1. INTRODUCTION

The three traditionally central areas of the theory of computation:
automata, computability, and complexity.
They are linked by the question:

What are the fundamental capabilities and limitations of computers?

In each of the three areas—automata, computability, and complexity—this question is interpreted differently, and the answers vary according to the interpretation.

1.1 COMPLEXITY THEORY

Computer problems come in different varieties; some are easy, and some are hard. For example, the sorting problem is an easy one. Say that you need to arrange a list of numbers in ascending order. Even a small computer can sort a million numbers rather quickly. Compare that to a scheduling problem. Say that you must find a schedule of classes for the entire university to satisfy some reasonable constraints, such as that no two classes take place in the same room at the same time. The scheduling problem seems to be much harder than the sorting problem. If you have just a thousand classes, finding the best schedule may require centuries, even with a supercomputer.

What makes some problems computationally hard and others easy?

This is the central question of complexity theory. Remarkably, we don't know the answer to it, though it has been intensively researched for over 40 years. Later, we explore this fascinating question and some of its ramifications.

In one important achievement of complexity theory thus far, researchers have discovered an elegant scheme for classifying problems according to their computational difficulty. It is analogous to the periodic table for classifying elements according to their chemical properties. Using this scheme, we can demonstrate a method for giving evidence that certain problems are computationally hard, even if we are unable to prove that they are.

You have several options when you confront a problem that appears to be computationally hard. First, by understanding which aspect of the problem is at the root of the difficulty, you may be able to alter it so that the problem is more easily solvable. Second, you may be able to settle for less than a perfect solution to the problem. In certain cases, finding solutions that only approximate the perfect one is relatively easy. Third, some problems are hard only in the worst-case situation, but easy most of the time. Depending on the application, you may be satisfied with a procedure that occasionally is slow but usually runs quickly. Finally, you may consider alternative types of computation, such as randomized computation, that can speed up certain tasks.

One applied area that has been affected directly by complexity theory is the ancient field of cryptography. In most fields, an easy computational problem is preferable to a hard one because easy ones are cheaper to solve. Cryptography is unusual because it specifically requires computational problems that are hard, rather than easy. Secret codes should be hard to break without the secret key or password. Complexity theory has pointed cryptographers in the direction of computationally hard problems around which they have designed revolutionary new codes.

1.2 COMPUTABILITY THEORY

During the first half of the twentieth century, mathematicians such as Kurt G¨odel, Alan Turing, and Alonzo Church discovered that certain basic problems cannot be solved by computers. One example of this phenomenon is the problem of determining whether a mathematical statement is true or false. This task is the bread and butter of mathematicians. It seems like a natural for solution by computer because it lies strictly within the realm of mathematics. But no computer algorithm can perform this task.

Among the consequences of this profound result was the development of ideas concerning theoretical models of computers that eventually would help lead to the construction of actual computers.

The theories of computability and complexity are closely related. In complexity theory, the objective is to classify problems as easy ones and hard ones; whereas in computability theory, the classification of problems is by those that are solvable and those that are not. Computability theory introduces several of the concepts used in complexity theory.

1.3 AUTOMATA THEORY

Automata theory deals with the definitions and properties of mathematical models of computation. These models play a role in several applied areas of computer science. One model, called the *finite automaton*, is used in text processing, compilers, and hardware design. Another model, called the *context-free grammar*, is used in programming languages and artificial intelligence.

Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a *computer*. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other non theoretical areas of computer science.

Automata theory is the study of abstract computing devices, or "machines." Before there were computers, in the 1930's, A. Turing studied an abstract machine that had all the capabilities of today's computers, at least as far as in what they could compute. Turing's goal was to describe precisely the boundary between what a computing machine could do and what it could not do; his conclusions apply not only to his abstract Turing machines, but to today's real machines. In the 1940's and 1950? simpler kinds of machines, which we today call “finite automata," were studied by a number of researchers. These automata, originally proposed to model brain function, turned out to be extremely useful for a variety of other purposes, (as told in applications of FA). Also, in the late 1950? the linguist N. Chomsky began the study of formal "grammars." While not strictly machines, these grammars have close relationships to abstract automata and serve today as the basis of some important software components, including parts of compilers.

In 1969, S. Cook extended Turing's study of what could and what could not be computed. Cook was able to separate those problems that can be solved efficiently by computer from those problems that can in principle be solved, but in practice take so much time that computers are useless for all but very small instances of the problem. The latter class of problems is called "intractable," or "NP-hard." It is highly unlikely that even the exponential improvement in computing speed that computer hardware has been following ("Moore’s Law") will have significant impact on our ability to solve large instances of intractable problems.

All of these theoretical developments bear directly on what computer scientists do today. Some of the concepts, like finite automata and certain kinds of formal grammars, are used in the design and construction of important kinds of software. Other concepts, like the Turing machine,

help us understand what we can expect from our software. Especially, the theory of intractable problems lets us deduce whether we are likely to be able to meet a problem "head-on" and write a program to solve it (because it is not in the intractable class), or whether we have to find some way to work around the intractable problem: find an approximation, use a heuristic, or use some other method to limit the amount of time the program will spend solving the problem.

Theory of Computation

Logical or Mathematical model of computing machines and their capabilities is TOC.

Formal Languages

The collection of the strings from the alphabet Σ where there can exist some restrictions or conditions in the formation of strings is called formal language.

Ex: $\Sigma = \{0,1\}$, then $L_1 = \{0^m1^n/m=n\}$

$L_2 = \{w/w = 0 \text{ to } *\}$

Formal languages are classified into 4 types

a) Regular Language (RL)
b) Context-Free Language (CFL)
c) Context -Sensitive Language (CSL)
d) Recursive Enumerable Language (REL)

Note:

- In formal languages, we give importance to only formation of strings based on the conditions rather than English meaning of the string.
- If $\Sigma = \{0,1\}$, -> called alphabet, then

  $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. -> set of all strings of all lengths.

  2^{Σ^*} = set of all formal languages

Grammar

The set of all the production rules which are used in the generation of a string is called grammar.

Grammar is a generating device.

Grammar is defined as 4-tuple G (V, T, P, S), where,

V is set of all variables, generally denoted by capital letters.

T is set of all terminals, generally denoted by small case letters.

P is set of all productions.

S is the start symbol of the Grammar.

Ex: Let Grammar G be,

S -> AB

A -> a

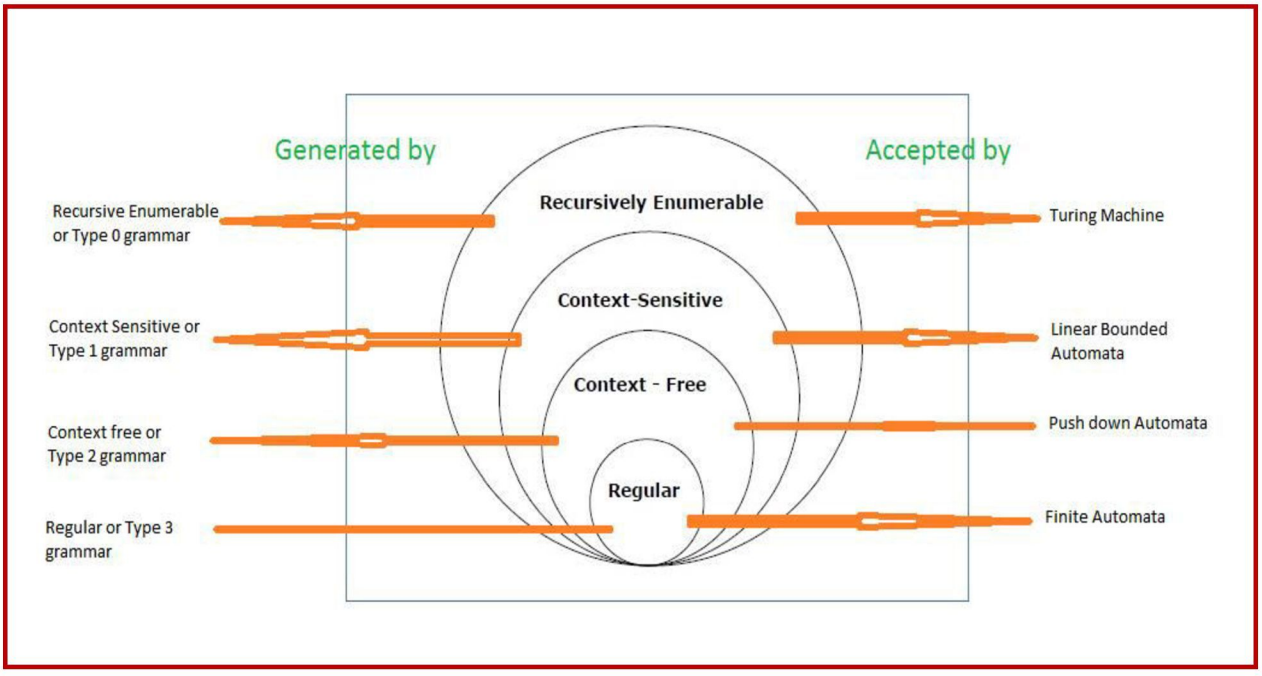
B -> b

Then L(G) = {ab}

Based on the form of productions, grammar is classifiedinto four types.

a) Regular Grammar or Type 3 Grammar
b) Context-Free Grammar or Type 2 Grammar
c) Context -Sensitive Grammar or Type 1 Grammar
d) Recursive Enumerable Grammar or Type 0 Grammar

The Chomsky hierarchy



Automaton

The logical or mathematical model of formal language is called as Automaton.

An automaton (plural of automaton) is a recognizing or accepting device.

Types of Automata:

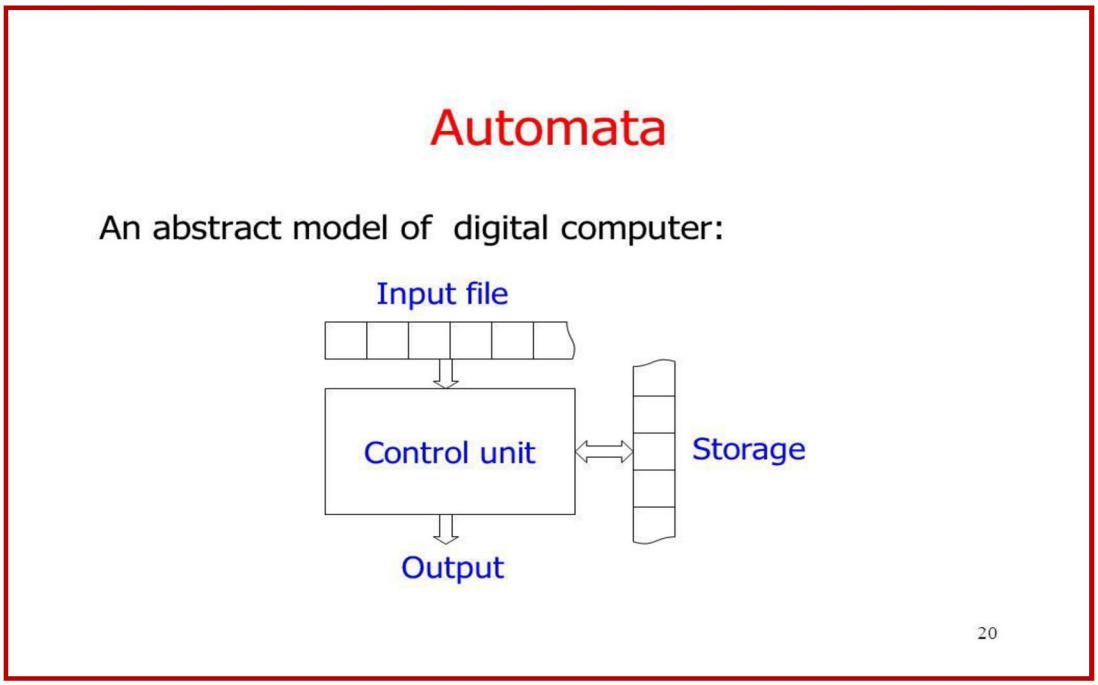
a) Finite Automata ---- accepts RL
b) Push down Automata ---- accepts CFL
c) Linear Bounded Automata ---- accepts CSL
d) Turing Machine ----- accepts REL

An automaton is an abstract model of a digital computer. As such, every automaton includes some essential features. It has a mechanism for reading input. It will be assumed that the input is a string over a given alphabet, written on an **input file**, which the automaton can read but not change. The input file is divided into cells, each of which can hold one symbol. The input mechanism can read the input file from left to right, one symbol at a time. The input mechanism can also detect the end of the input string (by sensing an end-of-file condition). The automaton can produce output of some form. It may have a temporary **storage** device, consisting of an unlimited number of cells, each capable of holding a single symbol from an alphabet (not necessarily the same one as the input alphabet). The automaton can read and change the contents of the storage cells. Finally, the automaton has a **control unit**, which can be in any one of a finite number of **internal states**, and which can change state in some defined manner. Figure below shows a schematic representation of a general automaton.

An automaton is assumed to operate in a discrete timeframe. At any given time, the control unit is in some internal state, and the input mechanism is scanning a particular symbol on the input file. The internal state of the control unit at the next time step is determined by the **next-state** or **transition function**. This transition function gives the next state in terms of the current state, the current input symbol, and the information currently in the temporary storage. During the transition from one time interval to the next, output may be produced or the information in the temporary storage changed. The

term **configuration** will be used to refer to a particular state of the control unit, input file, and temporary storage. The transition of the automaton from one configuration to the next will be called a **move**.

This general model covers all the automata discussed above. A finite-state control will be common to all specific cases, but differences will arise from the way in which the output can be produced and the nature of the temporary storage. As we will see, the nature of the temporary storage governs the power of different types of automata.



Expressive power of Automata

Number of languages accepted by automata is called Expressive power or Accepting power of Automata.

E (FA) = 1

E (PDA) = 2

E (LBA) = 3

E(TM) = 4

2. MATHEMATICAL NOTIONS AND TERMINOLOGY

As in any mathematical subject, we begin with a discussion of the basic mathematical objects, tools, and notation that we expect to use.

2.1. SETS

A ***set*** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even other sets. The objects in a set are called its ***elements*** or ***members***.

Sets may be described formally in several ways.

One way is by listing a set's elements inside braces. Thus, the set S = {7, 21, 57} contains the elements 7, 21, and 57. The symbols ∈ and ∉ denote set membership and non-membership. We write 7 ∈ {7, 21, 57} and 8 ∉ {7, 21, 57}.

Subset: For two sets A and B, we say that A is a ***subset*** of B, written A ⊆ B, if every member of A also is a member of B.

proper subset: set A is a ***proper subset*** of B, written A ⊊ B, if A is a subset of B and not equal to B.

multiset: A set where the elements can occur more than once is multiset. Multiset is not a set.

Ex 1: {1,2,3,4,4}

An ***infinite set*** contains infinitely many elements. We cannot write a list of all the elements of an infinite set, so we sometimes use the ". . ." notation to mean "continue the sequence forever." Thus we write the set of ***natural numbers*** N as {1, 2, 3, . . . }. The set of ***integers*** Z is written as { . . . ,−2,−1, 0, 1, 2, . . .}.

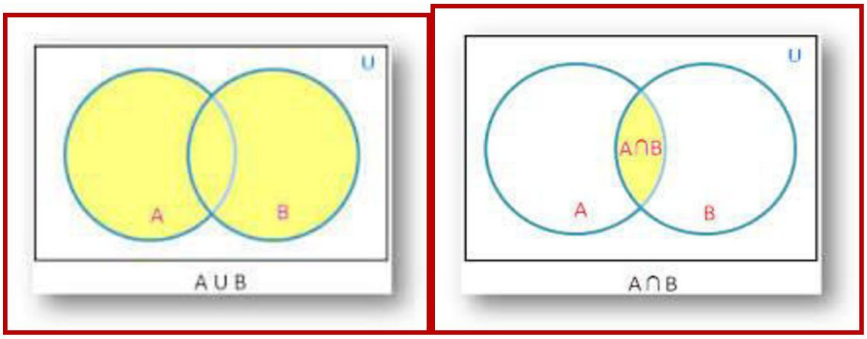
The set with zero members is called the ***empty set*** and is written ∅. A set with one member is sometimes called a ***singleton set***, and a set with two members is called an ***unordered pair***.

When we want to describe a set containing elements according to some rule, we write {n| rule about n}. Thus {n| n = m2 for some m ∈ N} means the set of perfect squares.

If we have two sets A and B, the ***union*** of A and B, written **A∪B**, is the set we get by combining all the elements in A and B into a single set. The ***intersection*** of A and B, written **A** ∩

B, is the set of elements that are in both A and B. The ***complement*** of A, written A, is the set of all elements under consideration that are *not* in A.

As is often the case in mathematics, a picture helps clarify a concept. For sets, we use a type of picture called a ***Venn diagram***. It represents sets as regions enclosed by circular lines. The next two Venn diagrams depict the union and intersection of sets A and B.



2.2. SEQUENCES AND TUPLES

A ***sequence*** of objects is a list of these objects in some order. We usually designate a sequence by writing the list within parentheses. For example, the sequence 7,21, 57 would be written (7, 21, 57).

The order doesn't matter in a set, but in a sequence it does. Hence (7, 21, 57) is not the same as (57, 7, 21). Similarly, repetition does matter in a sequence, but it doesn't matter in a set. Thus (7, 7, 21, 57) is different from both of the other sequences, whereas the set {7, 21, 57} is identical to the set {7, 7, 21, 57}.

As with sets, sequences may be finite or infinite. Finite sequences often are called ***tuples***. A sequence with k elements is a k-***tuple***. Thus (7, 21, 57) is a 3-tuple. A 2-tuple is also called an ***ordered pair***.

Sets and sequences may appear as elements of other sets and sequences. For example, the ***power set*** of A is the set of all subsets of A. If A is the set {0, 1}, the power set of A is the set {∅, {0}, {1}, {0, 1}}. The set of all ordered pairs whose elements are 0s and 1s is {(0, 0), (0, 1), (1, 0), (1, 1)}.

If A and B are two sets, the ***Cartesian product*** or ***cross product*** of A and B, written A × B, is the set of all ordered pairs wherein the first element is a member of A and the second element is a member of B.

Ex 2: If A = {1, 2} and B = {x, y, z}, then A × B = {(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)}.

We can also take the Cartesian product of k sets, A1, A2, . . ., Ak, written A1 × A2 ×· · ·×Ak. It is the set consisting of all k-tuples (a1, a2, . . ., ak) where ai ∈ Ai.

Ex 3: If A and B are as in Example 2,

A × B × A = {(1, x, 1), (1, x, 2), (1, y, 1), (1, y, 2), (1, z, 1), (1, z, 2), (2, x, 1), (2, x, 2), (2, y, 1),

(2, y, 2), (2, z, 1), (2, z, 2)}.

Ex 4: The set N^2 equals N ×N. It consists of all ordered pairs of natural numbers. We also may write

it as {(i, j)| i, j ≥ 1}.

2.3. FUNCTIONS AND RELATIONS

Functions are central to mathematics. A ***function*** is an object that sets up an input–output relationship. A function takes an input and produces an output. In every function, the same input always produces the same output. If f is a function whose output value is b when the input value is a, we write f(a) = b.

A **function** is a rule that assigns to elements of one set a unique element of another set. If *f* denotes a function, then the first set is called the **domain** of *f*, and the second set is its **range**. We write

*f: S*1 → *S*2 to indicate that the domain of *f* is a subset of *S*1 and that the range of *f* is a subset of *S*2. If the domain of *f* is all of *S*1, we say that *f* is a **total function** on *S*1; otherwise, *f* is said to be a **partial function**.

A function also is called a ***mapping***, and, if f(a) = b, we say that f maps a to b. For example, the absolute value function abs take a number x as input and returns x if x is positive and −x if x is negative. Thus abs (2) = abs (−2) = 2. Addition is another example of a function, written add. The input to the

addition function is an ordered pair of numbers, and the output is the sum of those numbers.

The set of possible inputs to the function is called its ***domain***. The outputs of a function come from a set called its ***range***. The notation for saying that f is a function with domain D and range R is f: D−>R.

In the case of the function abs, if we are working with integers, the domain and the range are Z, so we write abs: Z−>Z. In the case of the addition function for integers, the domain is the set of pairs of integers Z ×Z and the range is Z, so we write add: Z ×Z−>Z. Note that a function may not necessarily use all the elements of the specified range. The function abs never take on the value −1 even though −1 ∈ Z. A function that does use all the elements of the range is said to be ***onto*** the range.

We may describe a specific function in several ways. One way is with a procedure for computing an output from a specified input. Another way is with a table that lists all possible inputs and gives the output for each input.

3. THE CENTRAL CONCEPTS OF AUTOMATA THEORY

In this section we shall introduce the most important definitions of terms that pervade the theory of automata. These concepts include the "alphabet" (a set of symbols), "strings" (a list of symbols from an alphabet, and "language" (a set of strings from the same alphabet).

3.1. Alphabets

An alphabet is a finite, nonempty set of symbols. Conventionally, we use the symbol '**Σ**' for an alphabet. Common alphabets include:

1. Σ = {O, 1}, the binary alphabet.
2. Σ = {a, b…, z}, the set of all lower-case letters.
3. The set of all ASCII characters, or the set of all printable ASCII characters.
4. Σ = {a-z,0-9}
5. Σ = {0,1, 2….9}
6. Σ = {+, *, -, %}

3.2. Strings

A string (or sometimes a *word*) is a finite sequence of symbols chosen from some alphabet.

Conventionally, we use the symbol '**w**' for a string.

For example, 01101 is a string from the binary alphabet Σ = {O, 1}. The string 111 is another string chosen from this alphabet.

3.2.1. Length of a String

It is often useful to classify strings by their length, that is, the number of positions for symbols in the string. For instance, 01101 have length 5. It is common to say that the length of a string is "the number of symbols" in the string; this statement is colloquially accepted but not strictly correct. Thus, there are only two symbols, 0 and 1, in the string 01101, but there are five positions for symbols, and its length is 5. However, you should generally expect that "the number of symbols" can be used when "number of positions" is meant. The standard notation for the length of a string *w* is '|w|'.

For example, Let Σ = {0, 1} & w=10111, then |w| = 5 and I∈l = 0

3.2.2. Empty String

The empty string is the string with zero occurrences of symbols. This string, denoted by ∈, is a string that may be chosen from any alphabet whatsoever. A string of length *zero* is called Empty String.

For, w = ∈, |w| = 0.

3.2.3. Reverse of a String

If w has length n, we can write w = w1w2 · · ·wn where each wi ∈ Σ. The ***reverse*** of w, written w^R, is the string obtained by writing w in the opposite order (i.e., wnwn−1 · · ·w1).

3.2.4. Concatenation of strings

The **concatenation** of two strings *w* and *v* is the string obtained by appending the symbols of *v* to the right end of *w*, that is, if w = $a_1a_2a_3a_4\ldots a_n$ and v = $b1b2b_3b_4\ldots b_n$ then the concatenation of *w* and *v*, denoted by *wv*, is wv = $a_1a_2a_3a_4\ldots a_n$ $b1b2b_3b_4\ldots b_n$.

3.2.5. Substring

Let z, w be two strings over the alphabet Σ, then *z* is a ***substring*** of *w* if *z* appears consecutively within *w*.

Here |z| <= |w|.

For example, cad is a substring of abracadabra.

1. If w = CSE then substrings of w are

   Length zero substrings = {∈}

   Length one substrings = {C, S, E}

   Length two substrings = {CS, SE}

   Length three substrings = {CSE}

   Total no. of substrings = 1+3+2+1=7

Note:

- Every string is a substring of itself.
- Empty string∈ is a substring of every string.

- Substrings are of two types

  a) Trivial substring

  If *w* is any string, then *w* itself and empty string ∈ are called Trivial Substrings.

  Every string will have 2 Trivial Substrings.

  b) Non-trivial substring

  If *w* is any string, then any substring of *w* other than *w* itself and empty string ∈ are called Non-Trivial Substrings or Proper substrings.

- If *w* is any string with 'n' distinct symbols and |w| = n, then the no. of substrings = Σn + 1 & No. of non-trivial substrings = Σn – 1.

3.2.6. Prefix of a string

Say that string x is a ***prefix*** of string y if a string z exists where xz = y, and that x is a ***proper prefix*** of y if in addition x ≠ y. A ***language*** is a set of strings. A language is ***prefix-free*** if no member is a proper prefix of another member.

The sequence of starting or leading symbols of string *w* over alphabet Σ is called Prefix of string *w*.

Ex: If w = CSE then prefix of w = {∈, C, CS, CSE}.

If w = KITE then prefix of w = {∈, K, KI, KIT, KITE}.

3.2.7. Suffix of a string

Say that string x is a ***suffix*** of string y if a string z exists where zx = y, and that x is a ***proper suffix*** of y if in addition x ≠ y. A ***language*** is a set of strings. A language is ***suffix-free*** if no member is a proper suffix of another member.

The sequence of ending or tailing symbols of string *w* over alphabet Σ is called Suffix of string *w*.

Ex: If w = CSE then suffix of w = {∈, CSE, SE, E}.

If w = KITE then suffix of w = {∈, KITE, ITE, TE, E}.

Note:

- Trivial Substrings are both prefix &suffix.
- No. of Prefixes of a string *w* = No. of Suffixes of a string *w* = n +1, *where* n = |w|

- The ***lexicographic order*** of strings is the same as the familiar dictionary order. We'll occasionally use a modified lexicographic order, called ***shortlex order*** or simply ***string order*** that is identical to lexicographic order, except that shorter strings precede longer strings. Thus the string ordering of all strings over the alphabet {0,1} is (ε, 0, 1, 00, 01, 10, 11, 000, . . .).

3.3. Powers of an Alphabet

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ.

Note:

- Note that $\Sigma^0 = \{\varepsilon\}$, regardless of what alphabet Σ is. That is, ε is the only string whose length is 0.
- If Σ = {O, 1}, then $\Sigma^1 = \{0, 1\}$,
  $\Sigma^2 = \{00, 01, 10, 11\}$,
  $\Sigma^3 = \{000,001,010,011,100,101,110,111\}$ and so on.
- Note that there is a slight confusion between Σ and Σ^1. The former is an alphabet; its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1.
- The set of all the strings over alphabet Σ is conventionally denoted Σ*.
  For instance, {O, 1}* = {ε, 0, 1, 00, 01, 10, 11, 000, . . .}.
  Put another way, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots$
- Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty strings from alphabet Σ is denoted Σ^+.
- Thus, two appropriate equivalences are:
  a) $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots$ -> KLEEN CLOSURE
  b) $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots$ -> POSITIVE CLOSURE
  c) $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$
  d) $\Sigma^+ \subset \Sigma^*$

3.4. Languages

A set of strings all of which are chosen from some Σ*, where Σ is a particular alphabet, is called a language. If Σ is an alphabet, and L⊆ Σ*, then L is a language over Σ. Notice that a language over Σ need not include strings with all the symbols of Σ, so once we have established that L is a language over Σ,

we also know it is a language over any alphabet that is a superset of Σ.

The choice of the term "language" may seem strange. However, common languages can be viewed as sets of strings. An example is English, where the collection of legal English words is a set of strings over the alphabet that consists of all the letters. Another example is C, or any other programming language, where the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters. The exact alphabet may differ slightly among different programming languages, but generally includes the upper- and lower-case letters, the digits, punctuation, and mathematical symbols.

However, there are also many other languages that appear when we study automata. Some are abstract examples, such as:

Here are a few examples of languages over {a, b}:

a) The empty language ø.

b) {ε, *a, aab*}, another finite language.

c) The language *Pal* of palindromes over {*a, b*} (strings such as *aba* or *baab* that are unchanged when the order of the symbols is reversed).

d) $\{w \in \{a, b\}^* \mid n_a(w) > n_b(w)\}$.

e) $\{x \in \{a, b\}^* \mid |x| \geq 2$ and x begins and ends with $b\}$.

f) The language of all strings consisting of n 0' s followed by n l' s, for some n>=0:

   a. { ε ,01,0011,000111,. .}.

g) The set of strings of O's and l's with an equal number of each:

   a. { ε ,01,10,0011,0101,1001,.. .}

h) The set of binary numbers whose value is a prime:

   a. {10, 11, 101, 111, 1011,...}

i) Σ* is a language for any alphabet Σ.

j) ∅, the empty language, is a language over any alphabet.

k) {ε}, the language consisting of only the empty string, is also a language over any alphabet. Notice that ø ≠ { ε }; the former has no strings and the latter has one string.

Here are a few real-world languages, in some cases involving larger alphabets:

a) The language of legal Java identifiers.

b) The language *Expr* of legal algebraic expressions involving the identifier *a*, the binary operations + and *, and parentheses. Some of the strings in the language are *a*, *a* + *a* * *a*, and *(a + a * (a + a))*.

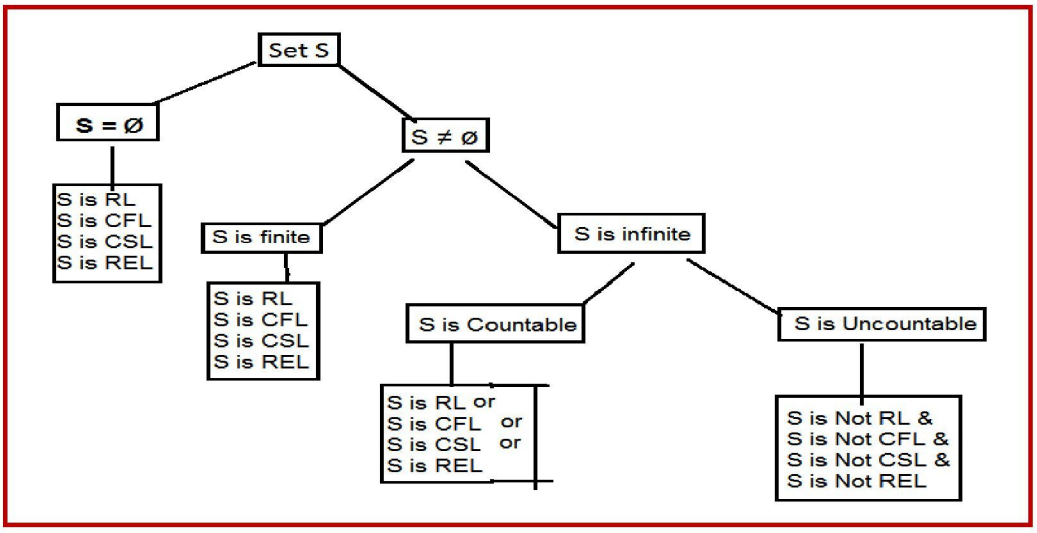
c) The language *Balanced* of balanced strings of parentheses (strings containing the occurrences of parentheses in some legal algebraic expression). Some elements are ε, *()(())*, and *(((((*********))))).*

d) The language of numeric "literals" in Java, such as −41, 0.03, and 5.0E−3.

e) The language of legal Java programs. Here the alphabet would include upper- and lowercase alphabetic symbols, numerical digits, blank spaces, and punctuation and other special symbols.

The only important constraint on what can be a language is that all alphabets are finite. Thus languages, although they can have an infinite number of strings, are restricted to consist of strings drawn from one fixed, finite alphabet.

Let S be any set.



3.5. Problems

In automata theory, a problem is the question of deciding whether a given string is a member of some particular language.

A "problem" can be expressed as membership in a language. That is, if Σ is an alphabet, and L is a language over Σ, then the problem L is:

"Given a string w in Σ, decide whether or not w is in L".*

4. FINITE AUTOMATA

4.1. Regular Language

A simple way of describing a language is to describe a finite automaton that accepts it.

A language L is called **regular** if and only if there exists some deterministic finite accepter M such that $L = L(M)$.

Note:

- A language is said to be regular if it is accepted by some FA or generated by Regular Grammar or Regular Expression.

  Ex:

  $L1$, the language of strings ending in aa;

  $L2$, the language of strings containing either the substring ab or the substring bba; and

  $L3$, the language $\{aa, aab\}^*\{b\}$.

  Like $L3$, both $L1$ and $L2$ can be expressed by a formula involving the operations of union, concatenation, and Kleene $*$: $L1$ is $\{a, b\}^*\{aa\}$ and $L2$ is $\{a, b\}^*(\{ab\} \cup \{bba\})\{a, b\}^*$.

  Languages that have formulas like these are called *regular* languages.

- The language which is not accepted by any FA is called as non-regular language.

  Ex:

  $L = \{a^n b^n / n >= 1\}$ ----------> FA +1 Stack

  $L = \{a^n b^n c^n / n >= 1\}$ ----------> FA +2 Stack

- Every language is either regular or non-regular. RL do not require any memory or require finite memory. NRL require extra memory or require infinite memory.

4.2. Finite Automata

The logical or Mathematical model of Regular language is finite automata.

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states.

The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language, a

list of five elements is often called a 5-tuple. Hence, we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a ***transition function***, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state *x* to a state *y* labeled with the input symbol 1, that means that if the automaton is in state *x* when it reads a 1, it then moves to state *y*. We can indicate the same thing with the transition function by saying that **δ (x, 1) = y**. This notation is a kind of mathematical shorthand. Putting it all together, we arrive at the formal definition of finite automata.

A **deterministic finite accepter** or **dfa** is defined by the quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is a finite set of **internal** states,

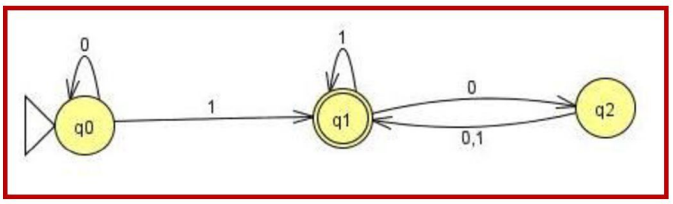
Σ is a finite set of symbols called the **input alphabet**,

δ: $Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**,

$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states.**

For example, The finite automaton M



We can describe M, formally by writing M = (Q, Σ, δ, q_1, F), where

1. Q = {q1, q2, q3},

2. Σ = {0,1},

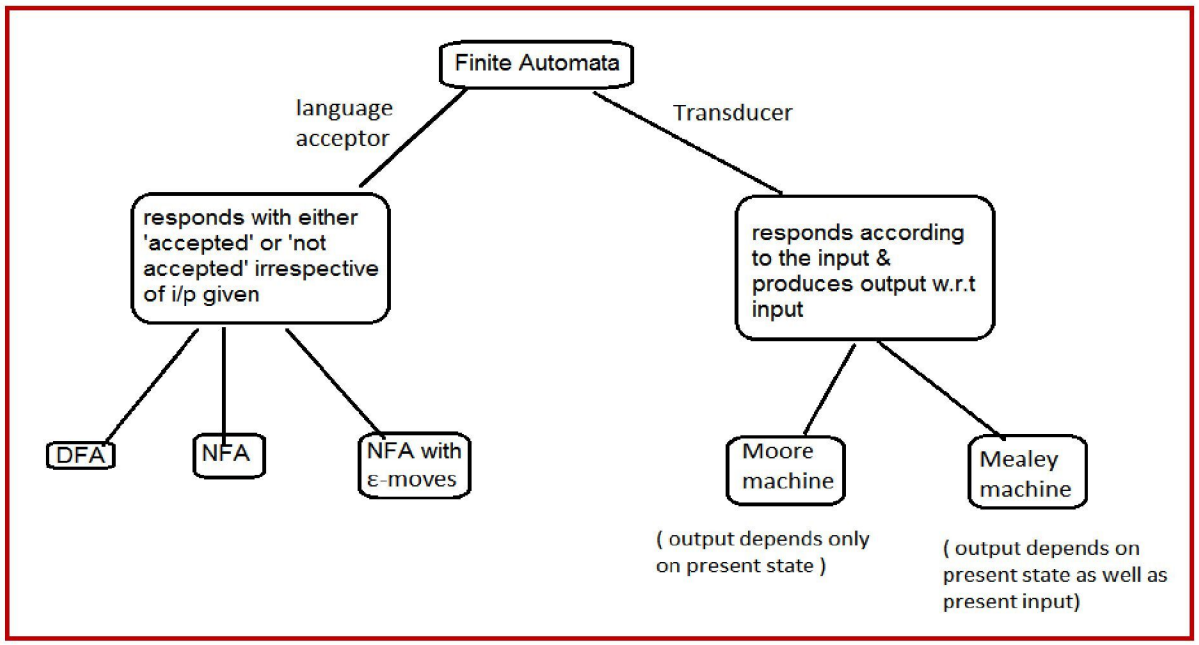
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

4. q_1 is the start state, and

5. F = {q2}.

If '*A*' is the set of all strings that machine 'M' accepts, we say that 'A' is the ***language of machine*** M and write L(M) = A. We say that M ***recognizes*** A or that M ***accepts*** A. Because the term "***accepts***'' has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term "***recognize***" for languages in order to avoid confusion.



Questions

1. FA model of ON/OFF switch.
2. FA model of working of Toll Plaza, for input Σ = {5/-, 10/-, 15/-} and vehicle can pass only if the amount is >= 25/-.
3. FA model of coin vending machine for cool drink, for the input Σ = {0.25/-, 1/-, select} and cool drink is obtained only if the amount given is >= 1.25/- and flavor is selected using 'select' input.
4. FA model of coin vending machine for printing tickets, for the input Σ = {1/-, 2/-, 5/-}. A person can take one or two tickets at a time by giving an amount of Rs 3/- or Rs 6/- respectively.

Note:

- In general FA is DFA.
- Intial state of DFA is unique and processing of string always starts from intial state.
- FA can be constructed with 0 or more final states.
- FA provides only single path to process a string.
- DFA is a complete system that responds to both valid & invalid inputs.
- The system is complete iff transition is defined for each & every symbol in the given input alphabet Σ at every state.
- Total no. of transitions defined at a state = |Σ| = no. of input symbols
- Total no. of input symbols defined in DFA = |Σ| * |Q|
- DFA processes input string character by character.
- DFA may change its present state after scanning each symbol in the input string.
- Sequence of transitions is called as transition path & the transition path represents the processing of input string.

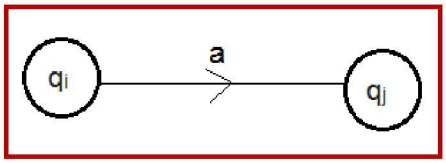
4.2.1. Instantaneous Description (ID)

ID defines next move of FA. The next move FA depends on two inputs.

a. present state
b. present input symbol

i.e $\delta(q_i, a) = q_j$ where q_i = present state & a = present input symbol & q_j is next state

Here q_j may or may not be equal to q_i.



4.2.2. Representation of FA

Specifying a DFA as a five-tuple with a detailed description of the δ transition function is both tedious and hard to read. There are two preferred notations for describing automata:

1. A transition diagram, which is a graph.
2. A transition table, which is a tabular listing of the δ function, which by implication tells us the set

of states and the input alphabet.

*A **transition diagram*** for a DFA, M = (Q, Σ, δ, q_0, F) is a graph defined as follows:

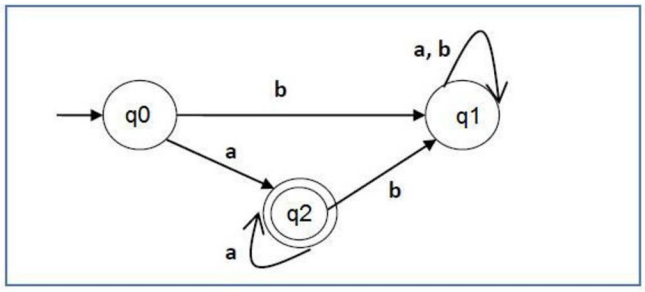
a) For each state in Q there is a node.

b) For each state q in Q and each input symbol a in Σ, let δ(q,a) = p.

c) The transition diagram has an arc from node q to node p, labeled a. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols separated by comma.

d) There is an arrow into the start state q_0, labeled Start. This arrow does not originate from any node.

e) Nodes corresponding to accepting states (those in F) are represented by two concentric circles. States not in F are represented by single circle.



DFA accepting language L

In the above transition diagram $M = (Q, \Sigma, \delta, q_0, F)$,

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b\}$

$\delta: Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**

i.e $\{q_0, q_1, q_2\} \times \{a, b\} \rightarrow \{q_2, q_1, q_1, q_1, q_2, q_1\}$ respectively.

$\delta(q_0, a) \rightarrow q_2$

$\delta(q_0, b) \rightarrow q_1$

$\delta(q_1, a) \rightarrow q_1$

$\delta(q_1, b) \rightarrow q_1$

$\delta(q_2, a) \rightarrow q_2$

$\delta(q_2, b) \rightarrow q_1$

$q_0 = \{q_0\}$

$F = \{q_2\}$

We have explained informally that the DFA defines a language: the set of all strings that result in a sequence of state transitions from the start state to an accepting state. In terms of the transition diagram, the language of a DFA is the set of labels along all the paths that lead from the start state to any
accepting state.

A ***transition table*** is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input *a* is the state
δ (q, a).

δ	**a**	**b**
q0	q2	q1
q1	q1	q1
q2	q2	q1

4.2.3. String Acceptance by FA

The first thing we need to understand about a DFA is how the DFA decides whether or not to "accept" a sequence of input symbols. A deterministic finite accepter operates in the following manner.

At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise, the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ.

i.e the "language" of the DFA is the set of all strings that the DFA accepts. Suppose $a_1a_2\ldots a_n$ is a sequence of input symbols. We start out with the DFA in its start state, q_0. We consult the transition function δ, say $\delta(q_0, a_1) == q_1$ to find the state that the DFA, M enters after processing the first input symbol a_1. We process the next input symbol, a_2, by evaluating $\delta(q_1,$

a_2); let us suppose this state is q_2. We continue in this manner, finding states q_3, q_4,…,q_n such that $\delta(q_{i-1},a_i) == q_i$ for each i. If q_n is a member of F, then the input $a_1a_2\ldots a_n$ is accepted, and if not then it is "rejected."

i.e. $L(M) = \{w \in \Sigma^* / \delta(q_0, w) = F\}$

4.2.4. Block Diagram of FA

It has three components.

1. Input tape (input file)
2. Tape header
3. Finite Control Unit (Process Unit)

Input tape:

It has a mechanism for reading input. It will be assumed that the input is a string over a given alphabet, written on an **input file**, which the automaton can read but not change. The input file is divided into cells, each of which can hold one symbol. The input mechanism can read the input file from left to right, one symbol at a time. The input mechanism can also detect the end of the input string (by sensing an end-of-file condition). Input tape can hold only one string at any point of time.

Tape header:

The header scans the input symbol from input tape character by character from left to right.

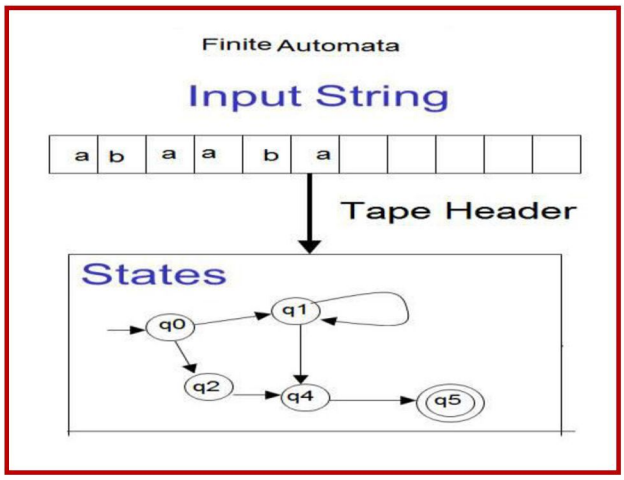
The header scans only one symbol at a time and moves exactly by one cell towards right.

The header movement is unidirectional (left to right).

The header can read the data from input tape but cannot write data.

Finite Control Unit (FCU):

At any given time, the control unit is in some internal state. The internal state of the control unit at is determined by the **ID** or **transition function**. This transition function is responsible for change in state of control unit depending on the current state & the current input symbol. During the transition, no output is produced but the relevant history of system

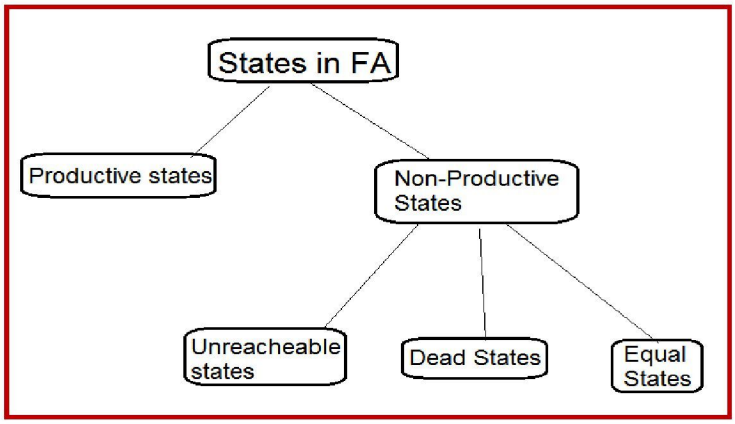


is remembered in terms of states. No external storage is required for storing the output, as the output is only 'yes' or 'no' based on last state of FCU.

Note:

a) Every state goes to itself on empty string 'ε'.
    a. i.e $\delta(q, \varepsilon) = q, q \in Q$
b) FA accepts empty string 'ε' iff intial state is final state.

.



Productive state

The state which is involved in the process of a acceptance of an valid input string is called ***Productive state***.

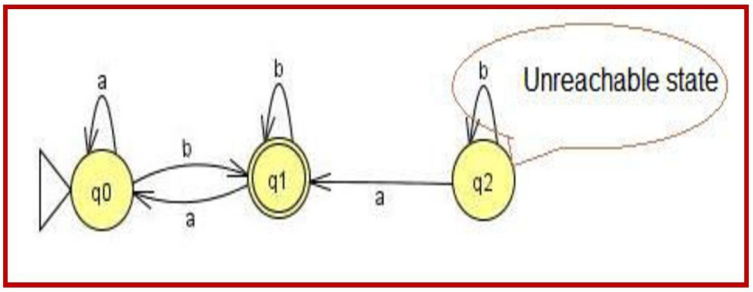
Non-Productive state

The state which is not involved in the process of an acceptance of a valid input string is called ***Non-Productive state***.

The state whose presence or absence will not affect the language accepted by the FA is called ***Non-Productive state***.

Unreachable state

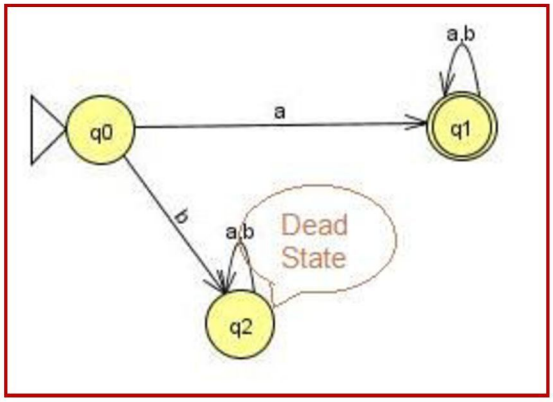
The state to which transition path doesn't exist from initial state is called as ***unreachable state***.



- No change in language acceptance & Structure after removal of unreachable states.

Dead state

It is also called as ***trap state***. The non-final state from which no transition path exists to any final state in the given FA is called as ***Dead State***.

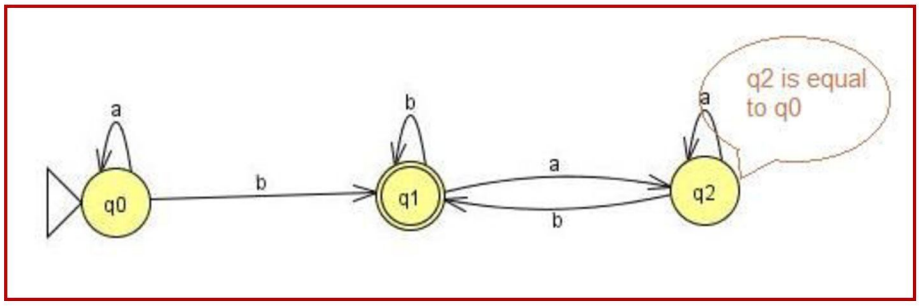


- No change in language acceptance & but change in Structure occurs after removal of dead states.

Equal States

Let Q denote set of all states of FA, $M = (Q, \Sigma, \delta, q0, F)$.Two states $p \in Q$, $q \in Q$ are said to be equal if $\delta(q,w) = \delta(p,w)\ \forall\ w \in \Sigma^*$.

i.e transition of both must result either final state or non-final state.

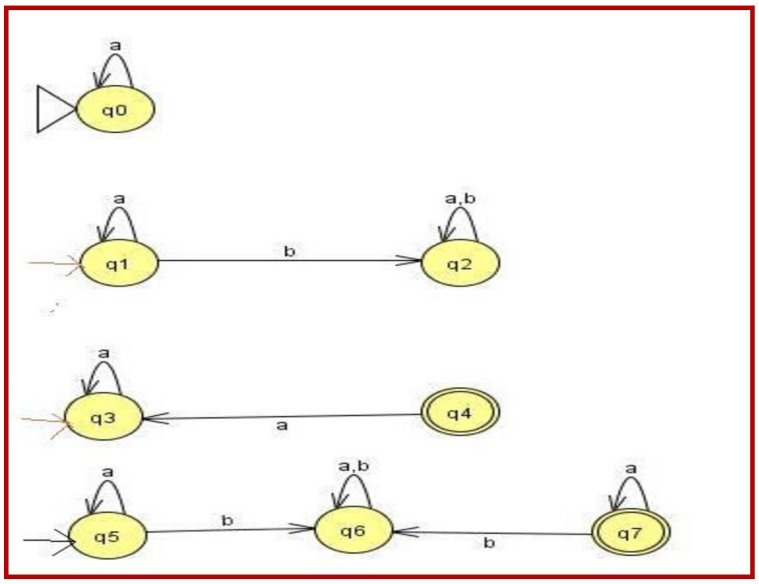


- No change in language acceptance & Structure after removal of Equal states.

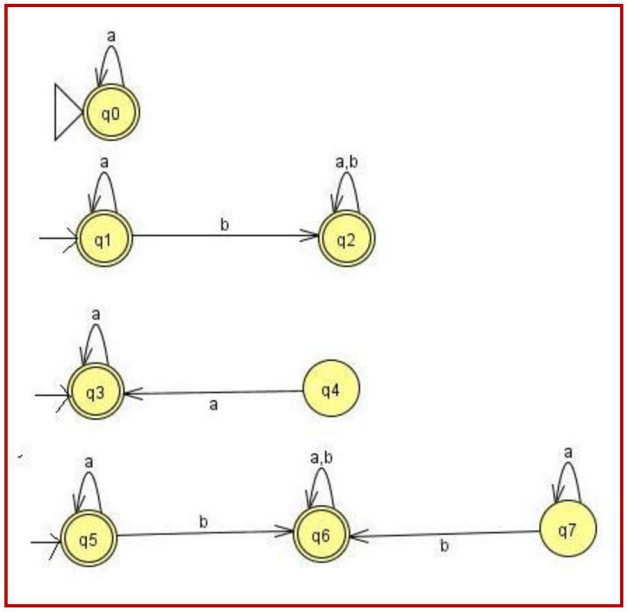
Note:

- The DFA which is free from unreachable states & equal States is called minimal FA. Dead state may or may not be part of minimal FA.

- Every FA represents only one regular language. But a regular language can be represented by more than one DFA.
- Every regular language is accepted by only one minimal DFA.
  i.e DFA is not unique but minimal DFA is unique.
- The FA that doesn't contain any final states (or it may include unreachable final states) accepts empty language 'ø'
  Ex:



- The FA accepts universal language over the considered Σ if all the states are final.
  Ex:



- No. of states in minimal FA to accept universal language Σ* over considered alphabet Σ is ***one.***
- No. of states in minimal FA to accept empty language ø over considered alphabet Σ is ***one or two.***

Questions

Design of FA for finite languages:

1) Design FA for the following finite languages?
   a) L = {a, aaa}, Σ = {a}
   b) L = { ε,aa}, Σ = {a}
   c) L = {ab,bb}, Σ = {a,b}
   d) L = {aba,bab,bba,bbb}, Σ = {a,b}
   e) L = $\{a^n b^n / 1 <= n <= 3\}$, Σ = {a,b} No. of states = 2n+2
   f) L = $\{(ab)^n / 1 <= n <= 3\}$, Σ = {a,b} No. of states = 2n+2
   g) L = $\{a^m b^n / 1 <= m <= 2, 1 <= n <= 3\}$, Σ = {a,b}
   h) L = $\{a^m b^n / m+n = 2\}$, Σ = {a,b}
   i) L = $\{a^m b^n / m*n = 4\}$, Σ = {a,b}
   j) L = $\{w \in \Sigma^* / |w| <= 2\}$, Σ = {a,b}

Note:

- Every path from initial state to final state represents a valid string.
- No. of valid input strings = No. of transition paths that start in initial state & end in one of the final states.
- FA accepts finite language if and only if transition diagram is free from cycles & loops except loops for dead state.
- FA accepts infinite language if transition diagram has cycles or loops or both.

4.2.5. Extending the Transition Function to Strings

We have explained informally, that the DFA defines a language: the set of all strings that result in a sequence of state transitions from the start state to an accepting state. In terms of the

transition diagram, the language of a DFA is the set of labels along all the paths that lead from the start state to any
accepting state.

Now, we need to make the notion of the language of a DFA precise. To do so, we define an extended transition function that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$.

The extended transition function is a function that takes a state q and a string *w* and returns a state p --- the state that the automaton reaches when starting in state q and processing the sequence of inputs *w*. We define $\hat{\delta}$ by induction on the length of the input string, as follows:

BASIS: $\hat{\delta}(q, \varepsilon) = q$. That is, if we are in state q and read no inputs, then we are still in state q.

INDUCTION: Suppose *w* is a string of the form xa; that is, '*a*' is the last symbol of w, and x is the string consisting of all but the last symbol.

For example,

w = 1101 is broken into x == 110 and a= 1. Then

$\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x),a)$

To compute $\hat{\delta}(q,w)$, first compute $\hat{\delta}(q,x)$, the state that the automaton is in after processing all but the last symbol of *w*. Suppose this state is p; that is, $\hat{\delta}(q,x) = p$. Then $\hat{\delta}(q,w)$ is what we get by making a transition from state p on input 'a', the last symbol of *w*. That is, $\hat{\delta}(q,w) = \delta(p,a)$.

Example 2.4: Let us design a DFA to accept the language

L = { w | w has both an even number of 0's and an even number of 1 's}

It should not be surprising that the job of the states of this DFA is to count both the number of 0's and the number of 1's, but count them modulo 2. That is, the state is used to remember whether the number of 0's seen so far is even or odd, and also to remember whether the number of 1's seen so far is even or odd.

There are thus four states, which can be given the following interpretations:

q_0: Both the number of 0's seen so far and the numbers of 1's seen so far are even.

q_1: The number of 0's seen so far is even, but the number of 1's seen so far is odd.

q_2: The number of 1's seen so far is even, but the number of 0's seen so far is odd.

q_3: Both the number of 0's seen so far and the numbers of 1's seen so far are odd.

State q_0 is both the start state and the lone accepting state. It is the start state, because before reading any inputs, the numbers of 0's and 1's seen so far are both zero, and zero is even. It is the only accepting state, because it describes exactly the condition for a sequence of 0's and 1's to be in language L.

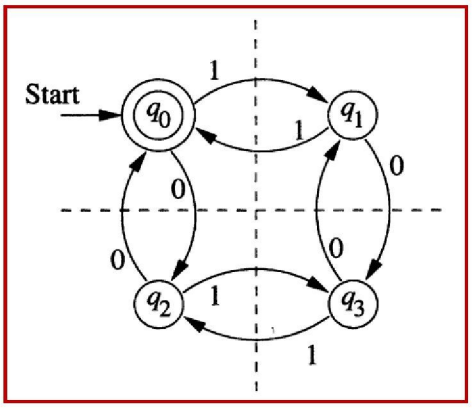


Figure 2.6: Transition diagram for the DFA of Example 2.4

We now know almost how to specify the DFA for language L. It is

$$M = (\{q_o, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_o\})$$

where the transition function δ is described by the transition diagram of Fig. 2.6.

Notice how each input 0 causes the state to cross the horizontal, dashed line. Thus, after seeing an even number of 0's we are always above the line, in state q_0 or q_1. While after seeing an odd number of 0's we are always below the line, in state q2 or q3. Likewise, every 1 causes the state to cross the vertical, dashed line. Thus, after seeing an even number of 1's, we are always to the left, in state q_o or q_2, while after seeing an odd number of 1's we are to the right, in state q_1 or q_3.

These observations are an informal proof that the four states have the interpretations attributed to them. However, one could prove the correctness of our claims about the states formally, by a mutual induction in the spirit of Example 1.23.

We can also represent this DFA by a transition table Fig 2.7.

	0	1
→ (q0)	q2	q1
q1	q3	q0
q2	q0	q3
q3	q1	q2

Figure 2.7: transition table for the DFA of Example 2.4

The check involves computing $\hat{\delta}(q_0, w)$ for each prefix w of 110101, starting at ϵ and going in increasing size. The summary of this calculation is:

- $\hat{\delta}(q_0, \epsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$.

Example 2.5: Let us design a DFA to accept the language

L = { w | w is of even length and begins with 01}

The Automaton needs to remember whether the string seen so far started with 01.It also has to keep track of the length of string. Hence it consists of five states which could be interpreted as follows.

q_0: The initial state.

q_1: The state entered on reading 0 in state q_0.

q_2: The state entered on reading 01 initially. The automaton subsequently returns to this state whenever the substring seen so far starts with 01 and is of even length.

q_3: The DFA enters this state whenever the substring seen so far starts with 01 and is of odd length.

q_4: This state is entered whenever a 1 is encountered in state q_0 or a 0 is encountered in state q_1.

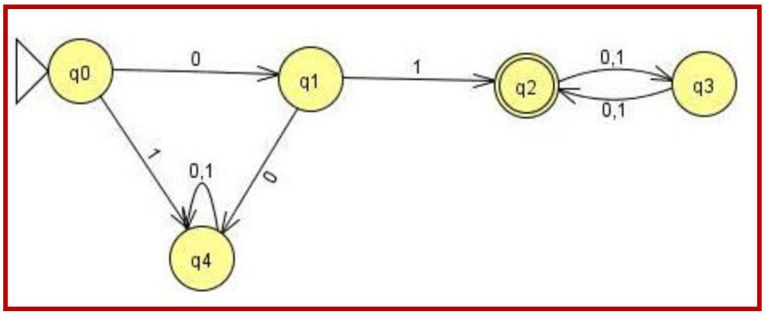


Figure 2.6: Transition diagram for the DFA of Example 2.4

Clearly q_2 is only accepting state. The DFA can thus be given as

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, \{q_2\})$

where the transition function δ is described by the transition diagram of Fig. 2.6.

We can also represent this DFA by a transition table.

δ	0	1
q_0	q_1	q_4
q_1	q_4	q_2
q_2	q_3	q_3
q_3	q_2	q_2
q_4	q_4	q_4

Let us construct $\hat{\delta}$ from its transition function δ. suppose the input is 011101. Since the string starts with 01 and is of even length, we expect it is in the language. Thus, we expect that $\hat{\delta}(q_0, 011101) = q_2$, since q_2 is only the accepting state. Let us verify it.

The check involves computing $\hat{\delta}(q, w)$ for each prefix w of 011101, starting at ε and going in increasing size. The summary of this calculation is:

$\hat{\delta}(q_0, \varepsilon) = q_0$

$\hat{\delta}(q_0, 0) = \delta(\hat{\delta}(q_0, \varepsilon), 0) = \delta(q_0, 0) = q_1$

$\hat{\delta}(q_0, 01) = \delta(\hat{\delta}(q_0, 0), 1) = \delta(q_1, 1) = q_2$

$\hat{\delta}(q_0, 011) = \delta(\hat{\delta}(q_0, 01), 1) = \delta(q_2, 1) = q_3$

$\hat{\delta}(q_0, 0111) = \delta(\hat{\delta}(q_0, 011), 1) = \delta(q_3, 1) = q_2$

$\hat{\delta}(q_0, 01110) = \delta(\hat{\delta}(q_0, 0111), 0) = \delta(q_2, 1) = q_3$

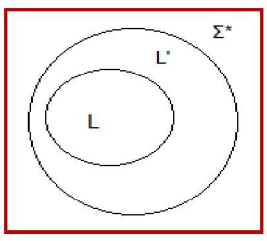
$\hat{\delta}(q_0, 0111101) = \delta(\hat{\delta}(q_0, 01110), 1) = \delta(q_3, 1) = q_2$

4.2.6. Compliment of FA

If '*L*' is a Regular language, accepting the strings over alphabet Σ, then compliment of *L* denoted by *L'* is defined as Regular language such that $L' = \Sigma^* - L$.

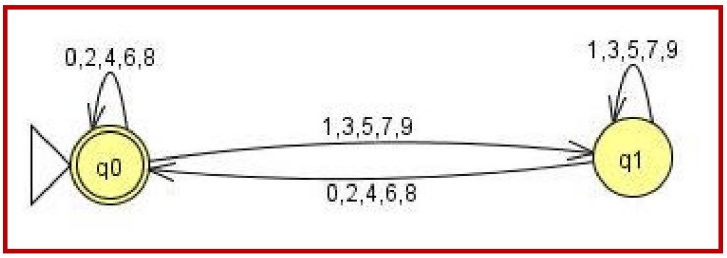
The FA for L' can be obtained by interchanging final & non-final states in FA of L.

Here L' = Σ* - L

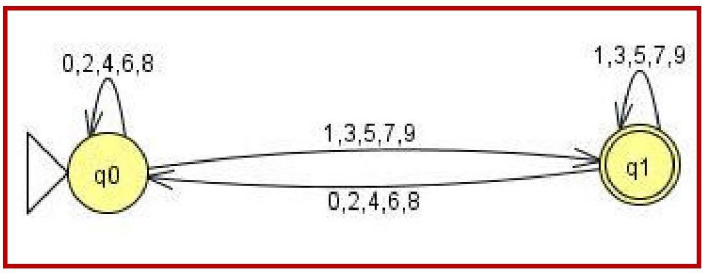


Ex:

L = {set of all even numbers over Σ = {0,1,2,3,4,5,6,7,8,9}}



L' = {set of all odd numbers over Σ = {0,1,2,3,4,5,6,7,8,9}}



Note:

- If L U L' = Σ* and L U L' = ø, then L & L' are called as equivalence classes in which all the elements of L satisfy a property & similarly L'.
- If the RL's L & L' are recognized by MFA's M & M' respectively, then no. of states in M = no. of states in M'.
- If the RL *L* is recognized by FA *M*, with 'n' states and 'k' final states, then RL *L'* is recognized by FA *M'* with 'n' states and 'n-k' final states.
- If L = ø, then L' = Σ* - L = Σ* - ø = Σ*

  If L = Σ*, then L' = Σ* - L = Σ* - Σ* = ø

  Therefore, Σ*, ø are compliment to each other.

4.3. Non-Deterministic Finite Automata

Non-determinism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

A "nondeterministic" finite automaton (NFA) has the power to be in several states at once. This ability is often expressed as an ability to "guess" something about its input. For

instance, when the automaton is used to search for certain sequences of characters (e.g., keywords) in a long text string, it is helpful to "guess" that we are at the beginning of one of those strings and use a sequence of states to do nothing but check that the string appears, character by character.

Before examining applications, we need to define nondeterministic finite automata and show that each one accepts a language that is also accepted by some DFA. That is, the NFA's accept exactly the regular languages, just as DFA's do. However, there are reasons to think about NFA's. They are often

more succinct (briefly and clearly expressed) and easier to design than DFA's.

Moreover, while we can always convert an NFA to a DFA, the latter may have exponentially more states than the NFA; fortunately, cases of this type are rare.

4.3.1. An Informal View of Nondeterministic Finite Automata

Like the DFA, an NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states. It also has a transition function, which we shall commonly call δ. The difference between the DFA and the NFA is in the type of δ. For the NFA, δ is a function that takes a state and input

symbol as arguments (like the DFA's transition function), but returns a set of zero, one, or more states (rather than returning exactly one state, as the DFA must). We shall start with an example of an NFA, and then make the definitions precise.

Figure 2.9 shows a nondeterministic finite automaton that accepts all and only the strings of 0's and 1's that end in 01. State q_0 is the start state, and we can think of the automaton as being in state q_0 (perhaps among other states) whenever it has not yet "guessed" that the final 01 has begun. It is always possible that the next symbol does not begin the final 01, even if that symbol is 0. Thus, state q_0 may transition to itself on both 0 and 1.

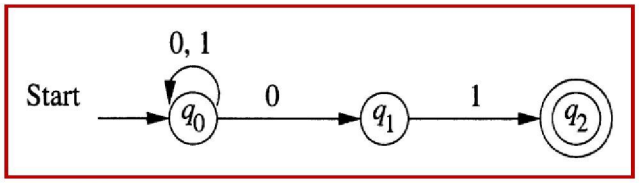


Figure 2.9: An NFA accepting all strings over Σ = {0, 1} that end in 01

However, if the next symbol is 0, this NFA also guesses that the final 01 has begun. An arc labeled 0 thus leads from q_0 to state q_1. Notice that there are two arcs labeled 0 out of q_0. The NFA has the option of going either to q_0 or to q_1, and in fact it does both, as we shall see when we make the definitions precise. In state q_1, the NFA checks that the next symbol is 1, and if so, it goes to state q_2 and accepts.

Notice that there is no arc out of q_1 labeled 0, and there are no arcs at all out of q_2. In these situations, the thread of the NFA's existence corresponding to those states simply "dies," although other threads may continue to exist. While a DFA has exactly one arc out of each state for each input symbol, an NFA has no such constraint; we have seen in Fig. 2.9 cases where the number of arcs is zero, one, and two, for example.

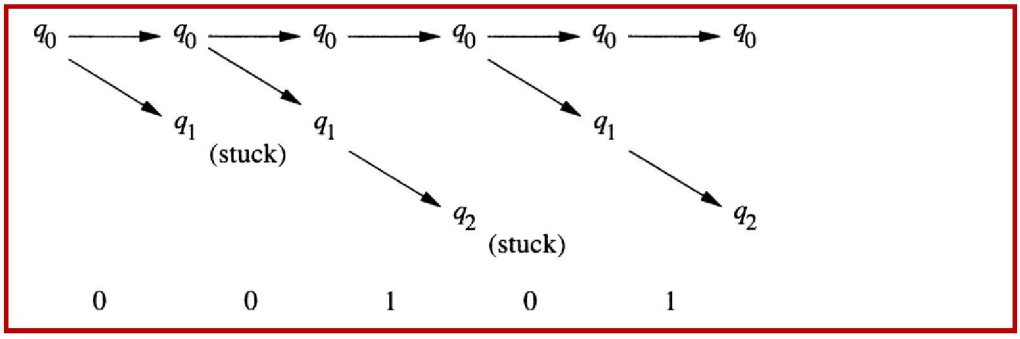


Figure 2.10: The states an NFA is in during the processing of input sequence 00101

Figure 2.10 suggests how an NFA processes inputs. We have shown what happens when the automaton of Fig. 2.9 receives the input sequence 00101. It starts in only its start state, q_0. When the first 0 is read, the NFA may go to either state q_0 or state q_1, so it does both. These two threads are suggested by the second column in Fig. 2.10.

Then, the second 0 is read. State q_0 may again go to both q_0 and q_1. However, state q_1 has no transition on 0, so it "dies." When the third input, a 1, occurs, we must consider transitions from both q_0 and q_1. We find that q_0 goes only to q_0 on 1, while q_1 goes only to q_2. Thus, after reading 001, the NFA

is in states q_0 and q_2. Since q_2 is an accepting state, the NFA accepts 001.

However, the input is not finished. The fourth input, a 0, causes q_2's thread to die, while q_0 goes to both q_0 and q_1. The last input, a 1, sends q_0 to q_0 and q_1 to q_2. Since we are again in an accepting state, 00101 is accepted.

4.3.2. Definition of Nondeterministic Finite Automata

A **Nondeterministic finite accepter** or **NFA** is defined by the quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

$\delta : Q \times \Sigma \rightarrow 2^Q$ is called the **transition function**,

δ, the transition function is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q. Notice that the only difference between an NFA and a DFA is in the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA.

$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states.**

The NFA of Fig. 2.9 can be specified formally as

$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

where the transition function δ is given by the transition table of Fig. 2.11

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Fig. 2.11: Transition table for an NFA that accepts all strings ending in 01

Notice that transition tables can be used to specify the transition function for an NFA as well as for a DFA. The only difference is that each entry in the table for the NFA is a set, even if the set is a singleton (has one member). Also notice that when there is no transition at all from a given state on a given input

symbol, the proper entry is ø, the empty set.

4.3.3. Acceptance of string by NFA

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only

if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Formally, if A = (Q, Σ, δ, q_0, F) is an NFA, then Language of NFA '*A*' denoted by L(A) is defined as

$$L(A) == \{w \in \Sigma^* \; I \; \hat{\delta}(q_0, w) \cap F \neq \phi\}$$

That is, L(A) is the set of strings w in Σ* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q_0 in NFA N1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ε symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ε-labeled arrows and one staying at the current state. Then the machine proceeds non-deterministically as before.

Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently. When the NFA splits to follow several choices, that corresponds to a process "forking" into several children, each proceeding separately. If at least one of these processes accepts, then the entire computation accepts.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure 1.28.

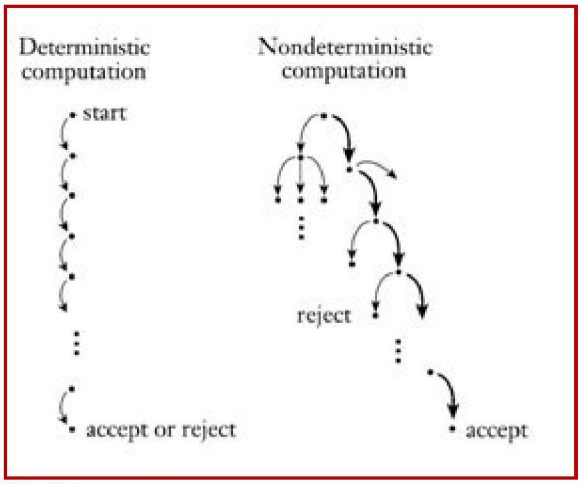


FIGURE **1.28** Deterministic and nondeterministic computations with an accepting branch

4.3.4. Why Nondeterminism?

Non-determinism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such non-mechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and back-track algorithms.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in Figure 2.8. It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2^n}: n \geq 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic

solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

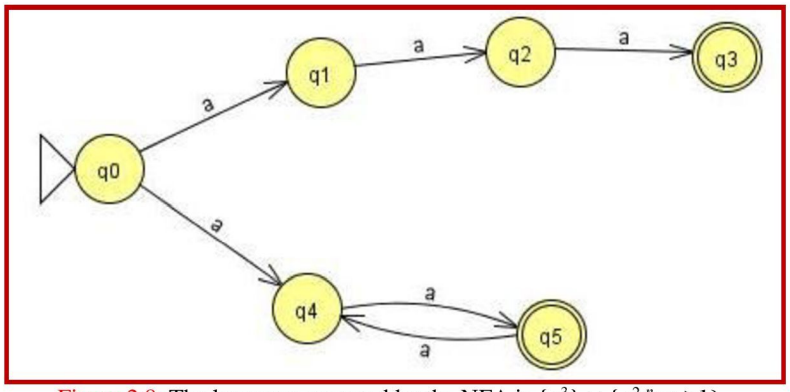


Figure 2.8: The language accepted by the NFA is $\{a^3\} \cup \{a^{2^n} : n \geq 1\}$

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely.

Notice that the definition of a grammar involves a nondeterministic element. In we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

$$S \rightarrow aSb|\lambda$$

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

Note:

- In a ***nondeterministic*** machine, several choices may exist for the next state at any point.
- Non-determinism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton.
- The capabilities of both NFA & DFA are same.
- DFA represents a complete system but NFA need not.
- No Dead state concept in NFA.
- Representation of RL by NFA is easier than DFA but DFA is more efficient & faster than NFA.

4.3.5. Differences between NFA & DFA

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent.

First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. In general, an NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

Third, in a DFA, the range of δ is single state of Q where as in a nondeterministic accepter; the range of δ is in the power set 2^Q, where Q is a finite set of **internal** states, so that its value is not a single element of Q but a subset of it. This subset defines the set of all possible states that can be reached by the transition.

Fourth, Let L be RL recognized by DFA M, then compliment of L, i.e L' is obtained by interchanging final and non-final states of M. whereas, this may not be true in case of NFA.

4.3.6. Equivalence of Deterministic and Nondeterministic Finite Automata

- The process of obtaining DFA for given NFA is called as ***subset construction***. This also stand as proof that DFA's can do whatever NFA's can do.
- If Q = n (no. of states), in NFA, then corresponding DFA may consist at most 2^n states. i.e if NFA consists of 'n' states and no. of states in its equivalent DFA would range from 1 to 2^n.
- There may be change in number of states.
- There will be no change in initial state.
- There may be change in final states.

Algorithm

Let N = (Q, Σ, δ, q_0, F) be the NFA recognizing some language A. We construct a DFA M = (Q′, Σ, δ′, q_0′, F′) equivalent to NFA 'N' recognizing A.

1) Q′ = P(Q).

   Every state of M is a set of states of N. Recall that P(Q) is the set of subsets of Q.

Often, not all these states are accessible from the start state of M. Inaccessible states can left out. So effectively, the number of states of M may be much smaller than 2n.

2) Notice that the input alphabets of the two automata are the same.

3) Initial state (q_0')

$q_0' = \{q_0\}$.

The start state of M is the set containing only the start state of N.

i.e there is no change in initial state of DFA M when compared with NFA N.

4) Construction of δ'

For each set $S \subseteq Q'$ and for each input symbol a in Σ,

$$\delta'(S, a) = \bigcup \delta(p, a), \text{ p in S}$$

i.e S is a singleton set consisting of only one state 'q', then

$$\delta'(q, a) = \delta(q, a)$$

else if S is a set consisting of multiple states (i.e $S = \{q_1, q_2, q_3, \ldots\ldots q_n\}$), then

$$\delta'(\{q_1, q_2, q_3, \ldots\ldots q_n\}, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_3, a) \ldots\ldots \cup \delta(q_n, a)$$

Start the construction of δ' with initial state obtained in step3 and continue for every new state that appears under any input symbol in the transition table.

5) Final state (F')

F' is all sets of N's states that include at least one accepting state of N. i.e Every subset which contain final state of NFA is a final state in DFA.

4.4. Definition of Nondeterministic Finite Automata with ε moves (ε- NFA)

A Nondeterministic Finite Automata with ε moves (ε- NFA) is defined by the quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

δ: $Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^Q$ is called the **transition function**,

δ, the transition function is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q. Notice that, ε- NFA will also define transitions for ε as input from a state.

$q_0 \in Q$ is the **initial state**,

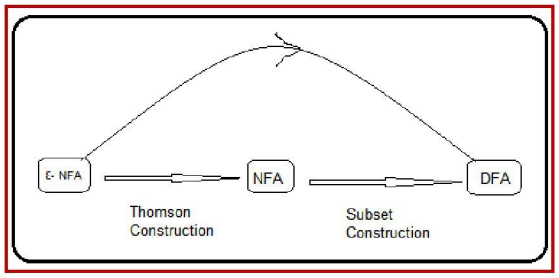
$F \subseteq Q$ is a set of **final states.**

Note:

- ε- NFA can have more than one initial state.
- The computing power of DFA, NFA & ε- NFA is same.

  E(DFA) = E(NFA) = E (ε- NFA)

- In ε- NFA, the machine changes its state even without operating on any input symbol.
- Construction of ε- NFA for RL is easier than NFA & DFA. But ε- NFA can be converted to NFA & DFA.



4.4.1. Epsilon - closure (q)

Let q∈Q in ε- NFA, then Epsilon - closure (q) represented as $\hat{\delta}(q, \varepsilon)$ = ECLOSE(q) is defined as set of all states that can be reached from 'q' along Epsilon(ε) transition path.

(or)

Let q∈Q in ε- NFA, then Epsilon - closure (q) represented as $\hat{\delta}(q, \varepsilon)$ = ECLOSE(q) is defined as set of all states that are at zero distance from 'q' .

Note:

- Every state is at zero distance from itself.
  - i.e $\hat{\delta}(A, \varepsilon) = A$
- ECLOSE(q) is a non-empty set.
- ECLOSE(ø) = ø
- ECLOSE(A∪B) = ECLOSE(A) ∪ ECLOSE(B)
- ECLOSE (A∪B ∪ C) = ECLOSE(A) ∪ ECLOSE(B) ∪ ECLOSE(C)
- ECLOSE(ECLOSE(A)) = ECLOSE(A)

4.4.2. Equivalence of NFA and ε- NFA

- The process of obtaining NFA for given ε- NFA is called as ***Thomson construction***.
- There will be no change in number of states.
- There will be no change in initial state.
- There may be change in final states.

Algorithm

Let N = (Q, Σ, δ, q_0, F) be the ε-NFA recognizing some language A. We construct a NFA M = (Q′, Σ, δ′, q_0′, F′) equivalent to ε-NFA 'N' recognizing A.

1) Q′ = Q.

   Both the automata will have same no. of states.

2) Notice that the input alphabets of the two automata are the same.

3) Initial state (q_0′)

   q_0′ = {q_0}.

   The start state of M is the start state of N.

   i.e there is no change in initial state of NFA M when compared with ε-NFA N.

4) Construction of δ′

   For each state q∈Q and for each input symbol '*a*' in Σ,

   δ′(q, a) = ECLOSE { δ (ECLOSE(q), a) }

5) Final state (F')

   F' is a set of all states whose Epsilon - closure contains final state of ε-NFA. i.e Every state whose Epsilon–closure contain final state of ε-NFA is a final state in NFA.

4.4.3. Equivalence of DFA and ε- NFA

- There may be change in number of states.
- There will be change in initial state.
- There may be change in final states.

Algorithm

Let N = (Q, Σ, δ, q_0, F) be the ε-NFA recognizing some language A. We construct a DFA M = (Q′, Σ, δ′, q_0′, F′) equivalent to ε-NFA 'N' recognizing A.

1) Q′ = set of subsets of Q and there may be additional state representing dead configuration of automata.
2) Notice that the input alphabets of the two automata are the same.
3) Initial state (q_0')

   q_0' = ECLOSE(q_0).
4) Construction of δ′

   For each state q∈Q and for each input symbol '*a*' in Σ,

   δ′(q, a) = ECLOSE { δ(q, a) }

   Start the construction of δ′ with initial state obtained in step3 and continue for every new state that appears under any input symbol in the transition table.
5) Final state (F')

   F' is a set of all states whose Epsilon - closure contains final state of ε-NFA. i.e Every state whose Epsilon–closure contain final state of ε-NFA is a final state in NFA.

4.5. Decision Properties of FA

The property which has an algorithm to decide is called as ***decidable*** or ***decision property***.

4.5.1. Emptiness

Identify all the unreachable states in FA and remove them.

If the resulting automaton contains at least one final state, then FA accepts non-empty language.

If the resulting automaton free from final states, then FA accepts empty language.

4.5.2. Finiteness

Identify all the unreachable states in FA and remove them.

Identify all the dead states in resulting FA of above step and remove them.

If the resulting automaton contains loops or cycles, then FA accepts infinite language.

If the resulting automaton is free from loops & cycles, then FA accepts finite language.

4.5.3. Equalness

Two FSMs are said to be equal if both of them accept the same language.

i.e Let M_1, M_2 be two FSMs, then $M_1 = M_2$ if and only if $L(M_1) = L(M_2)$

$M = (Q, \Sigma, \delta, q_0, F)$

$\delta : Q \times \Sigma \rightarrow Q$

$\delta(q,a) \neq$

$\varepsilon \quad \in \quad \Sigma^* \quad \text{ø} \quad \hat{\delta}$

$\cap \cup \neq \subseteq$

Note:

Questions

No. of states = 2n+2