Discrete Math for Computer Science

Peter Schaefer

Freshman - Kutztown

Contents

1 Logic

1.1 Propositions and Logical Operations

Proposition: a statement that is either true or false.
Some examples include ”It is raining today” and ”$3 \cdot 8 = 20$”.
However, not all statements are propositions, such as ”open the door”

Name	**Symbol**	**alternate name**
NOT	$\neg$	negation
AND	$\wedge$	conjunction
OR	$\vee$	disjunction
XOR	$\oplus$	exclusive or

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \oplus q$
T	T	F	T	T	F
T	F	F	F	T	T
F	T	T	F	T	T
F	F	T	F	F	F

XOR is very useful for encryption and binary arithmetic.

1.2 Evaluating Compound Propositions

p : The weather is bad.	$p \wedge q$: The weather is bad *and* the trip is cancelled
q : The trip is cancelled.	$p \vee q$: The weather is bad *or* the trip is cancelled
r : The trip is delayed.	$p \wedge (q \oplus r)$: The weather is bad *and* either the trip is cancelled *or* delayed

Order of Evaluation $\neg$, then $\wedge$, then $\vee$, but parenthesis always help for clarity.

Example Truth Table:

p	q	$p \wedge q$	$\neg q$	$(p \wedge q) \oplus \neg q$
T	T	T	F	T
T	F	F	T	T
F	T	F	F	F
F	F	F	T	T

1.3 Conditional Statements

$p \implies q$ where p is the hypothesis and q is the conclusion

Format	Terminology
$p \implies q$	given
$\neg q \implies \neg p$	contrapositive
$q \implies p$	converse
$\neg p \implies \neg q$	inverse

given	$p \implies q$	$\equiv$	$\neg q \implies \neg p$	contrapositive
inverse	$\neg p \implies \neg q$	$\equiv$	$q \implies p$	converse

p	q	$p \implies q$
T	T	T
T	F	F
F	T	T
F	F	T

p is a sufficient condition for q
q is a necessary condition for p

Phrase	Logic
q if p	$p \implies q$
q only if p	$q \implies p$
q if and only if p	$p \iff q$

Order of Operations: $p \wedge q \implies r \equiv (p \wedge q) \implies r$

1.4 Logical Equivalence

Tautology: a proposition that is always true **Contradiction**: a proposition that is always false

Logically equivalent: same truth value regardless of the truth values of their individual propositions

DeMorgan’s Laws:
$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$
$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

Verbally,
It is not true that the patient has migraines *or* high blood pressure ≡
≡ The patient does not have migraines *and* does not have high blood pressure

It is not true that the patient has migraines *and* high blood pressure ≡
≡ The patient does not have migraines *or* does not have high blood pressure

1.5 Laws of Propositional Logic

You can use **substitution** on logically equivalent propositions.

Law Name	$\vee$ or	$\wedge$ and
Idempotent	$p \vee p \equiv p$	$p \wedge p \equiv p$
Associative	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Distributive	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity	$p \vee \text{F} \equiv p$	$p \wedge \text{T} \equiv p$
Domination	$p \vee \text{T} \equiv \text{T}$	$p \wedge \text{F} \equiv \text{F}$
Double Negation	$\neg\neg p \equiv p$	
Complement	$p \vee \neg p \equiv \text{T}$	$p \wedge \neg p \equiv \text{F}$
DeMorgan	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Absorption	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Conditional	$p \implies q \equiv \neg p \vee q$	$p \iff q \equiv (p \implies q) \wedge (q \implies p)$

1.6 Predicates and Quantifiers

Predicate: a logical statement where truth value is a function of a variable.

P(x): x is an even number. P(5): false P(2): true

Domain: the set of all possible values for a variable in a predicate.

Ex. $\mathbb{Z}^+$ is the set of all positive integers.
*If domain is not clear from context, it should be given as part of the definition of the predicate.

Quantifier: converts a predicate to a proposition.

Quantifier	Symbol	Meaning
Universal	$\forall$	"for all"
Existential	$\exists$	"there exists"

$\exists\, x(x+1 < x)$ is false.

Counter Example: universally quantified statement where an element in the domain for which the predicate is false. Useful to prove a $\forall$ statement false.

1.7 Quantified Statements

Consider the two following two predicates:

$$P(x) : x \text{ is prime, } x \in \mathbb{Z}^+$$
$$O(x) : x \text{ is odd}$$

Proposition made of predicates: $\exists\, x(\text{P}(x) \wedge \neg\text{O}(x))$
Verbally: there exists a positive integer that is prime but is not odd.

Free Variable: a variable that is free to be any value in the domain.
Bound Variable: a variable that is bound to a quantifier.

P(x):	x came to the party
S(x):	x was sick

$\text{P}(x) \overset{?}{\equiv} \neg\text{S}(x)$
$\text{P}(x) \not\equiv \neg\text{S}(x)$

	P(x)	S(x)	¬S(x)
Joe	T	F	T
Theo	F	T	F
Gert	T	F	T
Sam	F	F	T

1.8 DeMorgan's law for Quantified Statements

Consider the predicate: $\mathrm{F}(x)$: "x can fly", where x is a bird. According to the DeMorgan Identity for Quantified Statements,

$$\neg\forall\ x\mathrm{F}(x) \equiv \exists\ x\neg\mathrm{F}(x)$$

"not every bird can fly ≡ "there exists a bird that cannot fly

Example using DeMorgan Identities:

$$\begin{aligned}\neg\exists\ x(\mathrm{P}(x) \implies \neg\mathrm{Q}(x)) &\equiv \forall\ x\neg(\mathrm{P}(x) \implies \neg\mathrm{Q}(x))\\ &\equiv \forall\ x(\neg\neg\mathrm{P}(x) \wedge \neg\neg\mathrm{Q}(x))\\ &\equiv \forall\ x(\mathrm{P}(x) \wedge \mathrm{Q}(x))\end{aligned}$$

1.9 Nested Quantifiers

A logical expression with more than one quantifier that binds different variables in the same predicate is said to have **Nested Quantifiers**.

Logic	Variable Boundedness
$\forall\ x\exists\ y\ \mathrm{P}(x,y)$	x, y bound
$\forall\ x\ \mathrm{P}(x,y)$	x bound, y free
$\exists\ x\exists\ y\ \mathrm{T}(x,y,z)$	x, y bound, z free

Logic	Meaning
$\forall\ x\forall\ y\ \mathrm{M}(x,y)$	"everyone sent an email to everyone"
$\forall\ x\exists\ y\ \mathrm{M}(x,y)$	"everyone sent an email to someone"
$\exists\ x\forall\ y\ \mathrm{M}(x,y)$	"someone sent an email to everyone"
$\exists\ x\exists\ y\ \mathrm{M}(x,y)$	"someone sent an email to someone"

There is a two-player game analogy for how quantifiers work:

Player	Action	Goal
Existential Player ∃	selects value for existentially-bound variables	tries to make expression <u>true</u>
Universal Player ∀	selects value for universally-bound variables	tries to make expression <u>false</u>

Consider the predicate $\mathrm{L}(x,y)$: "x likes y".

$\exists\ x\forall\ y\mathrm{L}(x,y)$ means "there is a student who likes everyone in the school".

$\neg\exists\ x\forall\ y\mathrm{L}(x,y)$ means "there is no student who likes everyone in the school".

After applying DeMorgan's Laws,

$\forall\ x\exists\ y\neg\mathrm{L}(x,y)$ means "there is no student who likes everyone in the school".

1.10 More Nested Quantifiers

$\mathrm{M}(x,y)$: "x sent an email to y". Consider $\forall\ x\forall\ y\ \mathrm{M}\ (x,y)$. It means that "email sent an email to everyone including themselves". Using $(x \neq y \implies \mathrm{M}(x,y))$ can fix this quirk.

$\forall\ x\forall\ y(x \neq y \implies \mathrm{M}(x,y))$ means "everyone sent an email to everyone else

Expressing Uniqueness in Quantified Statements

Consider $\mathrm{L}(x)$: x was late to the meeting. If someone was late to the meeting, how could you express that that someone was the only person late to the meeting? You want to express that there is someone where everyone else was not late, which can be done with

$$\exists\ x(\mathrm{L}(x) \wedge \forall\ y(x \neq y \implies \neg\mathrm{L}(y)))$$

Moving Quantifiers in Logical Statements

Consider M(x, y): "x is married to y" and A(x): "x is an adult". One way of expressing "For every person x, if x is an adult, then there is a person y to whom x is married to" is by this statement:

$$\forall\ x(\ \mathrm{A}(x) \implies \exists\ \ \mathrm{M}(x, y))$$

Since y does not appear in A(x), "$\exists\ y$" can be moved so that it appears just after the "$\forall$ ", resulting with

$$\forall\ x\exists\ y(\ \mathrm{A}(x) \implies \ \mathrm{M}(x, y))$$

When doing this, keep in mind that $\forall\ x\exists\ y \not\equiv \exists\ y\forall\ x$:

$\forall\ x\exists\ y(\ \mathrm{A}(x) \implies \ \mathrm{M}(x, y))$ means
for every x, if x is an adult, there exists y who is married to x.
$\exists\ y\forall\ x(\ \mathrm{A}(x) \implies \ \mathrm{M}(x, y))$ means
There exists a y, such that every x who is an adult is also married to y

1.11 Logical Reasoning

Argument: a sequence of propositions, called hypothesis, followed by a final proposition, called the conclusion.

An argument is **valid** if the conclusion is true whenever the hypothesis are all true, otherwise the argument is **invalid**.

$$\text{An argument is denoted as:}\quad \begin{array}{c} p_1 \\ p_2 \\ \vdots \\ p_n \\ \hline \therefore c \end{array} \quad \text{where} \quad \begin{array}{l} p_1, p_2, \ldots, p_n \text{ are hypothesis} \\ c \text{ is the conclusion} \end{array}$$

The argument is valid whenever the proposition $(p_1 \wedge p_1 \wedge \cdots \wedge p_n) \implies c$ is a tautology. Additionally, because of the commutative law, hypothesis can be reordered without changing the argument.

$$\begin{array}{l} p \\ p \implies q \\ \hline \therefore q \end{array} \equiv \begin{array}{l} p \implies q \\ p \\ \hline \therefore q \end{array}$$

The Form of an Argument

$$\begin{array}{l} \text{It is raining today.} \\ \text{If it is raining today, then I will not ride my bike to school.} \\ \hline \therefore \text{I will not ride my bike to school.} \end{array} \qquad \begin{array}{l} p \\ p \implies q \\ \hline \therefore q \end{array}$$

$$\text{The argument is valid because its form, } \begin{array}{l} p \\ p \implies q \\ \hline \therefore q \end{array} \text{ is an valid argument.}$$

$$\begin{array}{l} \text{5 is not an even number.} \\ \text{If 5 is an even number, then 7 is an even number.} \\ \hline \therefore \text{7 is not an even number.} \end{array} \qquad \begin{array}{l} p \\ p \implies q \\ \hline \therefore q \end{array}$$

$$\text{The argument is invalid because its form, } \begin{array}{l} \neg p \\ p \implies q \\ \hline \therefore \neg q \end{array} \text{ is an invalid argument.}$$

1.12 Rules of Inference with Propositions

Using truth tables to establish the validity of an argument can become tedious, especially if an argument uses a large number of variables.

$$\begin{array}{l} p \\ p \implies q \\ \hline \therefore q \end{array} \quad \text{Modus Ponens} \qquad \begin{array}{l} p \\ q \\ \hline \therefore p \land q \end{array} \quad \text{Conjunction}$$

$$\begin{array}{l} \neg q \\ p \implies q \\ \hline \therefore \neg p \end{array} \quad \text{Modus Tollens} \qquad \begin{array}{l} p \implies q \\ q \implies r \\ \hline \therefore p \implies r \end{array} \quad \text{Hypothetical Syllogism}$$

$$\begin{array}{l} p \\ \hline \therefore p \lor q \end{array} \quad \text{Addition} \qquad \begin{array}{l} p \lor q \\ \neg p \\ \hline \therefore q \end{array} \quad \text{Disjunctive Syllogism}$$

$$\begin{array}{l} p \land q \\ \hline \therefore p \end{array} \quad \text{Simplification} \qquad \begin{array}{l} p \implies q \\ q \implies r \\ \hline \therefore q \lor r \end{array} \quad \text{Resolution}$$

Example expressed in English:

$$\begin{array}{l} \text{If it is raining or windy or both, the game will be cancelled.} \\ \text{The game will not be cancelled} \\ \hline \therefore \text{ It is not windy.} \end{array} \qquad \begin{array}{l} (r \lor w) \implies c \\ \neg c \\ \hline \therefore \neg w \end{array}$$

Steps to Solve:

$(r \lor w) \implies c$	Hypothesis	(1)
$\neg c$	Hypothesis	(2)
$\neg(r \lor w)$	Modus Tollens: 1, 2	(3)
$\neg r \land \neg w$	DeMorgan's Law: 3	(4)
$\neg w \land \neg r$	Commutative Law: 4	(5)
$\neg w$	Simplification: 5	(6)

1.13 Rules of Inference with Quantifiers

In order to apply the rules of quantified expressions, such as $\forall\, x \neg(\text{ P}(x) \land \text{ Q}(x))$, we need to remove the quantifier by plugging in a value from the domain to replace the variable x.

For example:

$$\begin{array}{l} \text{Every employee who received a large bonus works hard.} \\ \text{Linda is an employee at the company.} \\ \text{Linda received a large bonus.} \\ \hline \therefore \text{ Some employee works hard.} \end{array} \qquad \begin{array}{l} \forall\, x(\text{B}(x) \implies \text{H}(x)) \\ \text{Linda} \in x \\ \text{B(Linda)} \\ \hline \therefore \exists\, x \text{ H}(x) \end{array}$$

Arbitrary Element: has no special properties other than those shared by all elements of the domain.

Particular Element: may have special properties that are not shared by all the elements of the domain. For example, if the domain is the set of all integers, $\mathbb{Z}$, a particular element is 3, because it is odd, which is not true for all integers.

Rules of Inference for Quantified Statements

$$\begin{array}{l} c \text{ is an element} \\ \forall\ x\ \mathrm{P}\ (x) \\ \hline \therefore \mathrm{P}(c) \end{array} \quad \text{Universal Instantiation} \qquad \begin{array}{l} \exists\ x\ \mathrm{P}(x) \\ \hline \therefore c \text{ is particular} \wedge \mathrm{P}\ (c) \end{array} \quad \text{Existential Instantiation*}$$

$$\begin{array}{l} c \text{ is arbitrary} \\ \mathrm{P}\ (c) \\ \hline \therefore \forall\ \ \mathrm{P}(x) \end{array} \quad \text{Universal Generalization} \qquad \begin{array}{l} c \text{ is an element} \\ \mathrm{P}(c) \\ \hline \therefore \exists\ x\ \mathrm{P}(x) \end{array} \quad \text{Existential Generalization}$$

*Each use of Existential Instantiation must define a new element with its own symbol or name.

Example of using the Laws of Inference for Quantified Statements

Consider the following argument: $\begin{array}{l} \forall\ x(\mathrm{P}(x) \vee\ \mathrm{Q}(x)) \\ 3 \text{ is an integer} \\ \neg\ \mathrm{P}(3) \\ \hline \therefore\ \ \mathrm{Q}(3) \end{array}$

Steps to Solve:

$\forall\ x(\mathrm{P}(x) \vee\ \mathrm{Q}(x))$	Hypothesis	(1)
3 is an integer	Hypothesis	(2)
$(\mathrm{P}(3) \vee \mathrm{Q}(3))$	Universal Instantiation: 1, 2	(3)
$\neg\ \mathrm{P}(3)$	Hypothesis	(4)
$\mathrm{Q}(3)$	Disjunctive Syllogism: 3, 4	(5)

Showing an Argument with Quantified Statements is Invalid

Consider the following argument: $\begin{array}{l} \exists\ x\mathrm{P}(x) \\ \exists\ x\mathrm{Q}(x) \\ \hline \therefore \exists\ x(\mathrm{P}(x) \wedge\ \mathrm{Q}(x)) \end{array}$

Using a supposed domain $\{c, d\}$, with truth values of $\begin{array}{c|cc} & \mathrm{P} & \mathrm{Q} \\ \hline \mathrm{c} & \mathrm{T} & \mathrm{F} \\ \mathrm{d} & \mathrm{F} & \mathrm{T} \end{array}$, the argument is invalid.

2 Proofs

2.1 Mathematical Definitions

- An integer x is *even* if there is an integer k such that $x = 2k$
- An integer x is *odd* if there is an integer k such that $x = 2k + 1$

Parity

The parity of a number is whether the number is odd or even.

- If 2 numbers are both even or both odd, they have the *same parity.*
- If 1 number is even and 1 number is odd, they have the *opposite parity.*

Rational

A number r is rational if there exists x and y such that

$$r = \frac{x}{y},\ y \neq 0$$

where the choice of x and y are not necessarily unique.

Divides

An integer x **divides** an integer y if and only if $x \neq 0$ and $y = kx$, for some integer k. x divides y is denoted as $x \mid y$. If x does not divide y, it is denoted as $x \nmid y$. If $x \mid y$, then y is said to be a multiple of x, and x is a **factor** or *divisor* of y.

Primality

An integer n is **prime** if and only if $n > 1$ and the only positive integers that divide n and 1 and n.

Composite

An integer n is **composite** if and only if $n > 1$ and there is an integer m such that $1 < m < n$ and $m \mid n$.

Properties of greater than and less than

If x and c are real numbers, then exactly 1 of the following is true:

$$x < c \quad x = c \quad x > c \quad \begin{array}{c} \neg(x < c) \Leftrightarrow x \geq c \\ \neg(x > c) \Leftrightarrow x \leq c \\ \hline x > c \implies x \geq c \\ x < c \implies x \leq c \end{array}$$

2.2 Introduction to Proofs

A **theorem** is a statement that can be proven to be true.

A **proof** consists of a series of steps, each of which follows logically from assumptions, or from previously proven statements, whose final step should result in the statement of the **theorem** being proven.

An **axiom** is a statement assumed to be true.

Example Theorem

Every positive integer is than or equal to its square.

Proof. let x be an integer, where $x > 0$.
Since x is an integer and $x > 0$, then $x \geq 1$.
Since $x > 0$, we can multiple both sides of the inequality by x to get

$$(x \cdot x \geq 1 \cdot x) = (x^2 \geq x)$$

□

Proof by Exhaustion

if $n \in \{-1, 0, 1\}$, *then* $n^2 = |n|$

Proof.

$$\begin{aligned} n &= -1 & (-1)^2 &= 1 = |-1| \\ n &= 0 & (0)^2 &= 0 = |-1| \\ n &= 1 & (1)^2 &= 1 = |1| \end{aligned}$$

□

Counter Example

An assignment of values to variables that shows that a universal statement is false.

2.3 Writing Proofs: Best Practices

Allowed assumptions in proofs

- the rules of algebra
- the set of integers is closed under addition, multiplication, and subtraction
- every integer is either even or odd
- if x is an integer, there is no integer between x and $x + 1$
- the relative order of any two real numbers, $x, y \in \mathbb{R}$
- the square of any real number is greater than or equal to 0

Best practices when writing proofs

- indicate when the proof starts and ends
- write proofs in complete sentences
- give the reader a road-map of what has been shown, what is assumed, and where the proof is going
- introduce each variable when the variable is used for the first time
- a block of equations should be introduced with English text and each step that does <u>not</u> follow from algebra should be justified

Common mistakes in proofs

- generalizing from examples
- skipping steps
- circular reasoning
- assuming facts that have not yet been proven

2.4 Writing Direct Proofs

In a **direct proof** of a conditional statement, the hypothesis p is assumed to be true and the conclusion c is proven as a direct result of the assumption.

After the assumptions are stated, a direct proof proceeds by proving the conclusion is true.

For example,

The square of every odd integer is also odd.
$$\downarrow$$
Let n be an integer that is odd. We will show that n^2 is also odd.

Direct Proof format

$$\boxed{\begin{array}{c}\text{Assume hypothesis}\\ \vdots \\ \text{Derive conclusion}\end{array}}$$

2.5 Proof by Contrapositive

A **proof by contrapositive** proves a conditional statement of the form $p \implies c$ by showing that the contrapositive $\neg c \implies \neg p$ is true. In other words, $\neg c$ is assumed to be true and $\neg p$ is proven as a result of $\neg c$.

For example,

The square of every odd integer is also odd.
$$\downarrow$$
Let n^2 be an integer that is *not* odd. We will show that n is also *not* odd.

Contrapositive Proof format

$$\boxed{\begin{array}{c}\text{Assume }\neg\text{conclusion}\\ \vdots \\ \text{Show }\neg\text{hypothesis}\end{array}}$$

2.6 Proof by Contradiction

A **proof by contradiction** starts by assuming that the theorem is false and then shows that some logical inconsistency arises as a result of the assumption. A proof by contradiction is sometimes called an **indirect proof**. A proof by contradiction shows the only option is for a theorem to be true to avoid logical errors.

For example,

The square of every odd integer is also odd.
$$\downarrow$$
Assume there is an even square of an odd integer. We will show there is a logical inconsistency.

Contradiction Proof format

Assume $\neg$theorem
$\vdots$
Show *logical inconsistency*

2.7 Proof by Cases

A **proof by cases** of a universal statement breaks the domain into different classes and gives a different proof for each class. The proof for each class is called a **case**. **Every** value in the domain *must* be included in at least one class.

For example,

The square of every odd integer is also odd.
$\downarrow$
Consider case n, where *condition.* We will show theorem is true for this case.
Consider case $n+1$, where *condition.* We will show theorem is true for this case.

Cases Proof format

Assume hypothesis, and partition domain
$\vdots$
Show *for each* case, the conclusion is true.

Without loss of generality

Sometimes the proof for two different cases are so similar, that it is repetitive to include both cases. When this happens, the two cases *can be merged into one case.* The term **without loss of generality, WLOG, or w.l.o.g.** is used in mathematical proofs to narrow the scope of the proof to one special case in situations when the proof can be easily adapted to apply the general case.

3 Sets

3.1 Sets and Subsets

A **set** is a collection of objects. Objects in a set are called **elements**. Order does not matter, and there are no duplicates.

Roster notation:

$$A = \{2, 4, 6, 10\}$$
$$B = \{4, 6, 10, 2\}$$
$$A = B$$

To show membership, use the $\in$ symbol. For example, $2 \in A$, while $7 \notin A$. The empty set, which contains nothing, typically uses the $\emptyset$ symbol, or $\{\}$. Sets can be finite, or infinite. **Cardinality** of a set is the number of elements in a set. For example, the cardinality of A is 4.

$$|A| = 4$$

Cardinality can be infinite. Consider the set of all the integers, $\mathbb{Z}$. $|\mathbb{Z}| = \infty$

$$\mathbb{N}: \text{ set of natural numbers} \quad = \{0, 1, 2, 3, \ldots\}$$

$$\mathbb{Z}: \text{ set of integers} \quad = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

$$\mathbb{B}: \text{ set of rational numbers} \quad = \{x | x = \frac{a}{b} \text{ where } a, b \in \mathbb{Z}, b \neq 0\}$$

$$\mathbb{R}: \text{ set of real numbers} \quad = \{x | x \text{ has a decimal representation}\}$$

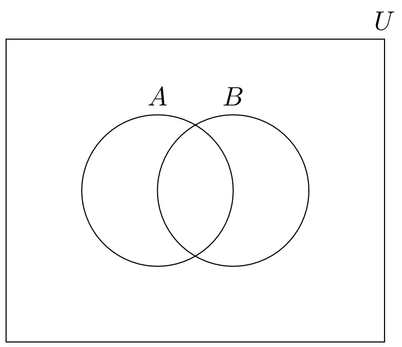
The subset operator is $\subseteq$

$$A \subseteq B \text{ if } \forall\, x(x \in A \implies x \in B)$$
$$A \subseteq A \text{ is true for } \underline{\text{any}} \text{ set}$$
$$\emptyset \subseteq A \text{ is true for } \underline{\text{any}} \text{ set}$$

Sometimes it is easier to define a set by defining properties that all the elements have. That is easy to do in **set builder notation**.

$$A = \{x \in S : A(x)\}, \text{ where } S \text{ is another set}$$
$$C = \{x \in \mathbb{Z} : 0 < x < 100 \text{ and } x \text{ is prime}\}.$$
$$D = \{x \in \mathbb{R} : |x| < 1\}$$

The **Universal Set**, usually called 'U', is a set that contains all elements mentioned in a particular context. For example, a discussion about certain types of real numbers, it would be understood that any element in the discussion is a real number. Sets are often represented pictorially with **Venn Diagrams**.



If $A \subseteq B$ and there is an element of B that is not an element of A, meaning $A \neq B$, then A is a **proper subset** of B, denoted as $A \subset B$. An important fact is that $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{B} \subset \mathbb{R}$

3.2 Sets of sets

Elements of sets can be sets themselves, consider $A = \{\{1,2\}, \emptyset, \{1,2,3\}, \{1\}\}$. The cardinality of A is 4, $|A| = 4$. Additionally, $\{1,2\} \in A$, but $1 \notin A$.

The **Powerset** of A, denoted as $P(A)$ is the set of all subsets of A. For example,

$$\begin{aligned} A &= \{1,2,3\} \\ P(A) &= \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\} \end{aligned}$$

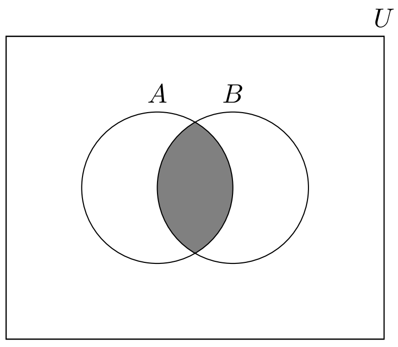
Cardinality of a Powerset

Let A be a finite set of cardinality n. Then the cardinality of the powerset of A is 2^n.

$$\begin{aligned} |A| &= n \\ |P(A)| &= 2^n \end{aligned}$$

3.3 Union and Intersection

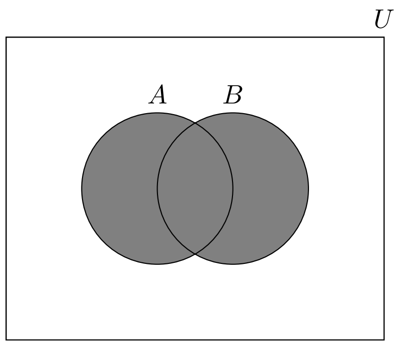
Intersection set operation: $\cap$. A intersected with B is defined to be the set containing elements which are in both A <u>and</u> B. That is, $A \cap B = \{x : x \in A \wedge x \in B\}$.



Intersection $\cap$ can also apply to infinite sets:

$$\begin{aligned} A &= \{x \in \mathbb{Z} : x \text{ is an integer multiple of } 2\} \\ B &= \{x \in \mathbb{Z} : x \text{ is an integer multiple of } 3\} \\ A \cap B &= \{x \in \mathbb{Z} : x \text{ is an integer multiple of } 6\} \end{aligned}$$

Union set operation $\cup$. A union with B is defined to be the set containing elements which are in A <u>or</u> B. That is, $A \cup B = \{x : x \in A \vee x \in B\}$.



A special notation, similar to $\sum$ or $\prod$ notation, allows for compound representation of the intersections or unions of a long sequence of sets.

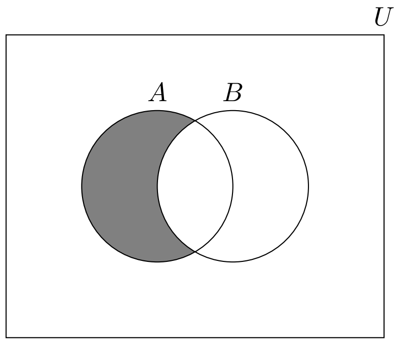
$$\bigcap_{i=1}^{n} A_i = A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_n = \{x : x \in A,\ \text{for } \underline{\text{all}}\ 1 \le i \le n\}$$
$$\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup A_3 \cup \cdots \cup A_n = \{x : x \in A,\ \text{for } \underline{\text{some}}\ 1 \le i \le n\}$$

Consider $A_j =$ a word with j letters, with $U =$ is the Oxford English Dictionary.

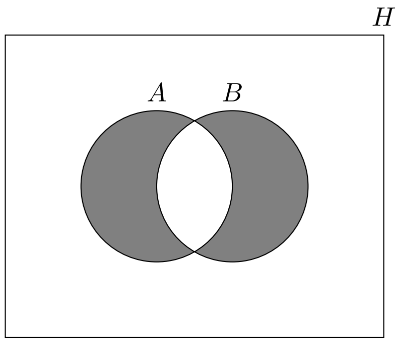
$$\bigcup_{j=1}^{10} A_j = \text{ the set of all words with 10 letters or fewer in the OED}$$
$$\bigcap_{j=1}^{45} A_j = \emptyset$$
$$\bigcup_{j=1}^{45} A_j = \text{ the set of all words in the OED.}$$

3.4 More set operations

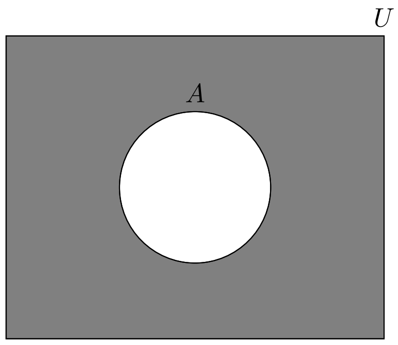
Difference set operation $-$. A difference with B is defined to be the set containing elements which are in A but not B. That is, $A - B = \{x : x \in A \wedge x \notin B\}$. A set difference is not strictly commutative, often $A - B \neq B - A$.



Symmetric Difference set operation $\triangle$. A symmetric difference with B is defined to be the set containing elements which are in A or B, but not A and B. That is, $A \triangle B = \{x : x \in A \oplus x \in B\}$.



Complement set operation . complement A is defined to be the set containing elements in U which are not in A. That is, $\overline{A} = \{x : x \in U \wedge x \notin A\}$.



Summary of Set Operations

Operation	Notation	Set Builder
Intersection	$A \cap B$	$\{x : x \in A \wedge x \in B\}$
Union	$A \cup B$	$\{x : x \in A \vee x \in B\}$
Difference	$A - B$	$\{x : x \in A \wedge x \notin B\}$
Symmetric Difference	$A \triangle B$	$\{x : x \in A \oplus x \in B\}$
Complement	$\overline{A}$	$\{x : x \in U \wedge x \notin A\}$

3.5 Set identities

The laws of propositional logic can be used to derive corresponding set identities. A **set identity** is an equation involving sets that is true, regardless of the contents of the sets used in the expression.

Law Name	$\cup$ Union	$\cap$ Intersection
Idempotent	$A \cup A = A$	$A \cap A = A$
Associative	$(A \cup B) \cup C = A \cup (B \cup C)$	$(A \cap B) \cap C = A \cap (B \cap C)$
Commutative	$A \cup B = B \cup A$	$A \cap B = B \cap A$
Distributive	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Identity	$A \cup \emptyset = A$	$A \cap U = A$
Domination	$A \cup U = U$	$A \cap \emptyset = \emptyset$
Double Complement	$\overline{\overline{A}} = A$	
Complement	$A \cup \overline{A} = \text{T}$	$A \cap \overline{A} = \text{F}$
DeMorgan	$\overline{A \cup B} = \overline{A} \cap \overline{B}$	$\overline{A \cap B} = \overline{A} \cup \overline{B}$
Absorption	$A \cup (A \cap B) = A$	$A \cap (A \cup B) = A$

3.6 Cartesian products

An **ordered pair** of items is written (x, y), where the first entry is x and the second entry is y. The use of () instead of {} indicates that order matters.
Cartesian Product of A and B, $A \times B = \{(a, b) : a \in A \wedge b \in B\}$

$$A = \{1, 2\} \qquad A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}$$
$$B = \{a, b, c\} \qquad B \times A = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$$

An ordered list of 3 items is called an **ordered triple**, denoted as (x, y, z). For a size of ≥ 4, use the term **n-tuple**. For example, (u, w, x, y, z).

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A \text{ for all } i \text{ such that} 1 \leq i \leq n\}$$

Another Example

$$A = \{a, b\} \qquad (a, 1, y, \beta) \in A \times B \times C \times D$$
$$B = \{1, 2\} \qquad (b, 1, x, \alpha) \in A \times B \times C \times D$$
$$C = \{x, y\} \qquad (1, b, x, \beta) \notin A \times B \times C \times D$$
$$D = \{\alpha, \beta\} \qquad \text{order matters}$$

$A \times A = A^2$, and in general,

$$A^k = \underbrace{A \times A \times \cdots \times A}_{k-times}$$

The **Cardinality of Cartesian Products**:

$$|A^n| = |A|^n$$
$$|A_1 \times A_2 \times \cdots| = |A_1| \cdot |A_2| \cdots$$

Strings

A sequence of characters is called a **string**. The set of characters used in a set of string is called the **alphabet** for the set of strings. The **length** of a string is the number of characters in the string. For example, the length of $'xxyxyx'$ is 6. The **empty string** is a string whose length is 0, and is usually denoted by λ. It is useful for A^0, for some alphabet A. $\{0,1\}^0 = \{\lambda\}$. If s and t are two strings, then the **concatenation** of s and t is the string obtained by putting s and t together.

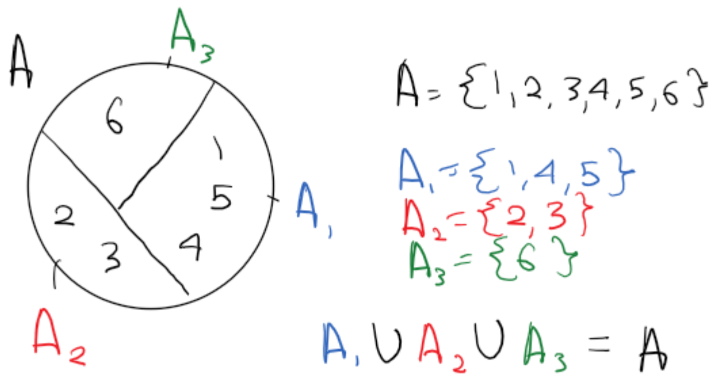
$$s = 010 \qquad st = 01011$$
$$t = 11 \qquad t0 = 110$$

Strings are used to specify passwords for computers or online accounts. Security systems vary with respect to the alphabet of characters allowed or required in a valid password. Strings also play an important rules in discrete mathematics as a mathematical tool to help count cardinality of sets.

3.7 Partitions

Two sets, A and B, are said to be **disjoint** if their intersection is empty ($A \cap B = \emptyset$). A sequence of sets, $A_1, A_2, A_3, \ldots, A_n$, is **pairwise disjoint** if every pair of distinct sets in the sequence is disjoint. A **partition** of a non-empty set A is a collection of non-empty subsets such that each element of A is in exactly one of the subsets. $A_1, A_2, A_3, \ldots, A_n$ is a partition for a nonempty set A if:

- For all i, $A_i \subseteq A$
- For all i, $A_i \neq \emptyset$
- $A_1, A_2, \ldots, A_n$ are pairwise disjoint
- $A = \bigcup_{i=1}^{n} A_i$, for some $n \in \mathbb{Z}^+$



4 Functions

4.1 Definition of functions

A **function** maps elements of one set X to elements of another set Y. A function from X to Y can be viewed as a subset of $X \times Y : (x, y) \in f$ if f maps x to y. The notation for a function is:

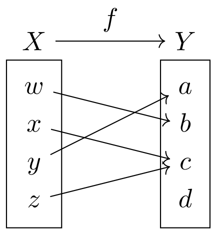
$$f : X \to Y, \text{ where } X \text{ is the } \textbf{domain} \text{ and } Y \text{ is the } \textbf{co-domain}.$$

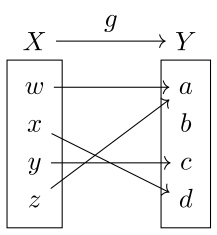
*if f maps an element of the domain to zero elements or more than one element of the target, then f is not *well-defined*

Arrow Diagram:

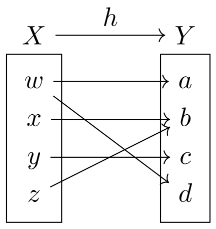
$$X = \{w, x, y, z\}$$
$$Y = \{a, b, c, d\}$$

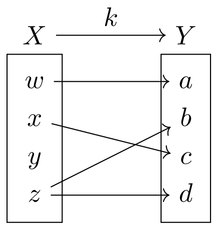
Well-defined functions:





Not well-defined functions:





For function $f : X \to Y$, an element y is in the **range** of f iff there is an $x \in X$ such that $(x, y) \in f$.

$$\text{Range of } f = \{y : (x, y) \in f, \text{ for some } x \in X\}$$

Two functions, f and g, are **equal** if f and g have the same domain and target and $f(x) = g(x)$ for every x in the domain.

$$\forall\ x : f(x) = g(x) \implies f = g$$

4.2 Floor and Ceiling functions

The **Floor** function, $\lfloor x \rfloor$

$$\text{floor} : \mathbb{R} \to \mathbb{Z}, \text{ where floor}(x) = \text{the largest integer } y \text{ such that } y \leq x.$$

Notation: $\text{floor}(x) = \lfloor x \rfloor$

The **Ceiling** function, $\lceil x \rceil$

$$\text{ceiling} : \mathbb{R} \to \mathbb{Z}, \text{ where ceiling}(x) = \text{the smallest integer } y \text{ such that } y \geq x.$$

Notation: $\text{ceiling}(x) = \lceil x \rceil$

Examples of floor and ceiling:

$$\begin{aligned} \lceil 4.32 \rceil &= 5 & \lfloor 4.32 \rfloor &= 4 \\ \lceil -4.32 \rceil &= -4 & \lfloor -4.32 \rfloor &= -5 \\ \lceil 4 \rceil &= 4 & \lfloor 4 \rfloor &= 4 \\ \lceil -4 \rceil &= -4 & \lfloor -4 \rfloor &= -4 \end{aligned}$$

4.3 Properties of functions

A function $f : X \to Y$ is **one-to-one** or **injective** if $x_1 \neq x_2$ implies that $f(x_1) \neq f(x_2)$. f maps different elements in x to different elements in y.

A function $f : X \to Y$ is **onto** or **surjective** if the range of f is equal to the target Y. That is, $\forall\, y \exists\, x (y \in Y \land x \in X \land f(x) = y)$

A function $f : X \to Y$ is **bijective** if it is both **injective** and **surjective**. A **bijective** function is called a **bijection**, or a **one-to-one correspondence**.

When the domain and target are finite sets, it is possible to infer information about their relative sizes based on whether a function is one-to-one or onto.

$$\begin{aligned} &f : D \to T \text{ is } \textbf{one-to-one} && \Longrightarrow && |D| \leq |T| \\ &f : D \to T \text{ is } \textbf{onto} && \Longrightarrow && |D| \geq |T| \\ &f : D \to T \text{ is } \textbf{bijective} && \Longrightarrow && |D| = |T| \end{aligned}$$

4.4 The inverse of a function

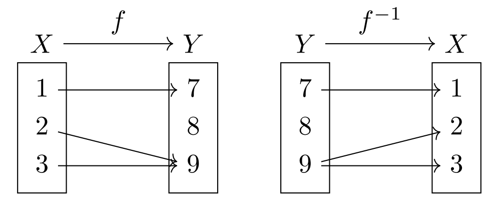
If a function $f : X \to Y$ is a *bijection*, then the **inverse** of f is obtained by exchanging the first and second entries in each pair in f.

$$\text{given } f : X \to Y$$
$$\text{inverse } f^{-1} : \{(y, x) : (x, y) \in f\}$$

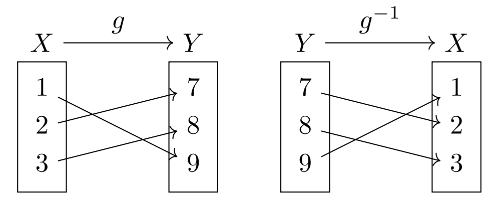
Reversing the cartesian pair does not always create a well-defined function. *Some functions do not have an inverse.*

Examples:

$$\begin{aligned} X &= \{1, 2, 3\} & f &= \{(1, 7), (2, 9), (3, 9)\} \\ Y &= \{7, 8, 9\} & g &= \{(1, 9), (2, 7), (3, 8)\} \end{aligned}$$



f^{-1} is not well defined, therefore f does not have an inverse.



g^{-1} is well defined, therefore g does have an inverse.

4.5 Composition of functions

The process of applying a function to the result of another function is called **composition**.

$$\begin{aligned} f &: X \to Y \\ g &: Y \to Z \\ (g \circ f) &: X \to Z, \text{ such that } \forall\ x : x \in X, (g \circ f)(x) = g(f(x)) \end{aligned}$$

Remember that order matters, as often $(g \circ f)(x) \neq (f \circ g)(x)$. However, composition is associative:

$$(f \circ g \circ h)(x) = ((f \circ g) \circ h)(x) = (f \circ (g \circ h))(x) = f(g(h(x)))$$

Identity Function

The **Identity Function** maps a set onto itself and maps every element to itself. It is notated as $I_A : A \to A$, where A is the set it maps. There are a number of identities about the Identity Function.

Let $f : A \to B$ be a bijection. Then,

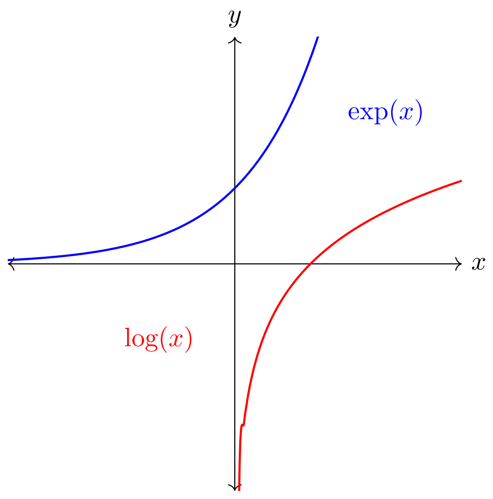
$$f \circ f^{-1} = I_B \text{ and } f^{-1} \circ f = I_A$$

4.6 Logarithms and exponents

The **Exponential** function, $\exp_b : \mathbb{R} \to \mathbb{R}^+, \exp_b(x) = b^x$. b is the base of the exponent and x is the exponent.

Properties of exponents:

$$\begin{aligned} b^x b^y &= b^{x+y} & b &\in \mathbb{R}^+ & c &\in \mathbb{R}^+ \\ (b^x)^y &= b^{xy} & x &\in \mathbb{R} & y &\in \mathbb{R} \\ \frac{b^x}{b^y} &= b^{x-y} \\ (bc)^x &= b^x c^x \end{aligned}$$



The **Logarithms** function, $\log_b : \mathbb{R} \to \mathbb{R}^+, \log_b(y) = x$. b is the base of the logarithm and x is the exponent.

Properties of exponents:

$$\begin{aligned}
\log_b(xy) &= \log_b x + \log_b y & \qquad & b \in \mathbb{R}^+ \\
\log_b\left(\frac{x}{y}\right) &= \log_b x - \log_b y & \qquad & c \in \mathbb{R}^+ \\
\log_b(x^y) &= y\log_b x & \qquad & x \in \mathbb{R} \\
\log_c x &= \frac{\log_b x}{\log_b c} & \qquad & y \in \mathbb{R}
\end{aligned}$$

5 Boolean Algebra

5.1 An introduction to Boolean Algebra

Boolean Algebra is a set of rules/operations for working with variables whose values are either 0 or 1. It corresponds highly to propositional logic.

Boolean Multiplication, denoted by $\cdot$.

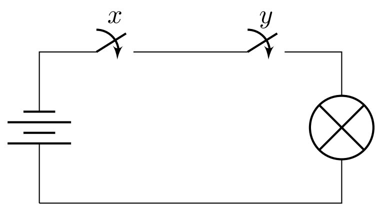
Boolean $\cdot$	Logic $\wedge$
$0 \cdot 0 = 0$	$\mathrm{F} \wedge \mathrm{F} = \mathrm{F}$
$0 \cdot 1 = 0$	$\mathrm{F} \wedge \mathrm{T} = \mathrm{F}$
$1 \cdot 0 = 0$	$\mathrm{T} \wedge \mathrm{F} = \mathrm{F}$
$1 \cdot 1 = 1$	$\mathrm{T} \wedge \mathrm{T} = \mathrm{T}$

Boolean Addition, denoted by $+$.

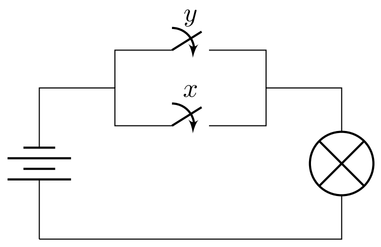
Boolean $\cdot$	Logic $\vee$
$0 + 0 = 0$	$\mathrm{F} \vee \mathrm{F} = \mathrm{F}$
$0 + 1 = 1$	$\mathrm{F} \vee \mathrm{T} = \mathrm{T}$
$1 + 0 = 1$	$\mathrm{T} \vee \mathrm{F} = \mathrm{T}$
$1 + 1 = 1$	$\mathrm{T} \vee \mathrm{T} = \mathrm{T}$

Boolean Complement, denoted by $\bar{}$.

Boolean $\bar{}$	Logic $\neg$
$\bar{0} = 1$	$\neg\mathrm{F} = \mathrm{T}$
$\bar{1} = 0$	$\neg\mathrm{T} = \mathrm{F}$



Shannon Circuit (AND $\cdot$)



Switching Circuit (OR $+$)

Variables that can have a value of either 1 or 0 are called **Boolean Variables**. Boolean expressions are made of boolean variables. There are also common shorthand ways of notating operations.

$$x \cdot y + 1 \cdot \bar{z} = xy + \bar{z}$$
$$x + z + \overline{0 + y} = x + z \cdot \bar{y}$$

Law Name	$+$ OR	$\cdot$ AND
Idempotent	$x+x=x$	$x\cdot x=x$
Associative	$(x+y)+z=x+(y+z)$	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$
Commutative	$x+y=y+x$	$x\cdot y=y\cdot x$
Distributive	$x+(y\cdot z)=(x+y)\cdot(x+z)$	$x\cdot(y+z)=(x\cdot y)+(x\cdot z)$
Identity	$x+0=x$	$x\cdot 1=x$
Domination	$x+1=1$	$x\cdot 0=0$
Double Complement	$\overline{\overline{x}}=x$	
Complement	$x+\overline{x}=1$	$x\cdot\overline{x}=0$
DeMorgan	$\overline{x+y}=\overline{x}\cdot\overline{y}$	$\overline{x\cdot y}=\overline{x}+\overline{y}$
Absorption	$x+(x\cdot y)=x$	$x\cdot(x+y)=x$

5.2 Boolean functions

A **boolean function** is a function which maps $B^k \to B$, where $B=\{0,1\}$. For example, consider $f : B^3 \to B$

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

x	y	z	$f(x,y,z)$	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{x}yz$
1	0	0	1	$x\overline{y}\overline{z}$
1	0	1	1	$x\overline{y}z$
1	1	0	0	
1	1	1	1	xyz

This function is equivalent to $f(x,y,z)=x\overline{y}+yz$. This can be determined using the boolean table and a strategy of combining the cases.

$$\begin{aligned} f(x,y,z) &= \bar{x}yz + x\overline{y}\overline{z} + x\overline{y}z + xyz \\ &= y(\bar{x}z+xz)+\overline{y}(x\bar{z}+xz) \\ &= y(z\cdot 1)+\overline{y}(x\cdot 1) \\ &= yz+\overline{y}x = \overline{y}x+yz \\ &= x\overline{y}+yz \end{aligned}$$

A **literal** is a boolean variable or the complement of a boolean variable, for example x or $\bar{x}$. In a boolean function whose input variables are $v_1, v_2, \ldots, v_k$, a *mini-term* is a product of literals $u_1, u_2, \ldots, u_k$, such that u_j is either v_j or $\overline{v_j}$

5.3 Disjunctive and conjunctive normal form

A boolean expression that is the sum of literals is said to be in *disjunctive normal form*, **DNF**. It has the following form:

$$c_1+c_2+\cdots+c_m, \text{ where } c_j \text{ for } j\in\{1,\ldots,m\} \text{ is a product of literals.}$$

For example, $\bar{x}y\bar{z}+xy+w+y\bar{z}w$. The complement only applies to a single variable and <u>no</u> addition within a term.

A boolean expression that is the product of sums of literals is said to be in *conjunctive normal form*, **CNF**. It has the following form:

$$d_1+d_2+\cdots+d_m, \text{ where } d_j \text{ for } j\in\{1,\ldots,m\} \text{ is a sum of literals.}$$

Each d_j is called a clause, and complements are only applied to a single variable. Additionally, there is no multiplication within variables. An example is $(\bar{x}+y+z)(x+\overline{y})(w)(y+\bar{z}+w)$.

5.4 Functional completeness

A set of operators is functionally complete if any boolean function can be expressed using only operations from the set. Two expressions can be added using only multiplication and complement.

$$x + y = \overline{\overline{x}\,\overline{y}} \text{ DeMorgan's Law}$$

DeMorgan's Law can be extended to more than two boolean variables:

$$x + y + w + z = \overline{\overline{x}\,\overline{y}\,\overline{w}\,\overline{z}}$$

The same can be said about the addition variant of the law:

$$xy = \overline{\overline{x} + \overline{y}}$$
$$xywz = \overline{\overline{x} + \overline{y} + \overline{w} + \overline{z}}$$

The **NAND** operation, $\uparrow$, and the **NOR** operation, $\downarrow$.

x	y	$x \uparrow y$	$x \downarrow y$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

{NAND} is functionally complete, it can create the complement, from which all other possible gates can be created.

x	$x \uparrow x$
0	1
1	0

From the complement, AND can be created, and {AND, COMPLEMENT} has already been proven to be functionally complete.

$$xy = \overline{x \uparrow y} = (x \uparrow y) \uparrow (x \uparrow y)$$

5.5 Boolean satisfiability

The **Boolean Satisfiability problem**, called the *SAT* for short, takes the boolean expression as an input and asks whether it is possible to set the values of the variables so that the expression evaluates to 1.

$$\text{If } \exists\, x \exists\, y \ldots B(x, y, \ldots), \text{ then the expression is } \textbf{satisfiable}$$
$$\text{If } \forall\, x \forall\, y \ldots \neg B(x, y, \ldots), \text{ then the expression is } \textbf{unsatisfiable}$$

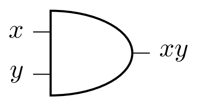
Expressions in DNF form are very easy to determine satisfiability. If there is <u>any</u> term which does *not* contain a variable and its complement, it is satisfiable. For example,

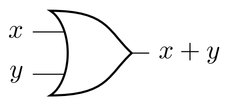
$$x\bar{y}z\bar{x} + \overbrace{\bar{w}xy\bar{z}}^{\text{no self-complements}} + \bar{w}xw\bar{x} + xy\bar{z}z$$

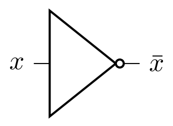
The above equation *is* satisfiable because there is a term which does not contain a self-complement.

5.6 Gates and circuits

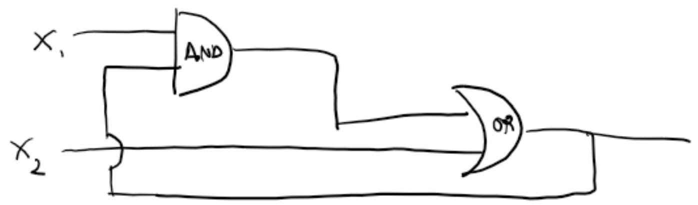
Circuits are built from electrical devices called **gates**.

AND: 

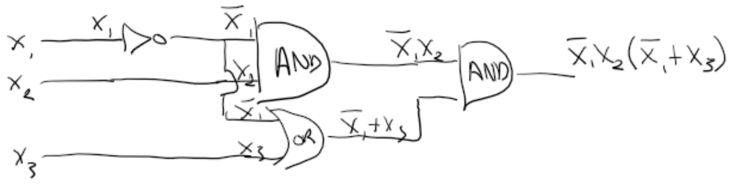
OR: 

NOT/INVERTER: 

The boolean function $f(x_1, x_2) : (f(x_1, x_2) \cdot x_1) + x_2$. Yes, circuits can contain recursion.

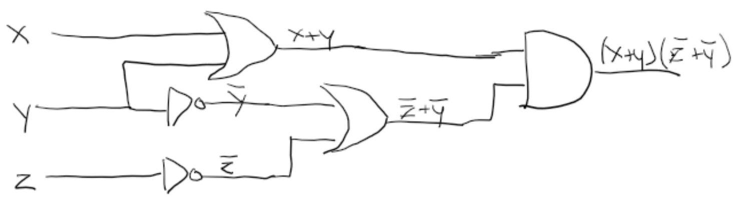


Boolean expressions can also be constructed by following the logic or a circuit.



$$f(x_1, x_2, x_3) = \bar{x}_1 x_2 (\bar{x}_1 + x_3)$$

An example of constructing a circuit from a boolean expression, $f(x, y, z) : (x + y)(\bar{z} + \bar{y})$



Designing Circuits

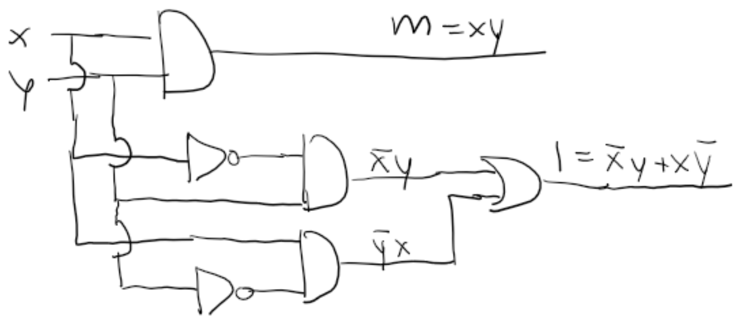
1. Build an input/output table with the desired output for every combination of input
2. Construct a boolean expression that computes the same function as the function specified in the input/output table
3. Construct a digital circuit that realizes the boolean expression

I/O for sum of two bits, x, y

x	y	m	l
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

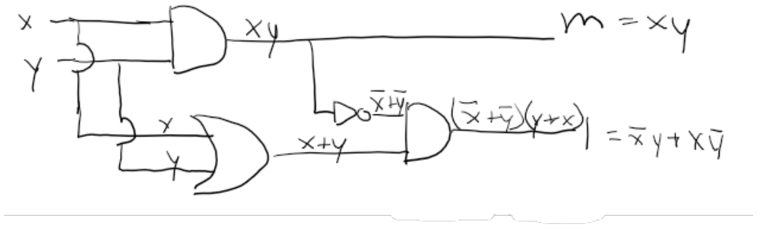
$m = xy$ most significant bit

$l = x\bar{y} + \bar{x}y$ least significant bit

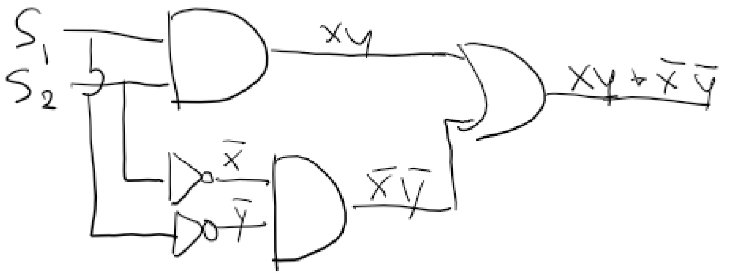


**this method does not necessarily give the most efficient circuit.*

Circuits with fewer gates cost less to manufacture. Here is a simplified circuit to add two bits:



Here is another example with boolean logic for light switches:



6 Relation and Digraphs

6.1 Introduction to binary relations

A **Binary Relation** between two sets A and B is a subset R of $A \times B$. It is binary because it is between two sets.

$$\text{for } a \in A \land b \in B, (a, b) \in R \text{ is denoted as } a\mathrm{R}b$$

For example, consider the relation C between $\mathbb{R}$ and $\mathbb{Z}$:

$$x\mathrm{C}y \text{ if } |x - y| \leq 1, \text{ where } x \in \mathbb{R} \text{ and } y \in \mathbb{Z}$$

If A and B are finite, then relation R between A and B can be represented by a set of ordered pairs.

Matrix Representation

$$P = \{\text{Sue}, \text{Harry}, \text{Sam}\}$$
$$\text{File} = \{\text{File A}, \text{File B}, \text{File C}, \text{File D}\}$$

$$\begin{array}{l|cccc|} & \text{File A} & \text{File B} & \text{File C} & \text{File D} \\ \text{Sue} & 0 & 1 & 1 & 1 \\ \text{Harry} & 1 & 0 & 0 & 0 \\ \text{Sam} & 0 & 0 & 0 & 0 \end{array}$$

$$\text{An element is } \begin{array}{l} 1 \text{ if } p\mathrm{R}f \text{ is true} \\ 0 \text{ if } p\mathrm{R}f \text{ is false} \end{array}$$

Arrow Diagram

$$A = \{a, b, c, d, e\}$$
$$R \subseteq A \times A$$
$$R = \{(a, b), (b, c), (e, c), (c, e), (d, a), (d, d)\}$$

$$\begin{tikzcd} & a \arrow[dl] & \\ b \arrow[dr] & & e \arrow[dl, bend left] \\ & c \arrow[ur, bend left] & d \arrow[uul] \arrow[loop right] \end{tikzcd}$$

Arrow Diagram vs. Matrix Representation

$$A = \{1, 2, 3, 4\}$$
$$R = \{(1, 2), (1, 3), (2, 2), (2, 3), (3, 4), (4, 3)\}$$

$$\begin{tikzcd} 1 \arrow[d] \arrow[dr] & 4 \arrow[d, bend left] \\ 2 \arrow[r] \arrow[loop left] & 3 \arrow[u, bend left] \end{tikzcd}$$

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 1 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

6.2 Properties of binary relations

A binary relation of R on set A is **Reflective** if for *every* $x \in A$, $x\mathrm{R}x$. For Arrow Diagrams, this means the graph contains self-loops:

$$\begin{tikzcd}
a \arrow[loop left] & & b \arrow[loop right]
\end{tikzcd}$$

For Matrix Representation, this means that the top left to bottom right diagonal are all 1's:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & - & 1 & - \\ - & - & - & 1 \end{pmatrix} \end{array}$$

A binary relation of R on set A is **Anti-reflective** if for *every* $x \in A$, $x\mathrm{R}x$ is *not* true. For Arrow Diagrams, this means the graph does not contain self-loops:

$$\begin{tikzcd}
a & & b
\end{tikzcd}$$

For Matrix Representation, this means that the top left to bottom right diagonal are all 0's:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & - & - & - \\ - & 0 & - & - \\ - & - & 0 & - \\ - & - & - & 0 \end{pmatrix} \end{array}$$

A binary relation of R on set A is **Symmetric** if and only if for *every* pair $x \in A$, $y \in Y$, either *both* $x\mathrm{R}y$ <u>and</u> $y\mathrm{R}x$, or *both* not $x\mathrm{R}y$ or not $y\mathrm{R}x$ is true. For Arrow Diagrams, this means that every arrow has an arrow going the other way:

$$\begin{tikzcd}
 & a \arrow[dl, bend left] \\
b \arrow[ur, bend left] \arrow[dr, bend left] & \\
 & c \arrow[ul, bend left]
\end{tikzcd}$$

For Matrix Representation, this means that the matrix is symmetric along the top left to bottom right diagonal:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} - & u & v & x \\ u & - & w & y \\ v & w & - & z \\ x & y & z & - \end{pmatrix} \end{array} \text{where} \quad \begin{array}{c} u \in \{0,1\} \\ \vdots \\ z \in \{0,1\} \end{array}$$

A binary relation of R on set A is **Anti-symmetric** if and only if for *every* pair $x \in A$, $y \in Y$, $x\mathrm{R}y$ xor $y\mathrm{R}x$. For Arrow Diagrams, this means that each arrow does not have an arrow going the other way:

$$\begin{tikzcd}
 & & a \arrow[loop right] \arrow[dl] \arrow[dddl] & \\
 & b \arrow[drr] \arrow[dd] & & \\
 & & & e \arrow[loop right] \\
 & c & & \\
 & & d \arrow[uur] \arrow[loop right] &
\end{tikzcd}$$

For Matrix Representation, this means that the matrix is anti-symmetric along the top left to bottom right diagonal:

$$\begin{array}{c|cccc|} & a & b & c & d \\ a & - & \bar{u} & \bar{v} & \bar{x} \\ b & u & - & \bar{w} & \bar{y} \\ c & v & w & - & \bar{z} \\ d & x & y & z & - \end{array} \quad \text{where} \quad \begin{array}{c} u \in \{0,1\} \\ \vdots \\ z \in \{0,1\} \end{array}$$

A binary relation of R on set A is **Transitive** if for *every* three elements $x, y, z \in A$, if $x\text{R}y$ and $y\text{R}z$, then $x\text{R}z$. Logically, $(x\text{R}y \wedge y\text{R}z) \implies x\text{R}z$. For Arrow Diagrams, this means the graph follows a hierarchy or kind of flow:

$$\begin{tikzcd}
& a \arrow[ld] \arrow[ldd] \arrow[loop above] & \\
b \arrow[d] & & \\
c & & e \arrow[ld, bend left] \\
& d \arrow[ru, bend left] \arrow[loop below] &
\end{tikzcd}$$

For Matrix Representation, it is much more difficult to determine transitivity, but here is an example:

$$\begin{array}{c|ccccc|} & a & b & c & d & e \\ a & 1 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 1 & 1 \\ e & 0 & 0 & 0 & 1 & 0 \end{array}$$

6.3 Directed graphs, paths, and cycles

A directed graph, or **Diagraph**, consists of a pair (V, E), where V is the set of vertices and E is the set of *directed edges*. It is a subset of $V \times V$.

- indegree: # of edges pointing towards a vertex, indegree$(u) = |\{v : (v, u) \in E\}|$
- outdegree: # of edges pointing away from a vertex, outdegree$(u) = |\{u : (v, u) \in E\}|$

A digraph is organized into a cartesian pair of the set of vertices and edge pairs:

$$\begin{aligned} &\text{Graph } G = (V, E) \\ &\qquad V = \{a, b, c, d\} \\ &\qquad E = \{(a, b), (b, c)(a, c), (d, d)\} \end{aligned}$$

$$\begin{tikzcd}
& a \arrow[ld] \arrow[dd] & \\
b \arrow[rd] & & d \arrow[loop right] \\
& c &
\end{tikzcd}$$

indegree$(c) = 2$	outdegree$(a) = 2$	a is the <u>tail</u> of edge (a, b)
indegree$(d) = 1$	outdegree$(d) = 1$	b is the <u>head</u> of edge (a, b)

A digraph is mathematically the same as a relation. Here is an example of the internet as a graph:

$$\begin{aligned} &\text{Graph } G = (V, E) \\ &\qquad V = \text{ set of all URLs} \\ &\qquad E = \text{ set of all hyperlinks from one URL to another URL} \end{aligned}$$

$$\begin{aligned} B &= \text{ Blog} \\ P &= \text{ Pediatrician website} \\ PC &= \text{ Pharmaceutical Company} \\ AoP &= \text{ Academy of Pediatrics} \end{aligned}$$

$$\begin{tikzcd}
& B \arrow[dl] \arrow[dr] \arrow[dd] & \\
P \arrow[rr] \arrow[dr] & & AoP \\
& PC \arrow[ur] &
\end{tikzcd}$$

Walks in Directed Graphs

A *walk* is a sequence of vertices and edges. For example, a walk from v_0 to v_l is notated as:

$$\langle v_0, (v_0, v_1), v_1, (v_1, v_2), \ldots, (v_{l-1}, v_l), v_l \rangle$$

where each edge in the sequence appears after its tail and before its head. A walk can also be a set of vertices:

$$\langle v_0, v_1, \ldots, v_l \rangle$$

provided that the edges between the vertices *actually exist.* The **length** of a walk is the number of edges traversed.

- **Open walk**: first and last vertices are *not* the same
- **Closed walk**: first and last vertices *are* the same.

Trails, Circuits, Paths, Cycles

- **Trail**: *open* walk in which no edge occurs more than once.
- **Circuit**: *closed* walk in which no edge occurs more than once. A circuit is a closed trail.
- **Path**: *trail* where no vertex occurs more than once.
- **Cycle**: *circuit* where no vertex occurs more than once, *except* for the first and last, which are the same.

Here are some examples:

$\langle a, c, d, a \rangle$ is a **cycle**: only the first and last vertices are repeated.
$\langle a, c, a, d, a \rangle$ is a **circuit**: vertices are repeated, but not edges
$\langle a, b, c, b, d \rangle$ is a **trail**: open, vertices are repeated, but not edges

6.4 Composition of relations

- one-to-one correspondence between digraphs and binary relations
- arrow diagram for a binary relation *is* a directed graph

The **composition** of relations R and S on set A is denoted as S∘R. Logically, this is what is means:

$$(a,c) \in S \circ R \iff \exists\, b : (b \in A \land (a,b) \in R \land (b,c) \in S)$$

Composition is applied *right to left*, much like composition of functions, or matrix transformations. Therefore, S∘R means R is applied first, then S.

R
$$\begin{tikzcd}
& a \arrow[dr] & \\
b \arrow[loop left] \arrow[dr] & & d \arrow[dl, bend left] \\
& c \arrow[ur, bend left] &
\end{tikzcd}$$

S
$$\begin{tikzcd}
& a & \\
b \arrow[ur] \arrow[rr] & & d \arrow[dl] \\
& c &
\end{tikzcd}$$

S∘R
$$\begin{tikzcd}
& a \arrow[dd] & \\
b \arrow[ur] \arrow[rr] & & d \\
& c \arrow[loop below] &
\end{tikzcd}$$

6.5 Graph powers and the transitive closure

A relation can be composed with itself. For example, consider relation P, which expresses parent-child relationship.

xPy means x is the parent of y.
xP∘Pz means x is the grandparent of z

A relation composed with itself also represents walks of different lengths.

P∘P represents all walks of length 2.
P∘P∘P represents all walks of length 3.

The Graph Power Theorem

: Let G be a directed graph. Let u and v be any two vertices in G. There is an edge from u to v in G^k if and only if there is a walk of length k from u to v in G.

$$R^1 = R$$
$$R^k = R \circ R^{k-1} \text{ for all } k \geq 2$$

Transitive Closure

$$G^+ = G^1 \cup G^2 \cup G^3 \cup \cdots$$

if G is not infinite, only up to the number of vertices are required for a complete graph of G^+

$$G^+ = G^1 \cup G^2 \cup G^3 \cup \cdots \cup G^{|V|}$$

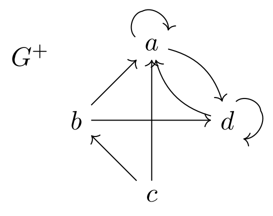
Here is an example of a series of graph powers:

G
$$\begin{tikzcd}
& a \arrow[dr, bend left] & \\
b \arrow[ur] & & d \arrow[ul, bend left] \\
& c \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

G^2
$$\begin{tikzcd}
& a \arrow[loop above] & \\
b \arrow[rr] & & d \arrow[loop right] \\
& c \arrow[uu] &
\end{tikzcd}$$

G^3
$$\begin{tikzcd}
& a \arrow[dr, bend left] & \\
b \arrow[ur] & & d \arrow[ul, bend left] \\
& c \arrow[ur] &
\end{tikzcd}$$

G^4
$$\begin{tikzcd}
& a \arrow[loop above] & \\
b \arrow[rr] & & d \arrow[loop right] \\
& c \arrow[uu] &
\end{tikzcd}$$

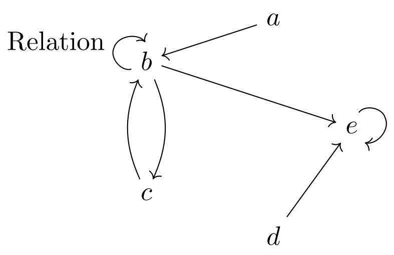


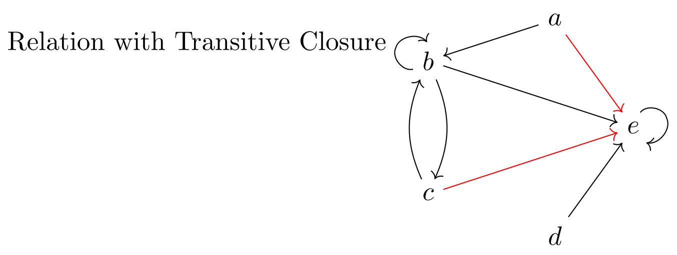
Finding the Transitive Closure of a Relation R on set A

Repeat until no pair is added to R.

If there are 3 elements $x, y, z \in A$ such that $x\mathrm{R}y$ and $y\mathrm{R}z$ but not $x\mathrm{R}z$, add $x\mathrm{R}z$ to R.

For example:





Edges added to find transitive closure are shown in red.

6.6 Matrix multiplication and graph powers

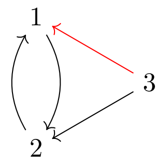
An $n \times m$ **matrix** over set S is an array of elements from S with n rows and m columns. Each element in a matrix is called an *entry*. A **square matrix** has the same number of rows and columns. Here are a number of example matrixes

$$\begin{bmatrix} 1 & 3 \\ 3 & -5 \\ -2 & -2 \end{bmatrix} \qquad \begin{bmatrix} 1.1 & 3.0 & -5.4 \\ -2.2 & -2.1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

3×2 matrix over $\mathbb{Z}$ $\quad$ 2×3 matrix over $\mathbb{Z}$ $\quad$ 2×2 matrix over $\{0, 1\}$

A directed graph G can be represented by a Matrix.

n vertices $\rightarrow$ $n \times n$ matrix over the set {0,1}, called an **adjacency matrix**



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

A **boolean matrix** is a matrix over the set {0,1}, and boolean addition and multiplication are used. The **dot product** of a matrix A and B is defined only if # of columns in A = # of rows in B

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\begin{array}{c}1\\ \times 1\\ \hline 1\end{array} + \begin{array}{c}0\\ \times 1\\ \hline 0\end{array} + \begin{array}{c}1\\ \times 1\\ \hline 1\end{array} = 1 = (A \times B)_{2,3}$$

Matrix Product

- denoted as AB or $A \cdot B$
- uses a series of dot products to compute

There are a number of properties of matrix multiplication:

~~Commutative~~	$AB \neq BA$
Associative	$(AB)C = A(BC)$
Distributive	$A(B + C) = AB + AC$
	$(B + C)A = BA + CA$
Multiplicative	$IA = A$ and $AI = A$
	$OA = A$ and $AO = O$
Dimension	$(m \times n) \cdot (n \times k) = (m \times k)$

k^{th} power of a matrix:

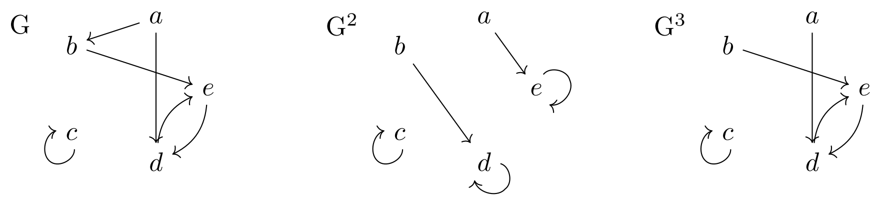
$$A^k = \underbrace{A \cdot A \cdots A}_{k \text{ times}}$$

If G is a digraph, G^k represents all walks of length k in G. There is an edge from vertex v to vertex w in G^k if and only if there is a walk of length *exactly* k from v to w in G. Matrix multiplication provides a systematic way of computing G^k.

1. Take *adjacency matrix* A for graph G
2. Compute A^k by repeated *matrix multiplication*
3. Matrix A^k is the *adjacency matrix* for graph G^k.

Relationship between powers of a graph and the powers of its adjacency matrix

Let G be a directed graph with n vertices and let A be the adjacency matrix for G. Then for $k \geq 1$, A^k is the adjacency matrix of G^k, where boolean addition and multiplication are used to compute A^k.



Example:

$$A = \begin{bmatrix}0&1&0&1&0\\0&0&0&0&1\\0&0&1&0&0\\0&0&0&0&1\\0&0&0&1&0\end{bmatrix} \quad A^2 = \begin{bmatrix}0&0&0&0&1\\0&0&0&1&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{bmatrix} \quad A^3 = \begin{bmatrix}0&0&0&0&0\\0&0&0&0&1\\0&0&1&0&0\\0&0&0&0&1\\0&0&0&1&0\end{bmatrix}$$

Matrix Sum

- denoted as $A + B$
- well-defined if same # of row and # of columns

$$(A+B)_{i,j} = A_{i,j} + B_{i,j} \text{ for all } i \text{ and } j \text{ in range of } A \text{ or } B$$

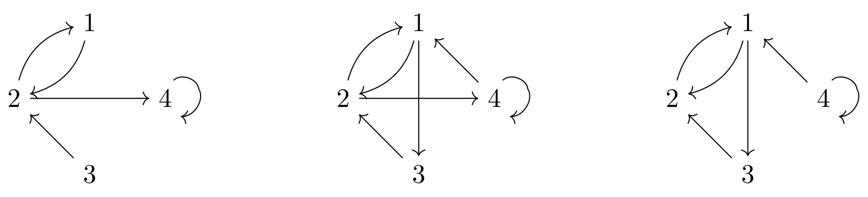
For example,

$$\begin{array}{ccccc} 0 & + & 1 & = & 1 \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & + & \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} & = & \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \\ A & + & B & = & A+B \end{array}$$

Addition and Graph Union

Let G and H be two directed graphs with the same vertex set. Let A be adjacency matrix for G and B the adjacency matrix for H.

Then the adjacency matrix for $G \cup H = A + B$, where boolean addition is used on the entries of matrices A and B.



$$\begin{array}{ccccc} & \cup & & = & \\ G & \cup & H & = & G \cup H \\ A & + & B & = & A+B \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & + & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} & = & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

For digraph G and adjacency matrix A for G:

$$G^+ = G^1 \cup G^2 \cup G^3 \cup \cdots \cup G^n$$
$$A^+ = A^1 + A^2 + A^3 + \cdots + A^n$$

Condition for well-defined matrix multiplication

$A_{m\times n} \times B_{s\times t}$ is only defined when $n = s$, and $A \times B$ has dimensions $m \times t$. For example:

$$A_{3\times 3} \begin{bmatrix} 0 & 2 & 3 \\ 1 & 0 & 1 \\ 2 & 0 & 1 \end{bmatrix} \times B_{3\times 1} \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} = (A \times B)_{3\times 1} \begin{bmatrix} 5 \\ 4 \\ 0 \end{bmatrix}$$

6.7 Partial orders

A relation on set A is a **partial order** if it is:

- reflexive
- transitive

- anti-symmetric

Notation: $a \preceq b$ used to express $a\mathrm{R}b$

Domain along with a partial order defined on it is denoted $(A, \preceq)$ and is called a **partial ordered set** or **poset**.

$$\preceq \neq \leq \text{ (notice the curves)}$$

Two elements of a poset are said to be *comparable* if $x \preceq y$ or $x \succeq y$. Otherwise they are said to be *incomparable*. A partial order is a *total order* if every two elements in the domain are *comparable*.

Here is an example of a partial order:

$x\mathrm{R}y$ if $x \leq y$

$$\begin{tikzcd}
 & 3 \arrow[dl] \arrow[dr] \arrow[dd] \arrow[loop above] & \\
6 \arrow[rr] \arrow[dr] \arrow[loop left] & & 15 \arrow[loop right] \\
 & 12 \arrow[ur] \arrow[loop below] &
\end{tikzcd}$$

- An element x is **minimal** element if there is no such $y \neq x$ such that $y \preceq x$
- An element x is **maximal** element if there is no such $y \neq x$ such that $x \preceq y$

Hasse Diagram

- useful way to depict a partial order on a finite set
- each element is represented by a point
- shows relationships by placing elements that are greater than others toward the top of the diagram.

Rules for placement and for connecting segments

- if $x \preceq y$, then make x appear lower in the diagram than y
- if $x \preceq y$, and there is no such z that $x \preceq z \preceq y$, then draw a segment connecting x and y

Examples: Hasse Diagrams for a partial order on the power set of $\{1, 2, 3\}$, and $\{1, 2, 3, 4, 5, 6\}$.

Hasse Diagram 1

$$\begin{tikzcd}
 & \{1,2,3\} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{1,2\} \arrow[d, dash] \arrow[dr, dash] & \{1,3\} \arrow[dl, dash] \arrow[dr, dash] & \{2,3\} \arrow[dl, dash] \arrow[d, dash] \\
\{1\} \arrow[dr, dash] & \{2\} \arrow[d, dash] & \{3\} \arrow[dl, dash] \\
 & \emptyset &
\end{tikzcd}$$

Hasse Diagram 2

$$\begin{tikzcd}
6 \arrow[d, dash] \\
5 \arrow[d, dash] \\
4 \arrow[d, dash] \\
3 \arrow[d, dash] \\
2 \arrow[d, dash] \\
1
\end{tikzcd}$$

The first example uses a rule of $A \preceq B \leftrightarrow A \subseteq B$, while the second example uses a rule of $x \preceq y \leftrightarrow x \leq y$.

In general, if two elements are incomparable, then they are not connected at all by a path of line segments or the only paths between x and y require a change in direction from up to down or down to up. Consider this partial order on the set $\{A, B, C, D, E, F, G\}$:

Hasse Diagram 3

$$\begin{tikzcd}
 & & & D \arrow[dl, dash] \arrow[dr, dash] & \\
G \arrow[d, dash] & & F \arrow[d, dash] & & E \arrow[ddl, dash] \\
B & & C \arrow[dr, dash] & & \\
 & & & A &
\end{tikzcd}$$

In this example, $B \preceq G$, and $A \preceq D$, but B and F are not comparable.

6.8 Strict orders and directed acyclic graphs

- Partial Order acts $\preceq$ on a domain
- Strict Order acts $\prec$ on a domain

A relation R is a **Strict Order** if R is *transitive, anti-symmetric,* and *anti-reflexive.*

- two elements are comparable if $x \prec y$ or $x \succ y$, and otherwise incomparable
- strict order is a *total order* if every pair of elements is comparable
- element x is **minimal** if no y exists such that $y \prec x$
- element x is **maximal** if no y exists such that $x \prec y$

Here is an example of a strict order:

$$\begin{tikzcd}
 & b \arrow[drr] & c \arrow[dr] & \\
a \arrow[ur] \arrow[urr] \arrow[rrr] & & & e \\
 & & & \\
 & d \arrow[r] & f &
\end{tikzcd}$$

maximal:	e and f
minimal:	a and d

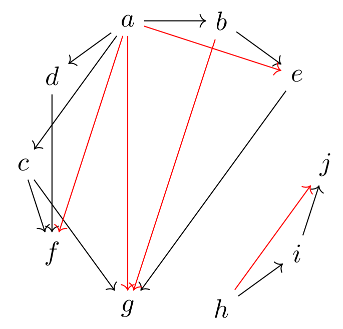
Strict orders are also anti-symmetric. Consider a relation R that is transitive and anti-reflexive. Then R is also anti-symmetric.

Directed Acyclic Graphs, DAGs

A directed acyclic graph is a directed graph that has no cycles. For example, consider this DAG:

$$\begin{tikzcd}
 & \text{Data Structs} \arrow[r] \arrow[dr] \arrow[dd] & \text{Operating Systems} \\
\text{Discrete Math} \arrow[ur] \arrow[dr] & & \text{Algorithms} \\
 & \text{Digital logic} \arrow[r] & \text{Computer Architecture}
\end{tikzcd}$$

Theorem: Directed Acyclic Graphs and Strict Orders Let G be a directed graph. G has no cycles if and only if G^+ is a strict order.

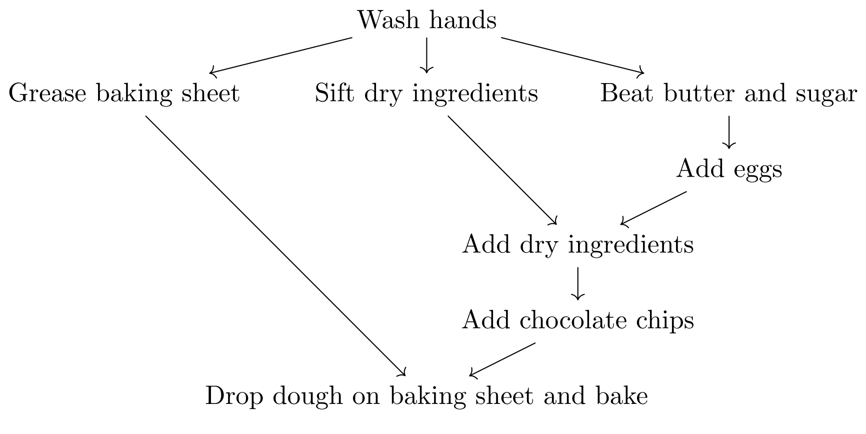


G is a DAG
Edges added by G^+ are shown in red
Minimal: a and h
Maximal: g, f, and j

Consider another example of precedence constraints for baking chocolate chip cookie:

- Wash hands
- Grease cooking sheet
- Sift together dry ingredients
- Beat together butter and sugar
- Add eggs to butter and sugar
- Add chocolate chips
- Drop spoonfuls of batter onto cookie sheet and bake

Converting this into a DAG, notice how incomparable tasks can be done simultaneously by different people:



Topological Sorts of DAGs

Consider a DAG which represents precedence constraints for a set of tasks. Need to find an ordering which does not violate any of the precedence constraints. A **topological** sort for a DAG is an ordering of vertices that is consistent with the edges in the graph.

- If there is an edge (u, v), then u must appear earlier than v in the topological sort.

$$\begin{tikzcd}
a \arrow[d] \arrow[dr] & \\
b & c
\end{tikzcd}$$

example topological sorts:
$\langle a, b, c \rangle$
$\langle a, c, b \rangle$

6.9 Equivalence relations

A relation R is an **equivalence relation** if:

- R is reflective
- R is transitive
- R is symmetric

For example:

$$\begin{tikzcd}
a \arrow[loop left] \arrow[dr, bend left] \arrow[dd, bend left] & & d \arrow[loop left] \arrow[dd, bend left] & \\
& b \arrow[loop right] \arrow[ul] \arrow[dl] \arrow[dl, bend left] & & e \arrow[loop right] \\
c \arrow[loop below] \arrow[uu, bend left] \arrow[ur] \arrow[uu] & & f \arrow[loop below] \arrow[uu, bend left] &
\end{tikzcd}$$

- Reflective
- Symmetric
- Transitive

If A is the domain of a equivalence relation and $a \in A$, then $[a]$ is called an **equivalence class**, defined to be the set of all $x \in A$, such that $a \sim x$.

For example, consider domain $\mathbb{Z}^+$ and $x \sim y$ if x and y have the same remainder when divided by 3.

$$[0] \text{ is } \{x \in \mathbb{Z}^+ : x \mod 3 = 0\}$$
$$[1] \text{ is } \{x \in \mathbb{Z}^+ : x \mod 3 = 1\}$$
$$[2] \text{ is } \{x \in \mathbb{Z}^+ : x \mod 3 = 2\}$$

Equivalence classes are either:

- completely identical, or
- completely disjoint

Theorem: Structure of Equivalence Relations Consider an equivalence relation on a set A. Let $x, y \in A$:

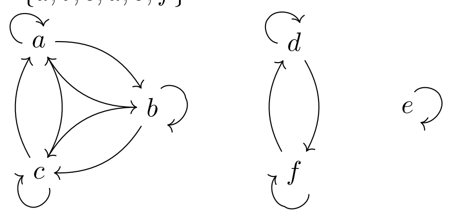
$$\text{If } x \sim y, \text{ then } [x] = [y]$$
$$\text{If } x \not\sim y, \text{ then } [x] \cap [y] = \emptyset$$

Theorem: Equivalence Relations define a Partition Consider and equivalence relation over a set A. The set of all distinct equivalence classes defines a partition of set A.

- The term "distinct" means that if there are two equivalent classes $[a] = [b]$, then set $[a]$ is the only included one.

For example:

$A = \{a, b, c, d, e, f\}$



$[a] = \{a, b, c\}$
$[d] = \{d, f\}$
$[e] = \{e\}$

6.10 N-ary relations and relational databases

A binary relation can be generalized to more than two sets. A relation on n sets is called an **N-nary Relation**. For example:

$(w, x, y, z) \in \mathbb{R}^4$ such that $wx = yz$
$(3, 12, 4, 9)$ would be in the relation because $3 \cdot 12 = 4 \cdot 9$
$(3, 8, 5, 6)$ would not be in the relation because $3 \cdot 8 \neq 5 \cdot 6$

Databases

A database is a large collection of data records that is searched and manipulated by a computer. The *regional database model* stores data records as relations.

The type of data stored in each entry of the n-tuple is called an attribute.

A query to a database is a request for a particular set of data.

A key is an attribute or set of attributes that uniquely identifies each n-tuple in the databases.

For example, Airlines use the combination of flight number, date, and departure time to uniquely identify a flight.

Selection

The **selection** operation chooses n-tuples from a relational database that satisfies particular conditions on their attributes. For example:

# boxes	Order Date	Complete?	Client City
8	6/19/2013	NO	Irvine
12	6/20/2013	YES	Huntington Beach
15	6/20/2013	YES	Huntington Beach
21	6/20/2013	NO	Irvine
3	6/21/2013	NO	Costa Mesa

Search[Complete=NO]

# boxes	Order Date	Complete?	Client City
8	6/19/2013	NO	Irvine
21	6/20/2013	NO	Irvine
3	6/21/2013	NO	Costa Mesa

Search[Complete=NO $\wedge$ Data $<$ 6/21/2013]

# boxes	Order Date	Complete?	Client City
8	6/19/2013	NO	Irvine
21	6/20/2013	NO	Irvine

Projection

The **projection** operation takes a subset of the attributes and deletes all other attributes in each of the n-tuples. For example:

# boxes	Order Date	Complete?	Client City
8	6/19/2013	NO	Irvine
12	6/20/2013	YES	Huntington Beach
15	6/20/2013	YES	Huntington Beach
21	6/20/2013	NO	Irvine
3	6/21/2013	NO	Costa Mesa

Project[Order Date, Client City]

Order Date	Client City
6/19/2013	Irvine
6/20/2013	Huntington Beach
6/20/2013	Irvine
6/21/2013	Costa Mesa

Select[Complete=NO], Project[City]

Client City
Irvine
Costa Mesa

7 Computation

7.1 An introduction to algorithms

An algorithm is a step by step method for solving a problem. It usually includes:

- name
- brief description
- description of input
- description of output
- sequence of steps to follow

Algorithms are often described in **pseudocode**

Assignment operator

```
x := y
```

Return statement

```
Return(value)
```

If-else statement

```
If(x = 5), y := 7

If(condition)                 If(condition)
  Step 1                        Step(s)
  Step 2                      Else
  ...                           Step(s)
  Step n                      End-if
End-if
```

For-loop

```
For i = s to t     <- first value is s, then s+1, until t is reached
  Step(s)
End-for
```

While-loop

```
While(condition)
  Step(s)
End-while
```

Nested Loops

```
Input: sequence a_1, ..., a_n; n

count := 0
For i = 1 to n-1
  For j = i+1 to n
    If(a_i = a_j) count := count+1
  End-for
End-for

Return(count)
```

7.2 Asymptotic growth of functions

Consider $f : \mathbb{Z}^+ \to \mathbb{R}^{\geq}$, where $\mathbb{R}^{\geq}$ denotes the set of non-negative real numbers. **Asymptotic growth** of a function f is a measure of how fast the object $f(n)$ grows as the input n grows. Classification of functions $\mathcal{O}, \Omega$, and Θ provide a way to concisely characterize the growth of a function.

$$f = \mathcal{O}(g) \ \text{"}f\text{ is Oh of }g\text{"}$$

Constant factors

$$\begin{aligned} 7n^3 &\to 7 \text{ is constant factor} \\ 5n^2 &\to 5 \text{ is constant factor} \\ 3 &\to 3 \text{ is constant factor} \end{aligned}$$

$\mathcal{O}$ notation

Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}^{\geq}$. Then $f = \mathcal{O}(g)$ if there is are positive real numbers c and n_0 such that for any $n \in \mathbb{Z}^+$ such that $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Constants c and n_0 are said to be a *witness* to the fact $f = \mathcal{O}(g)$

Ω notation

Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}^{\geq}$. Then $f = \Omega(g)$ if there is are positive real numbers c and n_0 such that for any $n \in \mathbb{Z}^+$ such that $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

$f = \Omega(g)$ is read "f is Omega of g"

Theorem: Relationship of $\mathcal{O}$-notation and Ω-notation

Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}^{\geq}$. Then $f = \Omega(g) \iff g = \mathcal{O}(f)$

Θ notation

Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}^{\geq}$.

$$f = \Theta(g) \text{ if} : f = \mathcal{O}(g) \land f = \Omega(g)$$

- $f = \Theta(g)$ is read "f is Theta of g"
- if $f = \Theta(g)$, then f is said to be the *order of* g.

Theorem: Asymptotic Growth of Polynomials

Let $p(n)$ be a degree-k polynomial of the form

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \text{ where } a_k > 0$$

Then $p(n)$ is $\Theta(n^k)$

Asymptotic Growth of Logarithm Functions with Different Bases

Let a and b be two real numbers greater than 1. Then

$$\log_a n = \Theta(\log_b n)$$

This is because of the fact that

$$\log_a n = \log_a b \cdot \log_b n, \text{ for } a, b > 1$$

when a function is said to be the $\mathcal{O}$ or Ω of a logarithm function, the base is often omitted because it is understood that as long as the base is greater than 1, the value of the base does not matter.

Growth rate of common functions

Constant Functions

- function that does not depend on n at all
- any constant function is $\Theta(1)$

Linear

- $\Theta(n)$

Function	Name
$\Theta(1)$	Constant
$\Theta(\log\log n)$	Log Log
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n\log n)$	$n\log n$
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^m)$ for $m \in \mathbb{Z}^+$	Power
$\Theta(c^n)$ for $c > 1$	Exponential
$\Theta(n!)$	Factorial

Rules about Asymptotic Growth

Let f, g, and h be functions from $\mathbb{Z}^+$ to $\mathbb{R}^{\geq}$.

- if $f = \mathcal{O}(h)$ and $g = \mathcal{O}(h)$, then $f + g = \mathcal{O}(h)$
- if $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$
- if $f = \mathcal{O}(g)$, $c \cdot f = \mathcal{O}(g)$, $c \in \mathbb{R}^{\geq}$
- if $f = \Omega$, $c \cdot f = \Omega(g)$, $c \in \mathbb{R}^{\geq}$
- if $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$, then $f = \mathcal{O}(h)$
- if $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$

7.3 Analysis of algorithms

Resources an algorithm requires to run

- time, called *time complexity*
- space, called *space complexity*
- Together called **computational complexity**

```
ComputeSum
Input: a_1,a_2,...,a_n   (n is length of sequence)
Output: the sum of the numbers in the sequence
sum := 0                 | 1 assignment operation
For i = 1 to n           | loop iterated n times
  sum := sum + a_i       |   for loop test and increments (2 operations)
End-for                  |   1 addition and 1 assignment (2 operations)
Return(sum)              | 1 op for return statement
```

$$\begin{aligned} f(n) &= 1 + n[2+2] + 1 \\ &= 1 + 4n + 1 \\ &= 4n + 2 \\ &= \mathcal{O}(n) \end{aligned}$$

Growth rates for different input sizes

$f(n)$	$n=10$	$n=50$	$n=100$	$n=1000$	$\cdots$
$\log_2 n$	$3.3\mu s$	$5.6\mu s$	$6.6\mu s$	$10\mu s$	$\cdots$
n	$10\mu s$	$50\mu s$	$100\mu s$	$1000\mu s$	$\cdots$
$n\log_2 n$	$.03ms$	$.28ms$	$.66ms$	$10ms$	$\cdots$
n^2	$.1ms$	$2.5ms$	$10ms$	$1s$	$\cdots$
n^3	$1ms$	$.125s$	$1s$	$16.7min$	$\cdots$
2^n	$1ms$	$35.7yrs$	$4\times10^{16}yrs$	$3.4\times10^{287}yrs$	$\cdots$

Worst-case analysis

Worst-case analysis evaluates the time complexity on the input which takes the longest time.

- upper bound: use $\mathcal{O}$-notation

  upper bound must apply for every input of size n
- lower bound: use Ω-notation

  lower bound need only apply for one possible input of n

Average-case analysis takes an average running time of algorithm on random inputs.

```
For(----)
  operations        -> linear (n)
End-for

For(----)
  For(----)
    operations      -> quadratic (n^2)
  End-for
End-for
```

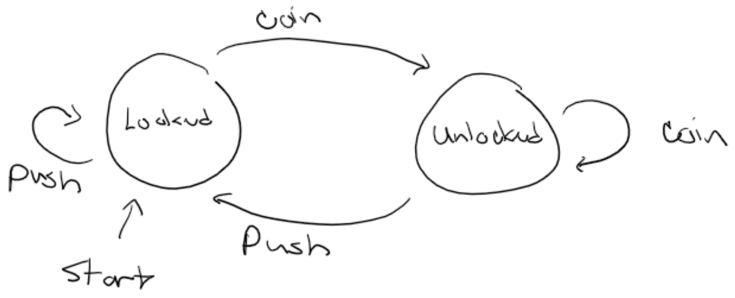
```
For(————)
   For(————)
      For(————)
         operations  —> cubic (n^3)
      End—for
   End—for
End—for
```

and so on. An algorithm runs in polynomial time if its time complexity is $\mathcal{O}^k$ for some fixed constant k. An algorithm is considered "efficient" if it runs in polynomial time. For example,

$$\mathcal{O}(n^5) \text{ is "efficient"}$$
$$\mathcal{O}(n^{\log n}) \text{ is \underline{not} "efficient"}$$

7.4 Finite state machines

A **finite state machine** consists of a finite set of states, with transitions between states triggered by different input actions. A finite state machine is sometimes called *finite state automation*.



states: Q ={locked, unlocked}

The reaction of a finite state machine to the input received is denoted by a **transitive function**, often denoted by the symbol 'δ'

$$\delta([\text{state}], [\text{action}]) = [\text{state}]$$

In the case of the coin machine,

$$\delta(\text{Locked}, \text{Coin}) = \text{Unlocked}$$

State transition table:

- rows represent current state
- columns represent possible inputs
- each entry for a particular row and column indicate the new state resulting from that state/input combination

For example, the state transition table for the coin machine is

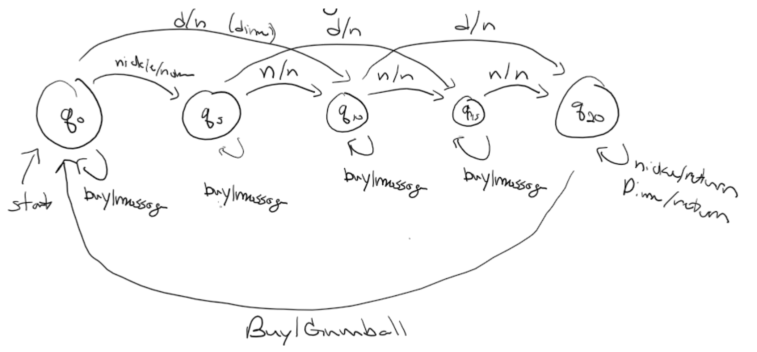
	Coin	Push
Locked	unlocked	locked
Unlocked	unlocked	locked

Components of a Finite State Machine

Notation	Description
Q	finite set of states
$q_0 \in Q$	q_0 is the start state
I	finite set of actions
$\delta : Q \times I \to Q$	transition function

FSM with Output

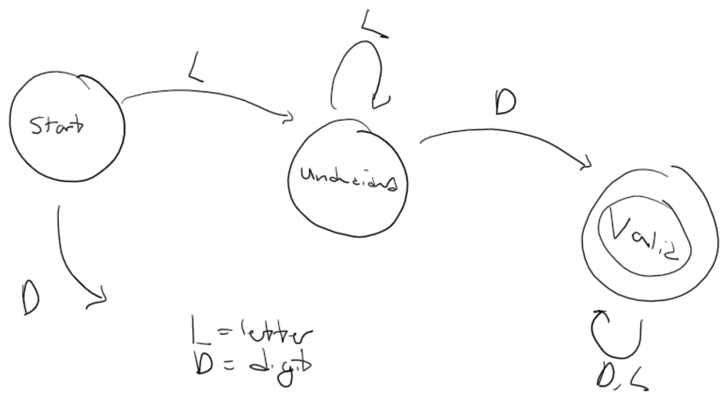
$$\begin{aligned} Q &= \{q_0, q_5, q_{10}, q_{15}, q_{20}\} \\ I &= \{\text{NICKLE}, \text{DIME}, \text{BUY}\} \\ O &= \{\text{Gumball}, \text{Return}, \text{Message}, \text{None}\} \end{aligned}$$



An accepted state is a state that is okay to end in.

$$A \subseteq Q,\ \text{Accepted states are a subset of the total states}$$

Example, recognizing valid password



A valid password must begin with a letter and contain at least one digit.

7.5 Turing machines

FSMs are unable to solve even simple computational tasks such as determining whether a binary string has more 0's than 1's.

Church-Turing conjecture

Any problem that can be solved efficiently on any computing device can be solved efficiently by a Turing Machine.

Definition of a Turing Machine

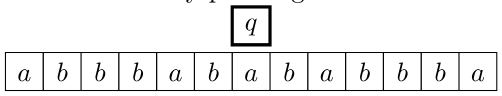
- memory is a 1-dimensional tape.

a	b	b	b	a	b	a	b	a	b	b	b	a

  example tape for $\{a, b, *\}$

- blank symbol (represented by a $*$ symbol)

- a configuration consists of the contents of the tape, the current state, and the tape cell to which the head is currently pointing



- action is determined by a transition function δ

Input to Turing Machine is the Input Alphabet, denoted by Σ, which much be a subset of the tape alphabet Γ

$$\Sigma \subset \Gamma$$

Components of a Turing Machine

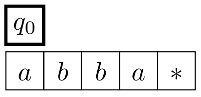
Notation	Description
Q	finite set of states
Γ	finite set of tape symbols
$\Sigma \subset \Gamma$	A subset of the tape symbols are input symbols
$q_0 \in Q$	q_0 is the start state
$q_{acc} \in Q$	q_{acc} is the accept state
$q_{rej} \in Q$	q_{rej} is the reject state
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$	Transition Function

Additional Rules

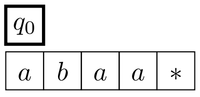
- if Turing machine reaches the accept state from a particular input x, the Turing machine **accepts** x
- if Turing machine reaches the reject state from a particular input x, the Turing machine **rejects** x
- if Turing machine *accepts* or *rejects* x, then the Turing machine **Halts** on x

Turing machine that accepts strings with 2 b's

	a	b	$*$
q_0	(q_0, a, R)	(q_1, b, R)	$(q_{rej}, *, L)$
q_1	(q_1, a, R)	(q_{acc}, b, R)	$(q_{rej}, *, L)$



$\uparrow$ Halts and accepts



$\uparrow$ Halts and rejects

Transition function for Turing machine that recognizes powers of 2:

-	a	x	$*$
q_0	$(q_{first}, *, R)$	$(q_{rej}, *, R)$	(q_{rej}, x, R)
q_{first}	(q_{even}, x, R)	(q_{first}, x, R)	$(q_{first}, *, R)$
q_{even}	(q_{odd}, a, R)	(q_{even}, x, R)	$(q_{first}, *, R)$
q_{odd}	(q_{even}, x, R)	(q_{odd}, x, R)	$(q_{first}, *, R)$
q_{ret}	(q_{ret}, a, R)	(q_{rej}, x, R)	$(q_{first}, *, R)$

7.6 Decision problems and languages

$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
2 & & 4 \\
 & 3 \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Symbol set	( ) , ; 0 1 2 3 4 5 6 7 8 9
Graph encoding	4; (1, 2)(1, 4)(3, 1)(3, 4)

Turing Machine can only accept or reject on input. This limits the class of problems answerable by a turing machine to **yes** or **no** problems.

- **Decision Problem**: given a boolean expression, is there an assignment to the boolean expression that causes the expression to evaluate to 1?
- **Search Problem**: given a boolean expression, find an assignment to the boolean expression that causes the expression to evaluate to 1 if one exists, or output that no such assignment exists.

If Σ is a finite alphabet, then a subset of Σ^* is called a *language* over Σ.

Language computed by a Turing Machine

Let Σ denote a finite alphabet and let L be a language over Σ. A turing machine M **computes language L**, or **decides language L** if for every $x \in \Sigma$, if $x \in L$, then M rejects x in a finite number of steps.

- **Time Complexity** is measured by how many steps taken by a Turing machine on a particular input.
- **Space Complexity** is measured by the number of tape cells that the turing machine uses in the course of it execution on a particular input.

A language is *incomputable* if there is no turing machine that computes the language.

8 Induction and Recursion

8.1 Sequences

A **sequence** is a special type of function in which the domain is the set of consecutive integers.

When a function is specified as a sequence, using subscripts to denote input is more common, so g_k is used instead of $g(k)$

A value g_k is called a **term**, and k is the *index* of g_k

For example:

$$g_1 = 3.67 \qquad g_2 = 2.88$$
$$g_3 = 3.25 \qquad g_4 = 3.75$$

$$g(k) = 3.67, 2.88, 3.25, 3.75$$

An entire sequence is denoted by $\{gk\}$, whereas g_k is used to denote a single term in the sequence.

A sequence commonly starts with 0 or 1, but it could be *any* integer.

Finite sequence

A sequence with a finite domain is a **finite sequence**. In a finite sequence, there is an *initial index* m and a *final index* n.

Infinite sequence

A sequence with an infinite domain is a **infinite sequence**. In an infinite sequence, there is an *initial index* m and the sequence is defined for indices $k \geq m$:

$$a_m, a_{m+1}, a_{m+2}, a_{m+3}, \ldots$$

A sequence can be specified by an **explicit formula**, such as $d_k = 2^k$ for $k \geq 1$.

$$\{d_k\} = 2, 4, 8, 16, \ldots$$

Increasing and Decreasing Sequences

- a sequence is *increasing* if for every two consecutive indices, k and $k+1$, $a_k < a_{k+1}$
- a sequence is *non-decreasing* if for every two consecutive indices, k and $k+1$, $a_k \leq a_{k+1}$

For example,

$$2 < 4 < 5 < 6 \text{ increasing } \textit{and} \text{ non-decreasing}$$
$$2 \leq 4 \leq 5 \leq 6 \text{ non-decreasing } \textit{but} \text{ not increasing}$$

The same relationship can be said for ***decreasing*** *and* ***non-increasing***.

Geometric Sequences

A **geometric sequence** is a sequence of real numbers where each term is found by taking the previous term and multiplying it by a fixed number called the **common ratio**.

For example, with an *initial term*: 4, and *common ratio*: $\frac{1}{2}$,

$$4, 2, \frac{1}{2}, \frac{1}{4}, \ldots$$

Arithmetic Sequence

An **arithmetic sequence** is a sequence of real numbers where each term after the initial term is found by taking the previous term and adding a fixed number called the **common difference**.

For example, with an *initial value*: 2, and *common difference*: 3,

$$2, 5, 8, 11, \ldots$$

8.2 Recurrence relations

A rule that defines a term a_n as a function of previous terms in the sequence is called a **recurrence relation**

For example,

$$\begin{aligned} a_0 &= a \text{ initial value} \\ a_n &= d + a_{n-1} \end{aligned}$$

Fibonacci Sequence:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \text{ for } n \geq 2 \end{aligned}$$

A **dynamical system** is a system that changes over time. The state of the system at any point is determined by a set of well-defined rules that depend on the past states of the system.

8.3 Summations

Summation Notation is used to express the sum of terms in a numerical sequence.

$$\sum_{i=s}^{t} a_i = a_s + a_{s+1} + \ldots + a_t$$

i is the *index*
s is the *lower limit*
t is the *upper limit*

Any variable name could be used for index, but i, j, k and ℓ are most common.

$$\begin{aligned} \sum_{j=1}^{4} j^3 &: \text{ summation form} \\ 1^3 + 2^3 + 3^3 + 4^3 &: \text{ expanded form} \end{aligned}$$

$$\sum_{k=m}^{n} a_k = \sum_{k=m}^{n-1} a_k + a_n, \text{ for } n > m$$

$$\sum_{j=1}^{n} (j+2)^3 = \sum_{k=1}^{n} (k+2)^3 = \sum_{i=3}^{n+2} i^3$$

Closed Form

A **closed form** for a mathematical sum expresses the value of the sum without summation notation. For example,

$$\sum_{k=0}^{n-1}(a+kd) = an + \frac{d(n-1)n}{2}$$

Arithmetic Sequence Closed Form:

$$\sum_{k=0}^{n-1}(a+kd) = an + \frac{d(n-1)n}{2}$$

Geometric Sequence Closed Form:

$$\sum_{k=0}^{n-1}(a\cdot r^k) = \frac{a(r^n-1)}{r-1}$$

8.4 Mathematical induction

Two Components of an inductive proof:

- Base Case:

  establishes that the theorem is true for the first value in the sequence

- Inductive Step:

  establishes that if the theorem is true for k, then the theorem holds for $k+1$

  For a $k \in \mathbb{Z}^+,\ s(k) \implies s(k+1) \iff [s(1) \implies s(2) \implies s(3) \implies s(4) \implies \cdots]$

The supposition that $s(k)$ is true is called the inductive hypothesis.

8.5 More inductive proofs

Explicit formula for a sequence defined by a recurrence relation

$$\{g_n\}:\ g_0 = 1,\ g_n = 3g_{n-1} + 2n \text{ then for any } n \geq 0,$$
$$g_n = \frac{5}{2}\cdot 3^n - n - \frac{3}{2}$$

Proof. by induction on n.
Base Case: $n = 0$

$$g_o = \frac{5}{2}\cdot 3^0 - 0 - \frac{3}{2} = 1$$
$$g_0 = 1 \checkmark \quad \text{(by initial condition)}$$

Inductive Step: assume for any integer $k \geq 0$, assume that $g_k = \frac{5}{2}\cdot 3^k - k - \frac{3}{2}$ is true.
For $k+1$:

$$\begin{aligned} g_{k+1} &= 3g_k + 2(k+1) \quad \text{by definition} \\ &= 3(\frac{5}{2}3^k - k - \frac{3}{2}) + 2(k+1) \quad \text{by induction hypothesis} \\ &= \frac{5}{2}3^{k+1} - 3k - \frac{9}{2} + 2k + 2 \\ &= \frac{5}{2}3^{k+1} - k - \frac{5}{2} \\ &= \frac{5}{2}3^{k+1} - (k+1) - \frac{5}{2} \quad \text{by algebra} \end{aligned}$$

$\therefore g_n = \frac{5}{2}\cdot 3^n - n - \frac{3}{2}$, for all $n \in \mathbb{Z}^{\geq}$ □

8.6 Strong induction and well-ordering

The **principle of strong induction** assumes the fact to be proven holds for all values less than or equal to k and proves the fact holds for $k+1$

$$\begin{array}{lccccccc}
\multicolumn{8}{c}{\text{Inductive Step for Weak Induction}} \\
\hline
 & \multicolumn{7}{c}{\text{For all } k \geq 1,\ S(k) \implies S(k+1)} \\
k=1 & S(1) & \implies & S(2) & & & & \\
k=2 & & & S(2) & \implies & S(3) & & \\
k=3 & & & & & S(3) & \implies & S(4) \\
\vdots & & & & & & & \ddots
\end{array}$$

$$\begin{array}{lccccccc}
\multicolumn{8}{c}{\text{Inductive Step for Strong Induction}} \\
\hline
\multicolumn{8}{l}{\text{For all } k \geq 1,\ S(0) \wedge S(1) \wedge \ldots \wedge S(k) \implies S(k+1)} \\
k=1 & S(1) & \implies & S(2) & & & & \\
k=2 & S(1) & \wedge & S(2) & \implies & S(3) & & \\
k=3 & S(1) & \wedge & S(2) & \wedge & S(3) & \implies & S(4) \\
\vdots & & & & & & & \ddots
\end{array}$$

for $n \geq 0,\ f_n \leq 2^n$

Proof. By Strong Induction of n

Base Case:

$$n = 0: \ f_0 = 0 \leq 2^0 \ \checkmark$$
$$n = 1: \ f_1 = 1 \leq 2^1 \ \checkmark$$

Inductive Step:

For $k \geq 1$, suppose that for any j in the range from 0 through k, $f_k \leq 2^j$. We will prove that $f_{k+1} \leq 2^{k+1}$.

Since $k \geq 1$, then $k-1 \geq 0$. Therefore both k and $k-1$ fall in the range from 0 through k, and by the induction hypothesis $f_{k-1} \leq 2^{k-1}$ and $f_k \leq 2^k$

$$\begin{aligned}
f_{k+1} &= f_k + f_{k-1} && \text{by definition} \\
&\leq 2^k + 2^{k-1} && \text{by inductive hypothesis} \\
&\leq 2^k + 2^{k-1} + 2^{k-1} && \text{since } 2^{k-1} \geq 0 \\
&\leq 2^k + 2 \cdot 2^{k-1} && \\
&\leq 2^k + 2^k && \\
&\leq 2 \cdot 2^k && \\
f_{k+1} &\leq 2^{k+1} && \text{by algebra}
\end{aligned}$$

$\therefore f_{k+1} \leq 2^{k+1}$ □

8.7 Loop invariants

Program Verification

- formally proving that programs perform correctly
- behavior is correct if:

  a pre-condition is true before the program starts

  a post-condition is true before the program ends

Loop Invariants for While Loops

A loop invariant is an assertion that is true before each iteration of a loop.

8.8 Recursive definitions

In a recursive definition of a function, the value of the function is defined in term of the output of the function on smaller input values.

For examples, the factorial:

$$\begin{aligned} n! &= f(n) \quad \text{such that} \\ f(0) &= 1 \\ f(1) &= n \cdot f(n-1) for n \geq 1 \end{aligned}$$

8.9 Structural induction

Structural induction is a type of induction used to prove theorem about recursively defined sets that follow the structure of the recursive definition.

For example, balanced parenthesis

$$\text{right[ ))()]} = 3 \qquad \text{left[ )))]} = 0$$
$$\text{left[((()))]} = 3$$

Recursive definition for the set of properly nested parenthesis:

Basis	$() \in P$
Recursive Rules	If $u, v \in P$, then
1	$(u) \in P$
2	$uv \in P$

Theorem: If string $x \in P$, then left[x] = right [x]

Proof. by structural induction

Base Case: $() \in P$

$$\text{left[()]} = \text{right[()]} = 1$$

Inductive Step: If $x \in P$, then x was constructed by apply a sequence of recursive rules starting with the string () given in the basis.

We consider two case, depending on the last recursive rule that was applies to construct x.

Case 1 Rules 1 was the last applied to construct x. Then $x = (u)$, where $u \in P$. We assume that left[u] = right [u].

$$\begin{aligned} \text{left[} x \text{]} &= \text{left[} (u) \text{]} && \text{because } x = (u) \\ &= 1 + \text{left[} u \text{]} && (u) \text{ has one more (than } u \\ &= 1 + \text{right[} u \text{]} && \text{by the inductive hypothesis} \\ &= \text{right[} (u) \text{]} && (u) \text{ has one more) than } u \\ &= \text{right[} x \text{]} && \text{because } x = (u) \end{aligned}$$

Case 2 Rule 2 was the last rule applied to construct x. Then $x = uv$, where $u, v \in P$. We assume that left[u] = right [u] and left[v] = right [v].

$$\begin{aligned} \text{left[} x \text{]} &= \text{left[} uv \text{]} && \text{because } x = uv \\ &= \text{left[} u \text{]} + \text{left[} v \text{]} \\ &= \text{right[} u \text{]} + \text{right[} v \text{]} && \text{by inductive hypothesis} \\ &= \text{right[} uv \text{]} \\ &= \text{right[} x \text{]} && \text{because } x = uv \end{aligned}$$

$\therefore$ left[x] = right[x] □

8.10 Recursive algorithms

A **recursive algorithm** is an algorithm that calls itself.

Ex. a recursive algorithm to compute the factorial formula:

```
Factorial(n)

Input:  A non-negative integer n
Output: n!

If(n=0), Return(1)
r := Factorial(n-1) //the recursive call

Return(r*n)
```

Ex. recursive algorithm to compute the powerset of a set:

```
PowerSet(A)

Input:  set A
Output: the powerset of A

If(A=[emptyset]), return {[emptyset]}

Select an element a in A
A' := A - {a}
P := PowerSet(A') //recursive call
P' := P
For each S in P'
  Add (S union {a}) to P
End-for

Return(P)
```

8.11 Induction and recursive algorithms

Nothing very useful in this section, only kept for counting reasons.

8.12 Analyzing the time complexity of recursive algorithms

The **time complexity** of an algorithm is a function $T(n)$, whose n is the input size

Ex. Determining recurrence relation for factorial(n)

```
Factorial(n)

If(n=0), Return(1)      // two operations, comparison and returning

r := Factorial(n-1)     // n-1 recursive calls, 1 operation for assignment

Return(r*n)             // 2 ops, multiplying and returning
```

Therefore, $T(0) = 2$, $T(n) = T(n-1) + 5$

8.13 Divide-and-conquer algorithms: Introduction and mergesort

A **divide-and-conquer** algorithm solves a problem by recursively breaking the original input into smaller sub-problems of roughly equal size

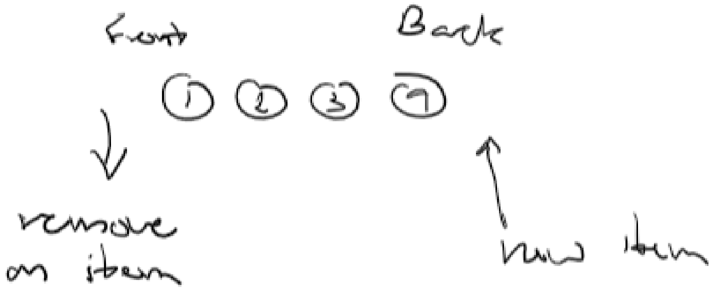
Ex.

```
Findmin(L)

If L has only one item x, return(x)

Break list L into two lists, A and B

a := Findmin(A)
b := Findmin(B)

If(a <= b) Return(a)
Else Return(B)
```

Queue

A **queue** maintains items in an ordered list

- new items are added to back of queue
- items are removed according to first-in, first-out



8.14 Divide-and-conquer algorithms: Binary Search

8.15 Solving linear homogeneous recurrence relations

8.16 Solving linear non-homogeneous recurrence relations

8.17 Divide-and-conquer recurrence relations

9 Integer Properties

9.1 The Division Algorithm

9.2 Modular arithmetic

9.3 Prime factorizations

9.4 Factoring and primality testing

9.5 Greatest common factor divisor and Euclid's algorithm

9.6 Number representation

9.7 Fast exponentiation

9.8 Introduction to cryptography

9.9 The RSA cryptosystem

10 Introduction to Counting

10.1 Sum and Product Rules

10.2 The Bijection Rules

10.3 The generalized product rule

10.4 Counting permutations

10.5 Counting subsets

10.6 Subset and permutation examples

10.7 Counting by complement

10.8 Permutations with repetitions

10.9 Counting multisets

10.10 Assignment problems: Balls in bins

10.11 Inclusion-exclusion principle

11 Advanced Counting

11.1 Generating permutations

11.2 Binomial coefficients and combinatorial identities

11.3 The pigeonhole principle

11.4 Generating functions

12 Discrete Probability

12.1 Probability of an event

12.2 Unions and complements of events

12.3 Conditional probability and independence

12.4 Bayes' Theorem

12.5 Random variables

12.6 Expectation of random variables

12.7 Linearity of expectations

12.8 Bernoulli trials and the binomial distribution

13 Graphs

13.1 Introduction to Graphs

13.2 Graph representations

13.3 Graph isomorphism

13.4 Walks, trails, circuits, paths, and cycles

13.5 Graph connectivity

13.6 Euler circuits and trails

13.7 Hamiltonian cycles and paths

13.8 Planar coloring

13.9 Graph coloring

14 Trees

14.1 Introduction to trees

14.2 Tree application examples

14.3 Properties of trees

14.4 Tree traversals

14.5 Spanning trees and graph traversals

14.6 Minimum spanning trees