CM1025: Fundamentals of Computer Science

Summary

Arjun Muralidharan

1st March 2020

Contents

List of Figures

List of Tables

List of Algorithms

1 Propositional Logic

Learning Outcomes

- ✓ Define propositional logic and learn some of its properties
- ✓ Give some examples of domains where propositional logic is used
- ✓ Defining a proposition in mathematical context and distinguish examples of sentences considered as propositions and other that are not propositions
- ✓ Learn how propositional variables help to simplify notations
- ✓ Learn how we can build a truth table and give an example of how it works
- ✓ Practice how to build compound statements using logical operators and take into consideration the order of precedence of the operators
- ✓ Define truth sets and learn some examples of truth sets for some compound propositions
- ✓ Define what is an implication and equivalence, what are their properties and build their truth table
- ✓ Learn some important laws of propositional logic including De Morgan's laws, and practice their use in building a reasoning, and proving equivalence

1.1 Connectives

A **proposition** is a declarative sentence that is either true or false, but not both. We use **propositional** or **sentential** variables such as $p, q, r, s \ldots$ to represent propositions. The **truth value of a proposition** is denoted by **T** or **F**. The simplest form of a proposition is an **atomic proposition**. Propositions can be placed in relation to one another using **connectives**.

1.1.1 Negation

Let p be a proposition. The *negation* of p is stated as

$$\neg \mathbf{p}$$

and proposes that "It is not the case that p".

The proposition $\neg p$ is read "not p". The truth value of the negation of $p, \neg p$, is the opposite of the truth value of p.

1.1.2 Conjunction

Let p and q be propositions. The *conjunction* of p and q, is denoted as

$$\mathbf{p} \wedge \mathbf{q}$$

and proposes that "p and q".

The *conjunction* $p \wedge q$ is true when both p and q are true and is false otherwise.

1.1.3 Disjunction

Let p and q be propositions. The *disjunction* of p and q, denoted as

$$\mathbf{p \vee q}$$

is the statement "p or q".

The *disjunction* $p \vee q$ is false when both p and q are false and is true otherwise. This is an **inclusive or**, which means that the disjunction is true if at least one of the propositions is true, or both are true.

1.1.4 Conditional Statements & Implication

Let p and q be propositions. The *conditional statement*

$$p \rightarrow q$$

is the proposition "if p, then q." The conditional statement $p \rightarrow q$ is false when p is true and q is false, and true otherwise. In the implication $p \rightarrow q$, p is the *hypothesis* (or *antecedent* or *premise*) and q is called the *conclusion* (or *consequence*).

The implication does not express causality in discrete mathematics. It only covers the truth value of the statement. It can also be read as:

1. "p is sufficient for q "
2. "a necessary condition for q is p "
3. "q unless $\neg p$ "
4. "q whenever p "

1.1.5 Exclusive Or

Let p and q be propositions. The *exclusive or* of p and q, denoted as

$$\mathbf{p \oplus q}$$

is the statement "either p or q, but not both".

The *exclusive or* $p \oplus q$ is true when exactly one of p and q is true and false otherwise. In other words, p and q may never have the same truth value.

1.1.6 Biconditional

Let p and q be propositions. The *biconditional statement*

$$\mathbf{p \leftrightarrow q}$$

is the proposition "p if and only if q".

The biconditional statement $p \leftrightarrow q$ is true when p and q have the same truth values, and is false otherwise.

1.2 Truth Tables

A truth table evaluates the truth values for propositions and connectives systematically. When given a compound proposition, a truth table can be used to evaluate the atomic propositions step-by-step and arrive at the correct truth value for the compound proposition.

Table 1. *Truth table for basic connectives*

p	**q**	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \rightarrow q$	$p \leftrightarrow q$
T	**T**	T	T	F	T	T
T	**F**	F	T	T	F	F
F	**T**	F	T	T	T	F
F	**F**	F	F	F	T	T

Truth values can also be denoted using binary bits, i.e. T is 1 and F is 0. When constructing a truth table, the number of rows is 2^n where n is the number of propositions involved. So compound proposition involving 3 atomic propositions p, q, r has $2^3 = 8$ rows. Further, The the first column (e.g. p) is constructed with all true values for the first half, and false values for the second half, the second (q) is true for the first 2 rows and false for next two rows, and the third is true for the first row and false for the second row of the table, and then this pattern repeats down the table for p, q and r.

The universal and existential quantifications $\forall$ and $\exists$ are described in subsection 2.1.

1.3 Precedence of Logical Operators

Logical operators have an order of precedence as follows.

$$\forall \quad \exists \quad \neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

1.4 Propositional Equivalences

1. **Tautology:** A compound proposition that is always true, no matter the truth values of the individual propositions

2. **Contradiction:** A compound proposition that is always false, no matter the truth values of the individual propositions. This is called **inconsistent**.

3. **Contingency:** A compound proposition that is true for at least one scenario of truth values. This is called **consistent**. All tautologies are consistent by definition.

1.5 Logical Equivalence

Compound propositions that have the same truth value in all possible cases are called **logically equivalent**. In other words, if $p \leftrightarrow q$ is a tautology, then $p \equiv q$.

Logical equivalence can be proved using truth tables.

1.6 De Morgan's Laws

$$\neg(p \wedge q) \equiv \neg p \vee \neg q \qquad \text{(De Morgan's First Law)}$$

$$\neg(p \vee q) \equiv \neg p \wedge \neg q \qquad \text{(De Morgan's Second Law)}$$

1.7 Special equivalences

1.7.1 Conjunction-Disjunction

The **conditional-disjunction equivalence** allows us to replace conditional statements with disjunctions. This is especially useful to convert a conditional to a form that can then be transformed using De Morgan's laws.

$$p \to q \equiv \neg p \vee q$$

1.7.2 Contrapositive

The **contrapositive** of a conditional statement is equivalent to the original conditional statement and is defined as follows.

$$p \to q \equiv \neg q \to \neg p$$

1.8 Summary of Logical Equivalences

Table 2. *Summary of Equivalence Laws*

Name	Equivalence
Identity Laws	$p \wedge \mathbf{T} \equiv p$
	$p \vee \mathbf{F} \equiv p$
Domination Laws	$p \vee \mathbf{T} \equiv \mathbf{T}$
	$p \wedge \mathbf{F} \equiv \mathbf{F}$
Idempotent Laws	$p \vee p \equiv p$
	$p \wedge p \equiv p$
Double Negation Law	$\neg(\neg p) \equiv p$
Commutative Laws	$p \vee q \equiv q \vee p$
	$p \wedge q \equiv q \wedge p$
Associative Laws	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Distributive Laws	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
De Morgan's Laws	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Absorption Laws	$p \vee (p \wedge q) \equiv p$
	$p \wedge (p \vee q) \equiv p$
Negation Laws	$p \vee \neg p \equiv \mathbf{T}$
	$p \wedge \neg p \equiv \mathbf{F}$

1.8.1 Logical Equivalences Involving Conditional Statements

$$p \to q \equiv \neg p \lor q$$
$$p \to q \equiv \neg q \to \neg p$$
$$p \lor q \equiv \neg p \to q$$
$$p \land q \equiv \neg(p \to \neg q)$$
$$\neg(p \to q) \equiv p \land \neg q$$
$$(p \to q) \land (p \to r) \equiv p \to (q \land r)$$
$$(p \to q) \land (q \to r) \equiv (p \lor q) \to r$$
$$(p \to q) \lor (p \to r) \equiv p \to (q \lor r)$$
$$(p \to r) \lor (q \to r) \equiv (p \land q) \to r$$
$$(p \lor \neg r) \to (\neg q \land r)$$

1.8.2 Logical Equivalences Involving Biconditional Statements

$$p \leftrightarrow q \equiv (p \to q) \land (q \to p)$$
$$p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q$$
$$p \leftrightarrow q \equiv (p \land q) \lor (\neg p \land \neg q)$$
$$\neg(p \leftrightarrow q) \equiv p \leftrightarrow \neg q$$

2 Predicate Logic

The **predicate** of a statement is the property assigned to a specific variable. In the statement "x is greater than 3", x is the variable and "is greater than 3" is the predicate. This statement can be noted as the **propositional function** $P(x)$ Predicate statements that describe valid input are **preconditions**, and statements describing valid output are **postconditions**.

2.1 Quantifiers

Quantification expresses to which extent a propositional function $P(x)$ is true over a range of elements.

2.1.1 Universal Quantification

The *universal quantification* is stated as

$$\forall x P(x)$$

and states "$P(x)$ for all values of x in the domain".

2.1.2 Existential Quantification

The *existential quantification* is stated as

$$\exists x P(x)$$

and states "There exists an element x in the domain such that $P(x)$".

The truth values of universal and existential quantifications are shown in Table 3.

2.1.3 Uniqueness Quantification

The *uniqueness quantifier* is stated as

$$\exists! x P(x)$$

and states "There exists one and only one element x in the domain such that $P(x)$ "

Table 3. *Truth values of quantifiers*

Statement	*When true?*	*When false?*
$\forall x P(x)$	$P(x)$ is true for every x.	$P(x)$ is false for at least one x.
$\exists x P(x)$	There exists at least an x such that $P(x)$.	$P(x)$ is false for all x.

2.1.4 De Morgan's Laws for Quantifiers

The negation of a universal quantification of a statement $P(x)$ is equivalent to an existential quantification of the negation of $P(x)$. Likewise, the negation of an existential quantification of a statement $P(x)$ is equivalent to the universal quantification of the negation of $P(x)$.

$$\neg \forall x P(x) \equiv \exists x \neg P(x)$$
$$\neg \exists x Q(x) \equiv \forall x \neg Q(x)$$

2.2 Rules of Inference

An **argument** in propositional logic is a sequence of propositions. All but the final proposition in the argument are called *premises* and the final proposition is called the *conclusion*. An argument is **valid** if the truth of all its premises implies that the conclusion is true. That is,

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow q$$

is a tautology where $p_1 \ldots p_n$ are the premises and q is the conclusion. A single premise can in itself be a conditional proposition.

An argument form in propositional logic is a sequence of compound propositions involving propositional variables. An **argument form** is valid no matter which particular propositions are substituted for the propositional variables in its premises, the conclusion is true if the premises are all true. The notation for two premises p and $p \to q$ and its conclusion q is shown below.

$$\begin{array}{ll} & p \\ & p \to q \\ \hline \therefore & q \end{array}$$

Instead of using truth tables, an argument's truth value can be evaluated by using **rules of inference**. The most common rules for logical propositions are listed in Table 4, while rules for quantified statements are shown in Table 5.

Rule	Tautology	Name
$\begin{array}{ll} & p \\ & p \to q \\ \hline \therefore & q \end{array}$	$(p \land (p \to q)) \to q$	Modus ponens
$\begin{array}{ll} & \neg q \\ & p \to q \\ \hline \therefore & \neg p \end{array}$	$(\neg q \land (p \to q)) \to \neg p$	Modus tollens
$\begin{array}{ll} & p \to q \\ & q \to r \\ \hline \therefore & p \to r \end{array}$	$((p \to q) \land (q \to p)) \to (p \to q)$	Hypothetical syllogism
$\begin{array}{ll} & p \lor q \\ & \neg p \\ \hline \therefore & q \end{array}$	$((p \lor q) \land \neg p) \to q$	Disjunctive syllogism
$\begin{array}{ll} & p \\ \hline \therefore & p \lor q \end{array}$	$p \to (p \lor q)$	Addition
$\begin{array}{ll} & p \land q \\ \hline \therefore & p \end{array}$	$((p \land q) \to p)$	Simplification
$\begin{array}{ll} & p \\ & q \\ \hline \therefore & p \land q \end{array}$	$(p) \land (q) \to (p \land q)$	Conjunction
$\begin{array}{ll} & p \lor q \\ & \neg p \lor r \\ \hline \therefore & q \lor r \end{array}$	$(p \lor q) \land (\neg p \lor r) \to (q \lor r)$	Resolution

Table 4. *Rules of inference*

Rule	Name
$\forall x P(x)$ ——— $\therefore \quad P(c)$	Universal Instantiation
$P(c)$ for an arbitrary c ——— $\therefore \quad \forall x P(x)$	Universal Generalization
$\exists x P(x)$ ——— $\therefore \quad P(c)$ for some element c	Existential Instantiation
$P(c)$ for some element c ——— $\therefore \quad \exists x P(x)$	Existential Generalization

Table 5. *Rules of Inference for Quantified Statements*

3 Proof Techniques

3.1 Terminology

Theorem A *theorem* is a formal statement that can be shown to be true.

Axiom An *axiom* is a statement that we assume to be true to

3.2 Proof Methods

A proof is a **valid argument** that shows the truth of a mathematical statement. It uses the *premise*, *axioms*, *theorems* to prove a *conjecture* to be true or false. Most commonly, we prove a statement of the form

$$\forall x P(x) \to Q(x)$$

which is often simplified to omit the universal quantification as

$$p \to q$$

This statement can be proven with various methods.

3.2.1 Direct Proof

A **direct proof** is constructed by *assuming that p is true* and showing that if p is true, q is true. This is done by expanding the definition underlying p and applying it to q.

3.2.2 Proof by Contrapositive

A **proof by contrapositive** uses the fact that

$$p \to q \equiv \neg q \to \neg p$$

to first assume that q is false. If the contrapositive can be shown to be true, then the original statement is also true.

Vacuous Proofs A proof is *vacuous* or *trivial* if e.g. in the statement $p \to q$ we can show that p is false, because $p \to q$ must be true if p is false.

3.3 Proof by Contradiction

We can show that p is true if

$$\neg p \to (r \wedge \neg r)$$

is true for some proposition r. Because the conclusion is false, the hypothesis $\neg p$ must be false for the conditional to be true. Therefore, p is true. This proof works by first assuming $\neg p$ and then constructing the conditional. We assume that p is false and then show that this assumption leads to a contradiction, therefore proving that p is true.

3.4 Proof by Induction

Proof by induction can be used to prove statements that assert that $P(n)$ is true for all positive integers n, where $P(n)$ is a propositional function. To prove this, there are two steps.

Basis Step We verify that $P(1)$ is true. Note that we cannot just assume that $P(1)$ is true, we need to show that it indeed is, by other proof methods. Note that the domain of the axiom to be proven may be restricted (e.g. $k > 3$). In this case, the basis step would be $P(4)$ instead of $P(1)$.

Inductive Step We show that the conditional statement $P(k) \to P(k+1)$ is true for all positive integers k. Here, we *assume* that $P(k)$ is true to show that $P(k+1)$ is true, leading the conditional to be true.

Proof by induction can be stated as a rule of inference:

$$(P(1) \wedge \forall k(P(k) \to P(k+1))) \to \forall n P(n)$$

The general template for proofs by induction is as follows.

1. Express the statement that is to be proved in the form "for all $n \geq b, P(n)$" for a fixed integer b. For statements of the form "$P(n)$ for all positive integers n", let $b = 1$, and for statements of the form "$P(n)$ for all non-negative integers n", let $b = 1$. For some statements of the form $P(n)$, such as inequalities, you may need to determine the appropriate value of b by checking the truth values of $P(n)$ for small values of n.

2. Write out the words "Basis Step". Then show that $P(b)$ is true, taking care that the correct value of b is used. This completes the first part of the proof.

3. Write out the words "Inductive Step" and state, and clearly identify, the inductive hypothesis, in the form "Assume that $P(k)$ is true for an arbitrary fixed integer $k \geq b$."

4. State what needs to be proved under the assumption that the inductive hypothesis is true. That is, write out what $P(k + 1)$ says.

5. Prove the statement $P(k + 1)$ making use of the assumption $P(k)$. (Generally, this is the most difficult part of a mathematical induction proof. Decide on the most promising proof strategy and look ahead to see how to use the induction hypothesis to build your proof of the inductive step. Also, be sure that your proof is valid for all integers k with $k \geq b$, taking care that the proof works for small values of k, including $k = b$.)

6. Clearly identify the conclusion of the inductive step, such as by saying "This completes the inductive step." After completing the basis step and the inductive step, state the conclusion, namely, "By mathematical induction, $P(n)$ is true for all integers n with $n \geq b$".

3.5 Strong Induction

When we cannot use induction to easily prove a result, we can consider using **strong induction**.

To prove that $P(n)$ is true for all positive integers n, wheren $P(n)$ is a propositional function, we complete two steps.

Basis Step We verify that the proposition $P(1)$ is true.

Inductive Step We show that the conditional statement $[P(1) \wedge P(2) \wedge \cdots \wedge P(k)] \rightarrow P(k + 1)$ is true for all positive integers k.

4 Combinatorics

Product Rule Suppose that a procedure can be broken down into a sequence of two tasks. If there are n_1 ways to do the first task and for each of these ways of doing the first task, there are n_2 ways to do the second task; then there are $n_1 n_2$ ways to do the procedure.

Sum Rule If a task can be done either in one of n_1 ways or in one of n_2 ways, where none of the set of n_1 ways is the same as any of the set of n_2 ways, then there are $n_1 + n_2$ ways to do the task.

Subtraction Rule If a task can be done in either n_1 ways or n_2 ways, then the number of ways to do the task is $n_1 + n_2$ minus the number of ways to do the task that are common to the two different ways. This is the same as the **principle of inclusion-exclusion** (see Discrete Mathematics — Set Theory).

4.1 Pigeonhole Principle

If N objects are placed into k boxes, then there is **at least** one box with **at least** $\lceil N/k \rceil$ objects.

Important: When calculating the number of objects N required to satisfy a specific $\lceil N/k \rceil$ outcome, keep in mind that you are rounding up N/k. For example, if $\lceil N/k \rceil = 6$ and $k = 4$, then

$$\begin{aligned} \lceil N/4 \rceil = 6 \Leftrightarrow N/4 &> 5 \\ N &> 20 \\ N &= 21 \end{aligned}$$

Therefore, when solving the inequality, you can add one to the result for N to arrive at the smallest number that will satisfy the desired N/k (if that is what is asked).

4.2 Permutations

A permutation of n different elements is an ordering of the elements such that one element is first, one is second, one is third, and so on.

The number of permutations of n elements is

$$n \cdot (n-1) \cdot \cdot 4 \cdot 3 \cdot 2 \cdot 1 = n!$$

In other words, there are $n!$ different ways of ordering n elements.

The number of permutations of n elements taken r at a time **without repetition** is

$$_nP_r = \frac{n!}{(n-r)!} = n(n-1)(n-2)\cdots(n-r+1)$$

The number or permutations of n elements taken r at a time **with repetition allowed** inclusion

$$n^r$$

Consider a set of n objects that has n_1 of one kind of object, n_2 of a second kind, and so on. The number of **distinguishable permutations** of the n objects is

$$\frac{n!}{n_1! \cdot n_2! \cdot n_3! \cdot \cdots \cdot n_k!} \tag{1}$$

This is equivalent to the number of ways n distinguishable objects can be placed into k boxes.

4.3 Combinations

Combinations consider only the possible sets of objects *regardless* of the order in which the members of the set are arranged.

The number of possible combinations of n elements taken r at a time **without repetition** is

$$ {}_nC_r = \frac{n!}{(n-r)!r!} = \frac{{}_nP_r}{r!} \tag{2}$$

The number of possible combinations of n elements taken r at a time **with repetition** is

$$ {}_nC_r = \frac{(n+r-1)!}{(n-1)!r!} $$

4.4 Binomial Theorem

The number of r-combinations from a set with n elements can be denoted as $\binom{n}{r}$. A **binomial** expression is the sum of two terms. The **binomial theorem** states that given two variables x and y, and a nonnegative integer n,

$$ (x+y)^n = \sum_{j=0}^{n} \binom{n}{j} x^{n-j}y^j = \binom{n}{0} x^n + \binom{n}{1} x^{n-1}y + \cdots + \binom{n}{n-1} xy^{n-1} + \binom{n}{n} y^n $$

Note that the exponent of n decreases with each term as the exponent of y increases with each term. Also note that if x or y is negative, then each alternating term will be negative by moving the negative sign from the negative term to the front (e.g. $x(-y)^3 = -xy^3$).

5 Automata Theory

5.1 Letters and Strings

Alphabet An alphabet Σ is a non-empty set of symbols. $\Sigma = \{0, 1\}$ is the binary alphabet.

Strings & Lengths A string $w = w_1w_2 \ldots w_i$ is a finite sequence of letters drawn from an alphabet where each w_i is an element of Σ. An empty string is denoted by ϵ. The **length of a string** w is denoted by $|w|$.

- The set of **all strings** composed from letters in Σ is Σ^*.
- The set of all **non-empty strings** composed from letters in Σ is Σ^+.
- The set of **all strings of length** k composed from letters in *Sigma* is Σ^k.
- The **size** of Σ^k is given as $|\Sigma^k| = |\Sigma|^k$.
- A **language** is a collection of strings w drawn from Σ.

Note that Σ^1 is not the same as Σ. The former is the *set of strings* of length one, while the latter is only the *set of symbols* in the alphabet. Σ^1 just happens to be a set of strings of length 1.

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Table 6. *Mapping a transition function*

5.2 Deterministic Finite Automata

A *finite automaton* or *state machine* is a simple mathematical machine. It is a representation of how computations are performed with limited memory space. Figure 1 shows an example of an automaton using a *state diagram*. The **start state** is shown with an incoming arrow, while transitions are shown as further arrows with their respective input symbols leading to the next state. The **accept state** is denoted by a double ring.

When an automaton processes an input string w, it outputs either *accept* or *reject*.

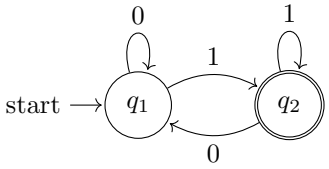

Figure 1. *State diagram of a finite automaton*

Definition A ***deterministic finite automaton*** M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states**,
2. Σ is a finite set called the **alphabet**,
3. $\delta : Q \times \Sigma \to Q$ is the **transition function**,
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accept states**.

The set A of all strings that a machine M accepts is the **language** $L(M) = A$ of the machine. We say that M *recognises* or *accepts* A. If a machine accepts no strings, it still accepts one language — the empty language $\emptyset$.

The transition function can be mapped using a two-dimensional table as shown for the function $Q \times \Sigma \to Q$ where $Q = \{q_1, q_2\}$ in Table 6.

If a machine M has a start state that is also an accept state, the machine will accept the empty string ϵ.

5.3 Non-deterministic Finite Automata

A deterministic finite automaton (DFA) has key differences from a non-deterministic finite automaton (NFA):

1. In a **DFA**, *every state* has exactly *one exiting transition* for *each symbol* in the alphabet.
2. In an **NFA**, states may have *none, one or more exiting transitions* for some or all symbols of the alphabet, including the empty string ϵ.

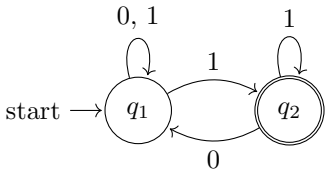

Figure 2. *Example of a non-deterministic finite automaton*

An example for an NFA is shown in Figure 2.

An NFA computes by making copies of itself to pursue every possible option in parallel. Therefore, if a state has two transitions for a given symbol, the machine copies itself and follows both routes. If a transition is not possible, the machine dies and returns a reject state.

Definition A ***non-deterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states**,
2. Σ is a finite set called the **alphabet**,
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the **transition function**,
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accept states**.

The **key difference** is the transition function, which maps the cross product of states and the alphabet, including the empty string ϵ, to a set of all possible next states, written as the power set of Q.

6 Regular Languages

A language is called a **regular language** if some finite automaton recognises it.

6.1 Regular Operations

Let A and B be languages. We define the regular operations **union**, **concatenation**, and **star** as follows:

1. **Union**: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
2. **Concatenation**: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
3. **Star**: $A^\star = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

The **star** operation is a **unary** operator, in that it only operates on one alphabet. It takes all symbols from the alphabet and generates all possible strings from language A to form a new language A^*. The empty string ϵ is always a member of A^*, as no string may be generated as one of the options.

Every NFA has an equivalent DFA in the sense that they both can recognise (accept) the same language.

A language is considered *regular* if and only if at least one NFA accepts it. If two languages are *regular*, then their **union**, **concatenation** and **star** are regular.

6.2 Regular Expressions

A *regular expression* is a way to describe a language in a concise, general notation using regular operations.

The following is an example of a regular expression in shorthand notation and it's more expanded form.

$$(0 \cup 1)0^* = (\{0\} \cup \{1\}) \circ \{0\}^*$$

In regular expressions, the order of precedence is given by *star* $\rightarrow$ *concatenation* $\rightarrow$ *union*. Concatenation is implicit, meaning that is is assumed when no operator is given between two terms (as in the example above).

Say that R is a **regular expression** if R is

1. a for some a in alphabet Σ,
2. ϵ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

Don't confuse the regular expressions ϵ and $\emptyset$. The expression ϵ represents the language containing a single string, the empty string. $\emptyset$ describes the language that doesn't contain any strings.

Kleene's Theorem A language is regular if and only if some regular expression describes it.

6.3 Pumping Lemma

If a language is not regular, then there is no DFA that recognises it. To prove that a language L is not regular, we can use a property of regular languages and prove non-regularity by contradiction.

A regular language can be *pumped*, which means that with a sufficiently long input string, it cycles at least once. That means that a state in the automaton is visited twice (based on the pigeonhole principle). The **pumping lemma** states that such a string, while cycling one or more times, will still remain in the language.

Pumping Lemma If A is a regular language, then there is a number p where if s is any string in A and $s \leq p$ then s may be divided into three pieces $s = xyz$ such that

1. for each $i \geq 0, xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

The first condition states that a string that has a cycling part remains in the language. The second part states that the cycling part may not be empty, it has to have at least one edge (meaning a node cycling onto itself). The this part states that the automaton needs to reach the start of the cycle before reaching the pumping length p.

The pumping lemma is only concerned with languages that generate very long (i.e. infinite strings).

7 Context-Free Languages

A grammar consists of **substitution rules** and describes how strings in a language are generated using such rules.

Definition A **context-free grammar** is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the ***variables*** ,
2. Σ is a finite set, disjoint from V, called the ***terminals*** ,
3. R is a finite set of ***rules*** , with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Example The language $L = \{o^n 1^n\}$ can be described by the grammar

$$S \to 0S1 \mid \epsilon$$

where S is the start variable, 0 and 1 are terminals and the entire statement is a substitution rule.

7.1 Chomsky Normal Form

Any context-free grammar has an equivalent grammar in **Chomsky normal form** which is a simplified version that removes ambiguity from the grammar.

A context-free grammar is in **Chomsky normal form** if every production is of the form

$$A \to BC$$

$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \epsilon$, where S is the start variable.

Converting to Chomsky normal form The steps to convert a grammar to Chomsky normal form are given as follows.

1. **Add a new start variable** S_0 and the rule $S_0 \to S$. This guarantees that the start variable occurs only on the left-hand side.
2. **Remove all** ϵ **rules** $A \to \epsilon$**.** For each occurrence of A on the right-hand side, add a new rule with that occurrence deleted.
3. **Remove all unit rules** of the form $A \to B$. Replace all occurrences of B on the right-hand side with the terminal that is produced by A. Eliminate the original unit rule.
4. **Reduce any rules with more than two variables or more than one terminal** to correct form. Replace variable pairs inside of n-tuples with new single variables and add corresponding rules. Replace any combinations of terminals and variables with variable pairs and add the corresponding rules.

8 Turing Machines

A Turing machine is a more powerful model of a computer, similar to a finite automaton but with unlimited and unrestricted memory.

It consists of a control unit with a read/write head that moves along a tape which is divided into cells. The machine can move along the tape and read from or write to cells. The tape contains the input string at the beginning, followed by a space $\sqcup$ and an infinite amount of empty tape following it. Cells can be overwritten.

Definition A ***Turing machine*** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where

1. Q is a set of states,
2. Σ is the input alphabet not containing the **blank symbol** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Configuration A *configuration* of a Turing machine describes it's current state, the current tape contents and the current head location. For a state q and two strings u and v over the tape alphabet Γ, we write uqv for the configuration where the currents state is q, the tape contents are uv and the head location is at the first symbol of v. For example, `1011`q_7`01111` is the configuration where the current state is q_7, the tape contents are `101101111` and the current head location is at the second `0`. The **start configuration** of M on input w is the configuration q_0w which indicates the machine is in the start state q_0 and the head is at the leftmost position.

A Turing machine M ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where

1. C_1 is the start configuration of M on input w
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

The collection of strings that M accepts is the **language of M**. The language is **Turing-recognisable** if some Turing machine *recognises* it. A Turing machine has three possible outcomes

- *accept*
- *reject*
- *loop*

A Turing machine that never loops is called a **decider**. A language is **Turing-decidable** if some Turing machine *decides* it.

8.1 Halting Problem

The **Halting Problem** is a kind of problem that cannot be solved by a Turing machine. Say we want to find out if a program P halts or loops for ever. No procedure exists to determine this, because if a program is still running after a fixed length of time, we do not know if it is still running or is looping forever. Hence we cannot computationally determine if the program will halt unless we let the program itself run to completion (or loop forever).

9 Algorithms

9.1 Bubble Sort

Bubble sort is popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

It starts at the head of a vector, compares the element to the adjacent elements, and swaps the elements if the first element is larger than the second element.

Passes The Bubble sort algorithm may need multiple passes to sort the entire vector. The maximum number of passes is given by the most difficult vectors to solve - such that are completely in reverse order. In this case, the number of required passes is $n-1$ where n is the number of elements. Therefore, **the maximum number of passes required** is always $n-1$.

Algorithm 1 Bubble Sort

```
function BUBBLESORT(vector)
    n ← LENGTH[vector]
    for 1 ≤ i ≤ n − 1 do
        count ← 0
        for 1 ≤ j ≤ n − 1 do
            if vector[j + 1] < vector[j] then
                Swap(vector, j, j + 1)
                count ← count + 1
            end if
        end for
        if count = 0 then
            break
        end if
    end for
    return vector
end function
```

9.2 Insertion Sort

Insertion sort works from the left of a vector, starting with the second element. It compares the element to the element on the left. If it needs to move, the element is moved along to the left as far as needed.

As an algorithm, we can store the current element in a temporary variable, while we "move" elements to the right by overwriting the slots to the right and then restoring the original values from the temporary variable.

Algorithm 2 Insertion Sort

```
function InsertionSort(v)
    j ← 2
    for j ≤ LENGTH[v] do
        key ← v[j]
        i ← j − 1
        while i > 0 ∧ v[i] > key do
            v[i + 1] ← v[i]
            i ← i − 1
        end while
        v[i + 1] ← key
        j ← j + 1
    end for
end function
```

9.3 Binary Search

Binary Search is a more efficient way of searching an item in a list. It proceeds by halving the list and comparing the middle element to the searched term. If the middle element is smaller than the term, the search continues on the right sublist, and if larger, on the left sublist. Then the procedure is repeated.

Binary search runs in logarithmic time $O(log_n)$.

Algorithm 3 Binary Search

```
function BinarySearch(v, item)
    L ← 1
    R ← LENGTH[v]
    while L ≤ R do
        M ← ⌊(L+R)/2⌋
        if v[M] < item then
            L ← M + 1
        else if v[M] > item then
            R ← M − 1
        else
            return M
        end if
    end while
    return false
end function
```

Binary search is faster than linear search because each additional search step works on a smaller input (half of the previous input). Hence the time to complete one additional step decreases by half with each step. Hence the algorithm is logarithmic.

9.4 Recursion

Recursion is the concept of a procedure referring to itself recursively. The main applications for using recursion are

- **Decrease and conquer**: Decreasing the problem to a *base case* and calling a procedure recursively on every decreasing inputs until the base case is reached.
- **Divide and conquer**: Split the problem into smaller partial problems and sort those first.

Algorithm 4 Recursive decrease-and-conquer procedure

```
function Factorial(n)
    if n = 0 then
        return 1
    end if
    return Factorial(n − 1)
end function
```

9.5 Quicksort

Quicksort is a recursive algorithm to sort a vector. It partitions the vector into sub-vectors at an arbitrary pivot, and then moves the elements smaller than the pivot to the left of it and elements larger to the right, and then moves the pivot element to the correct location in the partition. It then repeats this process recursively on the sub-vectors.

While Quicksort has a worst-case time-complexity of $\mathcal{O}(n^2)$, this case is unlikely to occur among possible inputs. The typical time complexity, or average-case complexity, is $\mathcal{O}(n \log(n))$

Algorithm 5 Quicksort — $\mathcal{O}(n^2)$

```
function QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end function

function PARTITION(A,p,r)
    x ← A[r] // Set the pivot to be the last element
    i ← p − 1 // Set i as the divider between elements smaller or bigger than x
    for j = p to r − 1 do // Loop through the vector
        if A[j] ≤ x then
            i ← i + 1 // Move divider to the right to include an element bigger than x
            exchange A[i] with A[j] // Move the jth element to the left partition
        end if
    end for
    exchange A[i + 1] with A[r] // Move the pivot to the right of the divider
end function
```

9.6 Merge Sort

Merge sort is another recursive algorithm that divides an array into subarrays, sorts them recursively and merges the resulting subarrays into an end result. This procedure relies on a MERGE function that compares two stacks element-by-element and merges them by placing the smaller of two elements into the result stack first. This comparison is then done recursively on ever-smaller subarrays, until the subarray length is one. In order to simplify code, the version below places *Sentinels* at the end of subarrays to terminate the comparison process and not have to track how many elements have been compared.

Algorithm 6 Merge Sort — $\mathcal{O}(n \log n)$

```
function MERGE-SORT(A,p,r )
    if p < r then// Check if the array is longer than 1
        q ← ⌊(p+r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
    end if
end function

function MERGE(A,p,q,r )
    n_1 ← q − p + 1 // Length of left subarray
    n_2 ← r − q // Length of right subarray
    let L[1 . . . n_1 + 1] and R[1 . . . n_2 + 1] be new arrays // Extend length by 1 for sentinels
    for i = 1 to n_1 do // Copy left subarray
        L[i] ← A[p + i − 1]
    end for
    for j = 1 to n_2 do // Copy right subarray
        R[j] ← A[q + j]
    end for
    L[n_1 + 1] ← ∞ // Insert sentinels
    R[n_2 + 1] ← ∞
    i ← 1
    j ← 1
    for k = p to r do
        if L[i] ≤ R[j] then // Pick left element
            A[k] ← L[i]
            i ← i + 1
        else A[k] ← R[j] // Pick right element
            j ← j + 1
        end if
    end for
end function
```

9.7 Gale-Shapley Algorithm

The idea is to iterate through all free men while there is any free man available. Every free man goes to all women in his preference list according to the order. For every woman he goes to, he checks if the woman is free, if yes, they both become engaged. If the woman is not free, then the woman chooses either says no to him or dumps her current engagement according to her preference list. So an engagement done once can be broken if a woman gets better option.

Algorithm 7 Gale-Shapley Algorithm for Stable Matching — $\mathcal{O}(n^2)$

```
function GALE-SHAPLEY(m,w,, lists of preferences for each m, w)
    Initialize all women w and all men m to free
    while ∃ a free man m who still has a woman w to propose to do
        w ← m's highest-ranked woman to whom he has not yet proposed
        if w is free then
            m, w become engaged
        else
            if w prefers m to m′ then
                m, w become engaged
                m′ becomes free
            else
                m′, w remain engaged
            end if
        end if
    end while
end function
```

10 Complexity Theory

10.1 Worst-Case Time Complexity

Let M be a RAM machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n . If $f(n)$ is the running time of M , we say that M runs in time $f(n)$ and that M is an $f(n)$ time random access machine. Customarily we use n to represent the length of the input.

10.1.1 Big-O Notation

The **running time** of an algorithm on a particular input is the number of steps executed. Machine-independently, we define "time-steps". The total running time of an algorithm is defined as

$$\text{time cost per statement} \times \text{no. of executions per statement} = \text{total running time}$$

Function growth can be described and compared using **Big-O Notation**. In *asymptotic analysis*, we can consider only the highest order terms of a function and ignore the constants. For example, given the function

$$f(n) = 6n^3 + 2n^2 + 20n + 45$$

then the *asymptotic* or *Big-O* notation is given as

$$f(n) = \mathcal{O}(n^3)$$

A table of common functions and their respective Big-O notation is shown in Table 7 and sorted by growth rate from slowest to fastest.

Type	**Notation**
Constant	$\mathcal{O}(1)$
Logarithmic	$\mathcal{O}(\log(n)) = \mathcal{O}(\log(n^c))$
Polylogarithmic	$\mathcal{O}((\log(n))^c)$
Linear	$\mathcal{O}(n)$
Quadratic	$\mathcal{O}(n^2)$
Polynomial	$\mathcal{O}(n^c)$
Exponential	$\mathcal{O}(c^n)$

Table 7. *Notation in Pseudocode*

Formally, this means that any function is asymptotically *upper-bounded* by its respective Big-O class.

Let f and g be functions. It holds that

$$\exists k \exists n_0 \forall n \{|f(n)| \leq k \cdot g(n) \mid k > 0, n > n_0\} \rightarrow f(n) \in \mathcal{O}(g(n))$$

stating that the size of the function $f(n)$ will eventually be overtaken by the Big-O class $g(n)$ multiplied by a constant k. It is an upper bound.

Because Big-O classes live inside each other, we always seek to ascertain the *smallest* Big-O class in which a function exists.

10.2 Summary of Sorting and Search Algorithms

The worst-case and average-case time complexities are shown in

Algorithm	**Worst Case**	**Average Case**
Bubble Sort	$\mathcal{O}(n^2)$	$\Theta(n^2)$
Insertion Sort	$\mathcal{O}(n^2)$	$\Theta(n^2)$
Quick Sort	$\mathcal{O}(n^2)$	$\Theta(n \log(n))$
Merge Sort	$\mathcal{O}(n \log(n))$	$\Theta(n \log(n))$
Linear Search	$\mathcal{O}(n)$	$\Theta(n)$
Binary Search	$\mathcal{O}(\log(n))$	$\Theta(\log(n))$

10.3 Master Theorem

In *divide-and-conquer* problems, a recurrence relation can be used to state the problem. Suppose a problem of size n is divided into a subproblems each of size n/b where b is some integer greater than 1. Additionally, there is some additional work $g(n)$ to be done in the *combine* phase at the end of a divide-and-conquer routine (see CLRS). The recurrence relation can be then stated as

$$f(n) = af\left(\frac{n}{b}\right) + g(n)$$

where $f(n)$ is the number of operations required to solve the problem of size n.

The **master theorem** allows estimation of the size of functions that satisfy divide-and-conquer relations easily.

$$f(n) \text{ is } \begin{cases} \mathcal{O}(n^d) & \text{if } a < b^d, \\ \mathcal{O}(n^d \log n) & \text{if } a = b^d, \\ \mathcal{O}(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$