**Theory of Computation**

**Department of Computer Science**

**Milan Kumar Nayak**

**Cont.6370158954**

**Basudev Godabari Degree College**

At /- Kesaibahal , Dist/- Sambalpur , Pin – 768228

[bgdegreecollege@gmail.com](mailto:bgdegreecollege@gmail.com) **,Mob – 9438000807**

Unit-1

**Introduction to Theory of Computation :-**

The **theory of computation** is a branch of **computer science** and mathematics combined that "deals with how efficiently problems can be solved on a model of **computation**, using an algorithm". It studies the general properties of **computation** which in turn, helps us increase the efficiency at which computers solve problems.

**Automata** theory (also known as **Theory Of Computation**) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.

Back to the computational theory,This theory is approached through three main fields:  
  
1. Automata theory  
2. Computability theory

3. Computational complexity theory  
  
 These three branches basically uses algorithms to exploit the powers of computers in solving problems.  
Here is an illustration of these fields :  
  
**Automata theory:**  
 This branch was established in the 20th century by mathematicians. The main aim in this branch is to analyze the behavior of machines and how they solve a problem.The most powerful model of automata is the Turing machine.

**Computability theory:**  
 It is when we are able to formulate the problem using the Turing machine easily, but we can never solve it.. In other words, It is when the computer is able to address the problem but unable to come up with the solution  
  
**Computational complexity theory:** The complexity theory discusses the efficiency at which a problem could be solved. This is done considering two major aspects: time complexity and space complexity, which are the measurees of the number of steps needed to  
  
analyze and solve the problem and thus determining the memory space needed to solve the problem.  
  
In order to pre-determine the space and time that will be needed,computer scientists linked these two factors to the amount of input that was recieved, as the time and space needed increase linearly as the amount of input

* The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a **Finite Automaton** (FA) or **Finite State Machine** (FSM).

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, ∑, δ, q0, F), where −

* **Q** is a finite set of states.
* **∑** is a finite set of symbols, called the **alphabet** of the automaton.
* **δ** is the transition function.
* **q0** is the initial state from where any input is processed (q0 ∈ Q).
* **F** is a set of final state/states of Q (F ⊆ Q).

### Alphabet

* **Definition** − An **alphabet** is any finite set of symbols.
* **Example** − ∑ = {a, b, c, d} is an **alphabet set** where ‘a’, ‘b’, ‘c’, and ‘d’ are **symbols**.

### String

* **Definition** − A **string** is a finite sequence of symbols taken from ∑.
* **Example** − ‘cabcad’ is a valid string on the alphabet set ∑ = {a, b, c, d}

### Length of a String

* **Definition** − It is the number of symbols present in a string. (Denoted by **|S|**).
* **Examples** −
  + If S = ‘cabcad’, |S|= 6
  + If |S|= 0, it is called an **empty string** (Denoted by **λ** or **ε**)

### Kleene Star

* **Definition** − The Kleene star, **∑\***, is a unary operator on a set of symbols or strings, **∑**, that gives the infinite set of all possible strings of all possible lengths over **∑** including **λ**.
* **Representation** − ∑\* = ∑0 ∪ ∑1 ∪ ∑2 ∪……. where ∑p is the set of all possible strings of length p.
* **Example** − If ∑ = {a, b}, ∑\* = {λ, a, b, aa, ab, ba, bb,………..}

### Kleene Closure / Plus

* **Definition** − The set **∑+** is the infinite set of all possible strings of all possible lengths over ∑ excluding λ.
* **Representation** − ∑+ = ∑1 ∪ ∑2 ∪ ∑3 ∪…….

∑+ = ∑\* − { λ }

* **Example** − If ∑ = { a, b } , ∑+ = { a, b, aa, ab, ba, bb,………..}

### Language

* **Definition** − A language is a subset of ∑\* for some alphabet ∑. It can be finite or infinite.
* Language: A (possibly infinite) set of strings all of which are chosen from some Σ∗ A language over Σ need not include strings with all symbols of Σ Thus, a language over Σ is also a language over any alphabet that is a superset of Σ
* **Example** − If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, aa, ba, bb }

**::::::::Basic Operation on Language::::::**

A simple but powerful collection of operations: – Union, Concatenation and Kleene Closure

**Union**

The union of two languages L and M, denoted L ∪ M, is the set of strings that are in either L, or M, or

both.

Example If

L = {001, 10, 111 } and M = {ǫ, 001 } then L ∪ M = {ǫ, 001, 10, 111 }

**Concatenation:**

The concatenation of languages L and M, denoted L.M or just LM , is the set of strings that can be formed by taking any string in L and concatenating it with any string in M.

Example If

L = {001, 10, 111 } and M = {ǫ, 001 } then

L.M = {001, 10, 111, 001001, 10001, 111001 }

**Closure :**

The closure of a language L is denoted L ∗ and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them.

Examples: If

L = { 0, 1 } then L ∗ is all strings of 0 and 1 If L = { 0, 11 } then L ∗ consists of strings of 0 and 1 such that the 1 come in pairs, e.g., 011, 11110 and ǫ. But not 01011 or 101.

***Unit-2***

**:::::::::::Regular Expression** **::::::::::**

A **Regular Expression** can be recursively defined as follows −

* **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
* **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
* **x** is a Regular Expression where **L = {x}**
* If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
  + **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.
  + **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**
  + **R\*** is a Regular Expression corresponding to the language **L(R\*)**where **L(R\*) = (L(R))\***
* If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

Some RE Examples

|  |  |
| --- | --- |
| **Regular Expressions** | **Regular Set** |
| (0 + 10\*) | L = { 0, 1, 10, 100, 1000, 10000, … } |
| (0\*10\*) | L = {1, 01, 10, 010, 0010, …} |
| (0 + ε)(1 + ε) | L = {ε, 0, 1, 01} |
| (a+b)\* | Set of strings of a’s and b’s of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….} |
| (a+b)\*abb | Set of strings of a’s and b’s ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, …………..} |
| (11)\* | Set consisting of even number of 1’s including empty string, So L= {ε, 11, 1111, 111111, ……….} |
| (aa)\*(bb)\*b | Set of strings consisting of even number of a’s followed by odd number of b’s , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, …………..} |
| (aa + ab + ba + bb)\* | String of a’s and b’s of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, …………..} |

**:::::::::::::::::::::Regular Set.::::::::::::::::**

Any set that represents the value of the Regular Expression is called a **Regular Set.**

### Properties of Regular Sets

**Property 1**. *The union of two regular set is regular.*

**Proof** −

Let us take two regular expressions

RE1 = a(aa)\* and RE2 = (aa)\*

So, L1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L2 ={ ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

L1 ∪ L2 = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.......}

(Strings of all possible lengths including Null)

RE (L1 ∪ L2) = a\* (which is a regular expression itself)

**Hence, proved.**

**Property 2.** *The intersection of two regular set is regular.*

**Proof** −

Let us take two regular expressions

RE1 = a(a\*) and RE2 = (aa)\*

So, L1 = { a,aa, aaa, aaaa, ....} (Strings of all possible lengths excluding Null)

L2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

L1 ∩ L2 = { aa, aaaa, aaaaaa,.......} (Strings of even length excluding Null)

RE (L1 ∩ L2) = aa(aa)\* which is a regular expression itself.

**Hence, proved.**

**Property 3.** *The complement of a regular set is regular.*

**Proof** −

Let us take a regular expression −

RE = (aa)\*

So, L = {ε, aa, aaaa, aaaaaa, .......} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L’ = {a, aaa, aaaaa, .....} (Strings of odd length excluding Null)

RE (L’) = a(aa)\* which is a regular expression itself.

**Hence, proved.**

**Property 4.** *The difference of two regular set is regular.*

**Proof** −

Let us take two regular expressions −

RE1 = a (a\*) and RE2 = (aa)\*

So, L1 = {a, aa, aaa, aaaa, ....} (Strings of all possible lengths excluding Null)

L2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

L1 – L2 = {a, aaa, aaaaa, aaaaaaa, ....}

(Strings of all odd lengths excluding Null)

RE (L1 – L2) = a (aa)\* which is a regular expression.

**Hence, proved.**

**Property 5.** *The reversal of a regular set is regular.*

**Proof** −

We have to prove **LR** is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

LR = {10, 01, 11, 01}

RE (LR) = 01 + 10 + 11 + 10 which is regular

**Hence, proved.**

**Property 6.** *The closure of a regular set is regular.*

**Proof** −

If L = {a, aaa, aaaaa, .......} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)\*

L\* = {a, aa, aaa, aaaa , aaaaa,……………} (Strings of all lengths excluding Null)

RE (L\*) = a (a)\*

**Hence, proved.**

**Property 7.** *The concatenation of two regular sets is regular.*

**Proof −**

Let RE1 = (0+1)\*0 and RE2 = 01(0+1)\*

Here, L1 = {0, 00, 10, 000, 010, ......} (Set of strings ending in 0)

and L2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, L1 L2 = {001,0010,0011,0001,00010,00011,1001,10010,.............}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)\*001(0 + 1)\*

Hence, proved.

**:::::::::::Transition Graphs::::::**

Transition graph can be interpreted as a flowchart for an algorithm recognizing a language.

A transition graph consists of three things:

1. A finite set of states, at least one of which is designated the start state and some of which are designated as final states.
2. An alphabet Σ of possible input symbols from which the input strings are formed.
3. A finite set of transitions that show the change of state from the given state on a given input.

A successful path through the transition graph is a series of edges forming a path beginning at the start state and ending at one of the final states.

**A transition graph is defined by a 5-tuple:**

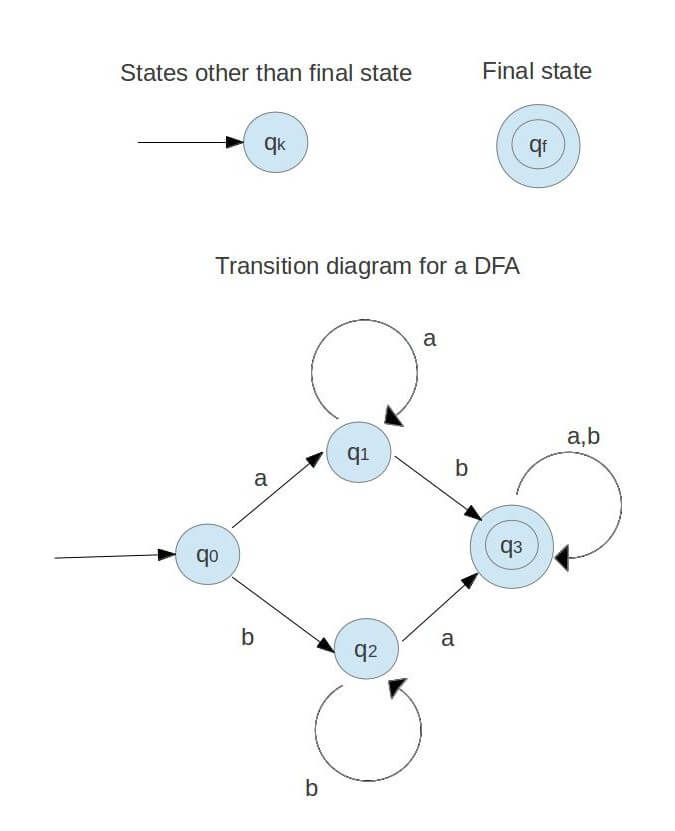
• A finite set of states, Q.

• A finite set of input symbols, Σ.

• A non-empty set set of start states, S ⊆ Q.

• A set of final or accepting states F ⊆ Q.

• A finite set, δ of transitions, (directed edge labels) (u, s, v), where u, v ∈ Q and s ∈ Σ ∗ .

Some of the transition graphs are given below:-

T he transition graph in the figure represents a DFA with at least one a and at least one b.

A transition diagram for DFA, M = ( Q, Σ, δ, F, q0) is a graph defined as follows:

(a) For each state in Q there is a node represented by the circle.  
(b) For each state q in Q each input symbol a in Σ, let δ(q, a) = q’. Then transition diagram has an arc from

q to q’, labelled a.  
(c) There is always an arrow into the start state that is not from any other state.  
(d) Nodes corresponding to accepting states are denoted by a double circle. Rest of the states are

depicted by a single circle.

**:::::::::Finite Automaton can be classified into two types ::::::::**

* Deterministic Finite Automaton (DFA)
* Non-deterministic Finite Automaton (NDFA / NFA)

## :::::::::Deterministic Finite Automaton (DFA) :::::::

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

## Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* **Q** is a finite set of states.
* **∑** is a finite set of symbols called the alphabet.
* **δ** is the transition function where δ: Q × ∑ → Q
* **q0** is the initial state from where any input is processed (q0 ∈ Q).
* **F** is a set of final state/states of Q (F ⊆ Q).

## Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram**.

* The vertices represent the states.
* The arcs labeled with an input alphabet show the transitions.
* The initial state is denoted by an empty single incoming arc.
* The final state is indicated by double circles.

### Example

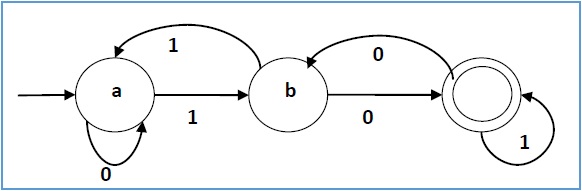
### Let a deterministic finite automaton be →

* Q = {a, b, c},
* ∑ = {0, 1},
* q0 = {a},
* F = {c}, and

Transition function δ as shown by the following table −

|  |  |  |
| --- | --- | --- |
| **Present State** | **Next State for Input 0** | **Next State for Input 1** |
| **a** | A | b |
| **b** | C | a |
| **c** | B | c |

Its graphical representation would be as follows −



# :::::::Non-deterministic Finite Automaton::::::

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* **Q** is a finite set of states.
* **∑** is a finite set of symbols called the alphabets.
* **δ** is the transition function where δ: Q × ∑ → 2Q

(Here the power set of Q (2Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)

* **q0** is the initial state from where any input is processed (q0 ∈ Q).
* **F** is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of an NDFA: (same as DFA)

An NDFA is represented by digraphs called state diagram.

* The vertices represent the states.
* The arcs labeled with an input alphabet show the transitions.
* The initial state is denoted by an empty single incoming arc.
* The final state is indicated by double circles.

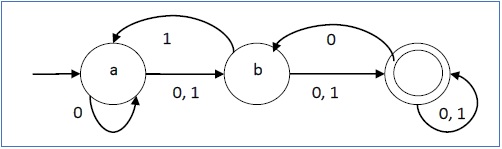
**Example**

Let a non-deterministic finite automaton be →

* Q = {a, b, c}
* ∑ = {0, 1}
* q0 = {a}
* F = {c}

The transition function δ as shown below −

|  |  |  |
| --- | --- | --- |
| **Present State** | **Next State for Input 0** | **Next State for Input 1** |
| a | a, b | b |
| b | C | a, c |
| c | b, c | c |

Its graphical representation would be as follows −

## DFA vs NDFA

The following table lists the differences between DFA and NDFA.

|  |  |
| --- | --- |
| **DFA** | **NDFA** |
| The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*. | The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*. |
| Empty string transitions are not seen in DFA. | NDFA permits empty string transitions. |
| Backtracking is allowed in DFA | In NDFA, backtracking is not always possible. |
| Requires more space. | Requires less space. |
| A string is accepted by a DFA, if it transits to a final state. | A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state. |

## :::::NFA to DFA Conversion::::::

## Problem Statement

Let **X = (Qx, ∑, δx, q0, Fx)** be an NDFA which accepts the language L(X). We have to design an equivalent DFA **Y = (Qy, ∑, δy, q0, Fy)** such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA −

## Algorithm

**Input** − An NDFA

**Output** − An equivalent DFA

**Step 1** − Create state table from the given NDFA.

**Step 2** − Create a blank state table under possible input alphabets for the equivalent DFA.

**Step 3** − Mark the start state of the DFA by q0 (Same as the NDFA).

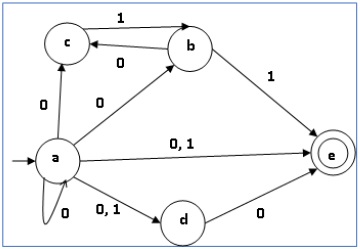
**Step 4** − Find out the combination of States {Q0, Q1,... , Qn} for each possible input alphabet.

**Step 5** − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

**Step 6** − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

## Example

Let us consider the NDFA shown in the figure below.

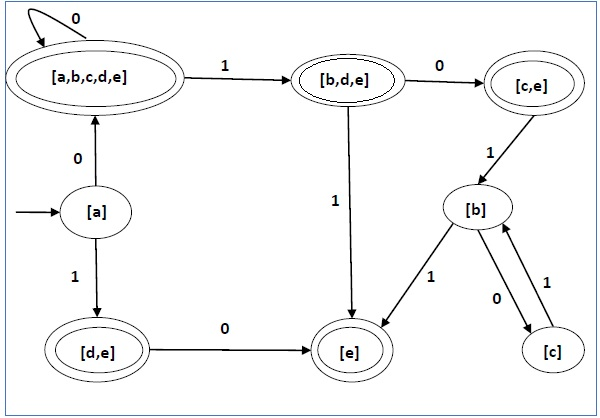


|  |  |  |
| --- | --- | --- |
| **q** | **δ(q,0)** | **δ(q,1)** |
| a | {a,b,c,d,e} | {d,e} |
| b | {c} | {e} |
| c | ∅ | {b} |
| d | {e} | ∅ |
| e | ∅ | ∅ |

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

|  |  |  |
| --- | --- | --- |
| **q** | **δ(q,0)** | **δ(q,1)** |
| [a] | [a,b,c,d,e] | [d,e] |
| [a,b,c,d,e] | [a,b,c,d,e] | [b,d,e] |
| [d,e] | [e] | ∅ |
| [b,d,e] | [c,e] | [e] |
| [e] | ∅ | ∅ |
| [c, e] | ∅ | [b] |
| [b] | [c] | [e] |
| [c] | ∅ | [b |

The state diagram of the DFA is as follows −



# ::::::::Pumping Lemma:::::::::

### Theorem

Let L be a regular language. Then there exists a constant **‘c’** such that for every string **w** in **L** −

**|w| ≥ c**

We can break **w** into three strings, **w = xyz**, such that −

* |y| > 0
* |xy| ≤ c
* For all k ≥ 0, the string xykz is also in L.

## Applications of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

* If **L** is regular, it satisfies Pumping Lemma.
* If **L** does not satisfy Pumping Lemma, it is non-regular.

## Method to prove that a language L is not regular

* At first, we have to assume that **L** is regular.
* So, the pumping lemma should hold for **L**.
* Use the pumping lemma to obtain a contradiction −
  + Select **w** such that **|w| ≥ c**
  + Select **y** such that **|y| ≥ 1**
  + Select **x** such that **|xy| ≤ c**
  + Assign the remaining string to **z.**
  + Select **k** such that the resulting string is not in **L.**

**Hence L is not regular.**

**Problem**

Prove that **L = {aibi | i ≥ 0}** is not regular.

***Solution*** −

* At first, we assume that **L** is regular and n is the number of states.
* Let w = *anbn*. Thus |w| = 2n ≥ n.
* By pumping lemma, let w = xyz, where |xy| ≤ n.
* Let x = ap, y = aq, and z = arbn, where p + q + r = n, p ≠ 0, q ≠ 0, r ≠ 0. Thus |y| ≠ 0.
* Let k = 2. Then xy2z = apa2qarbn.
* Number of as = (p + 2q + r) = (p + q + r) + q = n + q
* Hence, xy2z = an+q bn. Since q ≠ 0, xy2z is not of the form anbn.
* Thus, xy2z is not in L. Hence L is not regular.

**:::::::::::::Regular Languages & Closure Properties::::::::::::**

A language is regular if it can be expressed in terms of regular expression.  
   
**Closure Properties of Regular Languages**

**Union :**

 If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular. For example, L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}  
L3 = L1 ∪ L2 = {an ∪ bn | n ≥ 0} is also regular.

**Intersection :**

If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,  
L1= {am bn | n ≥ 0 and m ≥ 0} and L2= {am bn ∪ bn am | n ≥ 0 and m ≥ 0}  
L3 = L1 ∩ L2 = {am bn | n ≥ 0 and m ≥ 0} is also regular.

**Concatenation :**

 If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For

example,  
 L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}  
 L3 = L1.L2 = {am . bn | m ≥ 0 and n ≥ 0} is also regular.

**Kleene Closure :**

If L1 is a regular language, its Kleene closure L1\* will also be regular. For example,  
L1 = (a ∪ b)  
L1\* = (a ∪ b)\*

**Complement :** If L(G) is regular language, its complement L’(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings. For example,  
L(G) = {an | n > 3}  
L’(G) = {an | n <= 3}  
  
**Note :** Two regular expressions are equivalent if languages generated by them are same. For example, (a+b\*)\* and (a+b)\* generate same language. Every string which is generated by (a+b\*)\* is also generated by (a+b)\* and vice versa.

### How to solve problems on regular expression and regular languages?

* **Question 1 :** Which one of the following languages over the alphabet {0,1} is described by the regular expression?  
  (0+1)\*0(0+1)\*0(0+1)\*  
  (A) The set of all strings containing the substring 00.  
  (B) The set of all strings containing at most two 0’s.  
  (C) The set of all strings containing at least two 0’s.  
  (D) The set of all strings that begin and end with either 0 or 1.  
   **Solution :**

Option A says that it must have substring 00. But 10101 is also a part of language but it does not contain 00 as substring. So it is not correct option.  
Option B says that it can have maximum two 0’s but 00000 is also a part of language. So it is not correct option.  
Option C says that it must contain atleast two 0. In regular expression, two 0 are present. So this is correct option.  
Option D says that it contains all strings that begin and end with either 0 or 1. But it can generate strings which start with 0 and end with 1 or vice versa as well. So it is not correct.  
   
**Question 2 :** Which of the following languages is generated by given grammar?  
S -> aS | bS | ∊  
(A) {an bm | n,m ≥ 0}  
(B) {w ∈ {a,b}\* | w has equal number of a’s and b’s}  
(C) {an | n ≥ 0} ∪ {bn | n ≥ 0} ∪ {an bn | n ≥ 0}  
(D) {a,b}\*

**Solution :**

 Option (A) says that it will have 0 or more a followed by 0 or more b. But S -> bS => baS => ba is also a part of language. So (A) is not correct.  
Option (B) says that it will have equal no. of a’s and b’s. But But S -> bS => b is also a part of language. So (B) is not correct.  
Option (C) says either it will have 0 or more a’s or 0 or more b’s or a’s followed by b’s. But as shown in option (A), ba is also part of language. So (C) is not correct.  
Option (D) says it can have any number of a’s and any numbers of b’s in any order. So (D) is correct.  
   
**Question 3 :**The regular expression 0\*(10\*)\* denotes the same set as  
(A) (1\*0)\*1\*  
(B) 0 + (0 + 10)\*  
(C) (0 + 1)\* 10(0 + 1)\*  
(D) none of these  
 **Solution :**

 Two regular expressions are equivalent if languages generated by them are same.  
Option (A) can generate all strings generated by 0\*(10\*)\*. So they are equivalent.  
Option (B) string null can not generated by given languages but 0\*(10\*)\* can. So they are not equivalent.  
Option (C) will have 10 as substring but 0\*(10\*)\* may or may not. So they are not equivalent.