COMP1002 exam study sheet

Propositional logic

- *Propositional statement*: expression that has a truth value (true/false). It is a *tautology* if it is always true, *contradiction* if always false.
- *Logic connectives*: negation ("not") $\neg p$, conjunction ("and") $p \wedge q$, disjunction ("or") $p \vee q$, implication $p \rightarrow q$ (equivalent to $\neg p \vee q$), biconditional $p \leftrightarrow q$ (equivalent to $(p \rightarrow q) \wedge (q \rightarrow p)$). The order of precedence: $\neg$ strongest, $\wedge$ next, $\vee$ next, $\rightarrow$ and $\leftrightarrow$ the same, weakest.
- If $p \rightarrow q$ is an implication, then $\neg q \rightarrow \neg p$ is its *contrapositive*, $q \rightarrow p$ a *converse* and $\neg p \rightarrow \neg q$ an *inverse*. An implication is equivalent to its contrapositive, but not to converse/inverse or their negations. A negation of an implication $p \rightarrow q$ is $p \wedge \neg q$ (it is not an implication itself!)
- A *syntax tree* of a formula visually encodes its structure and the order of operations. Each occurrence of a variable or a logical connective is represented by a circle (a node in a tree). All variables are on the bottom of the tree. If an operation applied to subformula(s), the node for this operation is drawn above and connected to top nodes of subformulas to which this operation is applied.
- A *truth assignment* is a string of values of variables to the formula, usually a row with values of first several columns in the truth table (number of columns = number of variables). A truth assignment is *satisfying* the formula if the value of the formula on these variables is T, otherwise the truth assignment is *falsifying*. A formula is *satisfiable* if it has a satisfying assignment, otherwise it is *unsatisfiable* (a contradiction). A truth assignment can be encoded by a formula that is a $\wedge$ of variables and their negations, with negated variables in places that have F (false) in the assignment, and non-negated that have T (true). For example, $x = T, y = F, z = F$ is encoded as $(x \wedge \neg y \wedge \neg z)$.It is an encoding in a sense that this formula is true only on this truth assignment and nowhere else.
- A *truth table* has a line for each possible values of propositional variables (2^k lines if there are k variables), and a column for each variable and subformula, up to the whole statement.Its cells contain T and F depending whether the (sub)formula is true for the corresponding scenarios.
- Finding a method for checking if a formula has a satisfying assignment that is always significantly faster than using truth tables (that is, better than brute-force search) is a one of Clay Mathematics Institute $1,000,000 prize problems, known as "P vs. NP".
- Two formulas are *logically equivalent*, written $A \equiv B$, if they have the same truth value in all scenarios (truth assignments). $A \equiv B$ if and only if $A \leftrightarrow B$ is a tautology.
- There are several other important pairs of logically equivalent formulas, called *logical identities* or *logic laws*. We will talk more about them when we talk about Boolean algebras. The most famous example of logically equivalent formulas is $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$ (with a dual version $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$) where p and q can be arbitrary (propositional, here) formulas. These pairs of logically equivalent formulas are called *DeMorgan's law*. Here, remember that $FALSE \wedge p \equiv p \wedge \neg p \equiv FALSE$, $FALSE \vee p \equiv TRUE \wedge p \equiv p$ and $TRUE \vee p \equiv p \vee \neg p \equiv TRUE$.

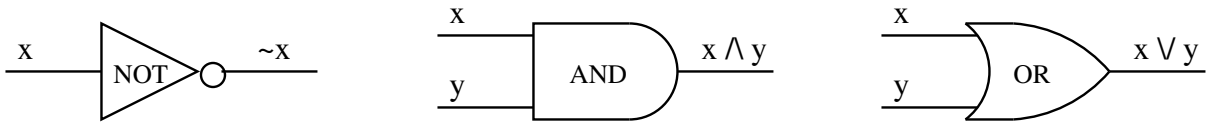


Figure 1: Types of gates in a digital circuit.

- A set of logic connectives is called *functionally complete* if it is possible to make a formula with any truth table out of these connectives. For example, $\neg, \wedge$ is a complete set of connectives, and so is the Sheffer's stroke $|$ (where $p|q \equiv \neg(p \wedge q)$), also called NAND for "not-and". But $\vee, \wedge$ is not a complete set of connectives since then it is impossible to express a truth table line with a 0 when all variables are 1.

- There are two main normal forms for the propositional formulas. One is called *Conjunctive normal form* (CNF, also known as Product-of-Sums) and is an $\wedge$ of $\vee$ of either variables or their negations (here, by $\wedge$ and $\vee$ we mean several formulas with $\wedge$ between each pair, as in $(\neg x \vee y \vee z) \wedge (\neg u \vee y) \wedge x$. A *literal* is a variable or its negation (x or $\neg x$, for example). A $\vee$ of (possibly more than 2) literals is called a *clause*, for example $(\neg u \vee z \vee x)$, so a CNF is true for some truth assignment whenever this assignment makes each of the clauses is true, that is, each clause has a literal that evaluates to true under this assignment. A *Disjunctive normal form* (DNF, Sum-of-Products) is like CNF except the roles of $\wedge$ and $\vee$ are reversed. A $\wedge$ of literals in a DNF is called a *term*.

- A CNF (DNF) is called *canonical* if it has a clause (respectively, term) for every falsifying (resp. satisfying) assignment. To construct canonical DNF and a CNF, start from a truth table and then for every satisfying truth assignment $\vee$ its encoding to a DNF, and for every falsifying truth assignment $\wedge$ the negation of its encoding to the CNF, and apply DeMorgan's law. This may result in a very large CNFs and DNFs, comparable to the size of the truth table itself ($2^{\text{number of variables}}$).

- *Boolean functions* are functions which take as argument boolean (ie, propositional) variables and return 1 or 0 (or, the convention here is 1 instead of T, and 0 instead of F). Each Boolean function on n variables can be fully described by its truth table. A size of a truth table of a function on n variables is 2^n. Even though we often can have a smaller description of a function, vast majority of Boolean functions cannot be described by anything much smaller. Every Boolean function can be described by a CNF or DNF, using the above construction.

- *Boolean circuits* is a generalization of Boolean formulas (or, rather, their syntax trees) in which we allow to reuse a part of a formula rather than writing it twice. To make a transition write Boolean formulas as trees and reuse parts that are repeating. The connectives become *circuit gates*. Here, we only look at circuits with AND, OR and NOT gates. It is possible to have more than 2 inputs into an AND or OR gates in a circuit, but a NOT gate always takes exactly one input.

- An *argument* consists of several logical statements (formulas) called *premises* and a final statement (formula) called a *conclusion*. If we call premises $A_1 \dots A_n$ and conclusion B, then an argument is *valid* iff premises imply the conclusion for all assignments to their free variables, that is, $A_1 \wedge \dots \wedge A_n \rightarrow B$. We usually write them in the following format:

  Today is either Thursday or Friday
  On Thursdays I have to go to a lecture

Today is not Friday (alternatively, On Friday I have to go to the lecture)

$\therefore$ I have to go to a lecture today

- A valid form of argument is called *rule of inference*. The most prominent such rule is called *modus ponens*.

  $p \to q$
  p ———
  $\therefore q$

- We studied three methods for proving that a formula is a tautology: *truth tables*, *natural deduction* and *resolution* (where resolution proves that a formula is a tautology by proving that its negation is a contradiction).

- A *natural deduction proof* consists of a sequence of applications of modus ponens (and other rules of inference) until a desired conclusion is reached, or there is nothing new left to derive. Example: treasure hunt, where "desired conclusion" is a statement that the treasure is in a specific location.

- A *resolution proof system* is used to find a contradiction in a formula (and, similarly, to prove that a formula is a tautology by finding a contradiction in its negation). Resolution starts with a formula in a CNF form, and applies the rule "from clause $(C \vee x)$ and clause $(D \vee \neg x)$ derive clause $(C \vee D)$ until a falsity F (equivalently, empty clause ()) is reached (so in the last step one of the clauses being *resolved* contains just one variable and another clause being resolved contains just that variable's negation.) Note that if a clause has opposing literals (e.g., from resolving $(x \vee y)$ with $(\neg x \vee \neg y)$ then it evaluates to true, and so is useless for deriving a contradiction. Resolution can be used to check the validity of an argument by running it on the $\wedge$ of all premises (converted, each, to a CNF) $\wedge$ together with the negation of the conclusion.

- *Pigeonhole principle* If n pigeons sit in $n-1$ holes, so that each pigeon sits in some hole, then some hole has at least two pigeons. There is no small resolution proof of the pigeonhole principle.

Predicate logic:

- A *set* is a well-defined collection of objects, called elements of a set. An object x belongs to set A is denoted $x \in A$ (said "x in A" or "x is a member of A"). Usually for every set we consider a bigger "universe" from which its elements come (for example, for a set of even numbers, the universe can be all natural numbers). A set is often constructed using *set-builder notation*: $A = \{x \in U | P(x)\}$ where U is a universe , and $P(x)$ is a predicate statement; this is read as "x in U such that $P(x)$" and denotes all elements in the universe for which $P(x)$ holds. Alternatively, for a small set, one can list its elements in curly brackets (e.g., $A = \{1, 2, 3, 4\}$.)

- *Cardinality* (size) of a set A, denoted $|A|$, is the number of elements in it.

- A *predicate* is like a propositional variable, but with *free variables*, and can be true or false depending on the values of these free variables. A *domain* (universe) of a predicate in a formula is a set from which the free variables can take their values (e.g., the domain of $Even(n)$ can be integers).

- *Quantifiers* For a predicate $P(x)$, a quantified statement "for all" ("every", "all") $\forall x P(x)$ is true iff $P(x)$ is true for every value of x from the domain (also called universe); here, $\forall$ is called a *universal quantifier*. A statement "exists" ("some", "a") $\exists x P(x)$ is true whenever $P(x)$ is true for at least one element x in the universe; $\exists$ is an existential quantifier. The word "any" means sometimes $\exists$ and sometimes $\forall$. A domain (universe) of a quantifier, sometimes written as $\exists x \in D$ and $\forall x \in D$ is the set of values from which the possible choices for x are made. If the domain of a quantifier is empty, then if the quantifier is universal then the formula is true, and if quantifier is existential, false. A *scope* of a quantifier is a part of the formula (akin to a piece of code) on which the variable under that quantifier can be used (after the quantifier symbol/inside the parentheses/until there is another quantifier over a variable with the same name). A variable is *bound* if it is under a some quantifier symbol, otherwise it is free.

- *First-order formula* A predicate is a first-order formula (possibly with free variables). If A and B are first-order formulas, then so are $\neg A$, $A \wedge B$, $A \vee B$. If a formula $A(x)$ has a free variable (that is, a variable x that occurs in some predicates but does not occur under quantifiers such as $\forall x$ or $\exists x$), then $\forall x\ A(x)$ and $\exists x\ A(x)$ are also first-order formulas. A first-order formula is in *prenex form* when all variables have different names and all quantifiers are in front of the formula.

- More precisely, a *signature* is a list of names of predicates with their arities (as well as names and arities of functions on elements, if we have them); once we specified a signature, we can write first-order formulas in this signature. A *structure* of a given signature consists of a domain, and *interpretations* of all predicate and function symbols (an *interpretation* tells us the values predicates and functions on elements of the domain). A *model* of a formula is an interpretation that makes this formula true.

  Example: in our Tarski world, the signature consists of 5 unary predicates Circle(), Square(), Triangle(), Big(), Little(), and three binary predicates NextTo(), Aligned() and EqualSize(). Each Tarski board is a structure of this signature, with the domain consisting of all pieces on the board, and interpretations of predicates reflecting what these pieces are and how they are positioned with respect to each other (for example, if the first piece, call it "a", is a triangle, then Triangle(a) would be true, and Circle(a), Square(a) would be false). A board which satisfies a given formula is a model of that formula.

- A set A is a *subset* of set B, denoted$A \subseteq B$, if $\forall x(x \in A \rightarrow x \in B)$. It is a *proper* subset if $\exists x \in B$ such that $x \notin A$. Otherwise, if $\forall x(x \in A \leftrightarrow x \in B)$ two sets are equal.

- Special sets are: *empty set* $\emptyset$, defined as $\forall x(x \notin \emptyset)$. Universal set U: all potential elements under consideration at given moment. Natural numbers $\mathbb{N}$ (here, $0 \in \mathbb{N}$), integers $\mathbb{Z}$, rationals $\mathbb{Q}$, reals $\mathbb{R}$, binary strings $\{0,1\}^*$.

- Basic set operations are a *complement* $\bar{A}$, denoting all elements in the universe that are *not* in A, then *union* $A \cup B = \{x | x \in A \text{ or } x \in B\}$, and *intersection* $A \cap B = \{x | x \in A \text{ and } x \in B\}$ and *set difference* $A - B = \{x | x \in A \text{ and } x \notin B\}$.

- The Cartesian product of two sets $A \times B = \{(a,b) | a \in A \text{ and } b \in B\}$.

- *Type checking*: there are different types of items and operations in a first-order formula: elements of the domains which occur only in inputs to predicates (and as variable names in quantifiers), sets (domains), and Booleans (returning true/false). All logical connectives take Booleans as inputs and

return Booleans (so the whole formula evaluates to a Boolean). Predicates take elements as inputs and return a Boolean. Functions such as addition take elements as inputs and return elements. Finally, quantifiers take a name of a variable denoting an element, the name of the domain (a set), and a Boolean (ie a formula or a predicate) and returns a Boolean.

- The order of quantifiers in a formula matters (as well as the order of variables in a predicate). $\forall x \exists y P(x, y)$ is not the same as $\exists y \forall x P(x, y)$, since in the first formula for different values of x different values of y can be chosen, whereas the second formula is true if there is a single value of y which should work for all x. If you have done programming, it might help to think of nested quantifiers as nested "for" loops: in the first case, the inner loop is on the y's, and in the second, the inner loop is on the x's.

- To evaluate a formula with nested quantifiers, think of a game between two players: a universal player is trying to make the formula false, and existential player trying to make it true. They go left to right through the formula, with universal player suggesting counterexamples, and existential player suggesting witnesses; if after all quantifiers have been instantiated the resulting formula is true, the existential player wins, if it is false, universal player wins. Now, a formula is true iff there is a way for an existential player to win no matter what universal player's choices are.

- *Negating quantifiers.* Remember that $\neg \forall x P(x) \equiv \exists x \neg P(x)$ and $\neg \exists x P(x) \equiv \forall x \neg P(x)$. This is because $\forall$ is like a big $\wedge$ over all scenarios, and $\exists$ is an $\vee$.

- *Prenex normal form* In a first-order formula, it is possible to rename variables under quantifiers so that they all have different names. Then, after pushing negations into the formulas under the quantifiers, the quantifier symbols can be moved to the front of a formula (making their scope the whole formula).

- *Formulas with finite domains* If the domain of a formula is finite, a formula can be converted into a propositional formula by changing each $\forall x$ quantifier with a $\wedge$ of the formula on all possible values of x; an $\exists$ quantifier becomes a $\vee$. Then terms of the form $P(value)$ (e.g., $Even(5)$) are treated as propositional variables.

- *Limitations of first-order logic* There are concepts that are not expressible by first-order formulas, for example, transitivity ("is there a flight from A to B with arbitrary many legs?" cannot be a database query described by a first-order formula).

- *Reasoning in predicate logic* There are four rules involving introducing and removing quantifiers, which together wth original rules of inference allow reasoning in predicate logic.
  - Universal instantiation: from $\forall x \in S, F(x)$ can derive $F(a)$ for any $a \in S$.
  - Universal generalization: from $F(a)$ where a is a name of an arbitrary element of S can derive $\forall x \in S, F(x)$.
  - Existential instantiation: from $\exists x \in S, F(x)$ can derive $F(k)$ where k is a variable name that has not occurred anywhere in the proof so far.
  - Existential generalization: from $F(k)$ where $k \in S$ can derive $\exists x \in S, F(x)$.

- The *universal modus ponens* rule combines universal instantiation and modus ponens. The classic example of this rule is "all men are mortal, Socrates is a man $\therefore$ Socrates is mortal"

- To prove that $A \subseteq B$, show that if you take an arbitrary element of A then it is always an element of B. To prove that two sets are equal, show both $A \subseteq B$ and $B \subseteq A$. You can also use set-theoretic identities.

Proof strategies

- Existential statement: $\forall x F(x)$. Constructive proof: give an example satisfying the formula under the quantifier (e.g, exists x which is both even and prime: take $n = 2$), then conclude by the *existential generalization rule* that $\exists x F(x)$ is true. Non-constructive proof: If the proof says $\exists n P(n)$, show that assuming $\forall n \neg P(n)$ leads to contradiction.

- Universal statement: $\forall x F(x)$. To prove that it is false, give a counterexample. To prove that it is true, start with the *universal instantiation*: take an arbitrary element, give it a name (say n) , and prove that $F(n)$ holds without any additional assumptions. By *universal generalization*, conclude that $\forall x F(x)$ holds.

- To prove $F(n)$

  - *Direct proof:* show that $F(n)$ holds directly, using definition, algebra, etc. If $F(n)$ is of the form $G(n) \rightarrow H(n)$, then assume $G(n)$ and derive $H(n)$ from this assumption. Examples: sum of even integers is even, if $n \equiv m(\mod d)$ then there is $k \in \mathbb{Z}$ such that $n = m + kd$, Pythagoras' theorem.
  - *Proof by cases* If $F(x)$ is of the form $(G_1(x) \vee G_2(x) \vee \cdots \vee G_k(x)) \rightarrow H(x)$, then prove $G_1(x) \rightarrow H(x)$, $G_2(x) \rightarrow H(x) \ldots G_k(x) \rightarrow H(x)$. Examples: sum of two consecutive integers is odd, $\forall x, y \in \mathbb{R} |x + y| \leq |x| + |y|$, $\min(x, y) = (x + y - |x - y|)/2$, $k^2 + k$ is even.
  - *Proof by contraposition* If $F(n)$ is of the form $G(n) \rightarrow H(n)$, can prove $\neg H(n) \rightarrow \neg G(n)$ (that is, assume that $H(n)$ is false, and derive that $G(n)$ is false). Examples: pigeonhole principle, if a square of an integer is even, then integer itself is even.
  - *Proof by contradiction* To prove $F(n)$, show that $\neg F(n) \rightarrow FALSE$. Examples: $\sqrt{(2)}$ is irrational, there are infinitely many primes.

- Some definitions:

  - An $n \in \mathbb{Z}$ is *even* if $\exists k \in \mathbb{Z}$ such that $n = 2k$. An $n \in \mathbb{Z}$ is *odd* if $\exists k \in \mathbb{Z}$ such that $n = 2k + 1$. An $n \in \mathbb{Z}$ is *divisible by* $m \in \mathbb{Z}$ if $\exists k \in \mathbb{Z}$ such that $n = km$.
  - Modular arithmetic: for any $n, d \neq 0 \in \mathbb{Z}$ $\exists q, r \in \mathbb{Z}$ such that $n = qd + r$ and $0 \leq r < d$. Here, q is a *quotient* and r is a *remainder*. *Congruence:* for $n, m, d \neq 0 \in \mathbb{Z}$, $n \equiv m(\mod d)$ ("n is congruent to m mod d") iff $\exists q_1, q_2, r \in \mathbb{Z}$ such that $0 \leq r \leq d$, $n = q_1 d + r$ and $m = q_2 d + r$. That is, n and m have the same remainder modulo d.
  - Absolute value of $x \in \mathbb{R}$, denoted $|x|$, is x if $x \geq 0$ and $-x$ if $x < 0$.
  - A ceiling of $x \in \mathbb{R}$, denoted $\lceil x \rceil$, is the smallest integer y such that $y \geq x$. Similarly, a floor of $x \in \mathbb{R}$, denoted $\lfloor x \rfloor$, is the largest integer y such that $y \leq x$.

Table 1: **Identities of logic and sets**

Name	**Logic law**	**Set theory law**
Double Negation	$\neg\neg p \equiv p$	$\overline{\overline{A}} = A$
DeMorgan's laws	$\neg(p \vee q) \equiv (\neg p \wedge \neg q)$ $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$	$\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\overline{A \cap B} = \overline{A} \cup \overline{B}$
Associativity	$(p \vee q) \vee r \equiv p \vee (q \vee r)$ $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
Commutativity	$p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Distributivity	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
Idempotence	$(p \vee p) \equiv p \equiv (p \wedge p)$	$A \cup A = A = A \cap A$
Identity	$p \vee F \equiv p \equiv p \wedge T$	$A \cup \emptyset = A = A \cap U$
Inverse	$p \vee \neg p \equiv T$ $p \wedge \neg p \equiv F$	$A \cup \bar{A} = U$ $A \cap \bar{A} = \emptyset$
Domination	$p \vee T \equiv T$ $p \wedge F \equiv F$	$A \cup U = U$ $A \cap \emptyset = \emptyset$

Basic structures

- An *alphabet* is a finite set of symbols (e.g.: binary alphabet $\{0, 1\}$, English alphabet, etc). An alphabet is usually denoted Σ (not to be confused with the summation sign, this is a capital Greek letter "Sigma"). A (finite) *string* (also called *word*) is a (finite) sequence of symbols (letters) from an alphabet (with repetition allowed). A special *empty string* is denoted λ (Greek letter "lambda"). The length of a string s, denoted $|s|$ (same notation as absolute value of a number or cardinality of a set) is the number of symbols in it: for example, string "00" has two symbols in it, so $|00| = 2$; $|\lambda| = 0$. A set of all strings over a given alphabet Σ is denoted Σ^* (pronounced "Sigma-star"; you will see why in COMP 1003). A *language* L over an alphabet Σ is a (possibly infinite) set of words from this language: $L \subseteq \Sigma^*$. The most common alphabet in Computer science is the binary alphabet $\{0, 1\}$, with the corresponding language of all binary strings $\{0, 1\}^*$.

- A *characteristic string* of a set A over the universe U is a way to represent sets on a computer. For that, put elements of U in some order, and represent A by a binary string of length $|U|$, with 0 for elements in A, and 0 for elements not in A. For example, if $U = a, b, c, d, e$, in that order, and $A = b, e$, then characteristic string for A is 01001.

- A *power set* for a given set A, denoted 2^A or $\mathcal{P}(A)$, is the set of all subsets of A. If A has n elements, then 2^A has 2^n elements (since for every element there are two choices, either it is in, or not).

- A k-ary **relation** R is a subset of Cartesian product of k sets $A_1 \times \cdots \times A_k$. We call elements of such R "k-tuples". A *binary* relation is a subset of a Cartesian product of two sets, so it is a set of *pairs* of elements. E.g., $R \subset \{2, 3, 4\} \times \{4, 6, 12\}$, where $R = \{(2, 4), (2, 6), (2, 12), (3, 6), (3, 12), (4, 4), (4, 12)\}$ is a binary relation consisting of pairs of numbers such that the first number in the pair divides the second.

- *Database queries* A *query* in a relational database is often represented as a first-order formula, where predicates correspond to the relations occurring in database (that is, a predicate is true on a tuple of values of variables if the corresponding relation contains that tuple). A query "returns" a set of values that satisfy the formula describing the query; a Boolean query, with no free variables, returns true or false. For example, a relation $StudentInfo(x, y)$ in a university database contains, say, all pairs x, y such that x is a student's name and y is the student number of student with the name x. A corresponding predicate $StudentInfo(x, y)$ will be true on all pairs x, y that are in the database. A query $\exists x StudentInfo(x, y)$ returns all valid student numbers. A query $\exists x \exists y StudentInfo(x, y)$, saying that there is at least one registered student, returns true if there is some student who is registered and false otherwise.

- Binary relation R (usually over $A \times A$) can be:
  - *reflexive*: $\forall x \in A \;\; R(x, x)$. For example, $a \leq b$ and $a = b$.
  - *anti-reflexive*: $\forall x \in A \;\; \neg R(x, x)$. For example, $a < b$ and $Parent(x, y)$.
  - *symmetric*: $\forall x, y \in A \;\; R(x, y) \rightarrow R(y, x)$. For example, $a = b$, $''sibling''$.
  - *antisymmetric*: $\forall x, y \in A \;\; R(x, y) \wedge R(y, x) \rightarrow x = y$. For example, $a < b$, $''parent''$.
  - *transitive*: $\forall x, y, z \in A \;\; (R(x, y) \wedge R(y, z) \rightarrow R(x, z)$. For example, $a = b$, $a < b$, $a|b$, $''ancestor''$.
  - *equivalence*: if R is reflexive, symmetric and transitive. For example, $a = b$, $a \equiv b$.
  - *order (total/partial)*: If R is antisymmetric, reflexive and transitive, then R is an order relation. If, additionally, $\forall x, y \in A \;\; R(x, y) \vee R(y, x)$, then the relation is a *total* order (e.g., $a \leq b$). Otherwise, it is a *partial* order (e.g., "ancestor", $a|b$.) An order relation can be represented by a Hasse diagram, which shows all connections between elements that cannot be derived by transitivity-reflexivity (e.g., "$p|n$" on $\{2, 6, 12\}$ will be depicted with just the connections 2 to 6 and 6 to 12.)

- A (reflexive, symmetric or transitive) *closure* of a relation R is the minimal relation containing R wiich is, respectively, reflexive, symmetric or transitive. In particular, the transitive closure of R is a relation R^{tc} that contains, in addition to R, all x, y such that there are $k \in \mathbb{N}, v_1, \ldots, v_k \in A$ such that $x = v_1, y = v_k$, and for i such that $1 \leq i < k$, $R(v_i, v_{i+1})$. For example, an "ancestor" relation is reflexive and transitive closure of the "parent" relation.

- A *lexicographic order* of tuples of elements $< a_1, \ldots, a_n >$ is the order in which the tuples are sorted on the first element first, then within the first on the second and so on. Here, the tuples of elements can be viewed as n-digit numbers, where digits can be any elements from the set on which the tuples are defined. For example, tuples $< 1, 2 >, < 1, 15 >, < 2, 1 >$ are listed here in the lexicographic order.

- *String order*: an order of strings in which shorter strings are listed before longer strings, and strings of the same length are listed in lexicographic order.

Mathematical induction.

- Let $n_0, n \in \mathbb{N}$, and $P(n)$ is a predicate with free variable n. Then the mathematical induction principle says:

$$(P(n_0) \wedge \forall n \geq n_0\ (P(n) \rightarrow P(n+1))) \rightarrow \forall n \geq n_0\ P(n)$$

  That is, to prove that a statement is true for all (sufficiently large) n, it is enough to prove that it holds for the smallest $n = n_0$ (*base case*) and prove that if it holds for some arbitrary $n > n_0$ (*induction hypothesis*) then it also holds for the next value of n, $n+1$ (*induction step*).

- Generally, *strong induction* is a variant of induction in which instead of assuming that the statement holds for just one value of n we assume it holds for several: $P(k) \wedge P(k+1) \wedge \cdots \wedge P(n) \rightarrow P(n+1)$ instead of just $P(n) \rightarrow P(n+1)$. Here, we usually use "strong induction" for the case also called "complete induction", when $k = n_0$, so we are assuming that the statement holds for *all* values smaller than $n+1$.

- A *well-ordering principle* states that every set of natural numbers has the smallest element. It is used to prove statements by counterexample: e.g., "define set of elements for which $P(n)$ does not hold. Take the smallest such n. Show that it is either not the smallest, or $P(n)$ holds for it".

  These three principles, Induction, Strong Induction and Well-ordering are equivalent. If you can prove a statement by one of them, you can prove it by the others.

  The following is the structure of an induction proof.

  1. $P(n)$. State which predicate $P(n)$ you are proving by induction. E.g., $P(n)$: $2^n < n!$.
  2. Base case: Prove $P(n_0)$ (usually just put n_0 in the expression and check that it works). E.g., $P(4)$: $2^4 < 4!$ holds because $2^4 = 16$ and $4! = 24$ and $16 < 24$.
  3. Induction hypothesis: "assume $P(n)$ for some $n > n_0$". I like to rewrite the statement for $P(n)$ at this point, just to see what I am using. For example, "Assume $2^n < n!$".
  4. Induction step: prove $P(n+1)$ under assumption that $P(n)$ holds. This is where all the work is. Start by writing $P(n+1)$ (for example, $2^{n+1} < (n+1)!$. Then try to make one side of the expression to "look like" (one side of) the induction hypothesis, maybe + some stuff and/or times some other stuff. For example, $2^{n+1} = 2 \cdot \mathbf{2^n}$, which is 2^n times additional 2. The next step is either to substitute the right side of induction hypothesis in the resulting expression with the left side (e.g., 2^n in $2 \cdot 2^n$ with $n!$, giving $2 \cdot n!$, or just apply the induction hypothesis assumption to prove the final result You might need to do some manipulations with the resulting expression to get what you want, but applying the induction hypothesis should be the main part of the proof of the induction step.

Recursive definitions, grammars, function growth, structural induction.

- A *recursive definition* (of a set) consists of

  1. The basis of recursion: "these several elements are in the set".
  2. The recursion rule or recurrence: "these rules are used to get new elements".

  Here, the underlying assumption (sometimes stated explicitly) is that there are no elements in the set other than the ones in the basis and introduced by the rules starting from the basis.

- A *Structural induction* is used to prove properties about recursively defined sets. The base case of the structural induction is to prove that $P(x)$ holds for the elements in the base, and the induction steps proves that if the property holds for some elements in the set, then it holds for all elements obtained using the rules in the recursion.

- Recursive definitions of functions are defined similar to sets: define a function on 0 or 1 (or several), and then give a rule for constructing new values from smaller ones. Some recursive definitions do not give a well-defined function (e.g., $G(n) = G(n/2)$ if n is even and $G(3n+1)$ if N is odd is not well defined). Some functions are well-defined but grow extremely fast: Ackermann function defined as $\forall m, n > 0, A(0, n) = n + 1, A(m, 0) = A(m - 1, 1), A(m, n) = A(m - 1, A(m, n - 1))$

- To compare grows rate of the functions, use $O()$-notation. $f(n) \in O(g(n))$ if $\exists n_0, c > 0$ such that $\forall n \geq n_0$ $f(n) \leq cg(n)$. In algorithmic terms, if $f(n)$ and $g(n)$ are running times of two algorithms for the same problem, $f(n)$ works faster on large inputs.

- A *context-free grammar* consists of

  1. Finite set Σ of terminals (letters in the alphabet).
  2. Finite set V of variables (also called non-terminals), including a specical starting variable.
  3. Finite set of rules, each of the form $A \to w$ for some variable A and a string of variables and terminals w (several rules for the same variable can also be written using symbol "—" for "or": $A \to w_1|w_2|\ldots|w_k$ has the same meaning as $A \to w_1$, $A \to w_2$, ..., $A \to w_k$.

- For example, the following grammar defines natural numbers in decimal notation:
  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $V = \{N, D\}$, with start variable N.
  $N \to 0|1D|2D|3D|4D|5D|6D|7D|8D|9D$
  $D \to \lambda|0D|1D|2D|3D|4D|5D|6D|7D|8D|9D$

  Note that this grammar avoids any number except for 0 starting with 0, and does not allow an empty number.

- A string is generated by a given grammar if it can be obtained by repeatedly applying the rules (represented by a parse tree); a language is recognized by a grammar is the set of all strings generated by it. If there is a context-free grammar recognizing a given language, that language is called *context-free*.

- A grammar is ambiguous if there is a string for which there is more than one derivation (parse tree).

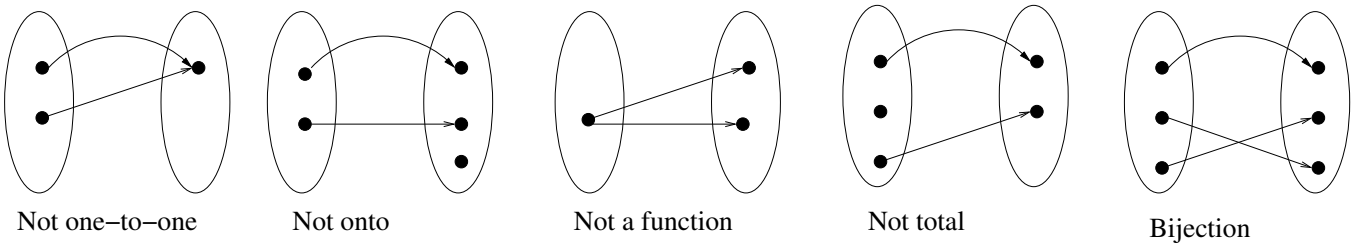
Counting

- *Rules of Sum and Product*: Choosing either one out of n *or* one out of m can be done $n + m$ ways. Choosing one out of n *and* one out of m can be done $n \cdot m$ ways.

- *Permutations*: The number of sequences of n distinct objects. Without repetition: $n!$, with repetition: n^k, where k is the length of the sequence.

- *Combinations*: The number of ways to choose k objects from n objects without repetition.

$$\text{Without order}: \quad C(n,k) = \binom{n}{k} = \frac{n!}{(n-k)!k!} \qquad \text{With order:} \quad P(n,k) = \frac{n!}{(n-k)!}$$



- *Combinations with repetition*: The number of ways to choose k elements out of n possibilities.

  Combinations of k elements from n categories ($n-1$ "dividers"): $\binom{k+(n-1)}{k} = \frac{(k+(n-1))!}{k!(n-1)!}$.

- *Binomial theorem.* For a non-negative integer n, $(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i$

- *Pascal's identity:*

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$$

- *Pascal's triangle:* each row contains binomial coefficients for the power binomial expansion. Each coefficient is the sum of two above it (above-right and above-left), using Pascal's identity.

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

- Other identities and corollaries of the binomial theorem:

$$\binom{n}{k} = \binom{n}{n-k} \qquad \sum_{i=1}^{n} \binom{n}{i} = 2^n \qquad \sum_{i=1}^{n} (-1)^i \binom{n}{i} = 0$$

- *Principle of inclusion-exclusion*: The number of elements in $A \cup B$, $|A \cup B| = |A| + |B| - |A \cap B|$. In general, add all odd-sized intersections and subtract all even-sized intersections.

- A **function** $f\colon A \to B$ is a special type of relation $R \subseteq A \times B$ such that for any $x \in A, y, z \in B$, if $f(x) = y$ and $f(x) = z$ then $y = z$. If $A = A_1 \times \ldots \times A_k$, we say that the function is k-ary. In words, a $k+1$-ary relation is a k-ary function if for any possible value of the first k variables there is at most one value of the last variable. We also say "f is a mapping from A to B" for a function f, and call $f(x) = y$ "f maps x to y".

  - A function is *total* if there is a value $f(x) \in B$ for every x; otherwise the function is *partial*. For example, $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^2$ is a total function, but $f(x) = \frac{1}{x}$ is partial, because it is not defined when $x = 0$.
  - If a function is $f\colon A \to B$, then A is called the *domain* of the function, and B a *codomain*. The set of $\{y \in B \mid \exists x \in A, f(x) = y\}$ is called the *range* of f. For $f(x) = y$, y is called the *image* of x and x a *preimage* of y.

- A *composition* of f: $A \to B$ and g: $B \to C$ is a function $g \circ f$: $A \to C$ such that if $f(x) = y$ and $g(y) = z$, then $(g \circ f)(x) = g(f(x)) = z$.
- A function g: $B \to A$ is an *inverse* of f (denoted f^{-1}) if $(g \circ f)(x) = x$ for all $x \in A$.
- A total function f is *one-to-one* if for every $y \in B$, there is at most one $x \in A$ such that $f(x) = y$. For example, the function $f(x) = x^2$ is not one-to-one when f: $\mathbb{Z} \to \mathbb{N}$ (because both $-x$ and x are mapped to the same x^2), but is one-to-one when f: $\mathbb{N} \to \mathbb{N}$.
- A total function f: $A \to B$ is *onto* if the range of f is all of B, that is, for every element in B there is some element in A that maps to it. For example, $f(x) = 2x$ is onto when f: $\mathbb{N} \to Even$, where $Even$ is the set of all even numbers, but not onto $\mathbb{N}$.
- A total function that is both one-to-one and onto is called a *bijection*.
- A function $f(x) = x$ is called the *identity* function. It has the property that $f^{-1}(x) = f(x)$. A function $f(x) = c$ for some fixed constant c (e.g., $f(x) = 3$) is called a *constant* function.

- A *cardinality* of a set is the number of elements in it. Two sets have the same cardinality if there is a bijection between them. If the cardinality of a set is the same as the cardinality of $\mathbb{N}$, the set is called *countable*. If it is greater, then *uncountable*.

- **Comparing set sizes**
  Two sets A and B have the same cardinality if exists f that is a bijection from A to B.
  If a set has the same cardinality as $\mathbb{N}$, we call it a *countable* set. If it has cardinality larger than the cardinality of $\mathbb{N}$, we call it *uncountable*. If it has k elements for some $k \in \mathbb{N}$, we call it *finite*, otherwise *infinite* (so countable and uncountable sets are infinite). E.g: $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, Even$, set of all finite strings, Java programs, or algorithms are all countable, and $\mathbb{R}, \mathbb{C}$, power set of $\mathbb{N}$, are all uncountable. E.g., to show that $\mathbb{Z}$ is countable, we prove that there is a bijection f: $\mathbb{Z} \to \mathbb{N}$: take $f(x) = \begin{cases} 2x & x \geq 0 \\ 1 - 2x & x < 0 \end{cases}$.
  It is one-to-one because $f(x) = f(y)$ only if $x = y$, and it is onto because for any $y \in \mathbb{N}$, if it is even then its preimage is $y/2$, if it is odd $-\frac{y-1}{2}$. Often it is easier to give instead two one-to-one functions, from the first set to the second and another from the second to the third. Also, often instead of a full description of a function it is enough to show that there is an enumeration such that every element of, say, $\mathbb{Z}$ is mapped to a distinct element of $\mathbb{N}$. To show that one finite set is smaller than another, just compare the number of elements. To show that one infinite set is smaller than another, in particular that a set is uncountable, use *diagonalization*: suppose that a there is an enumeration of elements of a set, say, $2^{\mathbb{N}}$ by elements of $\mathbb{N}$. List all elements of $2^{\mathbb{N}}$ according to that enumeration. Now, construct a new set which is not in the enumeration by making it differ from the k^{th} element of the enumeration in the k^{th} place (e.g., if the second set contains element 2, then the diagonal set will not contain the element 2, and vice versa).

Probability

- A *sample space* S is a set of all possible *outcomes* of an *experiment* ({heads, tails} for a coin toss, {1,2,3,4,5,6} for a die throw). An *event* is a subset of the sample space. If all outcomes are equally likely, probability of each is $1/|S|$ (uniform distribution). Otherwise, sum of probabilities of all outcomes is 1, and probability of each is between 0 and 1: probability distribution on the sample space. A probability of an event $Pr(A) = \Sigma_{a \in A} Pr(a)$. $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$, so if events A and B are disjoint, then $Pr(A \cup B) = Pr(A) + Pr(B)$.

- *Birthday paradox*: about 23 people enough to have 1/2 probability that two have birthday the same day (if birthdays are uniformly random days of the year).
- *Conditional probability*: $Pr(A|B) = Pr(A \cap B)/Pr(B)$. If $Pr(A \cap B) = Pr(A) \cdot Pr(B)$, events A and B are *independent*. Better to switch the door in Monty Hall puzzle.
- *Bayes theorem:* $Pr(B|A) = Pr(A|B) \cdot Pr(B)/Pr(A) = Pr(A|B) \cdot Pr(B)/(Pr(A|B) \cdot Pr(B) + Pr(A|\bar{B}) \cdot Pr(\bar{B})$. Generalizes to partition into arbitary many events rather than just B and $\bar{B}$. For a medical test, let A be an event that the test came up positive, and B that a person is sick. If this medical test has a false positive rate $Pr(A|\bar{B})$ (healthy mistakenly labeled sick, specificity $1 - Pr(A|\bar{B})$) and false negative rate $Pr(\bar{A}|B)$ (sick labeled healthy, sensitivity $1 - Pr(\bar{A}|B)$, and probability of being sick is $Pr(B)$, then probability of a person being sick if the test came up positive is $Pr(B|A) = Pr(A|B) \cdot Pr(B)/Pr(A)$, where $Pr(A) = Pr(A|B) \cdot Pr(B) + Pr(A|\bar{B}) \cdot Pr(\bar{B})$.
- A *random variable* is a function from outcomes to numbers. Distribution of a random variable X on a sample space S is a set of pairs $(x, Pr(X = x))$ where x ranges over all values X can take. Two random variables are *independent* when $\forall x, y \in \mathbb{R} Pr(X = x \wedge Y = y) = Pr(X = x) \cdot Pr(Y = y)$.
- *Expectaton*: let X be a random variable for some event over a sample space $\{a_1, \ldots, a_n\}$ (e.g., X is the number of coin tosses that came up heads, amount won in a lottery or X is 1 iff some event happened (indicator variable)). Then $E(X) = \Sigma_{i=1}^{n} a_k Pr(X(a_k))$ (if outcomes are numbers, often write $X = a_k$ in the equation).
- *Linearity of expectation*: $E(X_1 + X_2) = E(X_1) + E(X_2)$, and $E(aX + b) = aE(X) + b$. Example: hat check problem.
- *Bernoulli trials* Consider an experiment with probability p of success (and $1 - p$ of failure); this is one Bernoulli trial. Bernoulli trials consist of repeating this experiment n times independently. In Bernoulli trials
  - Probability of getting the first success exactly on k^{th} trial is $p(1-p)^{k-1}$
  - Probability of getting exactly k successes out of n trials is $\binom{n}{k} p^k (1-p)^{n-k}$
  - Expected number of trials until success is $1/p$.
  - Variance of the number of successes is $np(1-p)$.
- *Variance* of a random variable X over a sample space S is $V(X) = E_{s \in S}((X - E(X))^2) = \sum_{s \in S} (X(s) - E(X))^2 \cdot Pr(s) = E(X^2) - ((E(X))^2$. A *standard deviation* of a random variable X, denoted by σ (lowercase Greek letter sigma) is $\sigma(X) = \sqrt{V(X)}$.
- Markov's inequality: if X is a nonnegative random variable, $x > 0$. Then $Pr(X \geq x) \leq E(X)/x$.
- Chebyshev's inequality: Let $x > 0$. Then $Pr(|X| - E(X)| \geq x) \leq V(X)/x^2$.

Algorithm analysis.

- Preconditions and postconditions for a piece of code state, respectively, assumptions about input/values of the variables before this code is executed and the result after. If the code is correct, then preconditions + code imply postconditions.

- A *Loop invariant* is used to prove correctness of a loop. This is a statement implied by the precondition of the loop, true on every iteration of the loop (with loop guard variables as a parameter) and after the loop finishes implies the postcondition of the loop. A *guard* condition is a check whether to exist the loop or do another iteration. To prove total correctness of a program, need to prove that eventually the guard becomes false and the loop exists (however, it is not always possible to prove it given somebody's code), otherwise, the proof is of partial correctness.
- Correctness of recursive programs is proven by (strong) induction, with base case being the tail recursion call, and the induction step assumes that the calls returned the correct values and proves that the current iteration returns a correct value.
- Running time of an algorithm is a function of the length of the input. Usually we talk about worst-case running time, and give a bound on it using $O()$-notation.
- Average-case running time of an algorithm is an expectation over all inputs of a given length of the running time of this algorithm, also a function of the length of the input.