Lecture 1

Introduction to Theory of Computation, Mathematical Notions and Terminology

Lecture-1	**PO 1**

In this lecture, we will start our discussion with an overview of the different areas in the theory of computation that we present in this course. Following that, we will review some mathematical concepts that you will need later in this course. You may consider these mathematical concepts as a prerequisite of this course.

1.1 Automata, Computability and Complexity

In theoretical computer science, the theory of computation is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major areas: automata theory, computability theory and computational complexity theory.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine. We will study all the models in detail later during our discussions.

Now, we will introduce the different fields of Theory of computation in reverse order for your better understanding.

1.1.1 Complexity Theory

Computer problems come in different varieties; some are easy, and some are hard. Considering two problems, the first one to arrange a list of numbers in ascending order and the

second one to find a schedule of classes for the entire university to satisfy some reasonable constraints. 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.

In one important achievement of complexity theory thus far, researchers have discovered an elegant scheme for classifying problems according to their computational difficulty. 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. Cryptography 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.

1.1.2 Computability Theory

The problem of determining whether a mathematical statement is true or false falls out of the realm of computers and no computer algorithm can perform this task.These type of problems encouraged 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.

1.1.3 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.

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 nontheoretical areas of computer science.

1.2 Mathematical Notations and Terminologies

In this section, we will discuss the basic mathematical objects, tools, and notation that we expect to use during our study.

1.2.1 Sets

A collection of objects presented as a single unit is called a ***set***. Each object in a set is called a ***set element*** or ***set member*** and always written within a pair of braces. The order, sequence and repetition of set elements in a set doesn't matter. A set is represented using the ***set builder notation*** or ***roster notation***.

Example 1.1: A set of natural numbers equal to or less than 5 can be represented: in roster notation as $\{1, 2, 3, 4, 5\}$ and in set builder notation as $\{x | x \in N, x \leq 5\}$

Example 1.2: A set of even integer numbers greater than -3 and less than 5 can be represented: in roster notation as $\{-2, 0, 2, 4\}$ and in set builder notation as $\{x | x = 2n, n \in Z, -1 \leq x \leq 5\}$

Example 1.3: Prime numbers less than 20 can be represented: in roster notation as $\{2, 3, 5, 7, 11, 13, 17, 19\}$ and in set builder notation as $\{x |\ x \text{ is prime}, x < 20\}$

The symbols $\in$ and $\notin$ denote set membership and nonmembership respectively.

Considering two sets A and B, we can say that A is a ***subset*** of B, denoted as $A \subseteq B$, if every member of A also is a member of B. We can say that A is a ***proper subset*** of B, denoted as $A \subset B$, if A is a subset of B and not equal to B.

Example 1.4: Considering three sets, $A = \{1, 2, a, b\}$, $B = \{1, 2, a, b\}$, $C = \{1, 2, 3, a, b, c\}$. A is a subset of B and written as $A \subseteq B$, whereas A is a proper subset of C and written as $A \subset C$.

Repetition of a set element in a set doesn't matter but if we take it into consideration then it's called a ***multiset*** instead of a set.

Example 1.5: $\{3\}$ is a set but $\{3, 3\}$ is a multiset.

A set that contains finite number of elements is called a ***finite set***, whereas a set that contains infinitely many elements is called an ***infinite***. A set with zero members or no member is called an ***empty set***. A set with exactly one member is called a ***singleton set***. A set with exactly two members is called an ***unordered pair***.

Example 1.6:

$\{1, 2, 3, 4, 5\}$, $\{0, 1, 1, 2, 3\}$, $\{1, 2, 4, 8, 16\}$ are examples of some finite set. Set of ***natural numbers*** $N = \{1, 2, 3, \dots\}$, set of ***integers*** $Z = \{\dots, -2, -1, 0, 1, 2, \dots\}$ are examples of some infinite sets. Empty sets are denoted as $\emptyset$ or $\{\ \}$. $\{1\}$ and $\{a\}$ are two examples of singleton sets. $\{2, 4\}$ and $\{a, b\}$ are two examples of unordered pair.

Various type of operations can be made on sets such as ***union ($\cup$), intersection***

(∩), cartesian product (×), set difference (–), compliment ($^-$) etc. Considering two sets A and B, the ***union*** of A and B, written as $A \cup B$, is the set formed by combining all the elements in A and B into a single set. The ***intersection*** of A and B, written as $A \cap B$, is the set of elements that are common to both A and B. The ***cartesian product*** or ***cross product*** of A and B, written as $A \times B$, is the set of all possible order pairs in which the first element is in A and the second element is in B. The ***set difference*** of A and B, written as A–B, is the set of all elements that are in A but not in B. The ***complement*** of A, written as $\bar{A}$, is the set of all elements that are not in A.

Example 1.7: Let set $A = \{1, 2, 3\}$ and set $B = \{3, 4, 5\}$. Then

$A \cup B = \{\ x \mid x \in a \text{ or } x \in B\ \} = \{1, 2, 3, 4, 5\}$.

$A \cap B = \{x \mid x \in A \text{ and } x \in B\} = \{3\}$.

$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\} = \{(1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5)\}$.

A–$B = \{x \mid x \in A \text{ and } x \notin B\} = \{1, 2\}$ where as B–$A = \{4, 5\}$.

The pictorial representation of a set is called a ***Venn diagram***.An example of it given below.

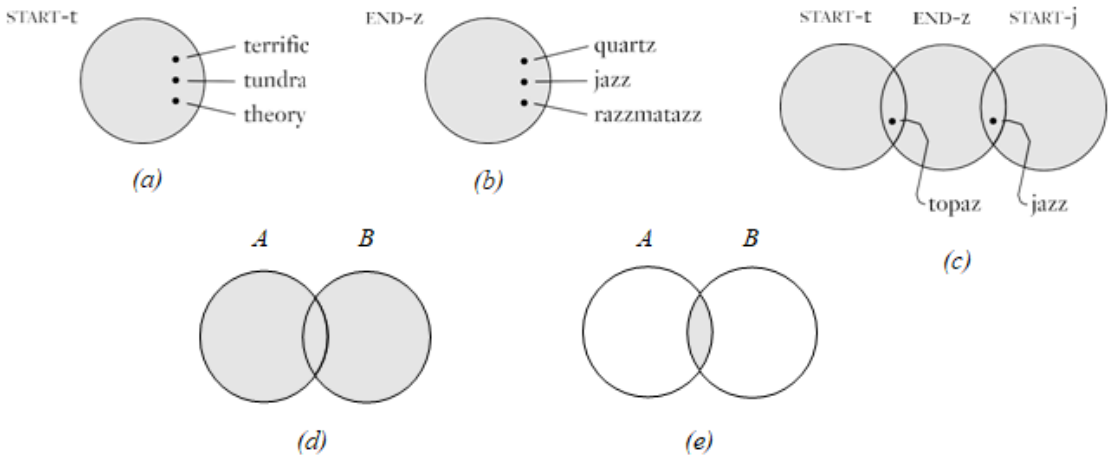

Figure 1.1: Venn diagram for (a) the set of English words starting with "t" (b) the set of English words ending with "z" (c) overlapping circles indicate common elements (d) $A \cup B$ (e) $A \cap B$.

Number of elements present in a set is called its ***cardinality***. Cardinality of a set A is denoted as $| A |$. The ***power set*** of A is the set of all possible subsets of A. If A is the set $\{0, 1\}$, then the power set of A denoted as $P(A)$ or 2^A is the set $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$. The cardinality of power set of a set A is $2^{|A|}$. As in this example, $| A |$ is 2, $P(A)$ is 2^2 i.e. 4.

1.2.2 Sequences and Tuples

A ***sequence*** is an ordered list of elements, which can also be infinite (e.g., the sequence composed of the real numbers). Unlike sets, the order and repetition matters in a se-

quence. Since order of elements matter, the sequence $(5, 6, 7)$ is different from the sequence $(5, 7, 6)$ or from the $(7, 6, 5)$ one. You may also have repeated elements sequence such as $(5, 5, 5)$ which is different from $(5, 5)$ and (5), whereas these three are same in case of a set. You may also have an empty sequence ( ).

Similar to sets, sequences may be finite or infinite. Finite sequences often are often called ***tuples***. A sequence with k elements is a ***k-tuple***. For example, $(7, 21, 57)$ is a 3-tuple. A 2-tuple is also called an ***ordered pair***. $(0, 1)$, (a, b) and $(21, 57)$ are some examples of ordered pair. The set of all ordered pairs whose elements are 0's and 1's is $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$.

A sequence is an ordered list whose elements are all of the same type, similar to an array. In contrast to arrays (and other standard container types), however, sequences are immutable, similar to strings in Java and other languages. There are operators to determine the length of a sequence s, to retrieve a single element, to select arbitrary subsequences, and to concatenate two sequences etc.

Tuples are also useful in programming languages as well. For example, if a function receives several parameters, then it is same as to receiving a tuple of values as a single parameter. Similarly, a function could easily return multiple values by returning a tuple of values.

1.2.3 Functions and Relations

A ***function*** is an object that sets input to output relationship, i.e. for every input there is an output. We will see many ways to think about functions, but there are always three main parts: the input, the relationship and the output.
The function always maps the input to output using a relationship and hence called a ***mapping***. In every function, the same input always produces the same output and if $f(a) = b$, we say that f maps a to b. For example, the absolute value function abs as described below, takes a number x as input and returns x if x is positive and $-x$ if x is negative. $f(x) = abs(x) = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$

Domain (D) of a function is the set of all the possible set of inputs to the function is possible. ***Range*** (R) is the set of all the possible outcomes for all possible inputs. The notation for saying that f is a function defined from the set 'A' to the set 'B' is $f : A \to B$, where the set A is called the domain of the function and the set B is called the co-domain of the function. The range of the function is subset of the co-domain set B. The elements in A which are mapped to the elements in B are called the pre-image while the elements in B which are having pre-image in A are called the image.

If every element of B has a pre-image in A then the function is called ***onto function***. If one or more elements of B does not have pre-image in A then the function is called ***into function***.
Example 1.8: Consider the function $f : A \to B$ where $A = \{2, 3, 4\}$ and $B = \{4, 9, 16\}$ and $f(x) = x^2$ then 2 have an image 4, 3 have an image 9 and 4 have an image 16 and

the function here is the onto function.

If the domain of a function is the Cartesian product of k number of sets i.e. $A_1 \times A_2 \times \ldots \times A_k$ then the function is called a ***k-ary function*** where k is called the ***arity*** of the function. For $k = 1$ it is ***unary*** function, $k = 2$ it is called ***binary*** function and so on. Let a function $f : A \times B \to C$ where $A = \{2, 4\}$ and $B = \{5, 9\}$ and $C = \{7, 9, 11, 13\}$ and $f(x, y) = x + y$. Here f is n example of binary function.

A ***predicate*** or ***property*** is a function whose range is $TRUE, FALSE$. Considering as example, the even be a property that checks a number is even or odd. It is $TRUE$ if the input is an even number and $FALSE$ if the input is an odd number. Thus $even(2) = TRUE$ and $even(3) = FALSE$.

A ***relation*** in mathematics defines the relationship between two different sets of information. If two sets are considered, the relation between them will be established if there is a connection between the elements of two or more non-empty sets.

In the morning assembly at schools, students are supposed to stand in a queue in ascending order of the heights of all the students. This defines an ordered relation between the students and their heights.

Therefore, mathematically we can say, A relation R from a set 'A' to a set 'B' is any subset of $A \times B$ and is represented as $R : A \to B$. Ex. let A=$\{a, b, c\}$ and B=$\{1, 2, 3\}$ then any subset of A×B becomes the relation from A to B. As there are 2^9 subset of $A \times B$ is possible, number of relations possible is 2^9. Some of those relations are R_1=$\{(a, 2), (b, 1)\}$ R_2=$\{(a, 3), (b, 2)\}$, R_3=$\{(c, 2), (b, 1)\}$ etc.

If a relation has k tuples as its domain then it is called a ***k-ary relation*** and the common case of $2 - aryrelation$ is called the ***binary relation***. A special type of binary relation R is called an ***equivalence relation***, if R satisfies the following three conditions. R is ***reflexive relation*** i.e $\forall x$, xRx. R is ***symmetric relation*** i.e if xRy then yRx $\forall x, y \in A$. R is ***transitive relation*** i.e if xRy and yRz then xRz $\forall x, y, z \in A$. The relation "is parallel to" defined on the set of straight lines, the relation "is congruent to" defined on the set of triangles are examples of equivalence relations.

Example 1.9: Define an equivalence relation on the set of natural numbers, written as $\equiv_7$.

For i, j ∈ N , let $i \equiv_7 j$, if $i - j$ is a multiple of 7. This is an equivalence relation because it satisfies the following three conditions. First, it is reflexive, as $i - i = 0$, which is a multiple of 7. Second, it is symmetric, as $i - j$ is a multiple of 7 if $j - i$ is a multiple of 7. Third, it is transitive, as whenever $i - j$ is a multiple of 7 and $j - k$ is a multiple of 7, then $i - k = (i - j) + (j - k)$ is the sum of two multiples of 7 and hence a multiple of 7, too.

1.2.4 Graphs

An ***undirected graph***, or simply a graph $G(V, E)$, is a set of points with lines connecting some of the points. The points are called ***nodes*** or ***vertices***(V), and the lines are called ***edges***(E), as shown in the Figure 1.2.

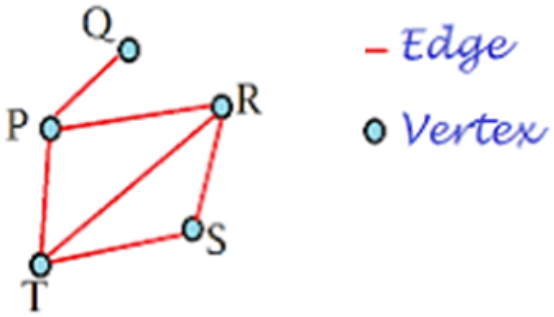

Figure 1.2: Diagram showing an undirected graph.

The number of edges at a particular node is the ***degree*** of that node. In the above figure the nodes P, Q, R, S and T have the degree 3, 1, 3, 2, and 3 respectively. No more than one edge is allowed between any two nodes. An edge from a node to itself is allowed and is called a ***self-loop***.

Sometimes, for convenience, the nodes and/or edges of a graph are labeled, which then is called a ***labeled graph***. The Figure 1.3 depicts a graph whose nodes and edges are labeled.

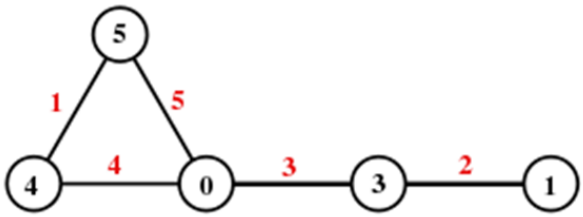

Figure 1.3: Diagram showing a labeled graph.

A ***subgraph*** S of a graph G is a graph whose vertex set $V(S)$ is a subset of the vertex set $V(G)$ that is $V(S) \subseteq V(G)$, and whose edge set $E(S)$, is a subset of the edge set $E(G)$, that is $E(S) \subseteq E(G)$. Generally, a subgraph is a graph within a larger graph. For example, the following graph S in Figure 1.4 is a subgraph of G.

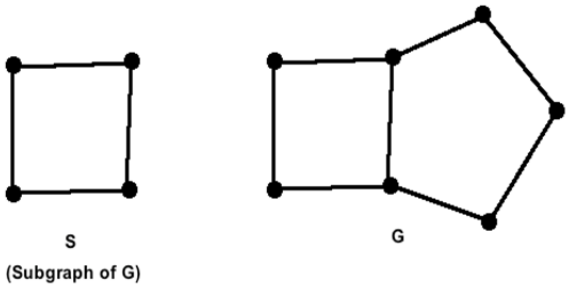

Figure 1.4: Diagram showing a subgraph.

A ***path*** in a graph is a sequence of nodes connected by edges. A ***simple path*** is a path that doesn't repeat any nodes. A graph is ***connected*** if every two nodes have a path between them. A path is a ***cycle*** if it starts and ends in the same node. A simple cycle is one that contains at least three nodes and repeats only the first and last nodes.

A graph is a ***tree*** if it is connected and has no simple cycles. A tree may contain a specially designated node called the ***root***. The nodes of degree 1 in a tree, other than the root, are called the ***leaves*** of the tree. Structure of a tree is shown in Figure 1.5.

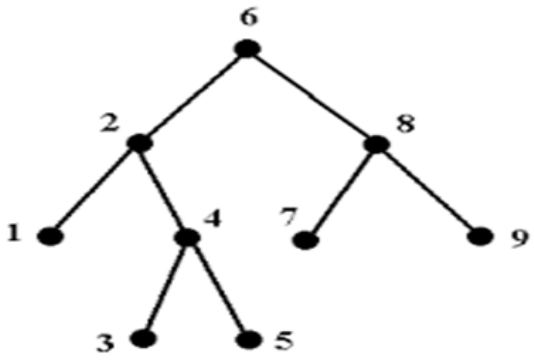

Figure 1.5: Diagram showing a tree.

A ***directed graph*** has arrows instead of lines, as shown in the Figure 1.6. The number of arrows pointing from a particular node is the ***outdegree*** of that node, and the number of arrows pointing to a particular node is the ***indegree***.

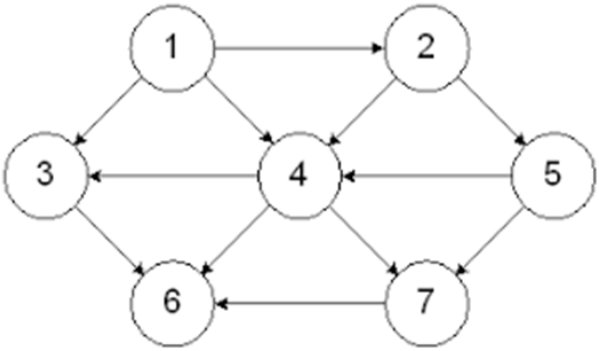

Figure 1.6: Diagram showing a directed graph.

A path in which all the arrows point in the same direction as its steps is called a directed path. A directed graph is ***strongly connected*** if a directed path connects every two nodes.

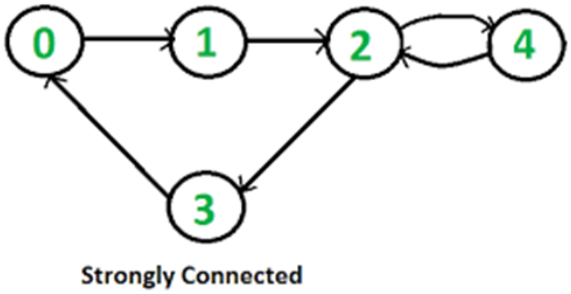

Figure 1.7: Diagram showing a strongly connected graph.

1.2.5 Strings and Languages

A non empty finite set of symbols is called ***alphabate*** . Generally capital Greek letters Σ and Γ are used to designate alphabets and a typewriter font for symbols. Some examples of alphabets are $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b, c\}$, etc.

A ***string*** over an alphabet is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas. If Σ_1 = {0,1}, then "01001" is a string over Σ_1. If $\Sigma_2 = \{a, b, c, ..., z\}$, then "abracadabra" is a string over Σ_2.

The number of symbols in a string is called its ***length***, if w is a string then its length is represented by $\mid w \mid$. The string of length zero is called the ***empty string*** and is written $\in$.

Reverse of a string, is the string obtained by writing a string w in the opposite order and is represented as w^R .

String z is a ***substring*** of string w if z appears consecutively within w. For example, "cad" is a substring of string "dfcadjhgi".

If we have string x of length m and string y of length n, the ***concatenation*** of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1x_2x_3 \ldots x_m y_1y_2y_3 \ldots .y_n$.

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 $x \neq y$.

A ***language*** is a set of strings. A language is prefix-free if no member is a proper prefix of another member.

1.2.6 Boolean Logic

Boolean logic is a mathematical system built around the two values ***TRUE*** and ***FALSE***. The values TRUE and FALSE are called the ***Boolean values*** and are often represented by the values 1 and 0.

Boolean values can be manipulated by using ***Boolean operations***,the simplest Boolean operation is the ***negation*** or ***NOT*** operation, designated with the symbol $\neg$. The negation of a Boolean value is the opposite value. Thus $\neg 0 = 1$ and $\neg 1 = 0$. The ***conjunction*** or ***AND*** operation is denoted with the symbol $\wedge$. The conjunction of two Boolean values is 1 if both of those values are 1. The ***disjunction*** or ***OR*** operation is designated with the symbol $\vee$. The disjunction of two Boolean values is 1 if either of those values is 1. These information can be summarized is as follows:

$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$\neg 0 = 1$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$\neg 1 = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	

The ***Boolean operations*** are used for combining simple statements into more complex Boolean expressions. For example, if P is the Boolean value representing the truth of the statement "the sun is shining" and Q represents the truth of the statement "today

is Monday", then $P \wedge Q$ represent the truth value of the statement "the sun is shining and today is Monday" and similarly $P \vee Q$ represent "the sun is shining or today is Monday" . The values P and Q are called the operands of the operation.

The ***exclusive or***, or ***XOR***, operation is designated by the $\oplus$ symbol and is 1 if either but not both of its two operands same. The ***equality*** operation, written with the symbol $\leftrightarrow$, is 1 if both of its operands have the same value. Finally, the ***implication*** operation is designated by the symbol $\rightarrow$ and is 0 if its first operand is 1 and its second operand is 0; otherwise, $\rightarrow$ is 1. We summarize this information as follows:

$0 \oplus 0 = 0$	$0 \leftrightarrow 0 = 1$	$0 \rightarrow 0 = 1$
$0 \oplus 1 = 1$	$0 \leftrightarrow 1 = 0$	$0 \rightarrow 1 = 1$
$1 \oplus 0 = 1$	$1 \leftrightarrow 0 = 0$	$1 \rightarrow 0 = 0$
$1 \oplus 1 = 0$	$1 \leftrightarrow 1 = 1$	$1 \rightarrow 1 = 1$

All Boolean operations can be expressed in terms of the AND and NOT operations, as the identities shown below. The two expressions in each row are equivalent. Each row expresses the operation in the left-hand column in terms of AND and NOT operations in its right-hand column.

$P \vee Q$	$=$	$\neg(\neg P \wedge \neg Q)$
$P \rightarrow Q$	$=$	$\neg P \vee Q$
$P \leftrightarrow Q$	$=$	$(P \rightarrow Q) \wedge (Q \rightarrow P)$
$P \oplus Q$	$=$	$\neg(P \leftrightarrow Q)$

The ***distributive law*** for AND and OR is
$P \wedge (Q \vee R)$ equals $(P \wedge Q) \vee (P \wedge R)$, and its dual
$P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$.