us first consider the following definitions.

iff is short for 'if and only if'.

Definition: An **ordered pair**, usually denoted by (x,y), is a pair of elements x and y of some sets. This is ordered in the sense that $(x,y) \neq (y,x)$ whenever $x \neq y$,that is , the order of placing of the element in the pair matters.

Any two ordered pairs (x,y) and (a,b) are equal iff x = a and y = b.

For example if, A= {a,b,c}and B= {x,y,z},then

$A \times B = \{a,b,c\} \times \{x,y,z\} = \{(a,x), (a,y),(a,z),(b,x),(b,y),(b,z),(c,x),(c,y), (c,z)\}$,and

$B \times A = \{x,y,z\} \times \{a,b,c\} = \{(x,a),(x,b),(x,c),(y,a),(y,b),(y,c),(z,a),(z,b),(z,c)\}$.

Now let us think about how $B \times A$ can be represented geometrically? For instance what is the geometric view of $\{2\} \times \mathbf{R}$? This is the line x =2 given in Fig.8(a).

Now, after seeing geometric representation of $\{2\} \times \mathbf{R}$, can you tell what $\{1,3\}\times\{2,3\} = \{(1,2),(3,2),(1,3),(3,3)\}$ looks like? You will get four points in the first quadrant, as shown in Fig.8(b).

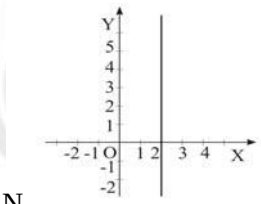


Fig. 8(a): {2} × R, i.e., x = 2.

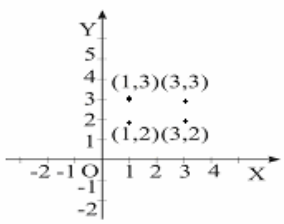


Fig.8(b): {1,3}×{2,3}

N of numbers is commutative. Is multiplication the Cartesian product of sets also commutative? For instance, is $\{1\}\times\{2\}=\{2\}\times\{1\}$? No, because $(1,2) \neq (2,1)$. So, $\mathbf{A}\times\mathbf{B} \neq \mathbf{B}\times\mathbf{A}$ **usually**.

We can extend the definition of $A \times B$ to define the Cartesian product of n sets A_1, A_2, ….,A_n as follows.

$A_1 \times A_2 \times A_3 \times \ldots \times A_n = \{(x_1,x_2,x_3,\ldots,x_n)\}: x_1 \in A_1, \wedge x_2 \in A_2 \wedge x_3 \in A_3 \wedge \ldots, \wedge x_n \in A_n\}$.

The element $(x_1,x_2,\ldots,x_n)$ is called an **n-tuple.** For instance, the 3–tuple $(1,1,3) \in \{1\} \times \{1,2\} \times \{2,3\}$.

Now you may try some exercises.

E18) If X= {a,b,c} and Y={1,2,3},find

i) $X \times X$, ii) $X \times Y$, and iii) $X \times \phi$.

E19) Under what conditions on A and B is $A \times B = B \times A$?

E20) Give the geometric representation of $\mathbf{R} \times \{2\}$.

With what you studied in this sub-section, you now have the background to discuss relations.

1.4.2 Relations and their Types

We often speak of relations which hold between two or more objects, e.g., discrete mathematics is one of the courses in the IGNOU MCA Ist semester, Chennai is the capital of Tamil Nadu. These are the relations in everyday situations. In these examples some sort of connections between pairs of objects are shown, and hence they express a relation between the pairs of objects.

Definition: A **relation between two sets A and B** is a subset of $A \times B$. Any subset of $A \times A$ is **a relation on the set A**.

For instance, if A ={1,2,3} and B = {p,q}, then the subset {(1,p),(2,q),(2,p)} is a relation on $A \times B$. And {(1,1), (2,3)} is a relation on A.

Also, $R= \{(x,y) \in \mathbf{N} \times \mathbf{N} : x > y\}$ is a relation on **N,** the set of natural numbers, since $R \subseteq \mathbf{N} \times \mathbf{N}$.

If $R \subseteq A \times B$,. we write $x \mathbf{R}_y$ if and only if $(x,y) \in R$($_xR_y$ is read as 'x is related to y').

Theorem 2: The total number of distinct relations between a finite set A and a finite set B is 2^{mn}, where m and n are the number of elements in A and B, respectively.

For example, $R_1 = \mathbf{N} \times L$, where L is set of straight lines, in this relation we can give different ordering of the straight lines.
If the relation $R_2 = \{1,2,3\} \times \{ l_1,l_2\}$, then line l_1 and l_2 can get three different ordering.

Proof: The number of elements of $A \times B$ is mn. Therefore, the number of elements of the power set of $A \times B$ is 2^{mn} (See Theorem 1). Thus, $A \times B$ has 2^{mn} different subsets. Now every subset of $A \times B$ is a relation from A to B, by definition. Hence the number of different relations from A to B is 2^{mn}.

As you have seen, any and every subset of $A \times A$ is a relation on A. However, some relations have special properties. Let us consider these types one by one.

1.4.3 Properties of Relations

Reflexive Relations: A relation R on a set A is called a **reflexive relation** if $(a,a) \in R \; \forall \; \mathbf{a} \in \mathbf{A}$.

In other words, R is reflexive if every element in A is related to itself.
Thus, R is **not reflexive** if there is at least one element $a \in A$ such that $(a,a) \notin R$.

For example, if A= {1,2,3,4}, then the relation R_1= {(1,1), (2,2), (3,3), (4,4)} in A is reflexive because for $x \in A$,$(x,x) \in R_1$. However, R_2= {(1,1), (2,1), (4,4)} is not reflexive since $2 \in A$, but $(2,2) \notin R_2$.

Symmetric Relations: A relation R on a set A is called a **symmetric relation** if $(a,b) \in R \Rightarrow (b,a) \in R$.Thus, R is symmetric if bRa holds whenever aRb holds.

A relation R in a set A is **not symmetric** if there exist two distinct elements $a, b \in A$, such that aRb, but not bRa.

For example, if L is the set of all straight lines in a plane, then the relation R in L, defined by 'x is parallel to y', is symmetric, since if a straight line a is parallel to a straight line b, then b is also parallel to a. Thus, $(a,b) \in R \Rightarrow (b,a) \in R$.

However, if R is the relation on **N** defined by 'xRy iff x–y>0', then R is not symmetric, since, 4−2>0 but 2−4 < 0.Thus, $(4,2) \in R$ but $(2,4) \notin R$.

Transitive Relations: A relation R on a set A is called a **transitive relation** if whenever $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$ for $a,b,c \in A$.
Thus, $[(a,b), \in R, (b,c) \in R \Rightarrow (a,c) \in R]$, $\forall$ $a,b,c \in A \Rightarrow$ R is transitive.

A relation R in a set A is **not transitive** if there exist elements $a,b,c \in A$, not necessarily distinct, such that $(a,b) \in R$, $(b,c) \in R$ but $(a,c) \notin R$.

For example, if L is the set of all straight lines in a plane and R is the relation on L defined by ' x is parallel to y' then if a is parallel to b and b is parallel to c, then a is parallel to c. Hence R is transitive. However, the relation 'xSy'on L defined by 'iff x intersects y' is not transitive.

Also, the relation R on A, the set of all Indians, defined by 'xRy iff x loves y', is not a transitive relation.

Equivalence Relations: A relation R on a set A is called an **equivalence relation** if and only if

(i) R is reflexive, i.e., for all $a \in R$, $(a, a) \in R$,

(ii) R is symmetric, i.e., $(a, b) \in R \Rightarrow (b, a) \in R$, for all $a, b \in A$, and

(iii) R is transitive, i.e., $(a, b) \in R$ and $(b, c) \in R \Rightarrow (a, c) \in R$, for all $a, b, c \in A$.

One of the most trivial examples of an equivalence relation is that of **'equality'.**
For any elements a,b,c in a set A,

(i) a = a, i.e., reflexivity

(ii) $a = b \Rightarrow b = a$, i.e., symmetricity

(iii) a = b and $b = c \Rightarrow a = c$, i.e., transitivity.

Now let us see if 'xRy iff ' '$x \leq y$' gives an equivalence relation on **R**.

(i) $x \leq x$, i.e., $(x,x) \in \mathbf{R}$, i.e., R is reflexive.

(ii) However, $2 \leq 3$ but 3 2. So, R is not symmetric.

Thus, R is not an equivalence relation.

Now you may try these exercises.

E 21) Let A be the set of all people on Earth. A relation R is defined on the set A by 'aRb if and only if a loves b' for $a,b \in A$.

Examine if R is i) reflexive, ii) symmetric, iii) transitive.

Now we shall study a particular kind of relation, which is very useful in mathematics, as well as in computer science, as you will soon see.

1.5 FUNCTIONS

A function is a special kind of relation. If we take the example of the set A of students of IGNOU, and the set B of their enrolment numbers. Now consider R = {(a,b)$\in$ A$\times$B | b is enrollment number of a },this is a relation between A and B. It is a 'special' relation, 'special' because to each a $\in$A $\exists$! b such that aRb. We call such a relation a function from A to B.

Let us define this term formally.

Definition: A **function from a non-empty set A to a non-empty set B** is a subset R of A$\times$B such that for each a $\in$A $\exists$a unique b$\in$B such that (a,b)$\in$R. So, this relation satisfies the following two conditions:

(i) for each a $\in$A, there is some b$\in$B such that (a,b) $\in$R

(ii) if (a,b) $\in$R and (a,b´) $\in$R then b = b´.

We usually present functions as a rule associating elements of one set with another. So, let us present the definition again, with this view.

Definition: Let A and B be non-empty sets. A **function** (or a **mapping**) f from A to B is a rule that assigns to each element x in A **exactly one** element y in B. We write this as f: A $\rightarrow$ B, read it as 'f is a function from A to B'.

Note that

(i) to each a $\in$A, f assigns an element of B; and

(ii) to each a $\in$A, f assigns only **one element** of B.

So, for example, suppose A ={1,2,3}, B= {1,4,9,11} and f assigns to each member in A its square values. Then f is a function from A to B. But if A={1,2,3,4}, B={1,4,9,10} and f is the same rule, then f is not a function from A to B since no member of B is assigned to the element 4 in A.

Note that the former example, 11$\in$B, but there is no element in A which is assigned to 11. This does not matter. It is not necessary that every element of B be related to some element of A.

Functions are not restricted to sets of numbers only. For instance, let A be the set of mothers and B be the set of human beings. Then the rule that assigns to every mother her eldest child is a **function**. But the rule that assigns to each mother her children is **not** a **function** because it does not relate a unique element of B to each element of A. Now, given a function, we have certain sets and terms that are associated with it. Let us give them some names.

Definitions: Let f be a function from A to B. The set A is called the **domain** of the function f and B is called the **co-domain** of f. The set {f(x)| x $\in$A} is called the **range** of f, and is also denoted by f(A).

Given an element x$\in$A, the unique element of B to which the function f associates, it is denoted by f (x) and is called the **f-image** (or **image**) of x or the value of the function f for x. We also say that f **maps** x to f(x). The element x is referred to as the **pre-image** of f (x).

For example, if A = {1,2,3,4}, B = {1,8,27,64,125}, and the rule f assigns to each member in A its cube, then f is a function from A to B. The domain of f is A, its codomain is B and its range is {1,8,27,64}.

Can you tell what will be the domain and codomain for rule f : $f(x) = \frac{x}{1-x}$?

You can see that 1–x = 0, if x = 1, in this case f(x) will be undefined. Domain of f can be taken as R~{1}and codomain can be **R**.

Remark: Each element of A has a **unique image,** and each element of B need not appear as the image of an element in A. Further, more than one element of A can have the same image in B.

Let us look at some examples of functions, and non-functions now.

i) If b is be a fixed element of B, then f: A→B : f(x)= b $\forall$ x $\in$A is called a **constant function**.

Note that if b=0, then f is called the **zero map**, and is denoted by **0**.

ii) f : A→ A : f(x) = x $\forall$ x $\in$A is called the **identity function** , and is denoted by **I**.

iii) Consider A= {1,2,3,4}, B= {1,4,5} and the rule f which associates 1→1, 2→ 4, 3→5, 4→5. Then f is a function from A to B.

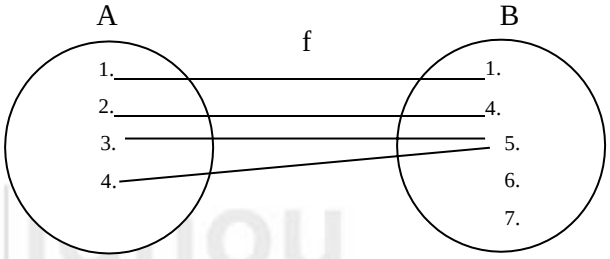


Fig.11: The rule f is a function

iv) The function f from **R** to **Z**, defined by the rule that f maps any real number x to the greatest integer less than or equal to x. is known as the **greatest integer** function or the **floor function**. We denote this function's action by f(x)=[x], where [x] is the greatest integer $\leq$x.

For example, if x = 0.6 then f(x) =[x]=0, if x= 2.3 then f(x)=[x]=2, and if x = –5, then [x]′= –5 .

v) Function f: **R→R**: f(x)= | x | is known as the modulus (or absolute value) function, where | x | is the absolute value of x.

For example, if x=10 then f(x)= | x | =10 and if x= –10, then f(x)= | x | =10.

vi) Now take, A={a,b,c} and B={1,2,3,4,5}. Consider the rule f which associates a→1, a→ 3, b→2, c→3. This is not a function from A to B because, elements 1 and 3 $\in$ B are assigned to the same element a $\in$A.

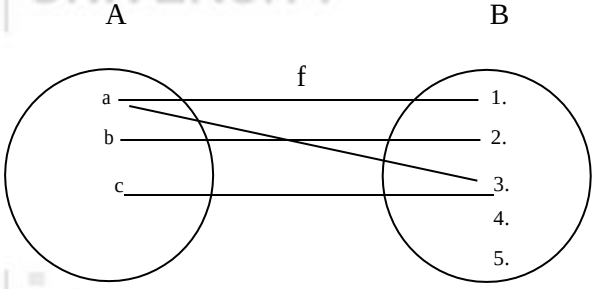


Fig.9: The rule f is not a function

vii) Consider A= {1,2,3}, B= {1,4,5,6,7} and the rule f which associates 1→1, 2→ 1, 2→4. Here f is not a function from A to B since no member of B is assigned to the element 3∈A.

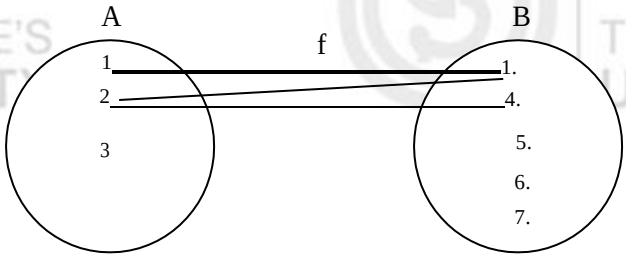


Fig.10: The rule is not a function

Now you may try these exercises.

E22) Let A= {a,b,c,d}, B= {1,2,3} and R= {(a,2), (b,1), (c,2), (d,1)}. Is R a function? Why.

E23) Every function is a relation. Is every relation a function? Why?

E24) Consider the following pseudocode.

1. read(n)
2. while n >1 do
3. begin
4. if n is even then n := n div 2
5. else n := 2n + 1;
6. end

Write a function of n that describes the operations performed.

E25) If A= {1,2,3,4}, B = {2,3,4,5,6,7} and the rule f assigned to each member in A is f(x)= x + 1, then find the domain and range of f.

Now let us discuss some types of functions.

1.5.1 Types of Functions

Here we shall look at different types of mappings.

'Surjective' comes from the French word 'sur', meaning 'on top of '.

Onto Mapping: A mapping f: A→ B is said to an **onto** (or **surjective**) **mapping** if f(A)= B, that is, the range and co-domain coincide. In this case we say that **f maps A onto B**.

For example, *f*: **Z**→ **Z** : f(x) = x+1, x∈**Z**, then every element y in the co-domain **Z** has a pre-image y–1 in the domain **Z**. Therefore, f(**Z**) = **Z** ,and f is an onto mapping.

Injective Mapping: A mapping *f*: A→ B is said to be **injective** (or **one-one**) if the images of distinct elements of A under f are distinct, i.e., if $x_1 \neq x_2$ in A, then $f(x_1) \neq f(x_2)$ in B. This is briefly denoted by saying f is **1–1**.

For example f: **R**→ **R** be defined by f(x) = 2x+1, x∈**R**, then for $x_1, x_2 \in$**R** ($x_1 \neq x_2$) we have $f(x_1) \neq f(x_2)$. So, f is **1–1**.

Bijective Mapping: A mapping f: A→ B is said to be **bijective** (or **one-one onto,**) if f is both injective and surjective, i.e., one-one as well as onto.

For example, f: **Z**→ **Z** : f(x) = x+2, x∈**Z** is both injective and surjective. So, f is bijective.

There is a particular kind of bijective function that we use very often. Let us define this.

Definition: A bijective mapping f : A→ A is said to be a **permutation** on the set A. Let A = $\{a_1,a_2,\ldots,a_n\}$, and f be a bijection from A onto A that maps a_i to $f(a_i)$, then we write f as

f = $\begin{pmatrix} a_1 & a_2 \ldots a_n \\ f(a_1) & f(a_2)\ldots f(a_n) \end{pmatrix}$. So, the identity mapping I= $\begin{pmatrix} a_1 & a_2 \ldots a_n \\ a_1 & a_2 \ldots a_n \end{pmatrix}$.

Now, associated with a bijective function, we get another function very naturally, which we now define.

Definition: Let f: A→ B be a bijective mapping. Then the mapping g : B→ A which associates to each element b ∈ B the unique element a ∈ A, such that f(a) = b, is called the inverse mapping of the mapping f: A→ B. We denote this function g by f^{-1}.

Note that a function f is invertible iff f^{-1} exists iff f is bijective.
Hence, if f: A→B is a one-one onto mapping, then f^{-1} : B→A exists, and is also 1-to-1.

Note the inverse of the permutation f = $\begin{pmatrix} a_1 & a_2 \ldots a_n \\ b_1 & b_2 \ldots b_n \end{pmatrix}$ is the permutation

$\begin{pmatrix} b_1 & b_2 \ldots b_n \\ a_1 & a_2 \ldots a_n \end{pmatrix}$.

For example, A= **R**~{3} and B= **R**~(1}, and the function f: A→ B is defined by

f(x)= $\frac{x-2}{x-3}$.

We can see that f is a one–to–one function.
∴ f inverse exists.

To get $f^{-1}(x)$ the following steps are required;

1. Replace f(x) by y in the equation describing the function. You will get

   y = $\frac{x-2}{x-3}$.

2. Interchange x and y. In other words, replace every x by y, and vice versa. You will get x = $\frac{y-2}{y-3}$.
3. Solve for y.
4. Replace y by $f^{-1}(x)$.

By applying these steps we get $f^{-1}(x)$= $\frac{3x-2}{x-1}$.

Now try these exercises.

E26) Explain why f : **Z** → **Z**: $f(x) = x^2$ is onto? Domain and range of f is **Z**.

E27) Which of the following kind of function would you use to provide photo identity numbers? Why?

i) Constant function, ii) one-to-one function, and iii) identity function.

E28) Find f inverse of rule f : $f(x) = x^3-3$.

Now we can see how different operations like addition, subtraction, multiplication and division can be applied on functions.

1.5.2 Operations on Functions

If given whose domains ranges are subsets of the **real numbers**, we define the function f+g by (f + g)(x) to be the function whose value at x is the sum of f(x) and g(x). Symbolically,

(f + g)(x) = f(x) + g(x). This is called pointwise addition of f and g.

The domain of **f+g** is the **intersection** of the domains of f and g since to compute (f + g)(x) it is necessary and sufficient to compute both f(x) and g(x).

Other operations on functions are defined similarly:

- (fg)(x) = f(x)g(x) (pointwise multiplication)
- $f^p(x) = (f(x))^p$ for any real exponent p with the domain of f^p consisting of those points for which the p-th power of f(x) makes sense.
- (f/g)(x) = f(x)/g(x), for $g(x) \neq 0$ (pointwise division)

For example, if f(x) = 3 sin (x) and $g(x) = x^2$, then

(f+g)(x) = 3 sin(x) + x^2
(fg)(x) = 3 sin(x)*x2
(f-g)(x) = 3 sin(x) – x^2
(f/g)(x) = 3 sin(x) / x^2

The domains of both f and g are all **real numbers**, but the domain of f/g is $\{ x \mid x \neq 0\}$.

Now let us consider two functions f and g from A= {1,2} to, B= {1,2,3,4}, where f={(1,1),(2,4)}. Let g be defined by the rule $g(x)= x^2$ where the domain of g is the set {1,2}. Here both have the same domain. Since f and g assign the same image to each element in the domain, they have the same effect throughout. This is why we treat them as the same, or equal.

Definition: If f and g are two functions defined on the same domain A and if **f(a)= g(a)** for every a $\in$A, then the functions **f** and **g** are **equal**, i.e., f = g.

For example $f(x) = x^2+5$, where x is a real number, and $g(x) = x^2+5$, where x is a complex number. Then the function f is not equal to the function g since they have different domains although $f(x)= x^2+5= g(x \forall x \in \mathbf{R})$. By this example we can conclude that even if f(a)= g(a), f and g may not be the same.

So far, the operations you have seen are the same as those for member systems. However, there is yet another operation on functions which we now define.

Definition: Let f and g be the operation of combining two functions by applying them one after the other. That is, the composition of f(x) and g(x), denoted by, fog .

For example, consider $f : \mathbf{R} \rightarrow \mathbf{R} : f(x) = (x^3 + 2x)^3$. We can write it as the composition of g and h, where the value of f(x) can be obtained by first calculating $x^3 + 2x$ and then taking its third power . We can write g for first or inside function $g(x) = x^3 + 2x$. We write h for the second function : $h(x) = x^3$. The use of the variable x is irrelevant, we could as well write $h(y) = y^3$ for $y \in \mathbf{R}$. We can see that $goh(x) = g(x^3 + 2x) = (x^3 + 2x)^3 = f(x)$.

In general (f g) $\neq$ (g f).

For example, if, $f(x) = x^2$ and g(x) = x+1, then $(f\ g)(x) = (x + 1)^2$ and $(g\ f)(x) = x^2+1$.
Here we can see that f g $\neq$ g f.

Let us see another example, where $f(x) = x^2$, g(x) = x +1, $h(x) = x^3$

Then, f (g h) (x) = $(x^3+1)^2$ and (f g) h (x) = $(x^3+1)^2$. Here we can see f (g h) = (f g) h.

Now let us see how you can get **product** of two permutations f and g of the same set,

Let f= $\begin{pmatrix} a_1 & a_2 \dots & a_n \\ f(a_1) & f(a_2) \dots & f(a_n) \end{pmatrix}$ and

g = $\begin{pmatrix} a_1 & a_2 \dots & a_n \\ g(a_1) & g(a_2) \dots & g(a_n) \end{pmatrix}$. Then fg= $\begin{pmatrix} a_1 & a_2 \dots & a_n \\ f[g(a_1)] & f[g(a_2)] \dots & f[g(a_n)] \end{pmatrix}$ is itself a permutation.

For example if, f= $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$, g= $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$ then

fg= $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$ and gf= $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$

Note that f g ≠ g f. Thus the **multiplication of permutations is not commutative** in general.

However, the multiplication of permutations is associative. For example, if f= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 4 & 2 \end{pmatrix}$, g= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$, h= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}$ be the permutations on A = {1,2,3,4}, then

fg= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}$, gf= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix}$, gh= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix}$,

f(gh)= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$, (fg)h= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$.

Here we can see the multiplication of permutation is commutative.

Now try these exercises.

E 29) Let f(x) = 1/x and g(x) = $x^3 + 2$. Find the following functions, where x ∈ **R**.

i) (f + g)(x)

ii) (f - g)(x)

iii) (fg)(x)

iv) (f/g)(x)

E30) Let f(x) = $\sqrt{x+1}$ ∀x ≥ –1 and g(x) = x^3 ∀x ∈ **R**. Define the following functions. Also give their domains.

i) (f + g)

ii) (f – g)

iii) (fg)

iv) (f/g)

v) (f g)

With this we have come to the end of this unit. Let us now summaries what we have covered in this unit.

1.6 SUMMARY

In this unit we have covered the following points:

1. We introduced basic concepts related to sets and different ways of representing them.
2. We worked at different operations on sets and there Venn diagram representations.
3. We explored some properties common to operations on sets and logical statements.
4. In the process we also documented the duality principle.
5. We defined relations as a Cartesian product of sets and looked at several examples and type of relations.
6. We defined a function as a particular kind of relation. Then we studied different types of functions as well as basic operations on functions. In the process we considered permutations and their product.

1.7 SOLUTIONS / ANSWERS

E1) A={x:x is a student of IGNOU.}

E2) i) The collection of all good teachers is **not** a set because this collection is not well-defined. The characteristic 'good' cannot be measured objectively.

ii) The set of points on a line is **not** finite because infinitely many points make a straight line.

E3) {…, –3, –2, –1,0,1,2,3,…}.

E4) Set of vowels of English alphabet V={a,e,i,o,u}. Two subsets of set V are V1={a,e}, and V2={i,o}. Two supersets of V are V3={a,b,c,…,z}, and V4={a,c,d,e,i,o,u,…,z}.

E5) Powerset of A={a,e,i,o,u} is

{ϕ,{a},{e},{i},{o},{u},{a,e},{a,i},{a,o},{a,u},{e,i},{e,o},{e,u},{i,o},{i,u}, {o,u},{a,e,i},{a,i,o},{a,o,u},{e,i,o},{i,o,u},{i,o,u},{a,e,i,o},{e,i,o,u},{a,e,i,o,u} .

E6) For empty set A={ } or ϕ, P(A)=1

E7) If $A \subseteq B$, then $P(A) \subseteq P(B)$ because every subset of A is a subset of B.

E8) If P(A) =P(B) then $A \in P(A) = P(B) = A \subseteq B$. Similarly, $B \subseteq A$. Therefore A = B.

E9)

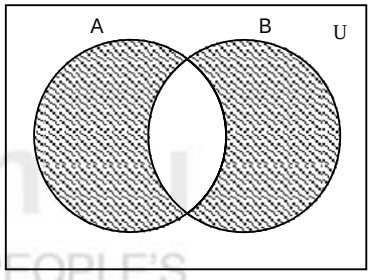


Fig.12: The Shaded portion is A Δ B

You can try them for the other situations. We are showing in Fig. 12 for the second situation.

E10) $A \cup (B \cap C) = \{Math, Physics, Science\} = A$.

E11) i) $A \sim B = \{1,2,3\}$

ii) $B \sim A = \{7,8,9\}$

iii) $A \Delta B = \{1,2,3,7,8,9\}$

E12) Only if A and B are ϕ.

E13) Write separate functions to find A~B, B~A and A Δ B with passing sets A and B as argument, return the resultant set.

E14) $A \cap B$ can be equal to $A \cup B$ if either $A \subseteq B$ or $B \subseteq A$.

E15) i) Dual of $A \cap (B \cap C) = (A \cap B) \cap C$ is $A \cup (B \cup C) = (A \cup B) \cup C$.

ii) Dual of $(A \cup B) \cap (A \cup C)$ is $(A \cap B) \cup (A \cap C)$.

E16)

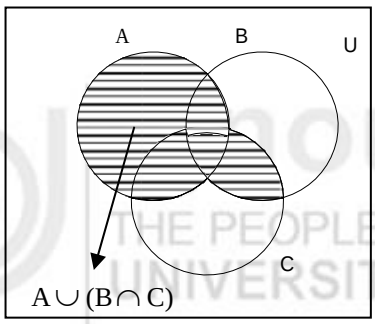


Fig.13: The lined portion represents A ∪ (B ∩ C)

E17)

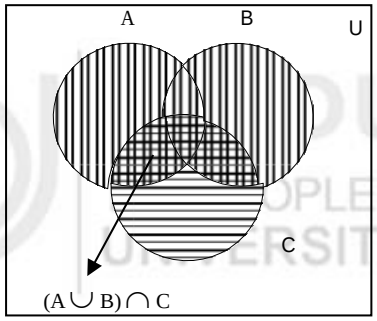


Fig.14(a)

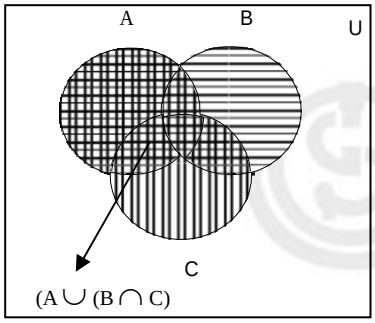


Fig.14(b)

Shaded area in Fig.14 (a) and Fig.14(b) are not same so $(A \cup B) \cap C$ is not equal to $(A \cup (B \cap C)$.

E18) i) X X X={(a,a),(a,b),(a,c),(b,b),(b,c),(c,c)}.

ii) X X Y={(a,1), (b,1), (c,1), (a,2), (b,2), (c,2), (a,3), (b,3), (c,3)}.

iii) X X ϕ = ϕ.

E19) $A \times B = B \times A$ iff $A = B$.

E20) The geometric diagram for $R \times \{2\}$ will be the line parallel to Y axis.

See Fig.15.

Fig.15: y=2

E21) i) For $a \in A$, aRa is reflexive because every one loves herself or himself.

ii) R is not symmetric because if a loves b then b need not love a, i.e., aRb does not always imply bRa. Thus R is not symmetric.

iii) R is not transitive, because if a loves b and b loves c then a need not love c; i.e., if aRb and bRc, aRc need not be. Thus, R is not transitive.

Hence, R is reflexive but is neither symmetric nor transitive.

E22) R is a function because each element of A is assigned to a unique element of B.

E23) Not every relation is a function. For example, this relation does not satisfy the property that,

a) Each element of A must have assigned one element in B.

b) If $a \in A$ is assigned $b \in B$ and $a \in A$ is assigned $b' \in B$ then $b = b'$.

That is why relations those who don't satisfy above properties are not a function

E24) We can see that the code has no effect on the value of $n \leq 0$.In the While loop, the value of n is halved whenever it is even. If n becomes odd before reaching 1, the second part of the while loop is invoked, and n remains odd and increases forever.

This shows that $f : N \to N$ is the function defined by $f(n) = \begin{cases} 0 & \text{if } n = 0. \\ 1 & \text{if } n \text{ is a power of 2,} \\ \text{undefined} & \text{otherwise} \end{cases}$

E25) The domain of f is {1,2,3,4} and range of f is {2,3,4,5}.

E26) Function $f(x) = x^2$ is one-to-one because for every value of x, x^2 will be a number that is different for different x. Hence, $f(x) = x^2$ is one-one mapping.

E27) One-to-one function will be used for providing identity card number, because each person must have unique identity numbers.

E28) Step 1: $y = x^3 - 3$

Step 2: $x = y^3 - 3$

Step 3: $y = \sqrt[3]{x+3}$

Step 4: $f^{-1}(x) = \sqrt[3]{x+3}$.

E29) i) $(f+g)(x) = \frac{1}{x} + x^3 + 2$

ii) $(f-g)(x) = \frac{1}{x} - (x^3 + 2)$

iii) $(f.g)(x) = \left(\frac{1}{x}\right)(x^3 + 2)$

iv) $(f/g)(x) = \dfrac{1}{x(x^3+2)}$ $\forall x \in R$.

E30) i) $(f+g)(x) = \sqrt{x+1} + x^3$ $\forall x \geq -1$

ii) $(f-g)(x) = \sqrt{x+1} - x^3$ $\forall x \geq -1$

iii) $(f.g)(x) = \sqrt{x+1}\ x^3$ $\forall x \geq -1$

iv) $(f/g)(x) = \sqrt{x+1}\ / x^3$ $\forall x \geq -1, x \neq 0$

v) $(f \circ g)(x) = f(x^3) = \sqrt{x^3+1}$ $\forall x \geq -1$.

UNIT 2 AUTOMATA AND LANGUAGES

2.0 INTRODUCTION

In the previous unit we have examined sets, relations and functions. In this unit we will introduce a special kind of a set i.e., a language which is a set of strings over an alphabet. First we describe what is a language and what are the operations we perform on languages. Certain set of strings or languages are represented in algebraic fashion, then these algebraic expressions of languages are called **regular expressions**. A language represented by a regular expression is called a regular language. Then we introduce a notion of finite automata, also called finite state machine or deterministic finite automata. that recognizes regular languages. Finally, we introduce another kind of a theoretical machine, called nondeterministic finite automata (NFA). In deterministic finite automata (DFA), there is a unique next state for transition on input in a given state. If we relax this condition of uniqueness of the next state in DFA, we get NFA. But any set accepted by NFA can be also accepted by DFA. However the concept of non-determinism plays an important role in both the theory of computation and design and analysis of algorithm, specially in defining complexity classes. In the next unit we will examine a more powerful machine and explain the terms computability and complexity classes.

2.1 OBJECTIVES

After studying this unit, you should be able to:

Define alphabet, substring;

Define a language and various operations on languages;

Define and use a regular expression;

Define a finite automata for computation of a language;

Design a finite automata for a known language;

Define the term nondeterministic finite automata

Design nondeterministic finite automata for a known language

2.2 REGULAR EXPENSION

In this unit, first we shall discuss the definitions of alphabet, string, and language with some important properties.

2.2.1 Introduction to Defining of Languages

For a language, defining rules can be of two types. The rules can either tell us how to test a string of alphabet letters that we might be presented with, to see if it is a valid word, i.e., a word in the language or the rules can tell us how to construct all the words in the language by some clear procedures.

Alphabet: A finite set of symbols/characters. We generally denote an alphabet by $\sum$. If we start an alphabet having only one letter, say, the letter z, then $\sum = \{z\}$

Letter: Each symbol of an alphabet may also be called a letter of the alphabet or simply a letter.

Language over an alphabet: A set of words over an alphabet. Languages are denoted by letter L with or without a subscript.

String/word over an alphabet: Every member of any language is said to be a string or a world.

Example 1: Let L_1 be the language of all possible strings obtained by L_1 = {z, zz, zzz, zzzz . . . . . .}

This can also be written as
$L_1 = \{z^n\}$ for n = 1, 2, 3, …..

A string of length zero is said to be **null string** and is represented by ∧.

Above given language L_1 does not include the null string. We could have defined it so as to include ∧. Thus, $L = \{z^n \mid n=0, 1, 2, 3\ldots\}$ contains the null string.

In this language, as in any other, we can define the operation of concatenation, in which two strings are written down side by side to form a new longer string. Suppose u = ab and v = baa, then uv is called concatenation of two strings u and v and is uv = abbaa and vu = baaab. The operation of concatenation is analogous to addition:

z^n concatenated with z^m is the word z^{n+m}.

Example 2: If the word zzz is called c and the word zz is called d, then the word formed by concatenating c and d is

cd = zzzzz

When two words in our language L_1 are concatenated they produce another word in the language L_1. However, this may not be true in all languages.

Example 3: If the language is L_2 = {z, zzz, zzzzz, zzzzzzz…..}

$= \{z^{odd}\}$

$= \{z^{2n+1}$ for n = 0, 1, 2, 3….}

then c = zzz and d = zzzzz are both words in L_2, but their concatenation cd = zzzzzzzz is not a word in L_2. The reason is simple that member of L_2 are of odd length while after concatenation it is of even length.

Note: The alphabet for L_2 is the same as the alphabet for L_1.

Example 4: A Language L_3 may denote the language having strings of even lengths include of length 0. In other words, L_3 = { ∧zz, zzzz, …..}

Another interesting language over the alphabet ∑ = {z} may be

Example 5: $L_4 = \{z^p : p \text{ is a prime natural number}\}$.
There are infinitely many possible languages even for a single letter alphabet ∑ = {z},

In the above description of concatenation we find very commonly, that for a single letter alphabet when we concatenate c with d, we get the same word as when we concatenate d with c, that is cd = dc **But this relationship does not hold for all languages**. For example, in the English language when we concatenate "Ram" and "goes" we get "Ram goes". This is, indeed, a word but distinct from "goes Ram".

Now, let us define the reverse of a language L. If c is a word in L, then reverse (c) is the same string of letters spelled backward.
The reverse (L) = {reverse (w), w ∈ L}

Example 6: Reverse (zzz) = zzz

Reverse (173) = 371

Let us define a new language called PALINDROME over the alphabet ∑ = {a,b}.

PALINDROME = {∧ , and all strings w such that reverse (w) = w}

= {∧ , a, b, aa, bb, aaa, aba, bab, bbb, aaaa, abba, …}

Concatenating two words in PALINDROME may or may not give a word in palindrome, e.g., if u = abba and v = abbcba, then uv = abbaabbcbba which is not palindrome.

2.2.2 Kleene Closure Definition

Suppose an alphabet ∑ , and define a language in which any string of letters from ∑ is a word, even the null string. We shall call this language the closure of the alphabet. We denote it by writing * after the name of the alphabet as a superscript, which is written as $\sum^*$. This notation is sometimes also known as **Kleene Star**.

For a given alphabet ∑ , the language $\sum^*$ ($sigma^*$) consists of all possible strings, including the null string.

For example, If ∑ = {z}, then, $\sum^* = L_1$ = {∧, z, zz, zzz …..}

Example 7: If ∑= {0, 1}, then, ∑ * = {∧, 0, 1, 00, 01, 10, 11, 000, 001 …..}
So, we can say that Kleene Star is an operation that makes an infinite language of strings of letters out of an alphabet, if the alphabet, ∑ ≠ ϕ. However, by the definition alphabet ∑ may also be ϕ. In that case, $\sum^*$ is finite. By "infinite language, we mean a language with infinitely many words.

Now, we can generalise the use of the star operator to languages, i.e., to a set of words, not just sets of alphabet letters.

Definition: If s is a set of words, then by s^* we mean the set of all finite strings formed by concatenating words from s, where any word may be used as often.

Example 8: If s = {cc, d}, then

s^* = {∧ or any word composed of factors of cc and d}
= {∧ or all strings of c's and d's in which c's occur in even clumps}.
The string ccdcccd is not in s^* since it has a clump of c's of length 3.
$\{x : x \wedge = \text{or } x = (cc)^{i_1} d^{j_1} (cc)^{i_2} d^{j_2} \ldots\ldots (cc)^{i_m} (d)^{j_m} \}$ where $i_1, j_1, \ldots\ldots i_m, j_m \geq 0$

Positive Closure: If we want to modify the concept of closure to refer to only the concatenation leading to non-null strings from a set s, we use the notation + instead of *. This plus operation is called positive closure.

Theorem 1: For any set s of strings prove that $s^* = (s^*)^* = s^{**}$

Proof: We know that every word in s^{**} is made up of factors from s^*.
Also, every factor from s^* is made up of factors from s.
Therefore, we can say that every word in s^{**} is made up of factors from s.

First, we show $s^{**} \subset s^*$. (i)
Let $x \in s^{**}$…. Then $x = x_1 \ldots\ldots x_n$ for some $x_1 \in s^*$ which implies $s^{**} \subset s^*$

Next, we show $s^* \subset s^{**}$.
$s^* \subset s^{**}$ (ii)

By above inclusions (i) and (ii), we prove that
$s^* = s^{**}$

Now, try some exercises.

Ex.1) If u = ababb and v = baa then find
(i) uv (ii) vu (iii) uv (iv) vu (v) uuv.

Ex.2) Write the Kleene closure of the following
(i) {aa, b}
(ii) {a, ba}

2.2.3 Formal Definition of Regular Expressions

Certain sets of strings or languages can be represented in algebraic fashion, then these algebraic expressions of languages are called **regular expressions**. Regular expressions are in **Bold** face. The symbols that appear in regular use of the letters of the alphabet ∑ are the symbol for the null string ∧ , parenthesis, the star operator, and the plus sign.

The set of regular expressions is defined by the following rules:

1. Every letter of ∑ can be made into a regular expression ∧ itself is a regular expression.
2. Φ is a regular expression

3 If **l** and **m** are regular expressions, then so are

(i) **(l)**

(ii) **lm**
(iii) **l+m**
(iv) $\mathbf{l}^*$
(v) $\mathbf{l}^+ = \mathbf{ll}^*$

4 Nothing else is regular expression.

For example, now we would build expression from the symbols 0,1 using the operations of union, concatenation, and Kleene closure.
(i) **01** means a zero followed by a one (concatenation)
(ii) **0+1** means either a zero or a one (union)
(iii) $\mathbf{0}^*$ means ∧**+0+00+000+**….. (Kleen closure) .

With parentheses, we can build larger expressions. And, we can associate meanings with our expressions. Here's how

Expression	**Set represented**
$\mathbf{(0+1)}^*$	all strings over {0,1}
$\mathbf{0}^*\mathbf{10}^*\mathbf{10}^*$	strings containing exactly two ones
$\mathbf{(0+1)}^*\mathbf{11}$	strings which end with two ones.

The language denoted/represented by the regular expression R is L(R).

Example 9: The language L defined by the regular expression **ab*a** is the set of all strings of a's and b's that begin and end with a's, and that have nothing but b's inside.
L = {∧ aa, aba, abba, abbba, abbbba, }

Example 10: The language associated with the regular expression $\mathbf{a}^*\mathbf{b}^*$ contains all the strings of a's and b's in which all the a's (if any) come before all the b's (if any).

L = {∧, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaa,…)

Note that ba and aba are not in this language. Notice also that there need not be the same number of a's and b's.

Example 11: Let us consider the language L defined by the regular expression $\mathbf{(a+b)}^*$ $\mathbf{a(a+b)}^*$**.** The strings of the language L are obtained by concatenating a string from the language corresponding to $\mathbf{(a+b)}^*$ followed by a followed by a string from the language associated with $\mathbf{(a+b)}^*$. We can also say that the language is a set of all words over the alphabet $\sum$ = {a,b} that have an a in them somewhere.

To make the association/correspondence/relation between the regular expressions and their associated languages more explicit, we need to define the operation of multiplication of set of words.

Definition: If S and T are sets of strings of letters (they may be finite or infinite sets), we define the product set of strings of letters to be. ST = {all combinations of a string from S concatenated with a string from T in that order}.

Example 12: If S = {a, aa, aaa}, T = {bb, bbb} Then, ST = {abb, abbb, aabb, aabbb, aaabb, aaabbb}.

Example 13: If S = {a bb bab}, T = {∧ bbbb}

Then, ST = {a, bb, bab, abbbb, bbbbbb ,babbbbb}

Ex.3) Find a regular expression to describe each of the following languages:

(a) {a,b,c}

(b) {∧,a,abb,abbbb,….}

Ex.4) Find a regular expression over the alphabet {0,1,} to describe the set of all binary numerals without leading zeroes (except 0 itself). So the language is the set

{0,1,10,11,100,101,110,111,…}.

2.2.4 Algebra of Regular Expressions

There are many general equalities for regular expressions. We will list a few simple equalities together with some that are not so simple. All the properties can be verified by using properties of languages and sets. We will assure that R,S and T denote the arbitrary regular expressions.

Properties of Regular Expressions

1. $(R+S)+T = R+(S+T)$
2. $R+R = R$
3. $R+\phi = \phi+R = R.$
4. $R+S = S+R$
5. $R\phi = \phi R = \phi$
6. $R\wedge = \wedge R = R$
7. $(RS)T = R(ST)$
8. $R(S+T) = RS+RT$
9. $(S+T)R = SR+TR$
10. $\phi^* = \wedge^* = \wedge$
11. $R^*R^* = R^* = (R^*)^*$
12. $RR^* = R^*R = R^* = \wedge+RR^*$
13. $(R+S)^* = (R^*S^*)^* = (R^*+S^*)^* = R^*S^* = (R^*S)^*R^* = R^*(SR^*)^*$
14. $(RS)^* = (R^*S^*)^* = (R^*+S^*)^*$

Theorem 2: Prove that R+R = R

Proof : We know the following equalities:

L(**R+R**) = L(**R**)UL(**R**) = L(**R**)

So **R**+**R** = **R**

Theorem 3: Prove the distributive property

R(S+T) = RS+RT

Proof: The following set of equalities will prove the property:

L(R(S+T)) = L(R)L(S+T)
= L(R)(L(S)UL(T))
= (L(R)L(S))U(L(R)L(T))
= L(RS+RT)

Similarly, by using the equalities we can prove the rest. The proofs of the rest of the equalities are left as exercises.

Example 15: Show that $R+RS^*S = a^*bS^*$, where $R = b+aa^*b$ and S is any regular expression.

$R+RS^*S = R \wedge+RS^*S$ (property 6)

$= R(\wedge+S^*S)$ (property 8)

$= R(\wedge+SS^*)$ (property 12)

$= RS^*$ (property 12)

$= (b+aa^*b)S^*$ (definition of R)
$= (\wedge+aa^*)\, bS^*$ (properties 6 and 8)

$= a^*bS^*$. (Property 12)

Try an exercise now.

Ex.5) Establish the following equality of regular expressions:
$b^*(abb^*+aabb^*+aaabb^*)^* = (b+ab+aab+aaab)^*$

As we already know the concept of language and regular expressions, we have an important type of language derived from the regular expression, called **regular language.**

2.3 REGULAR LANGUAGE

Language represented by a regular expression is called a regular language. In other words, we can say that a regular language is a language that can be represented by a regular expression.

Definition: For a given alphabet , the following rules define the regular language associated with a regular expression.

Rule 1: ϕ,{$\wedge$} and {a} are regular languages denoted respectively by regular expressions ϕ and $\wedge$.

Rule 2: For each a in , the set {a} is a regular language denoted by the regular expression **a**.

Rule 3: If **l** is a regular expression associated with the language L and **m** is a regular expression associated with the language M, then:

(i) The language = {xy : x ∈ L and y ∈ M} is a regular expression associated with the regular expression **lm**

(ii) The regular expression **l+m** is associated with the language formed by the union of the sets L and M.

language (**l+m**) = L∪M

(iii) The language associated with the regular expression $(\mathbf{l})^*$ is L^*, the Kleen Closure of the set L as a set of words:

language $(\mathbf{l}^*) = L^*$.

Now, we shall derive an important relation that, all finite languages are regular.

Theorem 4: If L is a finite language, then L can be defined by a regular expression.

In other words, all finite languages are regular.

Proof: A language is finite if it contains only finitely many words.

To make one regular expression that defines the language L, turn all the words in L into bold face type and insert plus signs between them. For example, the regular expression that defines the language L = {baa, abbba, bababa} is **baa + abbba + bababa**

Example16: If L = {aa, ab, ba, bb}, then the corresponding regular expression is **aa + ab +ba + bb.**

Another regular expression that defines this language is **(a+b) (a+b).**

So, a particular regular language can be represented by more than one regular expressions. Also, by definition, each regular language must have at least one regular expression corresponding to it.

Try some exercises.

Ex.6) Find a language to describe each of the following regular expressions:
(a) **a+b** (b) $\mathbf{a+b^*}$ (c) $\mathbf{a^*bc^*+ac}$

Ex.7) Find a regular expression for each of the following languages over the alphabet {a,b}:

(a) strings with even length.
(b) strings containing the sub string aba.

2.4 FINITE AUTOMATA

Finite automata are important in science, mathematics, and engineering. Engineers like them because they are superb models for circuits (and, since the advent of VLSI systems sometimes finite automata represent circuits.) computer scientists adore them because they adapt very likely to algorithm design. For example, the lexical analysis portion of compiling and translation. Mathematicians are introduced by them too due to the fact that there are several nifty mathematical characterizations of the sets they accept.
Can a machine recognise a language? The answer is yes for some machine and some an elementary class of machines called finite automata. Regular languages can be represented by certain kinds of algebraic expressions by Finite automaton and by certain grammars. For example, suppose we need to compute with numbers that are represented in scientific notation. Can we write an algorithm to recognise strings of symbols represented in this way? To do this, we need to discuss some basic computing machines called finite automaton.

An automata will be a finite automata if it accepts all the words of any regular language where language means a set of strings. In other words,

2.4.1 Finite Automata

A system where energy and information are transformed and used for performing some functions without direct involvement of man is called automaton. Examples are automatic machine tools, automatic photo printing tools, etc.

A finite automata is similar to a finite state machine. A finite automata consists of five parts:

(1) a finite set of states;
(2) a finite set of alphabets;
(3) an initial state;
(4) a subset of set of states (whose elements are called “yes” state or; accepting state;) and
(5) a next-state function or a transition state function.

A finite automata over a finite alphabet A can be thought of as a finite directed graph with the property that each node emits one labelled edge for each distinct element of A. The nodes are called states. There is one special state called **the start** (or **initial**) state, and there is a subset of states called **final states** (which could be possibly empty)

For example, the labelled graph in fig.1 given below represents a DFA over the alphabet A = {a,b} with start state 1 and final state 4.

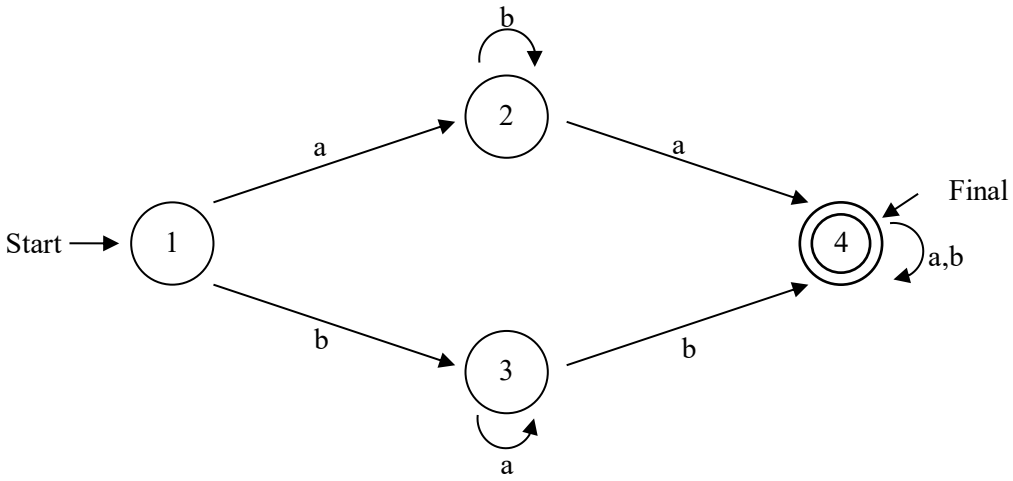


Fig. 1: Finite Automata

We always indicate the start state by writing the word start with an arrow painting to it. Final states are indicated by double circle.

The single arrow out of state 4 labelled with a,b is short hand for two arrows from state 4, going to the same place, one labelled a and one labelled b. It is easy to check that this digraph represents a DFA over {a,b} because there is a start state, and each state emits exactly two arrows, one labelled with a and one labelled with b.

So, we can say that a finite automaton is a collection of three tuples:

1. A finite set of states, one of which is designated as the initial state, called the start state, and some (may be none) of which we designated as final states.

2. An alphabet $\sum$ of possible input letters from which are formed strings that are to be read one letter at a time.

3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go to next.

For example the input alphabet has only two letters a and b. Let us also assume that there are only three states, x, y and z. Let the following be the rules of transition:

1. from state x and input a go to state y;
2. from state x and input b go to state z;
3. from state y and input b go to state x;
4. from state y and input b go to state z; and
5. from state z and any input stay at state z.

Let us also designate state x as the starting state and state z as the only final state.

Let us examine what happens to various input strings when presented to this FA. Let us start with the string aaa. We begin, as always, in state x. The first letter of the string is an a, and it tells us to go state y (by rule 1). The next input (instruction) is also an a, and this tells us (by rule 3) to go back to state x. The third input is another a, and (by Rule 1) again we go to the state y. There are no more input letters in the input string, so our trip has ended. We did not finish in the final state (state z), so we have an unsuccessful termination of our run.

The string aaa is not in the language of all strings that leave this FA in state z. The set of all strings that do leave as in a final state is called the language defined by the finite automaton. The input string aaa is not in the language defined by this FA. We may say that the string aaa is not accepted by this FA because it does not lead to a final state. We may also say "aaa is rejected by this FA." The set of all strings accepted is the language associated with the FA. So, we say that L is the language accepted by this FA. FA is also called a language recogniser.

Let us examine a different input string for this same FA. Let the input be abba. As always, we start in state x. Rule 1 tells us that the first input letter, a, takes us to state y. Once we are in state y we read the second input letter, which is a b. Rules 4 now tells us to move to state z. The third input letter is a b, and since we are in state z, Rule 5 tells us to stay there. The fourth input letter is an a, and again Rule 5 says state z. Therefore, after we have followed the instruction of each input letter we end up in state z. State z is designated as a final state. So, the input string abba has taken us successfully to the final state. The string abba is therefore a word in the language associated with this FA. The word abba is accepted by this FA.

It is not difficult for us to predict which strings will be accepted by this FA. If an input string is made up of only the letter a repeated some number of times, then the action of the FA will be jump back and forth between state x and state y. No such word can ever be accepted.

To get into state z, it is necessary for the string to have the letter b in it as soon as a b is encountered in the input string, the FA jumps immediately to state z no matter what state it was before. Once in state z, it is impossible to leave. When the input strings run out, the FA will still be in state z, leading to acceptance of the string.

So, the FA above will accept all the strings that have the letter b in them and strings not of this form are never accepted. Therefore, the language associated with this FA is the one defined by the regular expression $\mathbf{(a+b)^* b(a+b)^*}$.

The list of transition rules can grow very long. It is much simpler to summarise them in a table format. Each row of the table is the name of one of the states in FA, and each column of this table is a letter of the input alphabet. The entries inside the table are the new states that the FA moves into the transition states. The transition table for the FA we have described is:

Table 1

State	Input	
	a	b
Start x	y	z
y	x	z
Final z	z	z

The machine we have already defined by the transition list and the transition table can be depicted by the state graph in Figure 2.

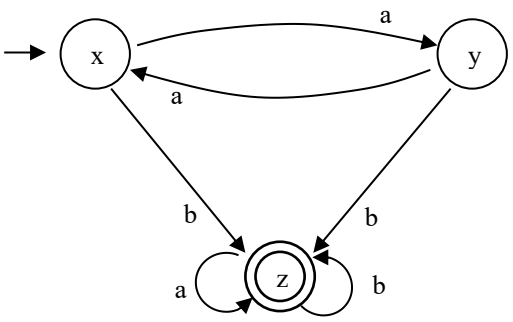


Fig. 2: State Transition graph

Note: A single state can be start as well as final state both. There will be only one start state and none or more than one final states in Finite Automaton.

2.4.2 Another Method to Describe FA

There is a traditional method to describe finite automata which is extremely intuitive. It is a picture called a graph. The states of the finite automaton appear as vertices of the graph while the transitions from state to state under inputs are the graph edges. The state graph for the same machine also appears in Figure3 given below.

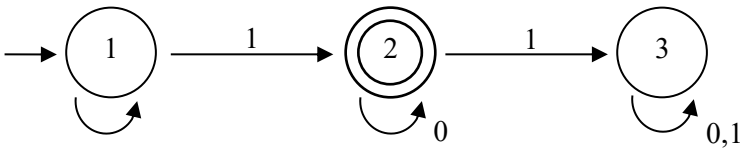


Fig. 3: Finite automata

The finite automata shown in Figure 3 can also be represented in Tabular form as below:

Table 2

	State	Input		Accept?
		0	1	
Start	1	1	2	No
Final	2	2	3	Yes
	3	3	3	No

Before continuing, let's examine the computation of a finite automaton. Our first example begins in state one and reads the input symbols in turn changing states as necessary. Thus, a computation can be characterized by a sequence of states. (Recall that Turing machine configurations needed the state plus the tape content. Since a finite automaton never writes, we always know what is on the tape and need only look at a state as a configuration.) Here is the sequence for the input 0001001.

Input Read:	0		0		0		1		0		0		1	
State:	1	→	1	→	1	→	1	→	2	→	2	→	2	→	3

Example 17 (An elevator controller): Let's imagine an elevator that serves two floors. Inputs are calls to a floor either from inside the elevator or from the floor itself. This makes three distinct inputs possible, namely:

0 - no calls
1 - call to floor one
2.- call to floor two

The elevator itself can be going up, going down, or halted at a floor. If it is on a floor, it could be waiting for a call or about to go to the other floor. This provides us with the six states shown in figure 4 along with the state graph for the elevator controller.

W1 Waiting on first floor
U1 About to go up
UP Going up
DN Going down
W2 Waiting-second floor
D2 About to go down

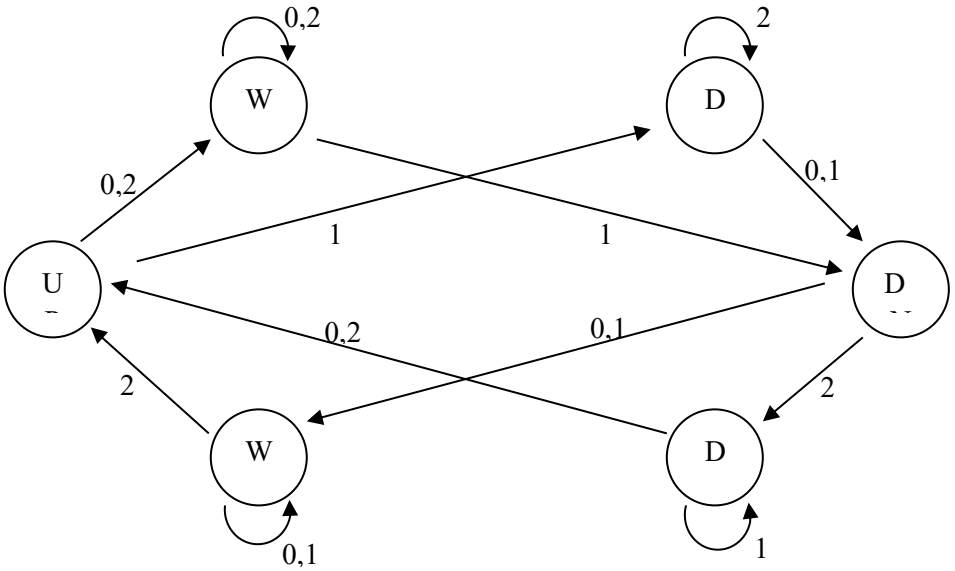


Fig. 4: Elevator Control

A transition state table for the elevator is given in table3:

Table 3 Elevator Control

State	Input		
	None	**call to 1**	**call to 2**
W1 (wait on 1)	W1	W1	UP
U1 (start up)	UP	U1	UP
UP	W2	D2	W2
DN	W1	W1	U1
W2 (wait on 2)	W2	DN	W2
D2 (start down)	DN	DN	D2

Accepting and rejecting states are not included in the elevator design because acceptance is not an issue. If we were to design a more sophisticated elevator, it might have states that indicated:

Finite automata

a) power failure
b) overloading, or
c) breakdown

In this case, acceptance and rejection might make sense.

Let us make a few small notes about the design. If the elevator is about to move (i.e., in state U1 or D2) and it is called to the floor it is presently on it will stay. (This may be good Try it next time you are in an elevator.) And, if it is moving (up or down) and gets called back the other way, it remembers the call by going to the U1 or D2 state upon arrival on the next floor. Of course, the elevator does not do things like open and close doors (these could be states too) since that would have added complexity to the design. Speaking of complexity, imagine having 100 floors.

That is our levity for this section. Now that we know what a finite automaton is, we must (as usual) define it precisely.

Definition : ***A finite automata M is a quintuple M = (Q, $\sum$,δ , q_O, F) where :***

Q is a finite set (of states)
$\sum$ is a finite alphabet (of input symbols)
$\boldsymbol{\delta}$: $Q \times \sum \rightarrow Q$ (next state function)
$q_O \in Q$ (the starting state)
$F \subseteq Q$ (the accepting states)

We also need some additional notation. The next state function is called the transition function and the accepting states are often called final states. The entire machine is usually defined by presenting a transition state table or a transition diagram. In this way, the states, alphabet, transition function, and final states are constructively defined. The starting state is usually the lowest numbered state. Our first example of a finite automaton is:

$M = (\{q_1, q_2, q_3\}, \{0,1\}, , q_1, \{q_2\}$

Let us look again at a computation by our first finite automaton. For the input 010, our machine begins in q_1, reads a 0 and goes to $\delta(q_2,0) = q_2$ after reading the final 0. All that can be put together as:

$\delta(\delta(\delta(q_1,0),1)0) = q_2$

We call this transition on strings δ^* and define it as follows:

Definition : Let M = ($Q,\sum,\delta,q_O,F$). For any input string x, input symbol a, and state q_i, the *transition function on strings δ^* takes the values:*

$\delta^*(q_i,(*e)) = w_i$
$\delta^*(q_i,a) = \delta(q_i,,a)$ a
$\delta^*(q_i,xa) = \delta(\delta^*(q_i,x),a) \forall a \in \sum , x \in \sum^*$

That certainly was terse. But δ^* is really just what one expects it to be. It merely applies the transition function to the symbols in the string.

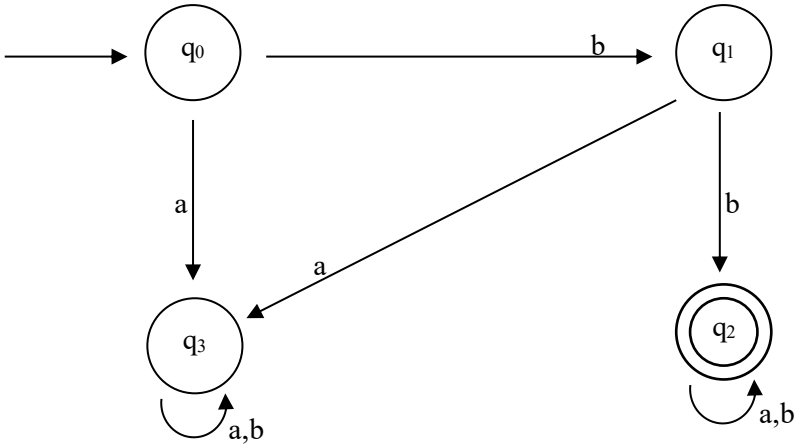


Fig. 5: Finite automata

This machine has a set of states = {q_0, q_1, q_2, q_3) and operates over the input alphabet {a,b}. In the above figure q_0 is the starting state and its set of final or accepting states, F = {q_2} an accepting state can also be shown by two concentric circles as shown in the fig.5

The transition function is fully described twice once in figure 6 as a state graph and once in tasble 4 as a state table.

Table 4

State	Input		Accept?
	A	b	
0	3	1	No
1	3	2	No
2	2	2	Yes
3	3	3	No

If the machine receives the input bbaa, it goes through the sequence of states:

q_0, q_1, q_2, q_2, q_2

While when it gets an input such as abab, it goes through the state transition:

q_0, q_3, q_3, q_3, q_3

Now we shall become a bit more abstract. When a finite automaton receives an input string such as:

$x = x_1 x_2 \ldots x_n$

where the x_i are symbols from its input alphabet, it progresses through the sequence:

$q_{k_1}, q_{k_2}, \ldots q_{k_{n+1}}$

where the states in the sequence are defined as:

$q_{k_1} = q_0$

$q_{k_2} = \delta(q_{k_1}, x_1) = \delta(q_0, x_0)$

$q_{k_3} = \delta(q_{k_2}, x_2) = \delta^*(q_0, x_1 x_2)$

$q_{k_{n+1}} = \delta(q_{k_n}, x_n) = \delta^*(q_0, x_1 x_2 \ldots . x_n)$

Getting back to a more intuitive reality, the following table provides an assignment of values to the symbols used above for an input of bbaba to the finite automaton of figure 3.

i	1	2	3	4	5	6
x_i	b	b	a	b	a	
q_{k_1}	q_0	q_1	q_2	q_2	q_2	q_2

Definition: *The **set** (of strings) **accepted** by the finite automaton* $M = (Q, \sum, \delta, q_0, F)$ *is* $T(M) = \{x / \delta^*(q_0,x) \in F\}$

This set of accepted strings (L(M) to mean for language accepted by M) is merely all of the strings for which M ended up in a final or accepting state after processing the string. For our first example (figure 1) this was all strings of 0's and 1's that contain exactly one 1. Our last example (figure 3) accepted the set of strings over the alphabet {a,b} which began with exactly two b's.

2.4.3 Finite Automata as Output Devices

The automata that we have discussed so far have only a limited output capability to the extent that only outputs are 'accepted' and 'not accepted' to indicating the acceptance or rejection of an input string. We want to introduce two classic models for finite automata that have additional output capability. We will consider machines that transform input strings into output strings. These machines are basically DFAs, except that we associate an output symbol with each state or with each state transition. But there are no final states because we are not interested in acceptance or rejection.

Mealy and Moore Machines

The first model invented by Mealy [1955] is called a Mealy machine. It associates an output letter with each transition. For example, if the output associated with the edge labelled with the letter a is x, we shall write a/x on that edge. A state transition for a Mealy machine can be presented in figure 7 as follows:

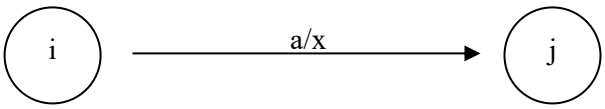


Fig. 7: Mealy machine

Indicating that the machine in state i and on input a gives output x and enters state j.

In a Mealy machine, an output always takes place during a transition of the states. The second model invented by Moore [1956], is called a Moore machine. It associates an output letter with each state. For example, if the output associated with state I is x, we will always write i/x inside the state circle. A typical state transition for a Moore machine can be presented in figure8 as follows:

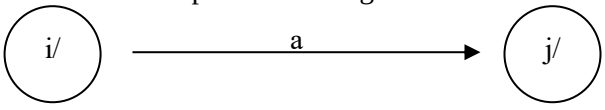


Fig. 8: Moore machine

In a Moore machine, each time a state is entered, simultaneously an output takes place. So, the first output always occurs as soon as the machine is started. Mealy and Moore machines are equivalent. In other words, any problem that is soluble by one type of machine can also be solved by the other type of machine.

Example 18: Suppose we want to compute the number of sub strings of the form bab that occurs in an arbitrary input string over the alphabet {a,b}.

The diagrammatic representation of a Mealy machine for the task is given below in figure 9:

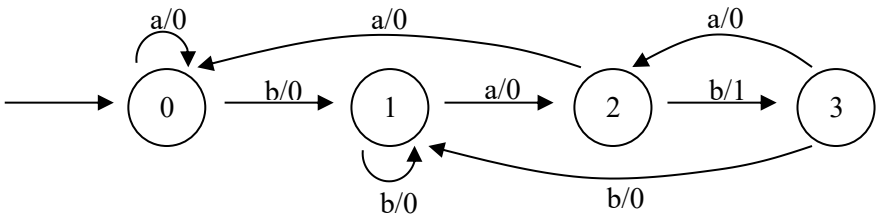


Fig. 9: Mealy machine

For example, the output of this Mealy machine for the sample string abababaababb is 000101000010, where each 1 indicates the availability of a (or an additional) substring up to that point. On the other hand, a 0 indicates that the three previous inputs including the current input do not form a substring of the form bab.

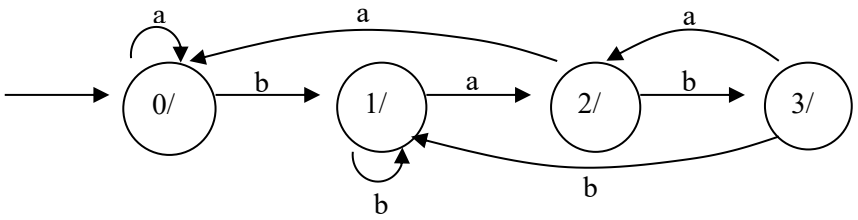


Fig. 10: Moore machine

For example, the output of this Moore machine for the simple string. Abababaababb is 0000101000010. We can count the number of 1's in the output string to obtain the number of occurrence of the sub string bab.

Example 19: A Simple Traffic Signal : Suppose we have a simple traffic intersection, where a north-south highway intersects an east-west highway. We will assume that the east-west highway always has a green light unless some north-south traffic is detected by sensors. When north-south traffic is detected, after a certain time delay the signals change and stay that way for a fixed period of time. We are required to design an appropriate circuit to capture the desired result stated above. We construct a Moore machine as a model of the required circuit as follows:

The input symbols for the required Moore machine are 0 (no traffic detected) and 1 (traffic detected). Let G, Y and R mean the colours Green, Yellow and Red, respectively. The output strings are GR, YR, RG, AND RY, where the first letter of a string is the colour of the east-west light and the second letter of a string is the colour of the north-sought light. The Moore machine model for this simple traffic intersection problem is given below diagrammatically:

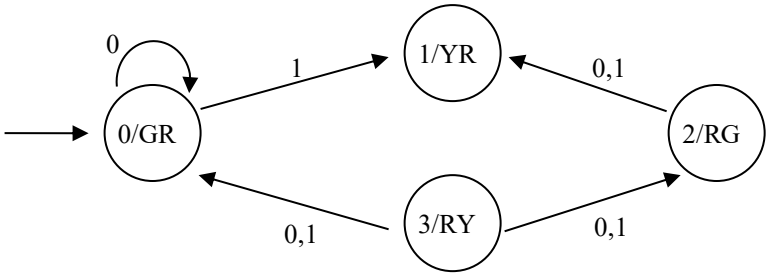


Fig. 11: Traffic signal transition diagram

Mealy machines appear to be more useful than Moore machines. But problems like traffic signal control have hic Moore machine solutions because each state is associated with a new output configuration.

Let us try some exercises:

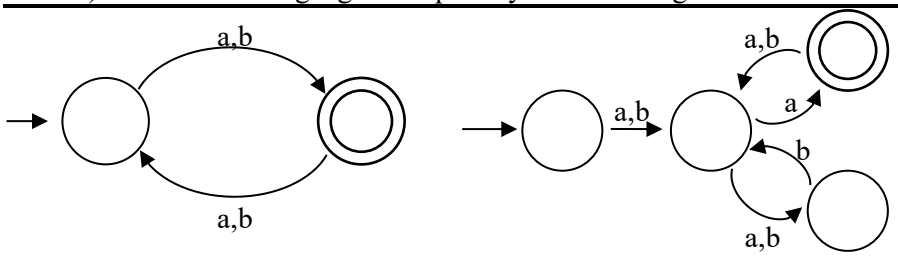
Ex.8) Build a new FA that accepts only the word ∧. Also write the corresponding regular expression.

Ex.9) Build an FA that accepts only those words that have even lengths. Also write the regular expression.

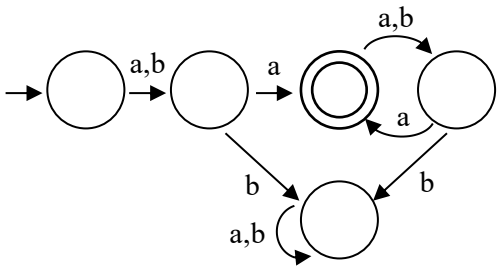
Ex.10) Build an FA that accepts only the word baa, ab and abb and no other words. Also write the corresponding regular expression.

Ex.11) Build an FA that will accept the language of all words each having twice as many a's as the number of b's. Also write the corresponding regular expression.

Ex.12) Describe the languages accepted by the following FA's:



(a) **(b)**



(c)

Figure 12:

2.5 NON DETERMINISTIC FINITE AUTOMATA

You have already studied finite automata (though 'automata' is a plural form of the

noun 'automaton', the word 'automata' is also used in singular sense). Now consider an automata that accepts all and only strings ending in 01, represented diagrammatically, as follows:

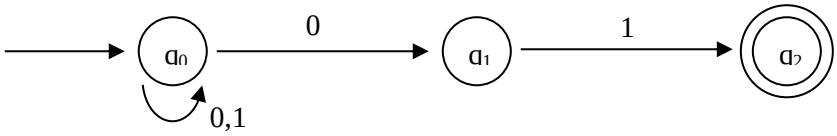


Fig. 13: Transition Diagram

In the case of the finite automata shown in figure 1, the following points may benoted:

(i) On input 0 in state q_0, the next state may be either of the two states viz., q_0 or q_1.
(ii) There is no next state on input 0 in the state q_1.
(iii) There is no next state on input 0 and 1 in the state q_2.

In this transition system, what happens when this automata processes the input .00101?

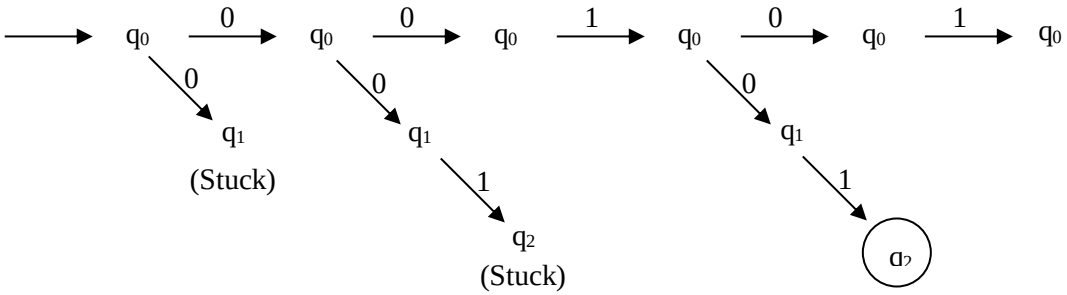


Fig. 14: Processing of string 00101

Here from the initial state q_0, for the processing of alphabet 0, there are two states at once or viewed another way, it can be 'guessed" which state to go to next. Such a finite automata allows to have a choice of 0 or more next states for each state input pair and is called a non-deterministic finite automata. An NFA can be in several states at once.

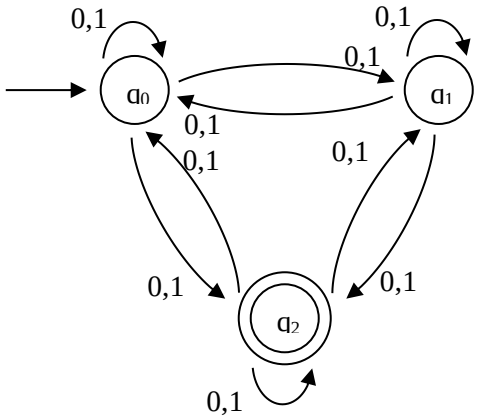


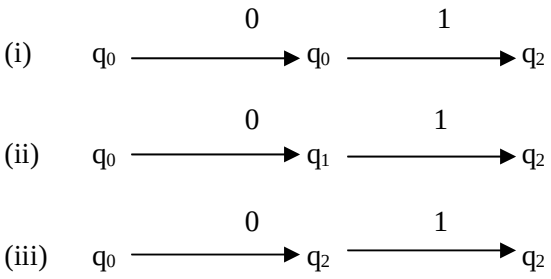
Fig. 15: Transition diagram

Before going to the formal definition of NFA, let us discuss one more case of non-determinism of finite automata. Suppose Q = {q_0, q_1, q_2}, $\sum$ = {0, 1}, q_0 is an initial state and q_2 is final state. Again, suppose the processing of any input symbol does not result in the transition to a unique state, but results a chain of states. Let us consider a

machine given in figure 3.

For the sake of convenience, let us check the processing of any input symbol. Fromthe state q_0, after processing 0, resulting states are q_0, q_1, q_2 and for input symbol 1, there are three possible states q_0, q_1 and q_2 not a unique state. It clarifies that a non-deterministic automata can have more than one possible state or none state after processing any input symbol from $\sum$.

Let us check how the string 01 is processed by the above automata. Here we havethree paths to reach to the final state:



A generalisation which is obtained here by allowing of several states as a result of the processing of an input symbol is called non-determinism. If from any state, we can reach to several states or none state, then the finite automata becomes non-deterministic in nature.

Formally, a non-deterministic finite automata is a quintuple

$A = (Q, \sum, \delta, q_0, F)$
Where

* Q is a finite set of states
* $\sum$ is a finite alphabet for inputs
* δ is a transition function from $Q \times \sum$ to the power set of Q i.e. to 2^Q
* $q_0 \in Q$ is the start/initial state
* $F \subseteq Q$ is a set of final/accepting states.

The NFA, for the example just considered, can be formally represented as:
$(\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$

Where the transition function, is given by the table 1:

Table1

States	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
q_2	Ø	Ø

Now, let us prove that the NFA

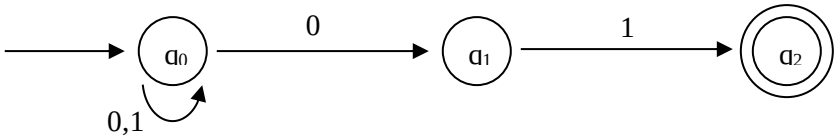


Fig. 16: NFA accepting x01

accepts the language $\{x01 : x \in \sum^*\}$ of all the strings that terminate with the sub-string 01. A mutual induction on the three statements below proves that the NFA accepts the given language.

1. $w \in \sum^* \Rightarrow q_0 \in \delta(q_0, w)$
2. $q_1 \in \delta(q_0, w) \Leftrightarrow w = x0$
3. $q_2 \in \delta(q_0, w)\ w = x01$

If $|w| = 0$ then $w = \wedge$. Then statement (1) follows from def., and statement & (2) and (3) show that all the string x01 will be accepted by the above non-deterministic automata.

Example 20: Consider the NFA with the formal description as $(Q, \sum, \delta, q_0, F)$ where $Q = \{q_0, q_1, q_2\}$, $\sum = \{a, b\}$, q_0 is the initial state and q_1 is only the final state, and δ is given by the following table:

Table 2

State	Input from	
	a	b
$\rightarrow q_0$	q_1, q_2	q_0
(q_1)	q_1	-
q_2	q_1	q_2

In NFA, though the function maps to a sub-set of the set of states, yet we generally drop braces, i.e., instead of $\{q_0, q_1\}$, we just write q_0, q_1.

The computation for an NFA is also similar to that of DFA. Let $N = (Q, \sum, \delta, q_0, F)$ be an NFA and w is a string over the alphabet $\sum$. The string w is accepted by NFA if corresponding to the input sequence, there exists a sequence of transitions from the initial state to any of the possible final states.

Now, let us check computations (in NFA, there are many possible computations) ofthe string aba.

$\delta(q_0, aba) = \delta(\delta(q_0, a), ba)$

$= \delta(q_1, ba)$ or $\delta(q_2, ba)$

$= \delta(\delta(q_1, b), a)$ or $\delta(\delta(q_2, b), a)$

= stuck or $\delta(q_2, a)$

= q_1 (an accepting state)

The above sequence of states shows the final state q_1 which is an accepting state. Hence, the string aba is accepted by the system and the input sequence of states for the input is $\rightarrow q_0 \xrightarrow{a} q_2 \xrightarrow{b} q_2 \xrightarrow{a} (q_1)$

Try some exercises:

Ex.13) Consider an NFA given in figure 5. Check whether the strings 001, 011101, 01110, 010 are accepted by the machine, or not?

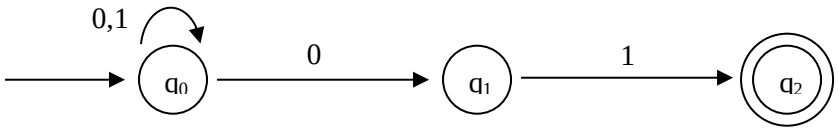


Fig. 17

Ex.14) Give an NFA which accepts all the strings starting with ab over {a,b}.

2.6 SUMMARY

In this unit we introduced the concepts of languages, regular expressions and regular langyages. Finite Automata are machines that recognize regular languages. From regular expressions, we can derive regular languages. In deterministic finite automata (DFA), there is a unique next state for transition on input in a given state. If we relax this condition of uniqueness of the next state in DFA, we get NFA.

2.7 SOLUTION/ANSWERS

Ex.1) (i) ababbbaa
(ii) baaababb
(iii) ab abb ab abb
(iv) baa baa
(v) ababbababb baa

Ex.2) (i) Suppose aa = x

Then $\{ x, b\}^* = \{\wedge , x, b, xx, bb, xb, bx, xxx, bxx, xbx, xxb, bbx, bxb, xbb, bbb\}$
substituting x = aa
$\{aa,b\}^* = \{\wedge , aa, b, aaaa, bb, aab, baa, aaaaaa, baaaa, aabaa, \ldots\}$

(ii) $\{a,ba\}^* = \{\wedge , a, ba, aa, baba, aba, baa, \ldots \}$

Ex.3) (a) **a+b+c**

(b) $\mathbf{ab^*+ba^*}$

(c) $\wedge+\mathbf{a(bb)^*}$

Ex.4) $\mathbf{0+1(0+1)^*}$

Ex.5) Starting with the left side and using properties of regular expressions, we get

$\mathbf{b^*(abb^* + aabb^*+aaabb^*)^*}$
$= \mathbf{b^*((ab+aab+aaab)b^*)^*}$ (property 9)
$= \mathbf{(b + ab + aab + aaab)^*}$ (property 7).

Ex.6) (a) {a,b}
(b) {a, ∧,b,bb,….b^n,….}
(c) {a,b,ab,bc,abb,bcc,…ab^n,bc^n,….}

Ex.7) (a) **(aa+ab+ba+bb)***
(b) **(a+b)*aba(a+b)***

Ex.8)

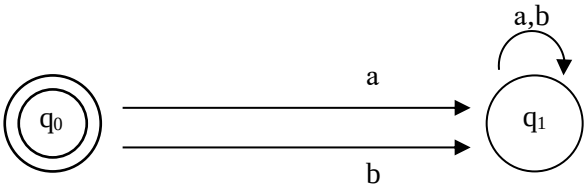


Fig. 18: Regular Expression of a null string

Ex.9)

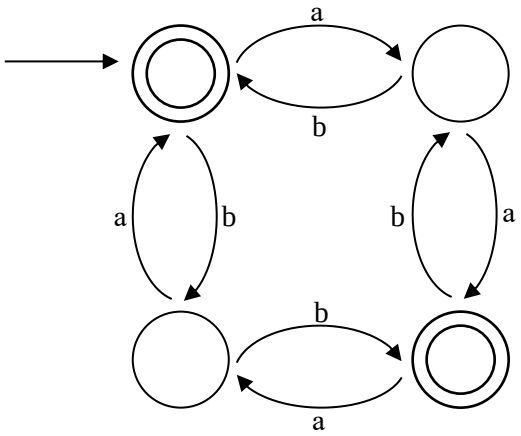


Fig. 19: Regular Expression is (aa+ba+ab+bb)

Ex.10) R.E. is **(baa + ab + abb)**

Ex.11) (i) All the words of odd lengths.
(ii) All the words ended with a.
(iii) All the words with a at even places.

Ex.12) is given by

State	Input	
	0	1
q_0	q_0, q_1	q_0
q_1	-	-
(q^2)	-	-

$\delta(q_0, 001) = \delta(q_0, 01) = \delta(q_1, 1) = (q_2)$ (Accepting state)

$\delta(q_0, 011101) = \delta(q_0, 11101)$

$= \delta(q_0, 1101)$

$= \delta (q_0, 101)$

$= \delta (q_0, 01)$

$= \delta (q_0, 1)$

$= q_2$ (accepting state)

$\delta (q_0, 01110) = \delta (q_0, 1110)$

$= \delta (q_0, 110)$

$= \delta (q_0, 10)$

$= \delta (q_0, 0)$

$= q_0$ or q_1 (Not an accepting state)

$\delta (q_0, 010) = \delta (q_0, 10)$

$= \delta (q_0, 0)$

$= q_0$ or q_1 (Not an accepting state)

So, strings 001 and 011101 are accepted by the given automata.

Ex.13)

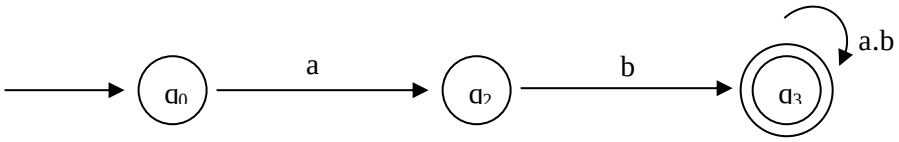


Fig. 20

UNIT 3 COMPUTABILITY AND COMPLEXITY

Structure

3.0 INTRODUCTION

Every system—natural or man-made, must be continuously, involved in some form of **computation** in its attempt at preserving its identity as a system.

In the previous unit, we discussed two major approaches to modeling of computation viz. the automata/machine approach and linguistic/grammatical approach. Under the automata approach, we discussed two models viz. Finite Automata and Nondeterministic Finite Automata. Under grammatical approach, we discussed only one model viz Regular Languages. We stated that the Finite Automata is a computational model which is equivalent to Regular Language Model and differentiated between a Deterministic Finite Automata and Nondeterministic Finite Automata.

In this unit we discuss a computational model called Turing Machine(TM) which is more powerful in terms of recognizing more languages and will help us **define and understand complexity classes** .

Turing Machine is named so, in Honor of its inventor Alan Mathison Turing (1921-1954). A.M. Turing, a British, was one of the greatest scholars of the twentieth century, and made profound contributions to the foundations of computer science.

We make an introduction to the concepts of a simple Turing Machine and Nondeterministic TM. Many notations and keywords are used to discuss the topics which are listed below:

Key words: Turing Machine (**TM**), Deterministic Turing Machine, Non-Deterministic Turing Machine, Turing Thesis, Computation, Configuration of TM, Turing-Acceptable Language, Turing DecidableLanguage,

Notations

TM	:	Turing Machine
⌜	:	Set of tape symbols, includes #, the blank symbol
Σ	:	Set of input/machine symbols, does not include #
Q	:	the finite set of states of TM
F	:	Set of final states
a,b,c…	:	Members of

σ	:	Variable for members of ∑
x or x	:	Any symbol of other than x
#	:	The blank symbol
α,β,γ	:	Variables for String over
L	:	Move the Head to the Left
R	:	Move the Head to the Right q : A state of TM, i.e, q ε Q
s or q_0	:	The start/initial state

Halt or h: The halt state. The same symbol h is used for the purpose of denoting halt state for all halt state versions of TM. And then h is not used for other purposes.

e or ∈ : The empty string

$C_1 \vdash_M C_2$: Configuration C_2 is obtained from configuration C_1 in *one* moveOf the machine M

$C_1 \vdash^* C_2$: Configuration C_2 is obtained from configuration C_1 in *finite* numberof moves.

$w_1 \underline{a}\ w_2$: The symbol a is the symbol currently being scanned by the Head

Or

w_1 a w_2: The symbol a is the symbol currently being scanned by the Head

↑

3.1 OBJECTIVES

After going through this unit, you should be able to:

Define all the terms listed under the keywords

Illustrate a basic Turing machine.

Explain the symbols used in a Turing Machine

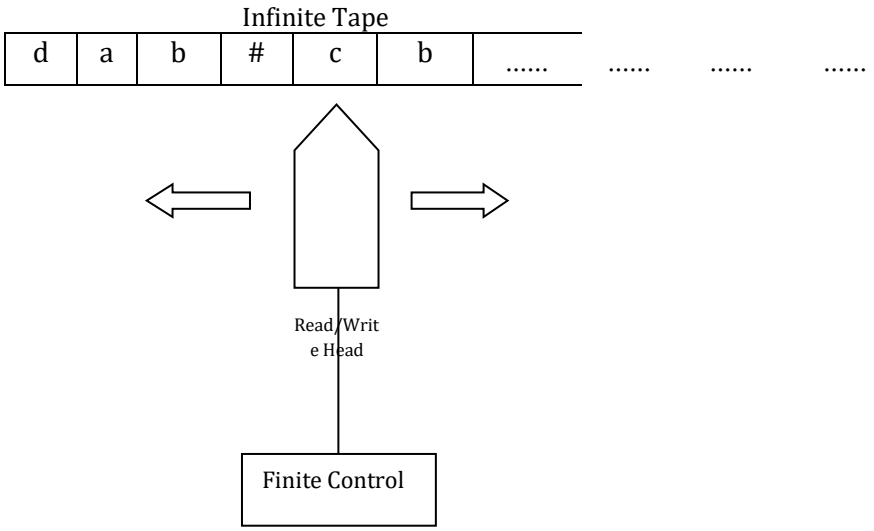
Explain the operations of a Turing Machine

Differentiate between TM and NDTM

Define Complexity Classes: P and NP

PRELUDE TO FORMAL DEFINITION

In the next section, we will notice through a formal definition of TM that a TM is an abstract entity constituted of mathematical objects like sets and a (partial) function. However, in order to help our understanding of the subject-matter of TMs, we can visualize a TM as a physical computing device that can be represented as a diagram as shown in below.



TURING MACHINE

Fig. 3.1

.

As shown in the above figure, TM consists of

(i) a **tape**, with an end on the left but infinite on the right side. The tape is divided into squares or cells, with each cell capable of holding one of the tape symbols including the blank symbol #. At any time, there can be only finitely many cells of the tape that can contain non-blank symbols. The set of **tape symbols** is denoted by

As the very first step in the sequence of operations of a TM, **the input, as a finite sequence of the input symbols is placed in the left-most cells of the tape.** The set of **input symbols** denoted by Σ , does not contain the blank symbol #. However, during operations of a TM, a cell may contain a **tape symbol** which is not necessarily an input symbol.
There are versions of TM, to be discussed later, in which the tape may be infinite in both left and right sides having neither left end nor right end.

(ii) **a finite control**, which can be in any one of the finite number of states.
The states in TM can be divided in three categories viz.

(a) the Initial state, the state of the control just at the time when TM starts its operations. The initial state of a TM is generally denoted by q_0 or s.

(b) the Halt state, which is the state in which TM stops all further operations.

The halt state is generally denoted by h. The halt state is distinct from the initial state. Thus, a TM HAS AT LEAST TWO STATES.

(c) Other states

(iii) **a tape head** (*or simply Head*), is always stationed at one of the tape cells and provides communication for interaction between the tape and the finite control. The Head can read or scan the symbol in the cell under it. The symbol is communicated to the finite control. The control taking into consideration the symbol and its current state decides for further course of action including

the change of the symbol in the cell being scanned and/or

change of its state and/or

moving the head to the Left or to the Right. The control may decide not to move the head.

The course of action is called a move of the Turing Machine. In other words, the move is a function of current state of the control and the tape symbol being scanned.

In case the control decides for change of the symbol in the cell being scanned, then the change is carried out by the head. This change of symbol in the cell being scanned is called writing of the cell by the head.

Initially, the head scans the left-most cell of the tape.

Now, we are ready to consider a **formal definition** of a Turing Machine in the next section.

3.2 TURING MACHINE

3.2.1 Turing Machine: Formal Definition

There are a number of versions of a TM. We consider below Halt State version of formal definition of a TM.

Definition: Turing Machine (Halt State Version)
A Turing Machine is a sextuple of the form (Q, Σ, ⌜,δ , q_o, h), where

(i) Q is the finite set of states,

(ii) Σ is the finite set of non-blank information symbols,

(iii) ⌜ is the set of tape symbols, including the blank symbol #

(iv) δ is the **next-move** partial function from Q x ⌜to Q x ⌜x {L, R, N},

where '**L**' denotes the tape Head moves to the left adjacent cell, '**R**' denotes tape Head moves to the Right adjacent cell and '**N**' denotes Head does not move, **i.e.**, continues scanning the same cell.

In other words, for $q_i \in Q$ and $a_k \in ⌜$, there exists (not necessarily always, Because δ is a partial function) some $q_j \in Q$ and some $a_l \in ⌜$such that $\delta (q_i a_k) = (q_j, a_l, x)$, where x may assume any one of the values 'L', 'R' and 'N'.

The **meaning** of δ (q_i, a_k) = (q_j, a_l, x) is that if q_i is the current state of the TM, and a_k is cell currently under the Head, then TM writes a_l in the cell currently under the Head, enters the state q_j and the *Head moves to the right adjacent cell, if the value of x is R, Head moves to the left adjacent cell, if the value of x is L* and continues scanning the same cell, if the value of x is N.

(v) $q_0 \in Q$, is the initial/start state.

(vi) h ϵ Q is the 'Halt State', in which the machine stops any further activity.

In order to illustrate the ideas involved, let us consider the following simple examples.

Example 1

Consider the Turing Machine (Q,Σ , ⌈ , δ , q_o, h) defined below that erases all the non-blank symbols on the tape, where the sequence of non-blank symbols does not contain any blank symbol # in-between:

Q= {q_o, h} Σ= {a, b}, ⌈ = {a, b, #}
and the next-move function is defined by the following table:

q: State	**σ: Input Symbol**	**δ (q, σ)** Type equation here. **)**
q_0	A	{q_0, #, R}
q_0	B	{q_0, #, R}
q_0	#	{h, #, N}
H	#	ACCEPT

Next, we consider how to design a Turing Machine to accomplish some computational task through the following example. For this purpose, we need the definition.

A string Accepted by a TM

A string over is said to be accepted by a TM M = (Q, Σ, ⌈, δ, q_0, h) **if** when the string is placed in the left-most cells on the tape of M and TM is started in the initial state q_0 then after a finite number of moves of the TM as determined by δ, TM is in state h (and hence stops any further operations. Further, a string is said to be rejected if under the conditions mentioned above, the TM enters a state q $\neq$ h .

3.2.2 INSTANTANEOUS DESCRIPTION AND TRANSITION DIAGRAMS

Instantaneous Description

Some authors use the term *Instantaneous Description* instead of *Total Configuration.*

Initial Configuration: The total configuration at the start of the (Turing) Machine is called the initial configuration.

Halted Configuration: is a configuration whose state component is the Halt state

There are various notations used for denoting the total configuration of a Turing Machine.

Notation 1: *We use the notations, illustrated below through an example:*

Let the TM be in state q_3 scanning the symbol g with the symbols on the tape as follows:

#	#	b	D	a	f	#	G	h	K	#	#	#	#

Then one of the notations is

#	#	b	D	a	f	#	g	h	k	#	#	#	#
							↑						
							q_3						

Notation 2: However, the above being a two-dimensional notation, is sometimes inconvenient. Therefore the following linear notations are frequently used: **(q_3,##bdaf#,g,hk), in which third component of the above 4-component vector, contains the symbol being scanned by the tape head.**

Alternatively, the configuration is also denoted by (q_3,## bdaf#g<u>h</u>k), where the symbol under the tape head is underscored but two last commas are dropped.

It may be noted that the sequence of blanks after the last non-blank symbol, is not shown in the configuration. The notation may be alternatively written (q_3, w, g, u) where w is the string to the left and u the string to the right respectively of the symbol that is currently being scanned.

In case g is the left-most symbol then we use the empty string e instead of w. Similarly, if g is being currently scanned and there is no non-blank character to the right of g then we use e, the empty string instead of u.

Notation 3: The next notation neither uses parentheses nor commas. Here the state is written just to the left of the symbol currently being scanned by the tape Head. Thus the configuration (q_3, ##bdaf#, g, h, k) is denoted as # # bdaf#$q_3$ghk

Thus if the tape is like

g	w	#	……
↑			
q_5			

then we may denote the corresponding configuration as (q_5, e, g, u). And, if the tape is like

A	b	c	g	#	#	…
			↑			
			q_6			

Then the configuration is (q_6, abc, g, e) or (q_6, abcg) or alternatively as abcq_6g by the following notation.

3.2.3 Transition Diagrams

In some situations, **graphical** representation of the next-move (partial) function δ of a Turing Machine may give better idea of the behavior of a TM in comparison to the **tabular** representation of δ .

A **Transition Diagram** of the next-move functions δ of a TM is a graphical representation consisting of a finite number of nodes and (directed) labelled arcs between the nodes. Each node represents a state of the TM and a label on an arc from one state (say p) to a state (say q) represents the information about the required **input symbol say x** for the transition from p to q to take place **and the action** on the part of the control of the TM. The action part consists of (i) the symbol say y to be written in the current cell and (ii) the movement of the tape Head.

Then the label of an arc is generally written as x/(y, M) where M is L, R or N.

Example 3.4.2.1

Let M ={Q, Σ, ⌜ , δ , q_0, h}
Where Q = { q_0, q_1, q_2, h}
Σ = { 0, 1}
⌜ = {0, 1, #}
and δ be given by the following table.

	0	1	#
q_0	-	-	(q_2, #, R)
q_1	(q_2, 0, R)	(q_1, #, R)	(h, #, N)
q_2	(q_2, 0, L)	(q_1, 1, R)	(h, #, N)
H	-	-	-

Then, the above Turing Machine may be denoted by the Transition Diagram shown below, where we assume that q_0 is the initial state and h is a final state.

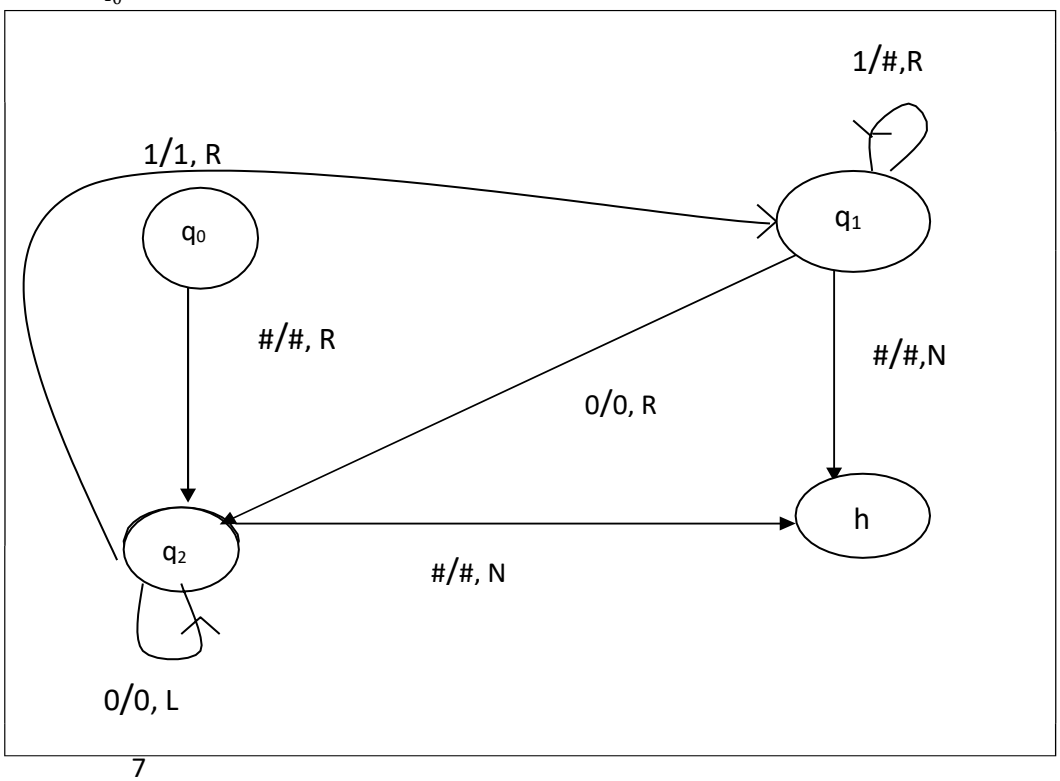


Fig. 3.2: Transition Diagram

3.2.4 SOME FORMAL DEFINITIONS

L(M), the language accepted by the TM M is the set of all finite strings over which are accepted by M.

Definition: Turing Acceptable Language

A language L over some alphabet is said to be Turing Acceptable Language, if there exists a Turing Machine M such that L = L (M)

Definition: Turing Decidable Language
A language over some alphabet is said to be Turing acceptable language if there exists a Turing Machine M such that L= L(M) and M halts on every input.
Remark 3.5.1

A very important fact in respect of Turing acceptability of a string (or a language) needs our attention. The fact has been discussed in very brief in a later section about undecidability. However, we briefly mention it below.

For a TM M and an input string $w \in \Sigma^*$, even after a large number of moves we may not reach the halt state. However, from this we can neither conclude that 'Halt state will be reached in a finite number of moves' nor can we conclude that Halt state will not be reached in a finite number moves.

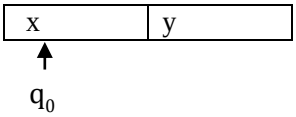
This raises the question of how to decide that an input string w is not accepted by a TM M.

An input string w is said to be *'not accepted'* by a TM M = $(Q,\Sigma,\Gamma,\delta,q_0,h)$ if any of the following three cases arise:

(i) There is a configuration of M for which there is no next move i.e., there may be a state and a symbol under the tape head, for which δ does not have a value.

(ii) The tape Head is scanning the left-most cell containing the symbol x and the state of M is say q and δ (x, q) suggests a move to the 'left' of the current cell. However, there is no cell to the left of the left-most cell. Therefore, move is not possible. The potentially resulting situation (can't say exactly configuration) is called **Hanging configuration**.

(iii) The TM on the given input w enters an infinite loop. For example if configuration is as



and we are given
$\delta(q_0, x) = (q_1, x, R)$
and $\delta(q_1, y) = (q_0, y, L)$
Then we are in an **infinite loop**.

3.2.5 OBSERVATIONS

The concept of TM is one of the most important concepts in the theory of Computation. In view of its significance, we discuss a number of issues in respect of TMs through the following remark:.

Turing Machine is not just another computational model, which may be further extended by another still more powerful computational model. It is not only the most powerful computational model known so far but also is conjectured to be the ultimate computational model. In this regard, we state below the

Turing Thesis: ***The power of any computational process is captured within the class of Turing Machines.***

It may be noted that Turing thesis is just a ***conjecture and not a theorem, hence, Turing Thesis*** can not be logically deduced from more elementary facts. However, the conjecture can be shown to be false, if a more powerful computational model is proposed that can recognize all the languages which are recognized by the TM model and also recognizes at least one more language that is not recognized by any TM.
In view of the unsuccessful efforts made in this direction since 1936, when Turing suggested his model, at least at present, it seems to be unlikely to have a more powerful computational model than TM Model.

Check Your Progress-1

Q1 What is the meaning of the following symbols:

(i) Q

(ii) Σ

(iii) Γ

(iv) δ

Q2 Explain what is a Turing Machine?

Q3 How TM is different from Finite Automata?

3.3 NONDETERMINISTRIC TURING MACHINE

Like nondeterministic finite automata, we can also imagine nondeterministic TM (NDTM). ***An* NDTM *is like the standard TM with the difference as described below***. In Standard TM, *to each* pair of the current state (except the halt state) and the symbol being scanned, *there is a* **unique** triplet comprising of *the next state*, *unique action* in terms of writing a symbol in the cell being scanned and *the motion*, if any, to the right or left. **However, in the case NDTM**, **to each pair** (q, s) with q as current state and s as symbol being scanned, *there may be a* **finite** *set of the triplets* $\{ (q_i, s_i, m_i) : I = 1,2,\ldots\ldots\}$ of possible next moves. *This set of triplets may be empty*, i.e. for some particular (q,s) the TM *may not have any next move*. Or alternatively the set $\{(q_i, s_i, m_i)\}$ may *have more than one* triplet, meaning thereby that the NDTM in the state q and scanning symbols s, has the *alternatives* for next move to choose from the set $\{(q_i, s_i, m_i)\}$ of next moves.

It can be easily seen that standard TM is a special case of the NDTM in which for each (q,s) the set {(q_i, s_i, }of next moves is a singleton set or empty.

In order to define formally the concept of Non-Deterministic TM (NDTM), and a configuration in NDTM etc, we assume *that the tape is one-way infinite*.

For the extensions of the standard TM, discussed so far, we did not state the full formal definition of each of the extension. We only discussed the definition only relative to the standard TM. Mainly we discussed configurations and partial move function δ for each of the extensions. However, in view of the significant though small, difference in the behaviour of an NDTMs, we provide below full formal definition of NDTM.

Remark 1:

An important point about the definition of NDTM needs to the highlighted. By the definition of δ which maps an element of (q, x) of Q x ⌈ to a set {(q_i, x_i, M_i) **} means that each element (q, x) of Q x ⌈ has the potential of leading to more than one configurations. In other words, there are various possible routes to a final configuration from one configuration. However, during one computation only one of these possible values (**q_i, x_i, M_i) will be associated with (q, x) through δ**. But we can not tell in advance which one out of the ordered triples from the set** {(q_i, x_i, M_i)**}**

This is why the adjective Non-Deterministic **is used for this version of the T.M.**

Remark 2:

The set **{(q_i, x_i, M_i)}** associated with (q, x) under δ, may be empty. This means there is no possible next move for (q, x), a situation that occurred even in the case of standard TM and other versions discussed so far. This is why δ was called a **partial** function from Q x ⌈ to Q x ⌈x**{*L,R,N*).**

Remark 3:

In the standard TM and the versions discussed before NDTM, we allowed δ as a partial function to Q x ⌈ x {L, R, N}. In other words, if a value under δ exists for (q, x) then the value has to be unique, i.e, can be determined. Therefore, the earlier versions are prefixed with the adjective *Deterministic*. The Non- Deterministic form of each of the earlier versions can be obtained by making suitable modifications in the corresponding definitions of δ etc on the lines of modifications suggested in the definition of NDTM from standard TM.

Remark 4:

Proper non-determinism means that at some stage, there are at least two next possible moves. Now, if we are engage two different persons or machines to work out further possible moves according to each of these two moves, the two can work **independent** of each other. **This means Non-Determination allows parallel computations**. This characteristic of Non-**Determinism,** also allows is further computations even if some of the sequences of moves may be locked as there may not be any next moves at some stages.

Definition: An Non-Deterministic Turing Machine
is a sextuple (Q, Σ, Γ, δ, q_o, h) where
Q: Set of States

Σ: Set of input symbols
Γ: Set of tape symbols
q_o: The initial state
h: The halt state and

δ: Q x Γ → **Power set of** (Q x Γ x {L, R, N})

The concept of a configuration *is same as in the case of standard TM. But the concept of 'yields in one step' denoted by* $\vdash_m$ *, has different meaning.* ***Here one configuration may yield more than one configurations.***

We explain these ideas through a suitable example, which also demonstrates the ***advantage of the Non- Deterministic Turing Machine*** *over the standard Turing Machine.* **The advantage is in respect of the relative ease of construction of NDTM**.

Remarks 5

Before coming to the example, showing advantage of an NDTM in solving some problems; we need to understand properly the concept of acceptance of a language by an NDTM. *First of all, let us recall below what is meant by acceptance of a language L by a standard TM M.*

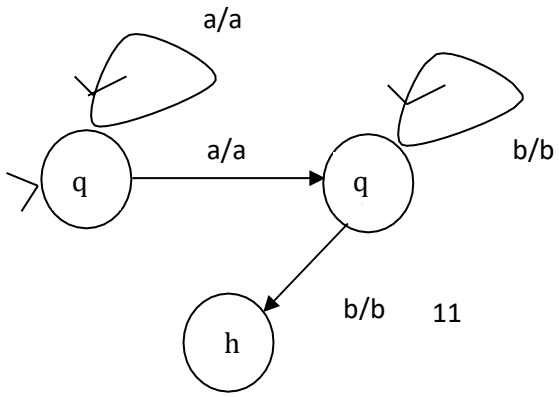
A language L is accepted by a **TM M** ***if each string*** $\alpha \in$ L, is acceptable by M.
Further a string α is acceptable M, if staring in the initial state q_0 of M, with α as input on the tape of M, if we are able to reach halt state in a finite number of moves,
A characteristic feature of the **standard TM**, in this case, is that if there is to be a sequence of moves from (q_0, α) to a final state, than that sequence might the unique. However **in the case of Non-Deterministic machines**, the halt state may be reached through any one of various permissible sequences of moves. Therefore in this version a string α over the set of input symbols of an NDTM is acceptable by an NDTM M, if **by at least one but by any one** of the sequences of moves halt state is reached from (q_0, α). Now we discuss the example showing advantage of NDTM over standard TM.

Example 1:

Construct an NDTM which accepts the language { $a^n b^m$: n≥1, m ≥1}, i.e., the language of all strings over {a,b}, in which there is at least one a and one b and all a's precede all b's.

Solution: The diagrammatic representation of the required NDTM is as given below:

In the proposed NDTM, as the motion of the head is always to the Right except in the Halt state. Therefore, R is not mentioned in the labels in the diagram below:



where the label i/j on an arc denotes that if symbol in the current cell is i then contents of the cell are to be replaced by j.

Formally the proposed NDTM may be defined as

M={ {q_0, q_1, h}, {a, b}, { a, b, #}, δ, q_0, h }

Where δ is defined as follows:

δ (q_0, a)= {(q_0, a, R), (q_1, a, R)}

δ (q_0, b)= empty

δ (q_1, a)= empty

δ (q_1, b)= {(q_1, b, R), (h, b, N)}

If the machine has no next move, then it halts without accepting the string.

Remarks 6:

Though we have already mentioned earlier on a number occasions, yet, in view of the significance of non-determinism in designing TMs comparatively *more easily*, we again bring to notice that in the state q_0 on scanning symbol a, the TM may move in any one of the two next possible states viz to q_0 after moving the head to the right or to q_1 (after moving the head to the right). And, if the TM is implemented as a parallel computer then the computer can presume independently both branches initiated by (q_0,a,R) and (q_1,a,R)

3.4 UNDECIDABLE AND HALTING PROBLEMS

UNDECIDABLE PROBLEM

A function g with domain D is said to be computable if there exists some Turing machine whose whose initial configuration is q0 w, i.e., the left-most symbol of the string w being scanned in state q0 and whose final configuration is q f g(w) for some final state qf , i.e., when the machine halts the only non-blank cells on the tape represent the function value g(w).

A function is said to be un-computable if no such machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain.

For some problems, we are interested in simpler solution in terms of "yes" or "no". We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

A class of problems with two outputs "yes" or "no" is said to be decidable (solvable) if there exists some definite algorithm which always terminates (halts) with one of two outputs "yes" or "no". Otherwise, the class of problems is said to be undecidable (unsolvable).

THE HALTING PROBLEM

There are many problems which are not computable. But, we start with a problem which is important and that at the same time gives us a platform for developing later results. One such problem is the halting problem. Algorithms may contain loops that may be infinite or finite in length. The amount of work done in an algorithm usually depends on the data input. Algorithms may consist of various numbers of loops, nested or in sequence. Informally, the **Halting problem** can be put as:

Given a Turing machine M and an input *w* to the machine M, determine if the machine M will eventually halt when it is given input *w*.

Trial solution: Just run the machine M with the given input *w*.

- If the machine M halts, we know the machine halts.
- But if the machine doesn't halt in a reasonable amount of time, we cannot conclude that it won't halt. Maybe we didn't wait long enough.

What we need is an algorithm that can determine the correct answer for any M and *w* by performing some analysis on the machine's description and the input. But, it is shown by Alan Turing that no such algorithm exists.

3.5 COMPLEXITY

In the previous unit, we introduced you to the fact that there are a large number of problems which cannot be solved by algorithmic means and discussed a number of issues about such problems.

The advantage of such a study is our becoming aware of the fact that in stead of attempting to write an algorithm for every problem that we are required to solve using a computer, we should first study the essential nature of the problem. In case the problem under consideration is not solvable by algorithmic means, we may adopt other computational techniques including use of heuristics, numerical and/or statistical techniques. Even out of problems, which though theoretically have algorithmic solutions, yet require such large amount of resources, that this type of problems are designated as *infeasible* for the purpose of computational solution. Out of the problems, which are *feasibly* solvable, there are problems each of which may have more than one algorithms to solve the problem. For us, it is desirable to know which one is better among the available ones. For example, we can use the algorithms viz, Bubble sort, Insertion sort, Heapsort and Quicksort, for sorting a list of numbers. Their designs are different but the outcome is the same for all, for a given list of numbers. As, there are more than one algorithms available to us to sort a list of numbers, it is natural for us to think of using the algorithm which solves a particular sorting problem, in some way better than the others. In context of practical disciplines like computer applications, an *efficient* solution is generally taken as a *better* solution. Efficiency of an algorithm can be considered in terms of the efficient use of computer resources, such as processor time and memory space used. In addition to the efficiency of execution of algorithms, other factors like time (taken by a team of software engineers and/or programmers) *required for developing algorithms* and *reliability* may also be taken into consideration as factors towards overall efficiency of an algorithm.

However, most of the time, in respect of efficiency of algorithms, we are only concerned with the time and space requirements of execution of algorithms.

In this unit, we will discuss the issue of efficiency of computation of an algorithm in terms of the *amount of time* used in its execution. On the basis of analysis of an algorithm, the amount of time that is estimated to be required in executing an algorithm, will be referred to as the **time complexity** of the algorithm. The time complexity of an algorithm is measured in terms of some (basic) **time unit** (not second or nano-second). Generally, time taken in executing one move of a TM, is taken as (basic) time unit for the purpose. or, alternatively, time taken in executing some elementary operation like addition, is taken as one unit. More complex operations like multiplication etc, are assumed to require an integral number of basic units. As mentioned earlier, given many algorithms (solutions) for solving a problem, we would like to choose the most efficient algorithm from amongst the available ones. For comparing efficiencies of algorithms, that solve a particular problem, time complexities of algorithms are considered as functions of the sizes of the problems (to be discussed). The *time complexity functions of the algorithms are compared in terms of their growth rates (to be defined)* as growth rates are considered important measures of *comparative efficiencies*.

The concept of the size of a problem, though a fundamental one, yet is difficult to define precisely. Generally, the size of a problem, is measured in terms of the size of the *input*. The concept of the size of an input of a problem may be explained informally through examples. In the case of multiplication of two nxn (squares) matrices, the size of the problem may be taken as n^2, i.e, the number of elements in each matrix to be multiplied. For problems involving polynomials, the degrees of the polynomials may be taken as measure of the sizes of the problems.

Also, we may have an intuitive idea about the term **growth rate** *and its significance* in the comparative study of algorithms that can be designed to solve problems. For the time being, in stead of attempting a formal definition, we illustrate the concept of *growth rate of time complexity function* of an algorithm and its significance through the following example.

Let us consider two algorithms to solve a problem P, having time-complexities respectively as $f_1(n) = 1000n^2$ and $f_2(n) = 5n^4$, where size of the problem is assumed to be n. Then

$f_1(n) \geq f_2(n)$ for $n \leq 14$ and

$f_1(n) \leq f_2(n)$ for $n \geq 15$.

Also, the increase in the ratio $(f_2(n)/f_1(n))$ is faster than increase in n. Thus, informally, growth rate of $f_2(n)$ is more than the growth rate of $f_1(n)$. In one sense, the algorithm having time complexity $f_2(n)$ is *inferior* to the algorithm having time complexity $f_1(n)$ as growth rate of $f_2(n)$ is faster than that of $f_1(n)$.

For a problem, a solution with time complexity which can be expressed as a polynomial of the size of the problem, is considered to have an **efficient solution**. Unfortunately, not many problems that arise in practice, admit any efficient algorithms, as these problems can be solved, if at all, by only non-polynomial time algorithms. A problem which does not have any (known) polynomial time algorithm is called an **intractable** problem.

At this stage, it is important to be aware of the following relevant facts

(i) **A non-polynomial function need not always be exponential:**. For example, the function $f(n) = n\log_2 n$ is neither polynomial function nor exponential function of n, but, somewhere between the two but n^log_2(n) is a polynomial function

(ii) **The term *solution* in its general form: need not be an algorithm.** If by tossing a coin, we get the correct answer to each instance of a problem, then the process of tossing the coin and getting answers constitutes a solution. But, the process is not an algorithm. Similarly, we solve problems based on **heuristics**, i.e, good guesses which, generally but not necessarily always, lead to solutions. All such cases of solutions are not algorithms, or algorithmic solutions. To be more explicit, by an algorithmic solution A of a problem L (*considered as a language*) from a problem domain Σ^*, we mean that among other conditions, the following are satisfied:

(a) A is a step-by-step method in which for each instance of the problem, there is a definite sequence of execution steps (*not involving any guesswork*).

(b) A *terminates* for each $x \in \Sigma^*$, irrespective of whether $x \in L$ or $x \notin L$.

In this sense of algorithmic solution, only a **solution by a Deterministic TM** is called an **algorithm**. A solution by a **Non-Deterministic TM may not be an algorithm.**

(iii) However, for every NTM solution, there is a Deterministic TM (DTM) solution of a problem. Therefore, if there is an NTM solution of a problem, then there is an algorithmic solution of the problem. However, *the symmetry may end here*.

The *computational equivalence* of Deterministic and Non-Deterministic TMs does not state or guarantee any *equivalence in respect of requirement of resources* like time and space by the Deterministic and Non-Deterministic models of TM, for solving a (solvable) problem. To be more precise, if a problem is solvable in polynomial-time by a Non-Deterministic Turing Machine, then it is, of course, *guaranteed* that there is a deterministic TM that solves the problem, but *it is not guaranteed* that there exists a Deterministic TM that solves the problem *in polynomial time*. Rather, **this fact forms the basis for one of the deepest open questions of Mathematics, which is stated as 'whether P = NP?'**(P and NP to be defined soon).

The question put in simpler language means: Is it possible to design a Deterministic TM to solve a problem in polynomial time, for which, a Non-Deterministic TM that solves the problem in polynomial time, has already been designed?

We summarize the above discussion from the intractable problem's definition onward. Let us begin with definitions of the notions of P and NP.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

We can define p and NP to be the classes of languages because problems from graph theory, combinatorics can often be formulated as language recognition problems. Consider a problem which requires an answer in form of " Yes" or "No" for each instance. Each instance of a problem can be encoded as a string and reformulate the problem as one of recognizing the language consisting of all the strings representing those instance of the problem whose answer is "Yes".

With this logic P and NP classes of complexities can be defined as classes of languages:

Definition

P = { L| L can be decided or accepted by a DTM(Deterministic TM) in polynomial time}

NP = { L| L can be decided or accepted by a Nondeterministic TM in polynomial time}

NP is a set of decision problems (with Yes or No answer) that can be solved by NDTM in polynomial time

Thus starting with two distinct classes of problems, viz, **tractable** problems and **intractable** problems, we introduced two classes of problems called **P and NP**. Some interesting relations known about these classes are:

(i) P = set of tractable problems

(ii) P⊆ NP.

(The relation (ii) above simply follows from the fact that every Deterministic TM is a special case of a Non-Deterministic TM).

Check your Progress-2

Q1 What is Nondeterministic Turing Machine?

Q2 Define P and NP Complexity Classes.

3.6 SUMMARY

In this unit, after giving the informal idea of what a Turing machine is, the concept is formally defined and illustrated through an example. A Nondeterministic TM was introduced next and also explained how it is different from a standard TM. Besides TM and NDTM, P and NP classes of complexities were defined.

3.7 Solutions/Answers

Check Your Progress-1

Q1 What is the meaning of the following symbols:

(i) Q

(ii) Σ

(iii) ⌈

(iv) δ

Ans1. (i) Q is the finite set of states,

(ii) Σ is the finite set of non-blank information symbols,

(iii) ⌈ is the set of tape symbols, including the blank symbol #

(iv) δ is the **next-move** partial function from Q x ⌈to Q x ⌈x {L, R, N},

where '**L**' denotes the tape Head moves to the left adjacent cell, '**R**' denotes tape Head moves to the Right adjacent cell and '**N**' denotes Head does not move, **i.e**., continues scanning the same cell.

Q2 Explain what is a Turing Machine?

Ans. 2: TM defines an abstract machine/mathematical model that consists of an infinite length tape divided into cells for storing input symbols and a head which scans and reads the input. If the TM reaches the final state, the input string is accepted, otherwise it is rejected. TMs are considered to be the most important formal models in the study of Computer Science.

Q3 How TM is different from Finite Automata?

Ans.3 The *Finite Automata* is used only as **accepting devices** for languages in the sense that the automata, when given an input string from a language, tells whether the string is acceptable or not. ***The Turing Machines are designed to play at least the following three different roles:***

(i) **As accepting devices for languages**, similar to the role played by FAs .

(ii) **As a computer of functions**. In this role, a TM represents a particular function (say the SQUARE function which gives as output the square of the integer given as input). Initial input is treated as representing an argument of the function. And the (final) string on the tape when the TM enters the Halt State is treated as

representative of the value obtained by an application of the function to the argument represented by the initial string.

(iii) **As an enumerator of strings of a language** that outputs the strings of a language, one at a time, in some systematic order, i.e, as a list.

Check your Progress-2

Q1 What is Nondeterministic Turing Machine?

Ans.1: In Standard TM, *to each* pair of the current state (except the halt state) and the symbol being scanned, *there is a* **unique** triplet comprising of *the next state*, *unique action* in terms of writing a symbol in the cell being scanned and *the motion*, if any, to the right or left. **However, in the case NDTM**, **to each pair** (q, s) with q as current state and s as symbol being scanned, *there may be a* **finite** *set of the triplets* { (q_i, s_i, m_i) : I =1,2,…….} of possible next moves.

Q2 Define P and NP Complexity Classes.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.
NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

We can define p and NP to be the classes of languages because problems from graph theory, combinatorics can often be formulated as language recognition problems. Consider a problem which requires an answer in form of " Yes" or "No" for each instance. Each instance of a problem can be encoded as a string and reformulate the problem as one of recognizing the language consisting of all the strings representing those instance of the problem whose answer is "Yes".

With this logic P and NP classes of complexities can be defined as classes of languages:

Definition

P = { L| L can be decided or accepted by a DTM(Deterministic TM) in polynomial time}

NP = { L| L can be decided or accepted by a Nondeterministic TM in polynomial time}

NP is a set of decision problems (with Yes or No answer) that can be solved by NDTM in polynomial time

3.7 SOLUTIONS AND ANSWERS