Contents

1 Review of Propositional Logic 4
1.1 Connectives 4
1.1.1 Truth Table of the Connectives 4
1.2 Important Tautologies 5
1.3 Indirect Arguments/Proofs by Contradiction/Reductio ad absurdum 5

2 Predicate logic and Quantifiers 6
2.1 Introduce quantifiers 6
2.1.1 $\exists$ existential quantifier 6
2.1.2 $\forall$ universal quantifier 6
2.1.3 $\exists!$ for one and only one (additional quantifier standard in maths) 6
2.2 Alternation of Quantifiers 7
2.3 Negation of Quantifiers 7

3 Set Theory 7
3.1 Two Ways to Describe Sets 8
3.2 Set Operations 8
3.2.1 Venn Diagrams 9
3.2.2 Properties of Set Operations 11
3.3 Example Proof in Set Theory 12
3.4 The Power Set 13
3.5 Cartesian Products 13
3.5.1 Cardinality (number of elements) in a Cartesian product . 15

4 Relations 15
4.1 Equivalence Relations 16
4.2 Equivalence Relations and Partitions 17
4.3 Partial Orders 20

5 Functions 21
5.1 Composition of Functions 22
5.2 Inverting Functions 22
5.3 Functions Defined on Finite Sets 24
5.4 Behaviour of Functions on Infinite Sets 25
5.4.1 Hilbert's Hotel problem (jazzier name: Hilbert's paradox of the Grand Hotel) 25

6 Mathematical Induction 26
6.1 Mathematical Induction Consists of Two Steps: 26

7 Abstract Algebra **28**
7.1 Binary Operations 28
7.2 Semigroups 29
7.2.1 General Associative Law 30
7.3 Identity Elements 30
7.4 Monoids 31
7.5 Inverses 32
7.6 Groups 34
7.7 Homomorphisms and Isomorphisms 36

8 Formal Languages **38**
8.1 Phrase Structure Grammars 42
8.2 Regular Languages 42
8.3 Finite State Acceptors and Automata Theory 44
8.4 Regular Grammars 47
8.5 Regular expressions 49
8.6 The Pumping Lemma 52
8.7 Applications of Formal Languages and Grammars as well as Automata Theory 54

9 Graph Theory **54**
9.1 Complete graphs 58
9.2 Bipartite graphs 58
9.3 Isomorphisms of Graphs 59
9.4 Subgraphs 59
9.5 Vertex Degrees 60
9.6 Walks, trails and paths 62
9.7 Connected Graphs 63
9.8 Components of a graph 64
9.9 Circuits 66
9.10 Bridge lecture between Michaelmas and Hilary terms 67
9.11 Eulerian trails and circuits 69
9.12 Hamiltonian Paths and Circuits 74
9.13 Forests and Trees 74
9.14 Spanning Trees 76
9.15 Constructing spanning trees 78
9.16 Kruskal's Algorithm 82
9.17 Prim's Algorithm 88
9.18 Directed Graphs 91
9.19 Directed Graphs and Binary Relations 93

10 Countability of Sets **93**
10.1 Applications of Countability of Sets to Formal Languages 103

11 Turing Machines **105**
11.1 Variants of Turing machines 115
11.2 Algorithms 118
11.3 Decidable Languages 119
11.4 Undecidability 125
11.5 Example of a language that is not Turing-recognizable 128

1 Review of Propositional Logic

Task: Recall enough propositional logic to see how it matches up with set theory.

Definition: A proposition is any declarative sentence that is either true or false.

1.1 Connectives

	Connectives		Notation in Maths
and	$\wedge$		
or	$\vee$	"Inclusive or"	
not	$\neg$	Sometimes denoted $\sim$	
implies	$\rightarrow$	if/then; called implication	$\Rightarrow$
if and only if	$\leftrightarrow$	Called equivalence	$\Leftrightarrow$

1.1.1 Truth Table of the Connectives

Let P, Q be propositions:

P	Q	P$\wedge$Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P$\vee$Q
F	F	F
F	T	T
T	F	T
T	T	T

P	$\neg$P
F	T
T	F

NB In some textbooks, T is denoted by 1, and F is denoted by 0.

P	Q	P$\rightarrow$Q
F	F	T
F	T	T
T	F	F
T	T	T

NB Note that the only instance when an implication (if/then statement) denoted by $P \rightarrow Q$ is false is when the hypothesis (P) is true, but the conclusion (Q) is false.

P	Q	P$\leftrightarrow$Q
F	F	T
F	T	F
T	F	F
T	T	T

NB The truth table for the equivalence says that both P and Q must have the same truth value, i.e. both be true or both be false for the equivalence to be true.

Priority of the Connectives

Highest to Lowest: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

1.2 Important Tautologies

$$\begin{array}{rcl}
(P \rightarrow Q) & \leftrightarrow & (\neg P \vee Q) \\
(P \leftrightarrow Q) & \leftrightarrow & [(P \rightarrow Q) \wedge (Q \rightarrow P)] \\
\neg(P \wedge Q) & \leftrightarrow & (\neg P \vee \neg Q) \\
\neg(P \vee Q) & \leftrightarrow & (\neg P \wedge \neg Q)
\end{array}
\left.\vphantom{\begin{array}{c}a\\b\end{array}}\right\} \text{De Morgan Laws}$$

(these have parallels in in set theory)

As a result, $\neg$ and $\vee$ together can be used to represent all of $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

Less obvious: One connective called the Sheffer stroke P|Q (which stands for "not both P and Q" or "P nand Q") can be used to represent all of $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ since $\neg$P $\leftrightarrow$ P|P and P $\vee$ Q $\leftrightarrow$ (P|P) | (Q|Q).

Recall that if P$\rightarrow$Q is a given implication, then Q$\rightarrow$P is called the <u>converse</u> of P$\rightarrow$Q, while $\neg$Q $\rightarrow$ $\neg$P is called the <u>contrapositive</u> of P $\rightarrow$ Q.

1.3 Indirect Arguments/Proofs by Contradiction/Reductio ad absurdum

Based on the tautology (P$\rightarrow$Q) $\leftrightarrow$ ($\neg$Q $\rightarrow$ $\neg$P)

Example: Famous argument that $\sqrt{2}$ is irrational.

Proof:

Suppose $\sqrt{2}$ is rational, then it can be expressed in fraction form as $\frac{a}{b}$ with a and b integers, $b \neq 0$. Let us **assume** that our fraction is reduced, **i.e.** the only common divisor of the numerator a and denominator b is 1.

Then,

$$\sqrt{2} = \frac{a}{b}$$

Squaring both sides, we have

$$2 = \frac{a^2}{b^2}$$

Multiplying both sides by b^2 yields

$$2b^2 = a^2$$

Therefore, 2 divides a^2, i.e. a^2 is even. If a^2 is even, then a is also even, namely $a = 2k$ for some integer k.

Substituting the value of $2k$ for a, we have $2b^2 = (2k)^2$ which means that $2b^2 = 4k^2$. Dividing both sides by 2, we have $b^2 = 2k^2$. That means 2 divides b^2, so b is even.

This implies that both a and b are even, which means that both the numerator and the denominator of our fraction are divisible by 2. This contradicts our **assumption** that the numerator a and the denominator b have no common divisor except 1. Since we found a contradiction, our assumption that $\sqrt{2}$ is rational must be false. Hence the theorem is true.

qed

2 Predicate logic and Quantifiers

Task: Understand enough predicate logic to make sense of quantified statements.

In predicate logic, propositions depend on variables x, y, z, so their truth value may change depending on which values these variables assume: $P(x), Q(x, y), R(x, y, z)$

2.1 Introduce quantifiers

2.1.1 $\exists$ existential quantifier

Syntax: $\exists x P(x)$

Definition: $\exists x P(x)$ is true if $P(x)$ is true for some value of x. It is false otherwise.

2.1.2 $\forall$ universal quantifier

Syntax: $\forall x P(x)$

Definition: $\forall x P(x)$ is true if $P(x)$ is true for all allowable values of x. It is false otherwise.

2.1.3 $\exists!$ for one and only one (additional quantifier standard in maths)

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

Definition: $\exists! x P(x)$ is true if $P(x)$ is true for exactly one value of x and false for all other values of x; otherwise, $\exists! x P(x)$ is false.

Example: $P(x) : x$ is/was the pope and x is Argentine.
(Compound statement; two sentences with connector $\wedge$ between them)
$\exists! x P(x)$ is true with x being Pope Francis.
Now, set $Q(x) : x$ is/was the pope and x is Brazilian.
$\exists! x Q(x)$ is false as there has not been a Brazilian pope so far.
In fact, $\exists x Q(x)$ is also false.

2.2 Alternation of Quantifiers

$\forall x \exists y \forall z \quad P(x, y, z)$

NB: The order <u>cannot</u> be exchanged as it might modify the truth value of the statement (think of examples with two quantifiers).

2.3 Negation of Quantifiers

$$\neg(\exists x P(x)) \quad \leftrightarrow \quad \forall x \neg P(x)$$
$$\neg(\forall x P(x)) \quad \leftrightarrow \quad \exists x \neg P(x)$$

3 Set Theory

Task: Understand enough set theory to make sense of other mathematical objects in abstract algebra, graph theory, etc.

Set theory started around 1870's $\rightarrow$ late development in mathematics but now taught early in one's maths education due to the Bourbaki school.

Definition: A set is a collection of objects. $x \in A$ means the element x is in the set A (**i.e.** belongs to A).

Examples:

1. All students in a class.
2. $\mathbb{N}$ the set of natural numbers starting at 0.

   $\mathbb{N}$ is defined via the following two axioms:
   (a) $0 \in \mathbb{N}$
   (b) if $x \in \mathbb{N}$, then $x + 1 \in \mathbb{N}$ ($x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$)
3. $\mathbb{R}$ set of real numbers also introduced axiomatically. The hardest axiom is the last one: completeness. $\mathbb{R}$ is constructed from $\mathbb{Q}$ in one of two ways: via Dedekind cuts or Cauchy sequences.

   $\mathbb{R}$ is the set of real numbers. The axioms governing $\mathbb{R}$ are:
   (a) Additive closure: $\forall x, y \, \exists z (x + y = z)$
   (b) Multiplicative closure: $\forall x, y, \exists z (x \times y = z)$
   (c) Additive associativity: $\forall x, y, z \quad x + (y + z) = (x + y) + z$
   (d) Multiplicative associativity: $\forall x, y, z \quad x \times (y \times z) = (x \times y) \times z$
   (e) Additive commutativity: $\forall x, y \quad x + y = y + x$
   (f) Multiplicative commutativity: $\forall x, y \quad x \times y = y \times x$
   (g) Distributivity: $\forall x, y, z \quad x \times (y + z) = (x \times y) + (x \times z)$ and $(y + z) \times x = (y \times x) + (z \times x)$
   (h) Additive identity: There is a number, denoted 0, such that for all $x, x + 0 = x$
   (i) Multiplicative identity: There is a number, denoted 1, such that for all $x, x \times 1 = 1 \times x = x$

(j) Additive inverses: For every x there is a number, denoted $-x$, such that $x + (-x) = 0$

(k) Multiplicative inverses: For every nonzero x there is a number, denoted x^{-1}, such that $x \times x^{-1} = x^{-1} \times x = 1$

(l) $0 \neq 1$

(m) Irreflexivity of $<$:$\sim (x < x)$

(n) Transitivity of $<$: If $x < y$ and $y < z$, then $x < z$

(o) Trichotomy: Either $x < y, y < x$, or $x = y$

(p) If $x < y$, then $x + z < y + z$

(q) If $x < y$ and $0 < z$, then $x \times z < y \times z$ and $z \times x < z \times y$

(r) Completeness: If a nonempty set of real numbers has an upper bound, then it has a *least* upper bound.

4. $\emptyset$ is the empty set (The set with no elements).

Definition: Let A, B be sets. A=B if and only if all elements of A are elements of B and all elements of B are elements of A,
i.e. $A = B \leftrightarrow [\forall x(x \in A \to x \in B)] \wedge [\forall y(y \in B \to y \in A)]$

3.1 Two Ways to Describe Sets

1. The enumeration/roster method: list all elements of the set.
NB: order is <u>irrelevant</u>.
$A = \{0, 1, 2, 3, 4, 5\} = \{5, 0, 2, 3, 1, 4\}$

2. The formulaic/set builder method: give a formula that generates all elements of the set.
$A = \{x \in \mathbb{N} \mid 0 \leq x \wedge x \leq 5\} = \{0, 1, 2, 3, 4, 5\} = \{x \in \mathbb{N} : 0 \leq x \wedge x \leq 5\}$

Using $\mathbb{N}$ and the set-builder method, we can define:
$\mathbb{Z} = \{m - n \mid \forall m, n \in \mathbb{N}\}$
$n = 0$ and m any natural number $\Rightarrow$ we generate all of $\mathbb{N}$
$m = 0$ and n any natural number $\Rightarrow$ we generate all negative integers
$0 - 1 = -1$
$0 - 2 = -2$
etc.
$\mathbb{Q} = \{\frac{p}{q} \mid p, q \in \mathbb{Z} \wedge q \neq 0\}$

Definition: A set A is called finite if it has a finite number of elements; otherwise, it is called infinite.

3.2 Set Operations

Task: Understand how to represent sets by Venn diagrams. Understand set union, intersection, complement, and difference.

Definition: Let A, B be sets. A is a subset of B if all elements of A are elements of B, **i.e.** $\forall x(x \in A \to x \in B)$. We denote that A is a subset of B by $A \subseteq B$

Example: $\mathbb{N} \subseteq \mathbb{Z}$

Definition: Let A, B be sets. A is a proper subset of B if $A \subseteq B \wedge A \neq B$, **i.e.** $A \subseteq B \wedge \exists x \in B \ s.t. \ x \notin A$.

Notation: $A \subset B$

Example: $\mathbb{N} \subset \mathbb{Z}$ since $\exists(-1) \in \mathbb{Z}$ such that $-1 \notin \mathbb{N}$.

NB: $\forall A$ a set, $\emptyset \subseteq A$

Recall: $B \subseteq C$ means $\forall x(x \in B \to x \in C)$, but $\emptyset$ has no elements, so in $\emptyset \subseteq A$ the quantifier $\forall$ operates on a domain with no elements. Clearly, we need to give meaning to $\exists$ and $\forall$ on empty sets.

$$\begin{array}{l} \underline{\text{Boolean Convention}} \\ \left.\begin{array}{l} \forall \text{ is true on the empty set} \\ \exists \text{ is false on the empty set} \end{array}\right\} \text{Consistent with common sense} \end{array}$$

Definition: Let A, B be two sets. The union $A \cup B = \{x \mid x \in A \vee x \in B\}$

Definition: Let A, B be two sets. The intersection $A \cap B = \{x \mid x \in A \wedge x \in B\}$

Definition: Let A, B be sets. A and B are called disjoint if $A \cap B = \emptyset$

Definition Let A, B be two sets. $A - B = A \backslash B = \{x \mid x \in A \wedge x \notin B\}$

Examples:

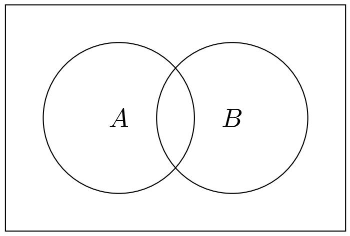
$$\begin{array}{ll} A = \{1,2,5\} & B = \{1,3,6\} \\ A \cup B = \{1,2,3,5,6\} & A \cap B = \{1\} \\ A \backslash B = \{2,5\} & B \backslash A = \{3,6\} \end{array}$$

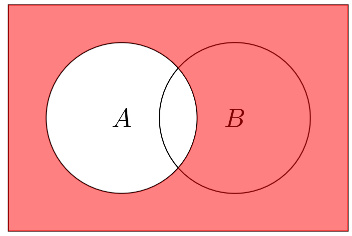
Definition: Let A, U be sets s.t. $A \subseteq U$. The complement of A in $U = U \backslash A = A^C = \{x \mid x \in U \wedge x \notin A\}$

Remark: The notation A^C is unambiguous only if the universe U is clearly defined or understood.

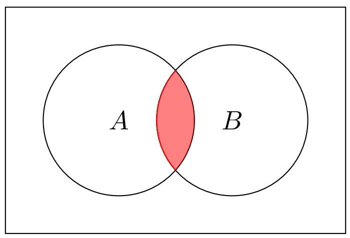
3.2.1 Venn Diagrams

Schematic representation of set operations.

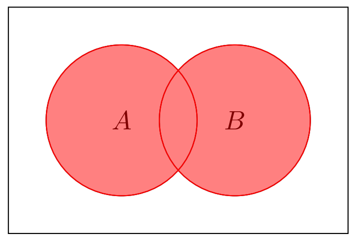




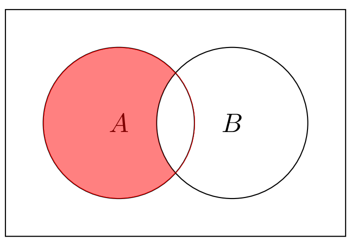
A^C



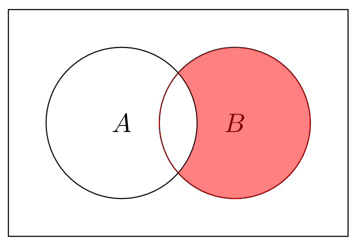
$A \cap B$



$A \cup B$



$A \backslash B$



$B \backslash A$

Pros of Venn diagrams:

Very easy to visualize

Cons of Venn diagrams:

1. Misleading if for example $A \subset B$ or sets are in some other non standard configuration;
2. Not helpful if a lot of sets are involved;
3. Not helpful if sets are infinite or have some peculiar structure.

Moral of the story: Venn diagrams will **NOT** be accepted as proof of any statement in set theory. Instead, we will introduce rigorous ways of proving assertions in set theory.

3.2.2 Properties of Set Operations

Correspondence between Logic and Set Theory

Logical Connective	Set operation/property
$\wedge$	intersection $\cap$
$\vee$	union $\cup$
$\neg$	complement $(\ )^C$
$\rightarrow$	subset $\subseteq$
$\leftrightarrow$	equality of sets $=$

Recall:

Definition: Let A, B be two sets. The intersection $A \cap B = \{x \mid x \in A \wedge x \in B\}$

Definition: Let A, B be two sets. The union $A \cup B = \{x \mid x \in A \ \vee \ x \in B\}$

Definition: Let A, U be sets s.t. $A \subseteq U$. The complement of A in $U = U \backslash A = A^C = \{x \mid x \in U \wedge x \notin A\}$

Definition: Let A, B be sets. A is a subset of B if all elements of A are elements of B, **i.e.** $\forall x(x \in A \rightarrow x \in B)$.

Definition: Let A, B be sets. A=B if and only if all elements of A are elements of B and all elements of B are elements of A,
i.e. $A = B \leftrightarrow [\forall x(x \in A \rightarrow x \in B)] \wedge [\forall y(y \in B \rightarrow y \in A)]$

As a result, various properties of set operations become obvious:

- Commutativity
  - $A \cap B = B \cap A$ comes from the tautology $(P \wedge Q) \leftrightarrow (Q \wedge P)$ (#31 on the list of tautologies posted in Course Documents)
  - $A \cup B = B \cup A$ comes from the tautology $(P \vee Q) \leftrightarrow (Q \vee P)$ (# 32 on the list of tautologies)
- Associativity
  - $(A \cup B) \cup C = A \cup (B \cup C)$ comes from the tautology $[(P \vee (Q \vee R)] \leftrightarrow [(P \vee Q) \vee R]$ (# 33 on the list of tautologies)
  - $(A \cap B) \cap C = A \cap (B \cap C)$ comes from the tautology $[(P \wedge (Q \wedge R)] \leftrightarrow [(P \wedge Q) \wedge R]$ (# 34 on the list of tautologies)
- Distributivity
  - $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ comes from the tautology $[(P \wedge (Q \vee R)] \leftrightarrow [(P \wedge Q) \vee (P \wedge R)]$ (# 29 on the list of tautologies)

- $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ comes from the tautology $[(P \vee (Q \wedge R)] \leftrightarrow [(P \vee Q) \wedge (P \vee R)]$ (# 30 on the list of tautologies)

- De Morgan Laws in Set Theory
  - $(A \cap B)^C = A^C \cup B^C$ comes from the tautology $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$ (# 18 on the list of tautologies)
  - $(A \cup B)^C = A^C \cap B^C$ comes from the tautology $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$ (# 19 on the list of tautologies)
- Involutivity of the Complement
  - $(A^C)^C = A$ comes from the tautology $\neg(\neg P) \leftrightarrow P$ (# 3 on the list of tautologies)

NB: An involution is a map such that applying it twice gives the identity. Familiar examples: reflecting across the x-axis, the y-axis, or the origin in the plane.

- Transitivity of Inclusion
  - $A \subseteq B \wedge B \subseteq C \rightarrow A \subseteq C$ comes from the tautology
  $$[(P \rightarrow Q) \wedge (Q \rightarrow R)] \rightarrow (P \rightarrow R)$$
  (# 14 on the list of tautologies)
- Criterion for proving equality of sets, which comes from the tautology $(P \leftrightarrow Q) \leftrightarrow [(P \rightarrow Q) \wedge (Q \rightarrow P)]$ (#22 on the list of tautologies)
  - $A = B \leftrightarrow A \subseteq B \wedge B \subseteq A$
- Criterion for proving non-equality of sets
  - $A \neq B \leftrightarrow (A \backslash B) \cup (B \backslash A) \neq \emptyset$

3.3 Example Proof in Set Theory

Proposition: $\forall A, B$ sets. $(A \cap B) \cup (A \backslash B) = A$

Proof: Use the criterion for proving equality of sets from above, **i.e.** inclusion in both directions.

Show $(A \cap B) \cup (A \backslash B) \subseteq A$: $\forall x \in (A \cap B) \cup (A \backslash B), x \in (A \cap B)$ or $x \in A \backslash B$. If $x \in (A \cap B)$, then clearly $x \in A$ as $A \cap B \subseteq A$ by definition. If $x \in A \backslash B$, then by definition $x \in A$ and $x \notin B$, so definitely $x \in A$. In both cases, $x \in A$ as needed.

Show $A \subseteq (A \cap B) \cup (A \backslash B)$: $\forall x \in A$, we have two possibilities, namely $x \in B$ or $x \notin B$. If $x \in B$, then $x \in A$ and $x \in B$, so $x \in A \cap B$. If $x \notin B$, then $x \in A$ and $x \notin B$, so $x \in A \backslash B$. In both cases, $x \in (A \cap B)$ or $x \in (A \backslash B)$, so $x \in (A \cap B) \cup (A \backslash B)$ as needed.

qed

3.4 The Power Set

Task: Understand what the power set of a set A is.

Definition: Let A be a set. The power set of A denoted $P(A)$ is the collection of all subsets of A.

Recall: $\emptyset \subseteq A$. It is also clear from the definition of a subset that $A \subseteq A$.

Examples:

1. $A = \{0, 1\}$
   $P(A) = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$
2. $A = \{a, b, c\}$
   $P(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$
3. $A = \emptyset$
   $P(A) = \{\emptyset\}$
   $P(P(A)) = \{\emptyset, \{\emptyset\}\}$

NB: $\emptyset$ and $\{\emptyset\}$ are different objects. $\emptyset$ has no elements, whereas $\{\emptyset\}$ has one element.

Remark: $P(A)$ and A are viewed as living in separate worlds to avoid phenomena like Russell's paradox.

Q: If A has n elements, how many elements does $P(A)$ have?

A: 2^n

Theorem: Let A be a set with n elements, then $P(A)$ contains 2^n elements.

Proof: Based on the on/off switch idea.

$\forall x \in A$, we have two choices: either we include x in the subset or we don't (on vs off switch). A has n elements $\Rightarrow$ we have 2^n subsets of A.

qed

Alternate Proof: Using mathematical induction.

NB: It is an axiom of set theory (in the ZFC standard system) that every set has a power set, which implies no set consisting of all possible sets could exist, else what would its power set be?

3.5 Cartesian Products

Task: Understand sets like $\mathbb{R}^1$ in a more theoretical way.

Remark: Cartesian products allow us to pair up sets that have no relationship to each other, which is essential in computer science as it allows us to define objects like finite state acceptors, Turing machines, etc.

Recall from Calculus:

$\mathbb{R} = \mathbb{R}^1 \ni x$

$\mathbb{R} \times \mathbb{R} = \mathbb{R}^2 \ni (x_1, x_2)$

$\vdots$

$\underbrace{\mathbb{R} \times \cdots \times \mathbb{R}}_{\text{n times}} = \mathbb{R}^n \ni (x_1, x_2, ..., x_n)$

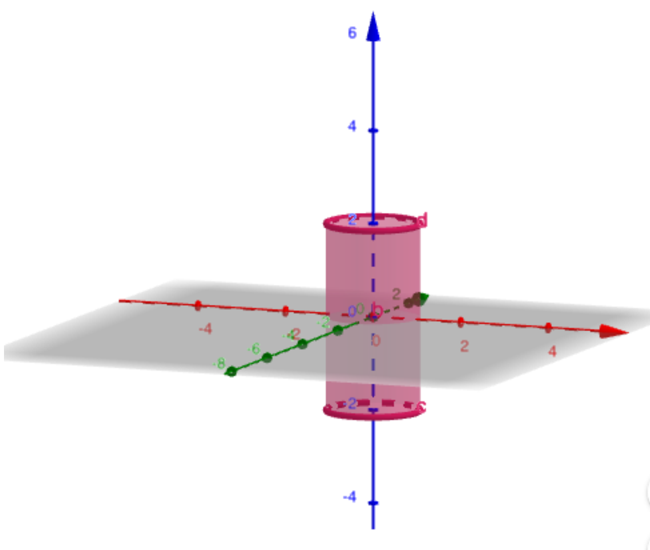
These are examples of Cartesian products.

Definition: Let A, B be sets. The Cartesian product denoted by $A \times B$ consists of all ordered pairs (x, y) $s.t.$ $x \in A \wedge y \in B$, **i.e.**

$$A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$$

Further Examples:

1. $A = \{1, 3, 7\}$
   $B = \{1, 5\}$
   $A \times B = \{(1, 1), (1, 5), (3, 1), (3, 5), (7, 1), (7, 5)\}$
   NB: The order in which elements in a pair matters: $(7, 1)$ is different from $(1, 7)$. This is why we call (x, y) an <u>ordered</u> pair.
2. $A = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\} \leftarrow$ circle of radius 1
   $B = \{z \in \mathbb{R} \mid -2 \leq z \leq 2\} = [-2, 2] \leftarrow$ closed interval
   $A \times B \leftarrow$ cylinder of radius 1 and height 4



3.5.1 Cardinality (number of elements) in a Cartesian product

If A has m elements and B has p elements, $A \times B$ has mp elements.

Examples:

1. $\#(A) = 3 \qquad A = \{1, 3, 7\}$
   $\#(B) = 2 \qquad B = \{1, 5\}$
   $\#(A \times B) = 3 \times 2 = 6$
2. Both A and B are infinite sets, so $A \times B$ is infinite as well.

Remark: We can define Cartesian products of any length, **e.g.** $A \times A \times B \times A$, $B \times A \times B \times A \times B$, etc. If all sets are finite, the number of elements is the product of the numbers of elements of each factor. If $\#(A) = 3$ and $\#(B) = 2$ as above, $\#(A \times B \times A) = 3 \times 2 \times 3 = 18$ and $\#(B \times A \times B) = 2 \times 3 \times 2 = 12$.

4 Relations

Task: Define subsets of Cartesian products with certain properties. Understand the predicates " $=$ " (equality) and other predicates in predicate logic in a more abstract light.

Start with $x = y$. The element x is some relation R to y (equality in this case). We can also denote it as xRy or $(x, y) \in E$

Let x, y in $\mathbb{R}$, then $E = \{(x, x) \mid x \in \mathbb{R}\} \subset \mathbb{R} \times \mathbb{R}$.

The "diagonal" in $\mathbb{R} \times \mathbb{R}$ gives exactly the elements equal to each other.

More generally:

Definition: Let A, B be sets. A subset of the Cartesian product $A \times B$ is called a relation between A and B. A subset of the Cartesian product $A \times A$ is called a relation on A.

Remark: Note how general this definition is. To make it useful for understanding predicates, we will need to introduce key properties relations can satisfy.

Example: $A = \{1, 3, 7\} \qquad B = \{1, 2, 5\}$

We can define a relation S on $A \times B$ by $S = \{(1, 1), (1, 5), (3, 2)\}$. This means $1S1$, $1S5$ and $3S2$ and no other ordered pairs in $A \times B$ satisfy S.

Remark: The relations we defined involve 2 elements, so they are often called <u>binary relations</u> in the literature.

4.1 Equivalence Relations

Task: Define the most useful kind of relation.

Definition: A relation R on a set A is called

1. reflexive iff (if and only if) $\forall x \in A, xRx$
2. symmetric iff $\forall x, y \in A, xRy \to yRx$
3. transitive iff $\forall x, y, z \in A, xRy \wedge yRz \to xRz$

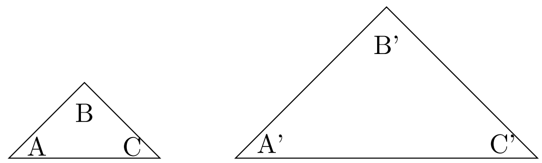
An equivalence relation on A is a relation that is reflexive, symmetric, and transitive.

Notation: Instead of xRy, an equivalence relation is often denoted by $x \equiv y$ or $x \sim y$.

Examples:

1. "=" equality is an equivalence relation.
   (a) $x = x$ reflexive
   (b) $x = y \Rightarrow y = x$ symmetric
   (c) $x = y \wedge y = z \Rightarrow x = z$ transitive
2. $A = \mathbb{N}$
   $x \equiv y \bmod 3$ is an equivalence relation. $x \equiv y \bmod 3$ means $x - y = 3m$ for some $m \in \mathbb{Z}$, **i.e.** x and y have the same remainder when divided by 3. The set of all possible remainders is $\{0, 1, 2\}$
   NB: In correct logic notation, $x \equiv y \bmod 3$ if $\exists m \in \mathbb{Z}\ s.t.\ x - y = 3m$
   (a) $x \equiv x \bmod 3$ since $x - x = 0 = 3 \times 0 \to$ reflexive
   (b) $x \equiv y \bmod 3 \Rightarrow y \equiv x \bmod 3$ because $x \equiv y \bmod 3$ means $x - y = 3m$ for some $m \in \mathbb{Z} \Rightarrow y - x = -3m = 3 \times (-m) \Rightarrow y \equiv x \bmod 3 \to$ symmetric
   (c) Assume $x \equiv y \bmod 3$ and $y \equiv z \bmod 3$
   $x \equiv y \bmod 3 \Rightarrow \exists m \in \mathbb{Z}$ s.t. $x - y = 3m \Rightarrow y = x - 3m$
   $y \equiv z \bmod 3 \Rightarrow \exists p \in \mathbb{Z}$ s.t. $y - z = 3p \Rightarrow y = z + 3p$
   Therefore, $x - 3m = z + 3p \Leftrightarrow x - z = 3p + 3m = 3(p + m)$
   Since $p, m \in \mathbb{Z}, p + m \in \mathbb{Z} \Rightarrow x \equiv z \bmod 3 \to$ transitive.
3. Let $f : A \to A$ be any function on a non-empty set A. We define the relation $R = \{(x, y) \mid f(x) = f(y)\}$
   (a) $\forall x \in A, f(x) = f(x) \Rightarrow (x, x) \in R \to$ reflexive
   (b) If $(x, y) \in R$, then $f(x) = f(y) \Rightarrow f(y) = f(x)$, **i.e.** $(y, x) \in R \to$ symmetric
   (c) If $(x, y) \in R$ and $(y, z) \in R$, then $f(x) = f(y)$ and $f(y) = f(z)$, which by the transitivity of equality implies $f(x) = f(z)$, **i.e.** $(x, z) \in R$ as needed, so R is transitive as well.
   $f(x)$ can be $e^x, \sin x, |x|$, etc.

4. Let Γ be the set of all triangles in the plane. $ABC \sim A'B'C'$ if ABC and $A'B'C'$ are similar triangles, **i.e.** have equal angles.



   (a) $\forall ABC \in \Gamma, ABC \sim ABC$ so $\sim$ is reflexive
   (b) $ABC \sim A'B'C' \Rightarrow A'B'C' \sim ABC$ so $\sim$ is symmetric
   (c) $ABC \sim A'B'C'$ and $A'B'C' \sim A"B"C" \Rightarrow ABC \sim A"B"C"$, so $\sim$ is transitive

Clearly (a), (b), (c) use the fact that equality of angles is an equivalence relation.

Exercise: For various predicates you've encountered, check whether reflexive, symmetric or transitive. Examples of predicates include $\neq$, $<$, $>$, $\leq$, $\geq$, $\subseteq$, $\rightarrow$, $\leftrightarrow$

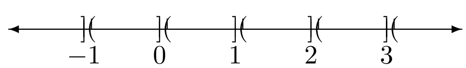
4.2 Equivalence Relations and Partitions

Task: Understand how equivalence relations divide sets.

Definition: Let A be a set. A partition of A is a collection of non-empty sets, any two of which are disjoint such that their union is A, **i.e.** $\lambda = \{A_\alpha \mid \alpha \in I\}$ s.t. $\forall \alpha, \alpha' \in I$ satisfying $\alpha \neq \alpha', A_\alpha \cap A_{\alpha'} = \emptyset$ and $\bigcup_{\alpha \in I} A_\alpha = A$

Here I is an indexing act (may be infinite). $\bigcup_{\alpha \in I} A_\alpha$ is the union of all the A_α's (possibly an infinite union)

Example $\{(n, n+1] \mid n \in \mathbb{Z}\}$ is a partition of $\mathbb{R}$



$\bigcup_{n \in \mathbb{Z}} (n, n+1] = \mathbb{R}$

$(n, n+1] \cap (m, m+1] = \emptyset$ if $n \neq m$

Definition: If R is an equivalence relation on a set A and $x \in A$, the equivalence class of x denoted $[x]_R$ is the set $\{y \mid xRy\}$. The collection of all equivalence classes is called A modulo R and denoted A/R.

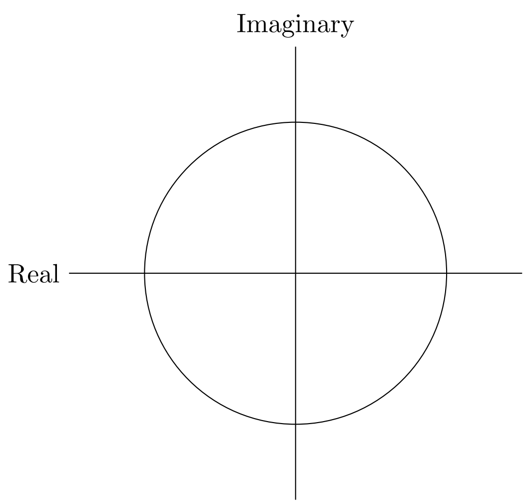
Examples:

1. $A = \mathbb{N}$ $\quad\quad x \equiv y \bmod 3$
   We have the equivalence classes $[0]_R, [1]_R$ and $[2]_R$ given by the three possible remainders under division by 3.
   $[0]_R = \{0, 3, 6, 9, ...\}$
   $[1]_R = \{1, 4, 7, 10, ...\}$
   $[2]_R = \{2, 5, 8, 11, ...\}$
   Clearly $[0]_R \cup [1]_R \cup [2]_R = \mathbb{N}$ and they are mutually disjoint $\Rightarrow R$ gives a partition of $\mathbb{N}$.

2. $ABC \sim A'B'C'$
   $[ABC] = \{$The set of all triangles with angles of magnitude $\measuredangle ABC, \measuredangle BAC, \measuredangle ACB\}$
   The union over the set of all $[ABC]$ is the set of all triangles and $[ABC] \cap [A'B'C'] = \emptyset$ if $ABC \nsim A'B'C'$ since it means these triangles have at least one angle that is different.

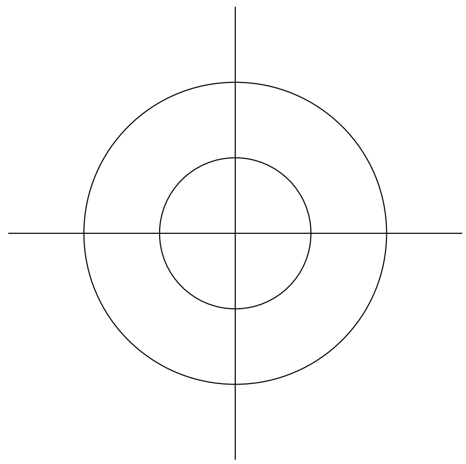
3. $A = \mathbb{C}$ $\quad\quad x \sim y$ if $|x| = |y|$ $\quad\quad$ equivalence relation
   $[x] = \{y \in \mathbb{C} \mid |x| = |y|\} = [r]$ for $r \in [0, +\infty)$ (meaning $r \geq 0$)

   circle of radius $|x|$

   $\bigcup\limits_{r \in [0,+\infty)} [r] = \mathbb{C}$
   $[r_1] \cap [r_2] \neq \emptyset$ if $r_1 \neq r_2$ since two distinct circles in $\mathbb{C} \simeq \mathbb{R}^2$ with empty intersection.

circles r_1 and r_2



Theorem: For any equivalence relation R on a set A, its equivalence classes form a partition of A, **i.e.**

1. $\forall x \in A, \exists y \in A$ s.t. $x \in [y]$ (every element of A sits somewhere)
2. $xRy \Leftrightarrow [x] = [y]$ (all elements related by R belong to the same equivalence class)
3. $\neg(xRy) \Leftrightarrow [x] \cap [y] = \emptyset$ (if two elements are not related by R, the they belong to disjoint equivalence classes)

Proof:

1. Trivial. Let $y = x$. $x \in [x]$ because R is an equivalence relation, hence reflexive, so xRx holds.
2. We will prove $xRy \Leftrightarrow [x] \subseteq [y]$ and $[y] \subseteq [x]$
   "$\Rightarrow$" Fix $x \in A, [x] = \{z \in A \mid xRz\} \Rightarrow \forall y \in A$ s.t. $xRy, y \in [x]$. Furthermore, $[y] = \{w \in A \mid yRw\}$
   $\Rightarrow \forall w \in [y], yRw$ but $xRy \Rightarrow xRw$ by transitivity. Therefore, $w \in [x]$. We have shown $[y] \subseteq [x]$.
   Since R is an equivalence relation, it is also symmetric. **i.e.** $xRy \Leftrightarrow yRx$. So by the same argument with x and y swapped $yRx \Rightarrow [x] \subseteq [y]$. Thus $xRy \Rightarrow [x] = [y]$.
   "$\Leftarrow$" $[x] = [y] \Rightarrow y \in [x]$ but $[x] = \{y \in A \mid xRy\}$
3. "$\Rightarrow$" We will prove the contrapositive. Assume $[x] \cap [y] \neq \emptyset \Rightarrow \exists z \in [x] \cap [y]$. $z \in [x]$ means xRz, whereas $z \in [y]$ means $yRz \Leftrightarrow zRy$ because R is symmetric. We thus have xRz and $zRy \Rightarrow xRy$ by the transitivity of R. xRy contradicts $\neg(xRy)$ so indeed $\neg(xRy) \Rightarrow [x] \cap [y] = \emptyset$
   "$\Leftarrow$" Once again we use the contrapositive:

Assume $\neg(\neg(xRy)) \Leftrightarrow xRy$. By part (2), $xRy \Rightarrow [x] = [y] \Rightarrow [x] \cap [y] \neq \emptyset$ since $x \in [x]$ and $y \in [y]$, **i.e.** these equivalence classes are non-empty. We have obtained the needed contradiction.

qed

Q: What partition does "=" impose on $\mathbb{R}$?

A: $[x] = \{x\}$ since $E = \{(x, x) \mid x \in \mathbb{R}\}$ the diagonal.
The one-element equivalence class is the smallest equivalence class possible (by definition, an equivalence class cannot be empty as it contains x itself). We call such a partition the finest possible partition.

Remark: The theorem above shows how every equivalence relation partitions a set. It turns out every partition of a set can be used to define an equivalence relation: xRy if x and y belong to the same subset of the partition (check this is indeed an equivalence relation!). Therefore, there is a 1-1 correspondence between partitions and equivalence relations: to each equivalence relation there corresponds a partition and vice versa.

4.3 Partial Orders

Task: Understand another type of relation with special properties.

Definition: Let A be a set. A relation R on A is called anti-symmetric if $\forall x, y \in A$ s.t. $xRy \wedge yRx$, then $x = y$.

Definition: A partial order is a relation on a set A that is reflexive, anti-symmetric, and transitive.

Examples:

1. $A = \mathbb{R}$ $\quad\quad \leq$ "less than or equal to" is a partial order
   (a) $\forall x \in \mathbb{R}$, $x \leq x \rightarrow$ reflexive
   (b) $\forall x, y \in \mathbb{R}$ s.t. $x \leq y \wedge y \leq x \implies x = y \rightarrow$ anti-symmetric
   (c) $\forall x, y, z \in \mathbb{R}$ s.t. $x \leq y \wedge y \leq z \implies x \leq z \rightarrow$ transitive
   Same conclusion if $A = \mathbb{Z}$ or $A = \mathbb{N}$
2. A is a set. Consider $P(A)$, the power set of A. The relation $\subseteq$ "being a subset of" is a partial order.
   (a) $\forall B \in P(A), B \subseteq B \rightarrow$ reflexive.
   (b) $\forall B, C \in P(A), B \subseteq C \wedge C \subseteq B \implies B = C$ (recall the criterion for proving equality of sets) $\rightarrow$ anti-symmetric
   (c) $\forall B, C, D \in P(A)$ s.t. $B \subseteq C \wedge C \subseteq D \implies B \subseteq D \rightarrow$ transitive

The most important example of a partial order is example (2) "being a subset of".

Q: Why is "being a subset of" a partial order as opposed to a total order?

A: There might exist subsets B, C of A s.t. neither $B \subseteq C$ nor $C \subseteq B$ holds, **i.e.** where B and C are not related via inclusion.

5 Functions

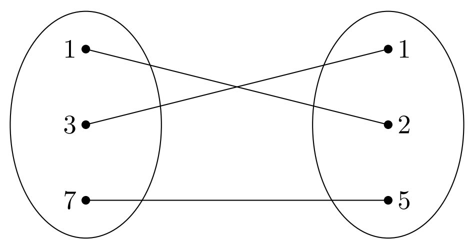
Task: Define a function rigorously and make sense of terminology associated to functions.

Definition: Let A, B be sets. A function $f : A \to B$ is a rule that assigns to every element of A one and only one element of B, **i.e.** $\forall x \in A\ \exists! y \in B$ s.t. $f(x) = y$. A is called the domain of f and B is called the codomain.

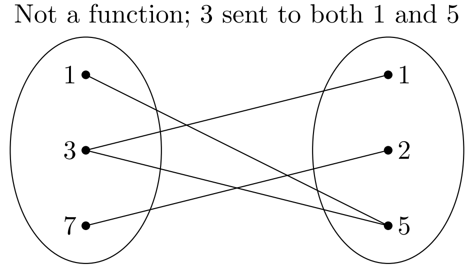
Examples:

1. $A = \{1, 3, 7\}$
   $B = \{1, 2, 5\}$

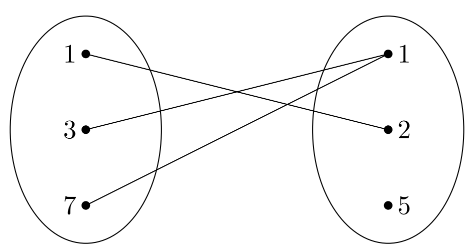
Is a function.



Not a function; 3 sent to both 1 and 5



Is a function.



2. $A = B = \mathbb{R}$ $\quad$ $F : \mathbb{R} \to \mathbb{R}$ given by $f(x) = x$ is called the identity function.

Definition: Let A, B be sets, and let $f : A \to B$ be a function. The range of f denoted by $f(A)$ is the subset of B defined by $f(A) = \{y \in B \mid \exists x \in A$ s.t. $f(x) = y\}$.

Definition: Let A be a set. A Boolean function on A is a function $f : A \to \{T, F\}$, which has A as its domain and the set of truth values $\{T, F\}$ as is codomain. $f : A \to \{T, F\}$ thus assigns truth values to the elements of A.

Function are often represented by graphs. If $f : A \to B$ is a function, the graph of f denoted $\Gamma(f)$ is the subset of the Cartesian product of the domain with the codomain $A \times B$ given by $\{(x, f(x)) \mid x \in A\}$.

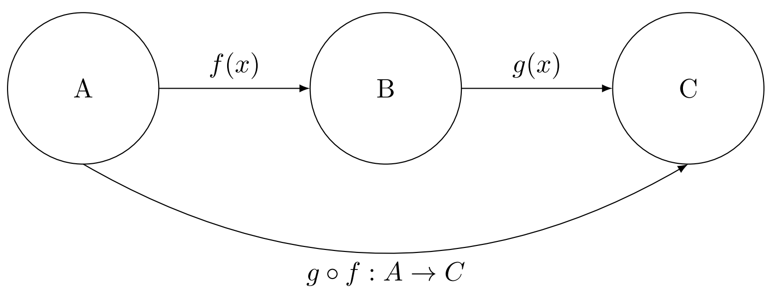
Q: Is it possible to obtain every subset of $A \times B$ as the graph of some function?

A: No! For $f : A \to B$ to be a function $\forall x \in A \quad \exists! y \in B$ s.t. $f(x) = y$, so for $\Gamma \subseteq A \times B$ to be the graph of some function, Γ must satisfy that $\forall x \in A \quad \exists! y \in B$ s.t. $(x, y) \in \Gamma$. Then we can define f by letting $y = f(x)$.

NB For the usual set-up of a function $f : \mathbb{R} \to \mathbb{R}$, this observation amounts to the "vertical line test," which you have seen before coming to university.

5.1 Composition of Functions

Task: Understand the natural operation that allows us to combine functions.



Example:

$f : \mathbb{R} \to \mathbb{R} \qquad f(x) = 2x$
$g : \mathbb{R} \to \mathbb{R} \qquad g(x) = \cos x$
$g \circ f(x) = g(f(x)) = g(2x) = \cos(2x)$
$f \circ g(x) = f(g(x)) = f(\cos x) = 2\,(\cos x) = 2\,\cos x$

5.2 Inverting Functions

Task: Figure out which properties a function has to satisfy so that its action can be undone, **i.e. when** we can define an inverse to the original function.

Given $f : A \to B$, want $f^{-1} : B \to A$ s.t. $f^{-1} \circ f : A \to A$ is the identity $f^{-1} \circ f(x) = f^{-1}(f(x)) = x$
$A \xrightarrow{f} B \xrightarrow{f^{-1}} A$
It turns out f has to satisfy two properties for f^{-1} to exist:

1. Injective
2. Surjective

Definition: A function $f : A \to B$ is called injective or an injection (sometimes called one-to-one) if $f(x) = f(y) \Rightarrow x = y$

Examples:
$\sin x : [0, \frac{\pi}{2}] \to \mathbb{R}$ is injective
$\sin x : \mathbb{R} \to \mathbb{R}$ is not injective because $\sin 0 = \sin \pi = 0$

Definition: A function $f : A \to B$ is called surjective or a surjection (sometimes called onto) if $\forall z \in B\ \exists x \in A$ s.t. $f(x) = z$.

Remark: f assigns a value to each element of A by its definition as a function, but it is not required to cover all of B. f is surjective if its range is all of B.

Examples:
$\sin x : \mathbb{R} \to [-1, 1]$ is surjective
$\sin x : \mathbb{R} \to \mathbb{R}$ is not surjective since $\nexists x \in \mathbb{R}$ s.t. $\sin x = 2$. We know $|\sin x| \leq 1\ \forall x \in \mathbb{R}$

Definition: A function $f : A \to B$ is called bijective or a bijection if f is both injective and surjective.

Example: $f : \mathbb{R} \to \mathbb{R} \qquad f(x) = 2x + 1$ is bijective.

- Check injectivity: $f(x_1) = f(x_2) \Rightarrow 2x_1 + 1 = 2x_2 + 1 \Leftrightarrow 2x_1 = 2x_2 \Leftrightarrow x_1 = x_2$ as needed.
- Check surjectivity: $\forall z \in \mathbb{R}\ f(x) = z$ means $2x + 1 = z$.
  Solve for x: $2x = z - 1 \Rightarrow x = \frac{z-1}{2} \in \mathbb{R} \Rightarrow f$ is surjective.

Remark: All bijective functions have inverses because we can define the inverse of a bijection and it will be a function:

- Surjectivity ensures f^{-1} assigns an element to every element of B (its domain).
- Injectivity ensures f^{-1} assigns to each element of B one and only one element of A.

Conclusion: $f : A \to B$ bijective $\Rightarrow f^{-1}$ exists, **i.e.** f^{-1} is a function. It turns out (reverse the arguments above) that f^{-1} exists $\Rightarrow f : A \to B$ is bijective.
Altogether we get the following theorem:

Theorem: Let $f : A \to B$ be a function. f^{-1} exists $\Leftrightarrow f : A \to B$ is bijective.

Q: How do we find the inverse function f^{-1} given $f : A \to B$?

A: If $f(x) = y$, solve for x as a function of y since $f^{-1}(f(x)) = f^{-1}(y) = x$ as $f^{-1} \circ f$ is the identity.

Example: $f(x) = 2x + 1 = y$. Solve for x in terms of y.
$f : \mathbb{R} \to \mathbb{R}$
$2x = y - 1$ $\qquad x = \frac{y-1}{2}$

5.3 Functions Defined on Finite Sets

Task: Derive conclusions about a function given the number of elements of the domain and codomain, if finite; understand the pigeonhole principle.

Proposition: Let A, B be sets and let $f : A \to B$ be a function. Assume A is finite. Then f is injective $\Leftrightarrow f(A)$ has the same number of elements as A.

Proof:

A is finite so we can write it as $A = \{a_1, a_2, ..., a_p\}$ for some p. Then $f(A) = \{f(a_1), f(a_2), ..., f(a_p)\} \subseteq B$. A priori, some $f(a_i)$ might be the same as some $f(a_j)$. However, f injective $\Leftrightarrow f(a_i) \neq f(a_j)$ whenever $i \neq j \Leftrightarrow f(A)$ has exactly p elements just like A.

qed

Corollary 1 Let A, B be finite sets such that $\#(A) = \#(B)$. Let $f : A \to B$ be a function. f is injective $\Leftrightarrow f$ is bijective.

Proof:

"$\Rightarrow$" Suppose $f : A \to B$ is injective. Since A is finite, by the previous proposition, $f(A)$ has the same number of elements as A, but $f(A) \subseteq B$ and B has the same number of elements as $A \Rightarrow \#(A) = \#(f(A)) = \#(B)$, which means $f(A) = B$, **i.e.** f is also surjective $\Rightarrow f$ is bijective.

"$\Leftarrow$" f is bijective $\Rightarrow f$ is injective.

qed

Corollary 2 (The Pigeonhole Principle) Let A, B be finite sets, and let $f : A \to B$ be a function. If $\#(B) < \#(A)$, $\exists\, a, a' \in A$ with $a \neq a'$ such that $f(a) = f(a')$.

Remark: The name pigeonhole principle is due to Paul Erdös and Richard Rado. Before it was known as the principle of the drawers of Dirichlet. It has a simple statement, but it's a very powerful result in both mathematics and computer science.

Proof: Since $f(A) \subseteq B$ and $\#(B) < \#(A)$, $f(A)$ cannot have as many elements as A, so by the proposition, f cannot be injective, namely $\exists\, a, a' \in A$ with $a \neq a'$ (**i.e.** distinct elements) s.t. $f(a) = f(a')$.

qed

Examples:

1. You have 8 friends. At least two of them were born the same day of the week. #(days of the week) $= 7 < 8$.
2. A family of five gives each other presents for Christmas. There are 12 presents under the tree. We conclude at least one person got three presents or more.
3. In a list of 30 words in English, at least two will begin with the same letter. #(Letters in the English alphabet) $= 26 < 30$.

5.4 Behaviour of Functions on Infinite Sets

Let A be a set, and $f : A \to A$ be a function. If A is finite, then corollary 1 tells us f injective $\Leftrightarrow$ f bijective. What if A is not finite?

5.4.1 Hilbert's Hotel problem (jazzier name: Hilbert's paradox of the Grand Hotel)

A fully occupied hotel with infinitely many rooms can always accommodate an additional guest as follows: The person in Room 1 moves to Room 2. The person in Room 2 moves to Room 3 and so on, **i.e.** if the rooms are x_1, x_2, x_3... define the function $f(x_1) = x_2, f(x_2) = x_3, ..., f(x_m) = x_{m+1}$.

Claim: As defined f is injective but not surjective (hence not bijective!). Let $H = \{x_1, x_2, ...\}$ be the hotel consisting of infinitely many rooms. $f : H \to H$ is given by $f(x_n) = x_{n+1}$. $f(H) = H \backslash \{x_1\}$. We can use this idea to prove:

Proposition: A set A is finite $\Leftrightarrow$ $\forall f : A \to A$ an injective function is also bijective.

Proof: "$\Rightarrow$" If the set A is finite, then it follows immediately from Corollary 1 that every injective function $f : A \to A$ is bijective.
"$\Leftarrow$" We prove the contrapositive. Suppose that the set A is infinite. We shall construct an injective function that is not bijective. Since A is infinite, there exists some infinite sequence $x_1, x_2, x_3, ...$ consisting of distinct elements of A, i.e. an element of A occurs at most once in this sequence. Then there exists a function $f : A \to A$ such that $f(x_n) = x_{n+1}$ for all integers $n \geq 1$ and $f(x) = x$ if x is an element of A that is not in the sequence $x_1, x_2, x_3, ...$ If x is not a member of the infinite sequence $x_1, x_2, x_3, ...$, then the only element of A that gets mapped to x is the element x itself; if $x = x_n$, where $n > 1$, then the only element of A that gets mapped to x is x_{n-1}. It follows that the function f is injective. It is not surjective, however, since no element of A gets mapped to x_1. This function f is thus an example of a function from the set A to itself, which is injective but not bijective.

qed

6 Mathematical Induction

Task: Understand how to construct a proof using mathematical induction.

$\mathbb{N} = \{0, 1, 2, ...\}$ set of natural numbers.
Recall that $\mathbb{N}$ is constructed using 2 axioms:

1. $0 \in \mathbb{N}$
2. If $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$

Remarks:

1. This is exactly the process of counting.
2. If we start at 1, then we construct $\mathbb{N}^* = \{1, 2, 3, 4, ...\} = \mathbb{N}\backslash\{0\}$

via the axioms

1. $1 \in \mathbb{N}^*$
2. if $n \in \mathbb{N}^*$, then $n + 1 \in \mathbb{N}^*$

$\mathbb{N}$ or $\mathbb{N}^*$ is used for mathematical induction.

6.1 Mathematical Induction Consists of Two Steps:

Step 1 Prove statement $P(1)$ called the base case.

Step 2 For any n, assume $P(n)$ and prove $P(n+1)$. This is called the inductive step. In other words, step 2 proves the statement $\forall n P(n) \to P(n + 1)$

Remark: Step 2 is not just an implication but infinitely many! In logic notation, we have:

Step 1 $P(1)$

Step 2 $\forall n(P(n) \to P(n + 1))$
Therefore, $\forall n P(n)$
Let's see how the argument proceeds:

1. $P(1)$ Step 1 (base case)
2. $P(1) \to P(2)$ by Step 2 with $n = 1$
3. $P(2)$ by Modus Ponens (tautology #10) applied to 1 & 2
4. $P(2) \to P(3)$ by Step 2 with $n = 2$
5. $P(3)$ by Modus Ponens (tautology #10) applied to 3 & 4
6. $P(3) \to P(4)$ by Step 2 with $n = 3$

7. $P(4)$ by Modus Ponens (tautology #10) applied to 5 & 6

   $\vdots$

8. $P(n)$ for any n.

   This is like a row of dominos: knocking over the first one in a row makes all the others fall. Another idea is climbing a ladder.

Examples:

1. Prove $1+3+5+...+(2n-1)=n^2$ by induction.

   Base Case: Verify statement for $n=1$

   When $n=1$, $2n-1=2\times 1-1=1^2$

   Inductive Step: Assume $P(n)$, **i.e.** $1+3+5+...+(2n-1)=n^2$ and seek to prove $P(n+1)$, **i.e.** the statement $1+3+5...+(2n-1)+[2(n+1)-1]=(n+1)^2$

   We start with LHS: $\underbrace{1+3+5+...+(2n-1)}_{n^2}+[2(n+1)-1]=$

   $n^2+2n+2-1=n^2+2n+1=(n+1)^2$

2. Prove $1+2+3+...+n=\frac{n(n+1)}{2}$ by induction.

   Base Case: Verify statement for $n=1$

   When $n=1, 1=\frac{1\times(1+1)}{2}=\frac{1\times 2}{2}=1$

   Inductive Step: Assume $P(n)$, **i.e.** $1+2+3+...+n=\frac{n\times(n+1)}{2}$ and seek to prove $1+2+3+...+n+(n+1)=\frac{(n+1)(n+2)}{2}$

   $\underbrace{1+2+3+...+n}_{\frac{n(n+1)}{2}}+n+1=\frac{n(n+1)}{2}+n+1=(n+1)(\frac{n}{2}+1)=$

   $(n+1)\frac{n+2}{2}=\frac{(n+1)(n+2)}{2}$ as needed.

Remarks:

1. For some arguments by induction, it might be necessary to assume not just $P(n)$ at the inductive step but also $P(1), P(2), ..., P(n-1)$. This is called <u>strong induction</u>.

   Base Case: Prove $P(1)$

   Inductive Step: Assume $P(1), P(2), ..., P(n)$ and prove $P(n+1)$.

   An example of result requiring the use of strong induction is the <u>Fundamental Theorem of Arithmetic</u>: $\forall n\in\mathbb{N}, n\geq 2, n$ can be expressed as a product of one or more prime numbers.

2. One has to be careful with arguments involving induction. Here is an illustration why:

   <u>Polya's argument that all horses are the same colour:</u>

   Base Case: $P(1)$ There is only one horse, so that has a colour.

   Inductive Step Assume any n horses are the same colour.

Consider a group of $n+1$ horses. Exclude the first horse and look at the other n. All of these are the same colour by our assumption. Now exclude the last horse. The remaining n horses are the same colour by our assumption. Therefore, the first horse, the horses in the middle, and the last horse are all of the same colour. We have established the inductive step.

Q: Where does the argument fail?

A: For $n = 2, P(2)$ is false because there are no middle horses to compare to.

3. The Grand Hotel Cigar Mystery

Recall Hilbert's hotel - the grand Hotel. Suppose that the Grand Hotel does not allow smoking and no cigars may be taken into the hotel. In spite of the rules, the guest in Room 1 goes to Room 2 to get a cigar. The guest in Room 2 goes to Room 3 to get 2 cigars (one for him and one for the person in room 1), etc. In other words, guest in Room N goes to Room N+1 to get N cigars. They will each get back to their rooms, smoke one cigar, and give the rest to the person in Room N-1.

Q: Where is the fallacy?

A: This is an induction argument without a base case. No cigars are allowed in the hotel, so no guests have cigars. An induction cannot get off the ground without a base case.

7 Abstract Algebra

Task: Understand binary operations, semigroups, monoids, and groups as well as their properties.

7.1 Binary Operations

Definition: Let A be a set. A binary operation $*$ on A is an operation applied to any two elements $x, y \in A$ that yields an element $x * y$ in A. In other words, $*$ is a binary operation on A if $\forall x, y \in A, x * y \in A$.

Examples:

1. $\mathbb{R}, +$ addition on $\mathbb{R} : \forall x, y \in \mathbb{R}, x + y \in \mathbb{R}$
2. $\mathbb{R}, -$ subtraction on $\mathbb{R} : \forall x, y \in \mathbb{R}, x - y \in \mathbb{R}$
3. $\mathbb{R}, \times$ multiplication on $\mathbb{R} : \forall x, y \in \mathbb{R}, x \times y \in \mathbb{R}$
4. $\mathbb{R}, /$, division on $\mathbb{R}$ is NOT a binary operation because $\forall x \in \mathbb{R}\ \exists\, 0 \in \mathbb{R}$ s.t. $\frac{x}{0}$ is undefined (not an element of $\mathbb{R}$)
5. Let A be the set of all lists or strings. Concatenation is a binary operation.

Definition: A binary operation $*$ on a set A is called commutative if $\forall x, y \in A, x * y = y * x$

Examples:

1. $\mathbb{R}, +$ is commutative since $\forall x, y \in \mathbb{R}, x + y = y + x$
2. $\mathbb{R}, \times$ is commutative since $\forall x, y \in \mathbb{R}, x \times y = y \times x$
3. $\mathbb{R}, -$ is not commutative since $\forall x, y \in \mathbb{R}, x - y \neq y - x$ in general. $x - y = y - x$ only if $x = y$
4. Let M_n be the set of n by n matrices with entries in $\mathbb{R}$, and let $*$ be matrix multiplication. $\forall A, B \in M_n, A * B \in M_n$, so $*$ is a binary operation, but $A * B \neq B * A$ in general. Therefore $*$ is not commutative.

Definition: A binary operation $*$ on a set A is called associative if $\forall x, y, z \in A$ $(x * y) * z = x * (y * z)$

Examples:

1. $\mathbb{R}, +$ is associative since $\forall x, y, z \in \mathbb{R}, (x + y) + z = x + (y + z)$
2. $\mathbb{R}, \times$ is associative since $\forall x, y, z \in \mathbb{R}, (x \times y) \times z = x \times (y \times z)$
3. Intersection $\cap$ on sets is associative since $\forall A, B, C$ sets $(A \cap B) \cap C = A \cap (B \cap C)$.
4. Union $\cup$ on sets is associative since $\forall A, B, C$ sets $(A \cup B) \cup C = A \cup (B \cup C)$
5. $\mathbb{R}, -$ is not associative since $(1-3)-5 = -2-5 = -7$ but $1-(3-5) = 1-(-2) = 1+2 = 3$

Remark: When we are dealing with associative binary operations we can drop the parentheses, **i.e.** $(x * y) * z$ can be written $x * y * z$.

7.2 Semigroups

Definition: A semigroup is a set endowed with an associative binary operation. We denote the semigroup $(A, *)$

Examples:

1. $(\mathbb{R}, +)$ and $(\mathbb{R}, \times)$ are semigroups.
2. Let A be a set and let $P(A)$ be its power set. $(P(A), \cap)$ and $(P(A), \cup)$ are both semigroups.
3. $(M_n, *)$, the set of $n \times n$ matrices with entries in $\mathbb{R}$ with the operation of matrix multiplication (which is associative $\rightarrow$ a bit gory to prove) forms a semigroup.

Since $*$ is associative on a semigroup, we can define a^n :
$a^1 = a$
$a^2 = a * a$
$a^3 = a * a * a$
$\vdots$
Recursively, $a^1 = a$ and $a^n = a * a^{n-1}, \forall n > 1$

NB: In $(\mathbb{R}, \times), \forall a \in \mathbb{R}, a^n = \underbrace{a \times a \times ... \times a}_{n\ times}$, whereas in $(\mathbb{R}, +), \forall a \in \mathbb{R}, a^n = \underbrace{a + a + ... + a}_{n\ times} = na$. Be careful what $*$ stands for!

Theorem: Let $(A, *)$ be a semigroup. $\forall a \in A, a^m * a^n = a^{m+n}, \forall m, n \in \mathbb{N}^*$.

Proof: By induction on m.

Base Case: $m = 1$ $\qquad a^1 * a^n = a * a^n = a^{1+n}$

Inductive Step: Assume the result is true for $m = p$, **i.e.** $a^p * a^n = a^{p+n}$ and seek to prove that $a^{p+1} * a^n = a^{p+1+n}$

$a^{p+1} * a^n = (a * a^p) * a^n = a * (a^p * a^n) = a * a^{p+n} = a^{p+1+n}$

Theorem: Let$(A, *)$ be a semigroup. $\forall a \in A, (a^m)^n = a^{mn}, \forall m, n \in \mathbb{N}^*$

Proof: By induction on n.

Base Case: $n = 1$ $\qquad (a^m)^1 = a^m = a^{m \times 1}$

Inductive Step: Assume the result is true for $n = p$, **i.e.** $(a^m)^p = a^{mp}$ and seek to prove that $(a^m)^{p+1} = a^{m(p+1)}$

$(a^m)^{p+1} = (a^m)^p * a^m = a^{mp} * a^m = a^{mp+m} = a^{m(p+1)}$ by the previous theorem.

7.2.1 General Associative Law

Let $(A, *)$ be a semigroup. $\forall a_1, ..., a_s \in A, a_1 * a_2 * ... * a_s$ has the same value regardless of how the product is bracketed.

Proof Use associativity of $*$.

qed

NB: Unless $(A, *)$ has a commutative binary operation, $a_1 * a_2 * ... * a_s$ does depend on the ORDER in which the $a_j's$ appear in $a_1 * a_2 * ... * a_s$

7.3 Identity Elements

Definition: Let $(A, *)$ be a semigroup. An element $e \in A$ is called an identity element for the binary operation $*$ if $e * x = x * e = x, \forall x \in A$.

Examples:

1. $(\mathbb{R}, +)$ has 0 as the identity element.
2. $(\mathbb{R}, \times)$ has 1 as the identity element.
3. Given a set A, $(P(A), \cup)$ has $\emptyset$ (the empty set) as its identity element, whereas $(P(A), \cap)$ has A as its identity element.
4. $(M_n, *)$ has I_n, the identity matrix, as its identity element.

Theorem A binary operation on a set cannot have more than one identity element, **i.e.** if an identity element exists, then it is unique.

Proof: Assume not (proof by contradiction). Let e and e' both be identity elements for a binary operation on a set A. $e = e * e' = e'$

qed

7.4 Monoids

Definition: A <u>monoid</u> is a set A endowed with an associative binary operation $*$ that has an identity element e. In other words, a monoid is a semigroup $(A, *)$, where $*$ has an identity element e.

Definition: A monoid $(A, *)$ is called <u>commutative</u> (or <u>Abelian</u>) if the binary operation $*$ is commutative.

Examples:

1. $(\mathbb{R}, +)$ is a commutative monoid with $e = 0$.
2. $(\mathbb{R}, \times)$ is a commutative monoid with $e = 1$.
3. Given a set A, $(P(A), \cup)$ is a commutative monoid with $e = \emptyset$.
4. $(M_n, *)$ is a monoid since $e = I_n$, but it is not commutative since matrix multiplication is not commutative.
5. $(\mathbb{N}, +)$ is a commutative monoid with $e = 0$, whereas $(\mathbb{N}^*, +)$ is merely a semigroup (recall $\mathbb{N}^* = \mathbb{N}\backslash\{0\}$)

Let $(A, *)$ be a monoid and let $a \in A$. We define $a^0 = e$, the identity element.

Theorem: Let $(A, *)$ be a monoid and let $a \in A$. Then $a^m * a^n = a^{m+n}, \forall m, n \in \mathbb{N}$.

Remark: Recall that we proved this theorem for semigroups if $m, n \in \mathbb{N}^*$. We now need to extend that result.

Proof: A monoid is a semigroup $\implies \forall a \in A, a^m * a^n = a^{m+n}$ whenever $m, n \in \mathbb{N}^*$, **i.e.** $m > 0$ and $n > 0$. Now let $m = 0$. $a^m * a^n = a^0 * a^n = e * a^n = a^n = a^{0+n}$
If $n = 0, a^m * a^n = a^m * a^0 = a^m * e = a^m = a^{m+0}$.

qed

Theorem: Let $(A, *)$ be a monoid, $\forall a \in A\ \forall m, n \in \mathbb{N}$, $(a^m)^n = a^{mn}$.

Remark: Once again, we had this result for semigroups when $m > 0$ and $n > 0$.

Proof: By the remark, we only need to prove the result when $m = 0$ or $n = 0$. If $m = 0, (a^0)^n = (e)^n = e = a^0 = a^{0\times n}$. If $n = 0$, then $(a^m)^0 = e = a^0 = a^{0\times m}$.

qed

7.5 Inverses

Task: Understand what an inverse is and what formal properties it satisfies.

Definition: Let $(A, *)$ be a monoid with identity element e and let $x \in A$. An element y of A is called the <u>inverse</u> of x if $x * y = y * x = e$. If an element $x \in A$ has an inverse, then x is called <u>invertible</u>.

Examples:

1. $(\mathbb{R}, +)$ has identity element 0. $\forall x \in \mathbb{R}, (-x)$ is the inverse of x since $x + (-x) = (-x) + x = 0$.
2. $(\mathbb{R}, \times)$ has identity element 1. $x \in \mathbb{R}$ is invertible only if $x \neq 0$. If $x \neq 0$, the inverse of x is $\frac{1}{x}$ since $x \times \frac{1}{x} = \frac{1}{x} \times x = 1$.
3. $(M_n, *)$ the identity element is I_n. $A \in M_n$ is invertible if $\det(A) \neq 0$. A^{-1} the inverse is exactly the one you computed in linear algebra. If $\det(A) = 0, A$ is <u>NOT</u> invertible.
4. Given a set $A, (P(A), \cup)$ has $\emptyset$ as its identity element. Of all the elements of $P(A)$, only $\emptyset$ is invertible and has itself as its inverse: $\emptyset \cup \emptyset = \emptyset \cup \emptyset = \emptyset$.

Theorem: Let $(A, *)$ be a monoid. If $a \in A$ has an inverse, then that inverse is unique.

Proof: By contradiction: Assume not, then $\exists a \in A$ s.t. both b and c in A are its inverses, **i.e.** $a * b = b * a = e$, the identity element of $(A, *)$, and $a * c = c * a = e$, where $b \neq c$. Then $b = b * e = b * (a * c) = (b * a) * c = e * c = c$. $\Rightarrow\Leftarrow$

qed

Since every invertible element a of a monoid $(A, *)$ has a unique inverse, we can denote the inverse by the more standard notation a^{-1}.

Next, we need to understand inverses of elements obtained via the binary operation:

Theorem: Let $(A, *)$ be a monoid, and let a, b be invertible elements of A. Then $a * b$ is also invertible, and $(a * b)^{-1} = b^{-1} * a^{-1}$.

Remark: You might remember this formula from linear algebra when you looked at the inverse of a product of matrices AB.

Proof: Let e be the identity element of $(A, *)$. $a * a^{-1} = a^{-1} * a = e$, and $b * b^{-1} = b^{-1} * b = e$. We would like to show $b^{-1} * a^{-1}$ is the inverse of $a * b$ by computing $(a * b) * (b^{-1} * a^{-1})$ and $(b^{-1} * a^{-1}) * (a * b)$ and showing both are e.

$(a * b) * (b^{-1} * a^{-1}) = a * (b * b^{-1}) * a^{-1} = a * e * a^{-1} = a * a^{-1} = e$

$(b^{-1}*a^{-1})*(a*b) = b^{-1}*(a^{-1}*a)*b = b^{-1}*e*b = (b^{-1}*e)*b = b^{-1}*b = e$

Thus $b^{-1} * a^{-1}$ satisfies the conditions needed for it to be the inverse of $a * b$. Since an inverse is unique, $a * b$ is invertible and $b^{-1} * a^{-1}$ is its inverse.

qed

Theorem: Let $(A, *)$ be a monoid, and let $a, b \in A$. Let $x \in A$ be invertible. $a = b * x \Leftrightarrow b = a * x^{-1}$. Similarly, $a = x * b \Leftrightarrow b = x^{-1} * a$

Proof: Let e be the identity element of $(A, *)$.

First $a = b * x \Leftrightarrow b = a * x^{-1}$:

"$\Rightarrow$" Assume $a = b*x$. Then $a*x^{-1} = (b*x)*x^{-1} = b*x*x^{-1} = b*e = b$ as needed.

"$\Leftarrow$" Assume $b = a*x^{-1}$. Then $b*x = (a*x^{-1})*x = a*(x^{-1}*x) = a*e = a$ as needed.

Apply the same type of argument to show $a = x * b \Leftrightarrow b = x^{-1} * a$.

qed

Let $(A, *)$ be a monoid. We can now make sense of a^n for $n \in \mathbb{Z}, n < 0$ for every $a \in A$ invertible. Since n is a negative integer, $\exists p \in \mathbb{N}$ s.t. $n = -p$. Set $a^n = a^{-p} = (a^p)^{-1}$.

Theorem: Let $(A, *)$ be a monoid, and let $a \in A$ be invertible. Then $a^m * a^n = a^{m+n}$ $\forall m, n \in \mathbb{Z}$.

Proof: When $m \geq 0$ and $n \geq 0$, we have already proven this result. The rest of the proof splits into cases.

Case 1: $m = 0$ or $n = 0$

If $m = 0$, $n \in \mathbb{Z}$, $a^m * a^n = a^0 * a^n = e * a^n = a^n = a^{0+n}$ as needed.

If $m \in \mathbb{Z}$, $n = 0$, $a^m * a^n = a^m * a^0 = a^m * e = a^m = a^{m+0}$ as needed.

Case 2: $m < 0$ and $n < 0$

$m < 0 \Rightarrow \exists p \in \mathbb{N}\, s.t.\, p = -m$. $n < 0 \Rightarrow \exists q \in \mathbb{N}\, s.t.\, q = -n$.

$a^m = a^{-p} = (a^p)^{-1}$ and $a^n = a^{-q} = (a^q)^{-1}$

$a^m * a^n = (a^p)^{-1} * (a^q)^{-1} = (a^q * a^p)^{-1} = (a^{p+q})^{-1} = a^{-(p+q)} = a^{-q-p} = a^{m+n}$

Case 3: m and n have opposite signs.

Without loss of generality, assume $m < 0$ and $n > 0$ (the case $m > 0$ and $n < 0$ is handled by the same argument). Since $m < 0, \exists p \in \mathbb{N}\, s.t.\, p = -m$. This case splits into two subcases:

Case 3.1: $m + n \geq 0$

Set $q = m + n$. Then $a^{m+n} = a^q = e * a^q = (a^p)^{-1} * a^p * a^q = (a^p)^{-1} * a^{p+q} = a^{-p} * a^{p+q} = a^m * a^{-m+m+n} = a^m * a^n$

Case 3.2: $m + n < 0$

Set $q = -(m+n) = -m-n \in \mathbb{N}^*$. Then $a^{m+n} = a^{-q} = (a^q)^{-1} * e = (a^q)^{-1} * (a^{-n} * a^n) = (a^q)^{-1} * (a^n)^{-1} * a^n = (a^n * a^q)^{-1} * a^n = (a^{n+q})^{-1} * a^n = (a^{n-m-n})^{-1} * a^n = (a^{-m})^{-1} * a^n = (a^p)^{-1} * a^n = a^m * a^n$

Theorem: Let $(A, *)$ be a monoid, and let a be an invertible element of A. $\forall m, n \in \mathbb{Z}, (a^m)^n = a^{mn}$.

Proof: We consider 3 cases:

Case 1: $n > 0$, **i.e.** $n \in \mathbb{N}^*$. $m \in \mathbb{Z}$ with no additional restrictions. We proceed by induction on n.

Base Case: $n = 1$ $\qquad (a^m)^1 = a^m = a^{m \times 1}$

Inductive Step: We assume $(a^m)^n = a^{mn}$ and seek to prove $(a^m)^{n+1} = a^{m(n+1)}$. Start with $(a^m)^{n+1} = (a^m)^n * (a^m)^1 = a^{mn} * a^m = a^{mn+m} = a^{m(n+1)}$

Case 2: $n = 0$; no restriction on $m \in \mathbb{Z}$

$(a^m)^n = (a^m)^0 = e = a^0 = a^{m \times 0} = a^{mn}$

Case 3: $n < 0$; no restriction on $m \in \mathbb{Z}$.

Since $n < 0, \exists p \in \mathbb{N}\, s.t.\, p = -n$. By case 1, $(a^m)^p = a^{mp}$

$(a^m)^n = (a^m)^{-p} = ((a^m)^p)^{-1} = (a^{mp})^{-1} = a^{-mp} = a^{m(-p)} = a^{mn}$

7.6 Groups

A notion formally defined in the 1870's even though theorems about groups were proven as early as a century before that.

Definition: A group is a monoid in which every element is invertible. In other words, a group is a set A endowed with a binary operation $*$ satisfying the following properties:

1. $*$ is associative, **i.e.** $\forall x, y, z \in A, (x * y) * z = x * (y * z)$
2. There exists an identity element $e \in A$, **i.e.** $\exists e \in A\, s.t. \forall a \in A, a * e = e * a = a$
3. Every element of A is invertible, **i.e.** $\forall a \in A\ \exists a^{-1} \in A\, s.t.\, a * a^{-1} = a^{-1} * a = e$

Notation for Groups: $(A, *)$ or $(\underbrace{A}_{set}, \underbrace{*}_{operation}, \underbrace{e}_{identity})$

Remark: Closure under the operation $*$ is implicit in the definition **i.e.** $\forall a, b \in A, a * b \in A$

Definition: A group $(A, *, e)$ is called commutative or Abelian if its operation $*$ is commutative.

Examples:

1. $(\mathbb{R}, +, 0)$ is an Abelian group.

   $-x$ is the inverse of $x, \forall x \in \mathbb{R}$

2. $(\mathbb{Q}^*, \times, 1)$ $\quad \mathbb{Q}^* = \mathbb{Q} \backslash \{0\}$ $\quad (\mathbb{Q}^*, \times, 1)$ is Abelian

   $\forall q \in \mathbb{Q}^*, q^{-1} = \frac{1}{q}$ is the inverse.

3. $(\mathbb{R}^3, +, 0)$ vectors in $\mathbb{R}^3$ with vector addition forms an Abelian group.

   $(x, y, z) + (x', y', z') = (x + x', y + y', z + z')$ vector addition.

   $0 = (0, 0, 0)$ is the identity. $(-x, -y, -z) = -(x, y, z)$ is the inverse of (x, y, z).

4. $(\widetilde{M}_n, *, I_n)$ $n \times n$ invertible matrices with real coefficients under matrix multiplication with I_n as the identity element forms a group, which is NOT Abelian.

5. Set $A = \mathbb{Z}$ and recall the equivalence relation $x \equiv y \bmod 3$ **i.e.** x and y have the same remainder under the division by 3. Recall that $\mathbb{Z}/\sim = \{0, 1, 2\}$, **i.e.** the set of equivalence classes under the partition determined by this equivalence relation. We denote $\mathbb{Z}/\sim = \{0, 1, 2\} = \mathbb{Z}_3$

   Consider $(\mathbb{Z}_3, \oplus_3, 0)$ where $\oplus_3$ is the operation of addition modulo 3, **i.e.** $1 + 0 = 1, 1 + 1 = 2, 1 + 2 = 3 \equiv 0 \bmod 3$.

Claim: $(\mathbb{Z}_3, \oplus_3, 0)$ is an Abelian group.

Proof of Claim: Associativity of $\oplus_3$ follows from the associativity of $+$, addition on $\mathbb{Z}$. Clearly, 0 is the identity (don't forget 0 stands for all elements with remainder 0 under division by 3, **i.e.** $\{0, 3, -3, 6, -6, ...\}$). To compute inverses recall that $a \oplus_3 a^{-1} = 0$, 0 is the inverse of 0 because $0 + 0 = 0$. 2 is the inverse of 1 because $1 + 2 = 3 \equiv 0 \bmod 3$, and 1 is the inverse of 2 because $2 + 1 = 3 \equiv 0 \bmod 3$.
More generally, consider the equivalence relation on $\mathbb{Z}$ given by $x \equiv y \bmod n$ for $n \geq 1$. $\mathbb{Z}/\sim = \{0, 1, ..., n-1\} = \mathbb{Z}_n$. All possible remainders under division by n are the equivalence classes. Let $\oplus_n$ be addition mod n. By the same argument as above, $(\mathbb{Z}_n, \oplus_n, 0)$ is an Abelian group.

Q: What if we consider multiplication mod n, **i.e.** $\otimes_n$. Is $(\mathbb{Z}_n, \otimes_n, 1)$ a group?

A: No! $(\mathbb{Z}_n, \otimes_n, 1)$ is not a group because 0 is not invertible: for any $a \in \mathbb{Z}_n$, $0 \otimes_n a = a \otimes_n 0 = 0 \neq 1$.

Q: Can this be fixed?

A: Troubleshoot how to get rid of 0.
Consider $\mathbb{Z}_n^* = \mathbb{Z}_n \backslash \{0\} = \{1, 2, ..., n-1\}$ all non-zero elements in $\mathbb{Z}_n^*$. This eliminates 0 as an element, but can 0 arise any other way from the binary operation? It turns out the answer depends on n. If n is not prime, say $n = 6$, we get **zero divisors**, i.e. elements that yield 0 when multiplied. These are precisely the factors of n. For $n = 6$, $\mathbb{Z}_6^* = \{1, 2, 3, 4, 5\}$ but $2 \otimes_6 3 = 6 \equiv 0 \bmod 6$, so 2 and 3 are zero divisors.

Claim: If n is prime, then $(\mathbb{Z}_n^*, \otimes_n, 1)$ is an Abelian group.
Used in cryptography $\rightarrow n$ is taken to be a very large prime number.
As an example, let us look at the multiplication table for $\mathbb{Z}_5^*$ to see the inverse of various elements: $\mathbb{Z}_5^* = \mathbb{Z}_5 \backslash \{0\} = \{0, 1, 2, 3, 4\} \backslash \{0\} = \{1, 2, 3, 4\}$

	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

$1^{-1} = 1 \quad 1 \otimes_5 1 = 1$
$2^{-1} = 3 \quad 2 \otimes_5 3 = 6 \equiv 1 \bmod 5$
$3^{-1} = 2 \quad 3 \otimes_5 2 = 6 \equiv 1 \bmod 5$
$4^{-1} = 4 \quad 4 \otimes_5 4 = 16 \equiv 1 \bmod 5$

The fact that $(\mathbb{Z}_n^*, \otimes_n, 1)$ is Abelian follows from the commutativity of multiplication on $\mathbb{Z}$.

6. Let $(A, *, e)$ be any group, and let $a \in A$.
Consider $A' = \{a^m \mid m \in \mathbb{Z}\}$ all powers of a. It turns out $(A', *, e)$ is a group called the cyclic group determined by a. $(A', *, e)$ is Abelian regardless of whether the original group was Abelian or not because of the theorem we proved on powers of a: $\forall m, n \in \mathbb{Z}\ a^m * a^n = a^{m+n} = a^{n+m} = a^n * a^m$.
Cyclic groups come in two flavours: finite (A' is a finite set) and infinite (A' is an infinite set).
For example, let $(A, *, e) = (\mathbb{Q}^*, \times, 1)$
If $a = -1$ $\quad A' = \{(-1)^m \mid m \in \mathbb{Z}\} = \{-1, 1\}$ is finite.
If $a = 2$ $\quad A' = \{2^m \mid m \in \mathbb{Z}\} = \{1, 2, \frac{1}{2}, 4, \frac{1}{4}, ...\}$ is infinite.

7.7 Homomorphisms and Isomorphisms

Task: Understand the most natural functions between objects in abstract algebra such as semigroups, monoids or groups.

Definition: Let $(A, *)$ and $(B, *)$ both be semigroups, monoids or groups. A function $f : A \rightarrow B$ is called a homomorphism if

$$f(x * y) = f(x) * f(y)\ \forall x, y \in A.$$

In other words, if f is a function that respects (behaves well with respect to) the binary operation.

Examples:

1. Consider $(\mathbb{Z}, +, 0)$ and $(\mathbb{R}^*, \times, 1)$.
   Pick $a \in \mathbb{R}^*$, then $f(n) = a^n$ is a homomorphism between $(\mathbb{Z}, +, 0)$ and $(\mathbb{R}^*, \times, 1)$ because $(\mathbb{R}^*, \times, 1)$ is a group, and we proved for groups that $a^{m+n} = f(m+n) = a^m * a^n = f(m) * f(n)\ \forall m, n \in \mathbb{Z}$.
2. More generally, $\forall a \in A$ invertible, where $(A, *)$ is a monoid with identity element e, $f(m) = a^m$ gives a homomorphism between $(\mathbb{Z}, +, 0)$ and $(A', *, e)$, where as before $A' = \{a^m \mid m \in \mathbb{Z}\} \subset A$.
   We get even better behaviour if we require $f : A \to B$ to be bijective.

Definition: Let $(A, *)$ and $(B, *)$ both be semigroups, monoids or groups. A function $f : A \to B$ is called an isomorphism if $f : A \to B$ is both bijective <u>AND</u> a homomorphism.

Examples:

1. Let $A' = \{2^m \mid m \in \mathbb{Z}\} = \{1, 2, \frac{1}{2}, 4, \frac{1}{4}, ...\}$
   $f(m) = 2^m$ from $(\mathbb{Z}, +, 0)$ to $(A', \times, 1)$ is an isomorphism since $2^m \neq 2^n$ if $m \neq n$.
2. Let $A' = \{(-1)^m \mid m \in \mathbb{Z}\} = \{-1, 1\}$
   $f(m) = (-1)^m$ from $(\mathbb{Z}, +, 0)$ to $(A', \times, 1)$ is <u>NOT</u> an isomorphism since it's not injective $(-1)^2 = (-1)^4 = 1$.

Theorem: Let $(A, *)$ and $(B, *)$ both be semigroups, monoids or groups. The inverse $f^{-1} : B \to A$ of any isomorphism $f : A \to B$ from A to B is itself an isomorphism.

Proof: If $f : A \to B$ is an isomorphism $\Rightarrow f : A \to B$ is bijective $\Rightarrow f^{-1} : B \to A$ is bijective (proven when we discussed functions).

To show $f^{-1} : B \to A$ is a homomorphism, let $u, v \in B$. $\exists x, y \in A$ s.t. $x = f^{-1}(u)$ and $y = f^{-1}(v)$, but then $u = f(x)$ and $v = f(y)$.

Since $f : A \to B$ is a homomorphism, $f(x * y) = f(x) * f(y) = u * v$. Then $f^{-1}(u * v) = f^{-1}(f(x * y)) = x * y = f^{-1}(u) * f^{-1}(v)$ as needed.

qed

Definition: Let $(A, *)$ and $(B, *)$ both be semigroups, monoids or groups. If $\exists f : A \to B$ an isomorphism betwen A and B, then $(A, *)$ and $(B, *)$ are said to be isomorphic.

Remark: "Isomorphic" comes from "iso" same and "morphē " form: the same abstract algebra structure on both $(A, *)$ and $(B, *)$ given to you in two different guises. As the French would say: "Même Marie, autre chapeau" same Mary, different hat.

8 Formal Languages

Task: Use what we learned about structures in abstract algebra in order to make sense of formal languages and grammars.
Let A be a finite set. When studying formal languages, we call A an alphabet and the elements of A letters.

Examples:

1. $A = \{0, 1\}$ binary digits
2. $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ decimal digits
3. $A =$ letters of the English alphabet

Definition: $\forall n \in \mathbb{N}^*$, we define a word of length n in the alphabet A as being any string of the form $a_1 a_2 \cdots a_n \; s.t. \; a_i \in A \quad \forall i, 1 \leq i \leq n$. Let A^n be the set of all words of length n over the alphabet A.

Remark: There is a one-to-one correspondence between the string $a_1 a_2 \cdots a_n$ and the ordered n-tuple $(a_1, a_2, ..., a_n) \in A^n = \underbrace{A \times ... \times A}_{n \; times}$, the Cartesian product of n copies of A.

Definition: Let $A^+ = \bigcup\limits_{n=1}^{\infty} A^n = A^1 \cup A^2 \cup A^3 \cup ....$ A^+ is the set of all words of positive length over the alphabet A.

Examples:

1. $A = \{0, 1\}, A^+$ is the set of all binary strings of finite length that is at least one, **i.e.** $0, 1, 01, 10, 00, 11$, etc.
2. If $A =$ letters of the English alphabet, then A^+ consists of all non-empty strings of finite length of letters from the English alphabet.

It is useful to also have the empty word ε in our set of strings. ε has length 0. Define $A^0 = \{\varepsilon\}$ and then adjoin the empty word ε to A^+. We get $A^* = \{\varepsilon\} \cup A^+ = A^0 \cup \bigcup\limits_{n=1}^{\infty} A^n = \bigcup\limits_{n=0}^{\infty} A^n$.

Notation: We denote the length of a word w by $\mid w \mid$.

Next introduce an operation on A^*.

Definition: Let A be a finite set, and let w_1 and w_2 be words in A^*. $w_1 = a_1 a_2 ... a_m$ and $w_2 = b_1 b_2 ... b_n$. The concatenation of w_1 and w_2 is the word $w_1 \circ w_2$, where $w_1 \circ w_2 = a_1 a_2 ... a_m b_1 b_2 ... b_n$. Sometimes $w_1 \circ w_2$ is denoted as just $w_1 w_2$. Note that $\mid w_1 \circ w_2 \mid = \mid w_1 \mid + \mid w_2 \mid$.
Concatenation of words is:

1. associative
2. NOT commutative if A has more than one element.

Proof of (1): Let $w_1, w_2, w_3 \in A^*$. $w_1 = a_1 a_2 ... a_m$ for some $m \in \mathbb{N}$, $w_2 = b_1 b_2 ... b_n$ for some $n \in \mathbb{N}$, and $w_3 = c_1 c_2 ... c_p$ for some $p \in \mathbb{N}$. $(w_1 \circ w_2) \circ w_3 = w_1 \circ (w_2 \circ w_3) = a_1 a_2 ... a_m b_1 b_2 ... b_n c_1 c_2 ... c_p$.

qed

Proof of (2): Since A has at least two elements, $\exists a, b \in A$ s.t. $a \neq b$.

$a \circ b = ab \neq ba = b \circ a$.

qed

A^* is closed under the operation of concatenation $\Rightarrow$ concatenation is a binary operation on A^* as $\forall w_1, w_2 \in A^*$, $w_1 \circ w_2 \in A^*$.

Theorem Let A be a finite set. $(A^*, \circ)$ is a monoid with identity element ε.

Proof: Concatenation $\circ$ is an associative binary operation on A^* as we showed above. Moreover, $\forall w \in A^*, \varepsilon \circ w = w \circ \varepsilon = w$, so ε is the identity element of A^*.

qed

Definition: Let A be a finite set. A language over A is a subset of A^*. A language L over A is called a formal language is $\exists$ a finite set of rules or algorithm that generates exactly L, **i.e.** all words that belong to L and no other words.

Theorem: Let A be a finite set.

1. If L_1 and L_2 are languages over $A, L_1 \cup L_2$ is a language over A.
2. If L_1 and L_2 are languages over $A, L_1 \cap L_2$ is a language over A.
3. If L_1 and L_2 are languages over A, the concatenation of L_1 and L_2 given by $L_1 \circ L_2 = \{w_1 \circ w_2 \in A^* \mid w_1 \in L_1 \wedge w_2 \in L_2\}$ is a language over A.
4. Let L be a language over A. Define $L^1 = L$ and inductively for any $n \geq 1$, $L^n = L \circ L^{n-1}$. L^n is a language over A. Furthermore, $L^* = \{\varepsilon\} \cup L^1 \cup L^2 \cup L^3 \cup ... = \bigcup_{n=0}^{\infty} L^n$ is a language over A.

Proof: By definition, a language over A is a subset of A^*. Therefore, if $L_1 \subseteq A^*$ and $L_2 \subseteq A^*$, then $L_1 \cup L_2 \subseteq A^*$ and $L_1 \cap L_2 \subseteq A^*$. $\forall w_1 \circ w_2 \in L_1 \circ L_2$, $w_1 \circ w_2 \in A^*$ because $w_1 \in A^n$ for some n and $w_2 \in A^m$ for some m, so $w_1 \circ w_2 \in A^{m+n} \subseteq A^* = \bigcup_{n=0}^{\infty} A^n$.

Applying the same reasoning inductively, we see that $L \subset A^* \Rightarrow L^* \subseteq A^*$ as $L^n \subseteq A^* \, \forall n \geq 0$.

qed

Remark: This theorem gives us a theoretic way of building languages, but we need a practical way. The practical way of building a language is through the notion of a grammar.

Definition: A (formal) grammar is a set of production rules for strings in a language.
To generate a language we use:

1. the set A, which is the alphabet of the language;
2. a start symbol <s>;
3. a set of production rules.

Example: $A = \{0, 1\}$; start symbol <s>; 2 production rules given by:

1. <s> $\to$ 0<s>1
2. <s> $\to$ 01

Let's see what we generate: via rule 2, <s> $\to$ 01, so we get <s> $\Rightarrow$ 01
Via rule 1, <s> $\to$ 0<s>1, then via rule 2, 0<s>1 $\to$ 0011. We write the process as <s> $\Rightarrow$ 0<s>1 $\Rightarrow$ 0011.
Via rule 1, <s> $\to$ 0<s>1, then via rule 1 again 0<s>1 $\to$ 00<s>11, then via rule 2, 00<s>11 $\to$ 000111.
We got <s> $\Rightarrow$ 0<s>1 $\Rightarrow$ 00<s>11 $\Rightarrow$ 000111.
The language L we generated thus consists of all strings of the form $0^m 1^m$ (m 0's followed by m 1's) for all $m \geq 1, m \in \mathbb{N}$
We saw 2 types of strings that appeared in this process of generating L:

1. terminals, **i.e.** the elements of A
2. nonterminals, **i.e.** strings that don't consist solely of 0's and 1's such as <s>, 0<s>1, 00<s>11, etc.

The production rules then have the form:

nonterminal $\to$ word over the alphabet V = {terminals, non-terminals}

<T> $\to$ w

In our notation, the set of nonterminals is $V \backslash A$, so <T>$\in V \backslash A$ and $w \in V^* = \bigcup_{n=0}^{\infty} V^n$. To the production rule <T>$\to w$, we can associate the ordered pair (<T>$, w) \in (V \backslash A) \times V^*$, so the set of production rules, which we will denote by P, is a subset of the Cartesian product $(V \backslash A) \times V^*$.
Grammars come in two flavours:

1. Context-free grammars where we can replace any occurrence of <T> by w if <T>$\to w$ is one of our production rules.
2. Context-sensitive grammars only certain replacements of <T> by w are allowed, which are governed by the syntax of our language L.

The example we had was of a context-free grammar. We can now finally define context free-grammars.

Definition: A context-free grammar $(V, A,$ <s>$, P)$ consists of a finite set V, a subset A of V, an element <s> of $V \backslash A$, and a finite subset P of the Cartesian product $V \backslash A \times V^*$.

Notation: $\Big(\underset{set\ of\ terminals\ and\ non\ terminals}{V}, \underset{set\ of\ terminals}{A}, \underset{start\ symbol}{<s>}, \underset{set\ of\ production\ rules}{P}\Big)$

Example: $A = \{0, 1\}$; start symbol <s>; 3 production rules given by:

1. <s>$\rightarrow$ 0<s>1
2. <s>$\rightarrow$ 01
3. <s>$\rightarrow$ 0011

We notice here that the word 0011 can be generated in 2 ways in this context free grammar:

By rule 3, <s>$\rightarrow$ 0011 so <s>$\Rightarrow$ 0011

$\vee$

By rule 1, <s>$\rightarrow$ 0<s>1 and by rule 2, 0<s>1 $\rightarrow$ 0011. Therefore, <s>$\Rightarrow$ 0<s>1 $\Rightarrow$ 0011.

Definition: A grammar is called ambiguous if it generates the same string in more than one way.
Obviously, we prefer to have unambiguous grammars, else we waste computer operations.
Next, we need to spell out how words relate to each other in the production of our language via the grammar:

Definition: Let w' and w'' be words over the alphabet $V = \{$terminals, non-terminals$\}$. We say that w' directly yields w'' if $\exists$ words u and v over the alphabet V and a production rule <T>$\rightarrow$ w of the grammar s.t. $w' = u$ <T> v and $w'' = uwv$, where either or both of the words u and v may be the empty word.
In other words, w' directly yields $w'' \Leftrightarrow \exists$ production rule <T>$\rightarrow$ w in the grammar s.t. w" may be obtained from w' by replacing a single occurrence of the nonterminal <T> within the word w' by the word w.

Notation: w' directly yields w'' is denoted by $w' \Rightarrow w''$

Definition: Let w' and w'' be words over the alphabet V. We say that w' yields w'' if either $w' = w''$ or else $\exists$ words $w_0, w_1, ...w_n$ over the alphabet V s.t. $w_0 = w', w_n = w'', w_{i-1} \Rightarrow w_i$ for all $i, 1 \leq i \leq n$. In other words, $w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow ... \Rightarrow w_{n-1} \Rightarrow w_n$

Notation: w' yields w'' is denotes by $w' \overset{*}{\Rightarrow} w''$.

Definition: Let $(V, A, <s>, P)$ be a context-free grammar. The language generated by this grammar is the subset L or A^* defined by $L = \{w \in A^* \mid <s> \overset{*}{\Rightarrow} w\}$

In other words, the language L generated by a context-free grammar $(V, A, <s>, P)$ consists of the set of all finite strings consisting entirely of terminals that may be obtained from the start symbol $<s>$ by applying a finite sequence of production rules of the grammar, where the application of one production rule causes one and only one nonterminal to be replaced by the string in V^* corresponding to the right-hand side of the production rule.

8.1 Phrase Structure Grammars

Definition: A phrase structure grammar $(V, A <s>, P)$ consists of a finite set V, a subset A of V, an element $<s>$ of $V \backslash A$, and a finite subset P of $(V^* \backslash A^*) \times V^*$

In a context-free grammar, the set of production rules $P \subset (V \backslash A) \times V^*$. In a phrase structure grammar, $P \subset (V^* \backslash A^*) \times V^*$. In other words, a production rule in a phrase structure grammar $r \to w$ has a left-hand side r that may contain more than one nonterminal. It is required to contain at least one nonterminal.

For example, if $A = \{0, 1\}$ and $<s>$ is the start symbol in a phrase structure grammar grammar, $0<s>0<s>0 \to 00010$ would be an acceptable production rule in a phrase structure grammar but not in a context-free grammar.

The notions $w' \Rightarrow w''$ (w' directly yields w'') and $w' \overset{*}{\Rightarrow} w''$ (w' yields w'') are defined the same way as for context-free grammars except that our production rules may, of course, be more general as we saw in the example above.

Definition: Let $(V, A <s>, P)$ be a phrase structure grammar. The language generated by this grammar is the subset L or A^* defined by $L = \{w \in A^* \mid <s> \overset{*}{\Rightarrow} w\}$

Remark: The term phrase structure grammars was introduced by Noam Chomsky.

Definition: A language L generated by a context-free grammar is called a context-free language.

We now want to understand a particularly important subclass of context-free languages called regular languages.

8.2 Regular Languages

Task: Understand when a language is regular and how regular languages are produced. Understand basics of automata theory.

History: The term regular language was introduced by Stephen Kleene in 1951. A more descriptive name is finite-state language as we will see that a language is regular $\Leftrightarrow$ it can be recognised by a finite state acceptor, which is a type of finite state machine.
The definition of a regular language is very abstract, though. First, describe what operations the collection of regular languages is closed under:

Let A be a finite set, and let A^* be the set of all words over the alphabet A. The regular languages over the alphabet A constitute the smallest collection C of subsets of A^* satisfying that:

1. All finite subsets of A^* belong to C.
2. C is closed under the Kleene star operation (if $M \subseteq A^*$ is inside C, **i.e.** $M \in C$, then $M^* \in C$)
3. C is closed under concatenation (if $M \subseteq A^*, N \subseteq A^*$ satisfy that $M \in C$ and $N \in C$, then $M \circ N \in C$)
4. C is closed under union (if $M \subseteq A^*$ and $N \subseteq A^*$ satisfy that $M \in C$ and $N \in C$, then $M \cup N \in C$)

Definition: Let A be a finite set, and let A^* be the set of words over the alphabet A. A subset L of A^* is called a regular language over the alphabet A if $L = L_m$ for some finite sequence $L_1, L_2, ..., L_m$ of subsets of A^* with the property that $\forall i, 1 \leq i \leq m, L_i$ satisfies one of the following:

1. L_i is a finite set
2. $L_i = L_j^*$ for some $j, 1 \leq j < i$ (the Kleene star operation applied to one of the previous $L_j's$)
3. $L_i = L_j \circ L_k$ for some j, k such that $1 \leq j, k < i$ (L_i is a concatenation of previous $L_j's$)
4. $L_i = L_j \cup L_k$ for some j, k such that $1 \leq k, j < i$ (L_i is a union of previous $L_j's$)

Example 1: Let $A = \{0, 1\}$. Let $L = \{0^m 1^n \mid m, n \in \mathbb{N} \quad m \geq 0, n \geq 0\}$
L is a regular language. Note that L consists of all strings of first 0's, then 1's or the empty string ε. $0^m 1^n$ stands for m 0's followed by n 1's, **i.e.** $0^m \circ 1^n$. Let us examine $L' = \{0^m \mid m \in \mathbb{N}, m \geq 0\}$ and $L'' = \{1^n \mid n \in \mathbb{N}, n \geq 0\}$

Q: Can we obtain them via operatons listed among 1-4?

A: Yes! Let $M = \{0\} \qquad M \subseteq A \subseteq A^*$ and $M^* = L' = \{0^m \mid m \in \mathbb{N} \quad m \geq 0\}$. Let $N = \{1\} \quad N \subseteq A \subseteq A^*$ and $N^* = L'' = \{1^n \mid n \in \mathbb{N}, n \geq 0\}$. In other words, we can do $L_1 = \{0\}, L_2 = \{1\}, L_3 = L_1^*, L_4 = L_2^*, L_5 = L_3 \circ L_4 = L$. Therefore, L is a regular language.

Example 2 Let $A = \{0, 1\}$. Let $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$. L is the language we used as an example earlier. It turns out L is NOT regular. This language consists of strings of 0's followed by an equal number of strings of 1's. For a machine to decide that the string $0^m 1^m$ is inside the language, it must store the number of 1's, as it examines the number of 0's or vice versa. The number of strings of the type $0^m 1^m$ is not finite, however, so a finite-state machine cannot recognise this language. Heuristically, regular languages correspond to problems that can be solved with finite memory, **i.e.** we only need to remember one of finitely many things. By contrast, nonregular languages correspond to problems that cannot be solved with finite memory.

Theorem: The collection of regular languages L is also closed under the following two operations:

1. Intersection, **i.e.** if L', L'' are regular languages (**i.e.** $L' \in C$ and $L'' \in C$), then their intersection $L' \cap L''$ is a regular language.
2. Complement, **i.e.** if L is a regular language (**i.e.** $L \in C$), then $A^* \backslash L$ is a regular language ($A^* \backslash L \in C$).

Remark: These two properties did not come into the definition of a regular language, but they are true and often quite useful.

8.3 Finite State Acceptors and Automata Theory

Definition: An automaton is a mathematical model of a computing device. Plural of automaton is automata.

Basic idea: Reason about computability without having to worry about the complexity of actual implementation.
It is most reasonable to consider at the beginning just finite states automata, **i.e.** machines with a finite number of internal states. The data is entered discretely, and each datum causes the machine to either remain in the same internal state or else make the transition to some other state determined solely by 2 pieces of information:

1. The current state
2. The input datum

In other words, if S is the finite set of all possible states of our finite state machine, then the transition mapping t that tells us how the internal state of the machine changes on inputting a datum will depend on the current state $s \in S$ and the input datum a, **i.e.** the machine will enter a (potentially) new state $s' = t(s, a)$.

Want to use finite state machines to recognise languages over some alphabet A. Let L be our language.

Input	Output
Word $w = a_1...a_n, a_i \in A\,\forall i$	Yes if $w \in L$
	No if $w \notin L$

Since our finite state machine accepts (**i.e.** returns yes to) w if $w \in L$, we call our machine a finite state acceptor. We want to give a rigorous definition of a finite state acceptor. To check $w = a_1...a_n$, we input each a_i starting with a_1 and trace how the internal state of the machine changes. S is our set of states of the machine (a finite set). The transition mapping t takes the pair(s, a) and returns the new state $s' = t(s, a)$ (where $s \in S$ and $a \in A$) that the machine has reached so $t : S \times A \to S$.
Some elements and subsets of S are important to understand:

1. The initial state $i \in S$ where the machine starts
2. The subset $F \subseteq S$ of finishing states

It turns out that knowing S, F, i, t, A specifies a finite state acceptor completely.

Definition: A finite state acceptor (S, A, i, t, F) consists of a finite set S of states, a finite set A that is the input alphabet, a starting state $i \in S$, a transition mapping $t : S \times A \to S$, and a set F of finishing states, where $F \subseteq S$.

Definition: Let (S, A, i, t, F) be a finite state acceptor, and let A^* denote the set of words over the input alphabet A. A word $a_1...a_n$ of length n over the alphabet A is said to be recognised or accepted by the finite state acceptor if $\exists s_0, s_1, ..., s_n \in S$ states s.t. $s_0 = i$ (the initial state), $s_n \in F$, and $s_i = t(s_{i-1}, a_i)\,\forall i \quad 1 \leq i \leq n$.

Definition: Let (S, A, i, t, F) be a finite state acceptor. A language L over the alphabet A is said to be recognised or accepted by the finite state acceptor if L is the set consisting of all words recognized by the finite state acceptor.

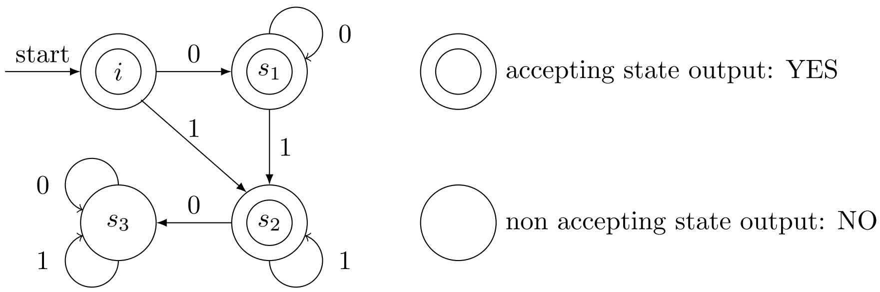
In the definition of a finite state acceptor, t is the transition mapping, which may or may not be a function (hence the careful terminology). This is because finite state acceptors come in 2 flavours:

1. Deterministic: every state has exactly one transition for each possible input, **i.e.** $\forall (s, a) \in S \times A\ \exists!\ t(s, a) \in S$. In other words, the transition mapping is a function.
2. Non-deterministic: an input can lead to one, more than one or no transition for a given state. Some $(s, a) \in S \times A$ might be assigned to more than one element of S, **i.e.** the transition mapping is not a function.

Surprisingly $\exists$ algorithm that transforms a non-deterministic (though more complex one) into a deterministic one using the power set construction. As a result, we have the following theorem:

Theorem: A language L over some alphabet A is a regular language $\Leftrightarrow$ L is recognised by a deterministic finite state acceptor with input alphabet A $\Leftrightarrow$ L is recognised by a non-deterministic finite state acceptor with input alphabet A.

Example: Build a deterministic finite state acceptor for the regular language $L = \{0^m 1^n \mid m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$



Accepting states in this examples: i, s_1, s_2
Non accepting states: s_3
Start state: i
Here $S = \{i, s_1, s_2, s_3\}$ $\quad F = \{i, s_1, s_2\}$ $\quad A = \{0, 1\}$ $\quad t : S \times A \to S$ $\quad t(i, 0) = s_1$ $\quad t(i, 1) = s_2$ $\quad t(s_1, 0) = s_1$ $\quad t(s_1, 1) = s_2$ $\quad t(s_2, 0) = s_3$ $\quad t(s_2, 1) = s_2$ $\quad t(s_3, 0) = s_3$ $\quad t(s_3, 1) = s_3$
Let's process some strings:

String	ε (empty string)
State (i)	i
Output	YES

String	0	0	1	1	1
State i	s_1	s_1	s_2	s_2	s_2
Output	YES				

String	1	1
State i	s_2	s_2
Output	YES	

String	1
State i	s_2
Output	YES

String	0	1	0	1
State i	s_1	s_2	s_3	s_3
Output	NO			

Now that we really understand what a finite state acceptor is, we can develop a criterion for recognising regular languages called the Myhill-Nerode theorem based on an equivalence relation we can set up on words in our language over the alphabet A.

Definition: Let $x, y \in L$, a language over the alphabet A. We call x and y equivalent over L denoted by $x \equiv_L y$ if $\forall w \in A^*, xw \in L \Leftrightarrow yw \in L$.

Note: xw means the concatenation $x \circ w$, and yw is the concatenation $y \circ w$.

Idea: If $x \equiv_L y$, then x and y place our finite state acceptor into the same state s.

Notation: Let $L/\equiv$ be the set of equivalence classes determined by the equivalence relation $\equiv_L$.

The Myhill-Nerode Theorem: Let L be a language over the alphabet A. If the set $L/\equiv$ of equivalence classes in L is infinite, then L is not a regular language.

Stretch of Proof: All elements of one equivalence class in $L/\equiv$ place our automaton into the same state s. Elements of distinct equivalence classes place the automaton into distinct states, **i.e.** if $[x], [y] \in L/\equiv$ and $[x] \neq [y]$, then all elements of $[x]$ place the automaton into some state s, while all elements of $[y]$ place the automaton into some state s', with $s \neq s' \Rightarrow$ an automaton that can recognise L has as many states at the number of equivalence classes in $L/\equiv$, but $L/\equiv$ is NOT finite $\Rightarrow L$ cannot be recognised by a finite state automaton $\Rightarrow L$ is not regular by the theorem above.

qed

8.4 Regular Grammars

Task: Understand what is the form of the production rules of a grammar that generates a regular language.

Recall: that a context-free grammar is given by $(V, A, <s>, P)$ where every production rule <T>$\to$ w in P causes one and only one nonterminal to be replaced by a string in V^*.

Definition: A context-free grammar $(V, A, <s>, P)$ is called a regular grammar if every production rule in P is of one of the three forms:

(i) <A>$\to$ b<B>

(ii) <A>$\to$ b

(iii) <A>$\to \varepsilon$

where <A> and <B> are nonterminals, b is a terminal, and ε is the empty word. A regular grammar is said to be in normal form if all its production rules are of types (i) and (iii).

Remark: In the literature, you often see this definition labelled left-regular grammar as opposed to right-regular grammar, where the production rules of type (i) have the form <A>→<B>b, (**i.e.** the terminal is one the right of the nonterminal). This distinction is not really important as long as we stick to one type throughout since both left-regular grammars and right-regular grammars generate regular languages.

Lemma: Any language generated by a regular grammar may be generated by a regular grammar in normal form.

Proof: Let <A>→b be a rule of type (ii). Replace it by two rules: <A>→b<F> and <F>→ ε, where <F> is a new nonterminal. Add <F> to the set V. We do the same for every rule of type (ii) obtaining a bigger set V, but now our production rules are only of type (i) and (iii) and we are generating the same language.

qed

Example: Recall the regular language $L = \{0^m 1^n \mid m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$. We can generate it from the regular grammar in normal form given by production rules:

1. <s>→ 0<A>
2. <A>→ 0<A>
3. <A>→ ε
4. <s>→ ε
5. <A>→ 1<B>
6. <B>→ 1<B>
7. <s>→ 1 <B>
8. <B>→ ε

Rules (1), (2), (5), (6), (7) are of type (i), while rules (3), (4) and (8) are of type (iii).
(1) and (3) give 0. (1), (2) applied $m-1$ times and (3) give 0^m for $m \geq 2$.
(7) and (8) give 1. (7), (6) applied $n-1$ times and (8) give 1^n for $n \geq 2$.
(1), (5) and (8) give 01. (1), (5), (6) applied $n-1$ times and (8) give 01^n for $n \geq 2$.
(1), (2) applied $m-1$ times, (5) and (8) give $0^m 1$ for $m \geq 2$.
(1), (2) applied $m-1$ times, (5), (6) applied $n-1$ times, and (8) give $0^m 1^n$ for $m \geq 2, n \geq 2$.
Rule (4) gives the empty word $\varepsilon = 0^0 1^0$.

Q: Why does a regular grammar yield a regular language, **i.e.** one recognised by a finite state acceptor?

A: Not obvious from the definition, but we can construct the finite state acceptor from the regular grammar as follows: our regular grammar is given by $(V, A, <s>, P)$. Want a finite state acceptor (S, A, i, t, F). Immediately, we see the alphabet A is the same and $i =$<s>. This gives us the idea of associating to every nonterminal symbol in $V\backslash A$ a state. <s>$\in V\backslash A$, so that's good. Next we ask:

Q: Is it sufficient for $S = V\backslash A$?

A: No! Our set F of finishing/accepting states should be nonempty. So we add an element $\{f\}$ to $V\backslash A$, where our acceptor will end up when a word in our language. Thus, $S = (V\backslash A) \cup \{f\}$ and $F = \{f\}$. $F \subseteq S$ as needed.

Q: How do we define t?

A: Use the production rules in P! For every rule of type (i), which is of the form <A>$\rightarrow$ b<B> set t(<A>,b) =<B>. This works out well because our nonterminals <A> and <B> are states of the acceptor and the terminal $b \in A$ so t takes an element of $S \times A$ to an element of S as needed. Now look at production rules of type (ii), <A>$\rightarrow b$ and of type (iii), <A>$\rightarrow \varepsilon$. Those are applied when we finish constructing a word w in our language L, **i.e.** at the very last step, so our acceptor should end up in the finishing state f whenever a production rule of type (ii) or (iii) is applied. Write a production rule of type (ii) or (iii) as <A>$\rightarrow w$, then we can set $t($<A>$, w) = f$. We have finished constructing t as well. Technically, $t : S \times (A \cup \{\varepsilon\}) \rightarrow S$ instead of $t : S \times A \rightarrow S$, but we can easily fix the transition function t by combining the last two transitions for each accepted word.

Remark: The same general principles as we used above allow us to go from a finite state acceptor to a regular grammar. This gives us the following theorem:

Theorem: A language L is regular $\Leftrightarrow$ L is recognised by a finite state acceptor $\Leftrightarrow$ L is generated by a regular grammar.

8.5 Regular expressions

Task: Understand another equivalent way of characterizing regular languages due to Kleene in the 1950's.

Definition: Let A be an alphabet.

1. $\emptyset$, ϵ, and all elements of A are regular expressions;
2. If w and w' are regular expressions, then $w \circ w'$, $w \cup w'$, and w^* are regular expressions.

Remark: This definition is an inductive one.

NB It is important not to confuse the regular expressions $\emptyset$ and ϵ. The expression ϵ represents the language consisting of a single string, namely ϵ, the empty string, whereas $\emptyset$ represents the language that does not contain any strings. Recall that a language L is any subset of

$$A^* = \bigcup_{n=0}^{\infty} A^n = A^0 \cup A^1 \cup A^2 \cup \cdots ,$$

where $A^0 = \{\epsilon\}$, the set of words of length 0, $A^1 =$ the set of words of length 1, and $A^2 =$ the set of words of length 2.

Precedence order of operations if parentheses are not present:

First $*$, then $\circ$ (concatenation), then $\cup$ (union).

Examples: (1) $A = \{0, 1\}$

$$\begin{aligned} 1^* \circ 0 &= \{w \in A^* \mid w = 1^m 0 \text{ for } m \in \mathbb{N}, m \geq 0\} = \{0, 10, 110, 1110, \dots\} \\ &= 1^*0. \end{aligned}$$

We can omit the concatenation symbol.

(2) $A = \{0, 1\}$

$$\begin{aligned} A^* \circ 1 \circ A^* &= \{w \in A^* \mid w \text{ contains at least one } 1\} \\ &= \{u \circ 1 \circ v \mid u, v \in A^*\} = A^* 1 A^* \end{aligned}$$

(3) $A = \{0, 1\}$

$$(A \circ A)^* = \{w \in A^* \mid w \text{ is a word of even length}\}.$$

Recall that $L^* = \bigcup_{n=0}^{\infty} L^n$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and inductively $L^n = L \circ L^{n-1}$. Here $L = \{00, 01, 10, 11\}$.

(3') $(A^* \circ A^*)^* = A^*$.

(4) $A = \{0, 1\}$ $\quad (0 \cup \epsilon) \circ (1 \cup \epsilon) = \{\epsilon, 0, 1, 01\}$.

(5) $\epsilon^* = \{\epsilon\}$.

(6) $\emptyset^* = \{\epsilon\}$. The star operation concatenates any number of words from the language. If the language is empty, then the star operation can only put together 0 words, which yields only the empty word.

Use of regular expressions in programming:

$\rightarrow$ design of compilers for programming languages

Elemental objects in a programming language, which are called tokens (for example variables names and constants) can be described with regular expressions. We get the syntax of a programming language this way. There exists an algorithm for recognizing regular expressions that has been implemented $\Longrightarrow$ an automatic system generates the lexical analyzer that checks the input in a compiler.

$\rightarrow$ eliminate redundancy in programming

The same regular expression can be generated in more than one way (obvious from the definition of a regular expression) $\Longrightarrow$ there exists an equivalence relation on regular expressions and algorithms that check when two regular expressions are equivalent.

Theoretical importance of regular expressions

For the study of formal languages and grammars, the importance of regular expressions comes from the following theorem:

Theorem: A language is regular $\Longleftrightarrow$ some regular expression describes it.

Sketch of proof: Recall the definition of a regular language as the language obtained in finitely many steps from finite subsets of words via union, concatenation or the Kleene star. We can construct a regular expression from the definition of the regular language in question, and vice versa starting with a regular expression, we can define a finite sequence of L_i's such that each L_i is a finite set of words or is obtained from previous L_i's via union, concatenation or the Kleene star.

qed

Finally, we can state the complete characterization of regular languages:

Theorem: The following are equivalent:

(i) L is a regular language.

(ii) L is recognized by a (deterministic or non-deterministic) finite state acceptor.

(iii) L is produced by a regular grammar.

(iv) L is given by a regular expression.

Remark: It is possible to prove directly that (iv) $\Longleftrightarrow$ (ii), but the construction is rather complicated. Instead, we sketched above the proof that (i) $\Longleftrightarrow$ (iv), and we had previously stated that (i) $\Longleftrightarrow$ (ii) $\Longleftrightarrow$ (iii), so we now have that (i) $\Longleftrightarrow$ (ii) $\Longleftrightarrow$ (iii) $\Longleftrightarrow$ (iv).

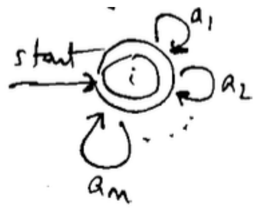
Example: Let $L = \{0^m 1^n \;\big|\; m, n \in \mathbb{N}, m \geq 0, n \geq 0\}$ be the regular language we considered before. We now give a regular expression for L: $L = 0^* \circ 1^*$. Recall we previously show this language is regular from the definition of a regular language, so solving this problem is a direct illustration of the implication (i) $\Longleftrightarrow$ (iv).

8.6 The Pumping Lemma

Task: Understand another criterion for figuring out when a language is regular.

Let a finite set A be the alphabet, and let L be a language over A. Then $L \subset A^*$. We make the following two crucial observations:

1. If L is finite, then clearly there exists a finite state acceptor that recognizes $L \Rightarrow L$ is regular.
2. If $L = A^*$, then L is likewise regular. Here is why: Let $A = \{a_1, \ldots, a_n\}$. The acceptor



with just one state i recognizes A^*.

Question: If L is infinite, but $L \subsetneqq A^*$, how can we tell whether L is regular?

Answer: The Myhill-Nerode Theorem would have us look at equivalence classes of words, but that analysis can be complicated at times. The Pumping Lemma provides another way of checking whether L is regular.

The Pumping Lemma: If L is a regular language, then there is a number p (the pumping length) where if w is any word in L of length at least p, then $w = xuy$ for words x, y, and u satisfying:

1. $u \neq \epsilon$ (i.e., $|u| > 0$, the length of u is positive);
2. $|xu| \leq p$;
3. $xu^n y \in L \; \forall \, n \geq 0$.

Remark: p can be taken to equal the number of states of a deterministic finite state acceptor that recognizes L (we know such a finite state acceptor exists because L is regular).

Sketch of proof: The name of the lemma comes from the fact that if L is regular, then all of its words can be pumped through a finite state acceptor that recognizes L. We assume this acceptor is deterministic and has p states. We will show the Pumping Lemma is a consequence of the Pigeonhole Principle we studied in the unit on functions. If a word w has length l, then the finite state acceptor must process l pieces of information ($w = a_1 a_2 \cdots a_l$, where $a_k \in A \; \forall \, k, \; 1 \leq k \leq l$) $\implies$ it passes through $l+1$ states starting with the initial state. In the hypotheses of the lemma, we assume $|w| = l \geq p$, but $p = \#(\text{states of the acceptor}) \implies$ the acceptor passes through $l + 1 \geq p + 1$ states to process w and therefore at least one state is repeated among the first $p+1$. Let s_1, s_2, ..., s_{l+1} be the sequence of states. $|w| = l \geq p \implies s_i = s_j$ with $i < j \leq p+1$. Now we set x to be the part of w that makes the acceptor pass through states s_1, s_2, ..., s_i, i.e., $x = a_1 a_2 \cdots a_{i-1}$ (the first $i-1$ letters in w). We set u to be the part of w that makes the acceptor pass through states s_i, s_{i+1}, s_{i+2}, ..., s_j. In other words, $u = a_i a_{i+1} \cdots a_{j-1}$. Since $i < j$, $|u| \geq 1$ $\implies u \neq \epsilon$. Finally, set y to be the part of w (the tail end) that makes the acceptor pass through states s_j, s_{j+1}, ..., s_{l+1}, i.e., $y = a_j a_{j+1} \cdots a_l$. Since $j \leq p+1$, $j-1 \leq p$, so $|xu| = |a_1 a_2 \cdots a_{j-1}| = j-1 \leq p$ as needed. Furthermore, $s_i = s_j$, so at the beginning of u and at its end the acceptor is in the same state $s_i = s_j \implies xu^n y$ is accepted for every $n \geq 0 \implies$ $xu^n y \in L$ as needed. We have obtained conditions (1)-(3).

qed

Applications of the Pumping Lemma

As a statement, the Pumping Lemma is the implication $P \to Q$ with P being the sentence "L is a regular language" and Q being the decomposition of every w, $|w| \geq p$ as $w = xuy$. We use the contrapositive $\neg Q \to \neg P$ (tautologically equivalent to $P \to Q$) as our criterion for detecting non-regular languages.

Examples:

1. $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 0\}$ is not regular. Let $w = 0^m 1^m$. We cannot decompose w as $w = xuy$ because whatever we let u be, we get a contradiction to $xu^n y \in L \; \forall n \geq 0$. If $u \in 0^*$ (string of 0's), $x \in 0^*$ and $y = 0^s 1^m$ (string of s 0's with $s \geq 0$ and m 1's). If $n \geq 2$, $xu^n y \notin L$ because $xu^n y$ has more 0's than 1's.
   If $u \in 1^*$, we get a contradiction the same way (more 1's than 0's in this case).
   If $u \in 0^* 1^*$, $xu^2 y \notin L$ for any x, y words!
2. $L = \{0^m \mid m \text{ is prime}\}$ is not regular.
   Since $w = 0^m$, x, u, y can consist only of 0's, so then $x = 0^i$, $u = 0^j$, $y = 0^k$. If $xu^n y \in L \; \forall n \geq 0$, then $i + nj + k$ is prime $\forall n \geq 0$, which is impossible.
   Set $n = i + 2j + k + 2$, then
   $$\begin{aligned} i + nj + k &= i + (i + 2j + k + 2)j + k = i + ij + 2j^2 + jk + 2j + k \\ &= i(j+1) + 2j(j+1) + k(j+1) = (j+1)(i + 2j + k), \end{aligned}$$
   where $|u| > 0$, so $j \geq 1$. Therefore, $n = (j+1)(i+2j+k)$ is not prime!

Practice at home: weitz.de/pump (on Edi Weitz's website)

The pumping game, an online game to help you understand the Pumping Lemma.

8.7 Applications of Formal Languages and Grammars as well as Automata Theory

1. Compiler architecture uses context-free grammars
2. Parsers - recognise if commands comply with the syntax of a language
3. Pattern matching and data mining - guess the language from a given set of words (applied in CS, genetics, etc.)
4. Natural language processing - example in David Wilkins' notes pp.40-44
5. Checking proofs by computers/automatic theorem proving - simpler example of this kind in David Wilkins' notes pp.45-57 that pertains to propositional logic
6. The theory of regular expressions enables
   (a) grep/awk/sed in Unix
   (b) More efficient coding (avoiding unnecessary detours in your code)
7. Biology - John Conway's game of life is a cellular automaton
8. Modelling of AI characters in games uses the finite state automaton idea. Our character can choose among different behaviours based on stimuli - like a finite state automaton reacting to input
9. Strategy and tactics in games - teach the opposition to recognise certain patterns, then suddenly change them to gain an advantage and score - used in football, fencing, etc.
10. Learning a sport/a numerical instrument/a new field or subject - split the information into blocks and learn how to combine them into meaningful patterns - uses notions from context-sensitive grammars.
11. Finite state automata and probability $\rightsquigarrow$ Markov chains - chaos theory, financial mathematics.

etc...

9 Graph Theory

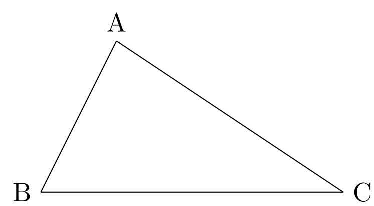
Task: Introduce terminology related to graphs; understand different types of graphs; learn how to put together arguments involving graphs.
An undirected graph consists of:

1. A finite set of points V called vertices
2. A finite set E of edges joining two distinct vertices of the graph.

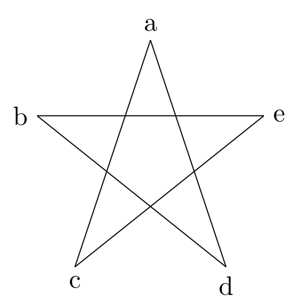
Understand the meaning of an edge better: Let V be the set of vertices. Consider $P(V)$, the power set of V. Let $V_2 \subseteq P(V)$ consist of all subsets of V containing exactly 2 points, **i.e.** $V_2 = \{A \in P(V) \mid \#(A) = 2\}$ Identify each element in V_2 with the edge joining the two points. In other words, if $\{a, b\} \in V_2$, then we can let ab be the edge corresponding to $\{a, b\}$.

Examples:

1. A triangle is an undirected graph.
   $V = \{A, B, C\}$

   3 possible 2 element subsets of V: $\{A, B\} \to AB$
   $\{A, C\} \to AC$
   $\{B, C\} \to BC$
   $E = \{AB, AC, BC\}$
2. A pentagram is an example of an undirected graph.
   $V = \{a, b, c, d, e\}$

   $E = \{ac, ad, be, ce, bd\}$

Convention: The set of vertices cannot be empty, **i.e.** $V \neq \emptyset$.

Q: If $V \neq \emptyset$, what is the simplest possible undirected graph?

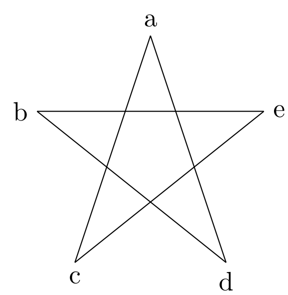
A: A graph consisting of a single point, **i.e.** with one vertex and zero edges.

Definition: A graph is called <u>trivial</u> if it consists of one vertex and zero edges. Next, study how vertices and edges relate to each other.

Definition: If v is a vertex of some graph, if e is an edge of that graph, and it $e = vv'$ for v' another vertex, then the vertex v is called incident to the edge e and the edge e is called incident to the vertex v.

Example:

b is incident to edges be and bd
be is incident to vertices b and e



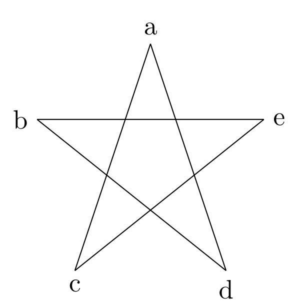
Definition: Let (V, E) be an undirected graph. Two vertices $A, B \in V$ $A \neq B$ are called adjacent if $\exists$ edge $AB \in E$.
We represent the incidence relations among the vertices V and edges E of an undirected graph via:

1. An incidence table
2. An incidence matrix

Legend:

1 an incidence relation holds
0 an incidence relation does not hold

From the pentagram:
$V = \{a, b, c, d, e\}$
$E = \{ac, ad, be, bd, ce\}$



The incidence table is:

	ac	ad	be	bd	ce
a	1	1	0	0	0
b	0	0	1	1	0
c	1	0	0	0	1
d	0	1	0	1	0
e	0	0	1	0	1

Correspondingly, the incidence matrix is:

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

Note that for the incidence matrix to make sense, we need to know that vertices were considered in the order $\{a, b, c, d, e\}$ and edges in the order $\{ac, ad, be, bd, ce\}$. If we shuffle either set, the incidence matrix changes. With this in mind, we can now rigorously define the incidence matrix:

Definition: Let (V, E) be an undirected graph with m vertices and n edges. Let vertices be ordered as $v_1, v_2, ..., v_m$, and let the edges be ordered $e_1, e_2, ..., e_n$. The incidence matrix for such a graph is given by $\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & ... & a_{mn} \end{pmatrix}$, where the entry a_{ij} in row i and column j has the value 1 if the i^{th} vertex is incident to the j^{th} edge and has value 0 otherwise.
Similarly, we can define the adjacency table and the adjacency matrix of a graph:

Definition: Let (V, E) be an undirected graph with m vertices, and let these vertices be ordered as $v_1, v_2, ..., v_m$. The adjacency matrix for this graph is given by $\begin{pmatrix} b_{11} & b_{12} & ... & b_{1m} \\ b_{21} & b_{22} & ... & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & ... & b_{mm} \end{pmatrix}$ where $b_{ij} = 1$ if v_i and v_j are adjacent to each other and $b_{ij} = 0$ if v_i and v_j are not adjacent to each other.

Remark: "Being adjacent to" is a symmetric relation on the set of vertices V, so the adjacency matrix is symmetric, **i.e.** $b_{ij} = b_{ji} \quad \forall i, j \quad 1 \leq i, j \leq m$. It is not reflexive so all the entries on the diagonal are zero.

9.1 Complete graphs

Definition: A graph (V, E) is called complete if $\forall v, v' \in V$ s.t. $v \neq v'$, the edge $vv' \in E$. In other words, a complete graph has the highest number of edges possible given its number of vertices.

Examples:

1. The triangle is a complete graph.
2. The pentagram is not a complete graph.

Notation: A complete graph with n vertices is denoted by K_n.

Q: How does the adjacency matrix of a complete graph look like?

A: All entries are 1 except on the diagonal, where they are all zero.

9.2 Bipartite graphs

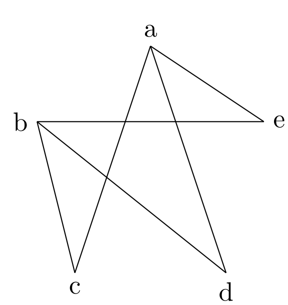
Definition: A graph (V, E) is called bipartite is $\exists$ subsets V_1 and V_2 s.t.

1. $V_1 \cup V_2 = V$
2. $V_1 \cap V_2 = \emptyset$
3. Every edge in E is of the form vw with $v \in V_1$ and $w \in V_2$.

A bipartite graph is called a complete bipartite graph if $\forall v \in V_1$
$\forall w \in V_2 \quad \exists vw \in E$.

Notation: A complete bipartite graph where the set V_1 has p elements and the set V_2 has q elements is denoted by $K_{p,q}$.

Example:
$V_1 = \{a, b\}$
$V_2 = \{c, d, e\}$
$V = \{a, b, c, d, e\}$
$E = \{ac, ad, ae, bc, bd, be\}$
is a complete bipartite graph.



Next, relate graphs to each other via functions with special properties.

9.3 Isomorphisms of Graphs

Definition: An isomorphism between two graphs (V, E) and (V', E') is a bijective function $\varphi : V \to V'$ satisfying that $\forall a, b \in V$ with $a \neq b$ the edge $ab \in E \Leftrightarrow$ the edge $\varphi(a)\varphi(b) \in E'$.

Recall: A function $\varphi : V \to V'$ is bijective $\Leftrightarrow$ it has an inverse $\varphi^{-1} : V' \to V$. The bijection $\varphi : V \to V'$ that gives the isomorphism between (V, E) and (V', E') thus sets up the following:

1. A 1-1 correspondence of the vertices V of (V, E) with the vertices V' of $(V', E') \rightsquigarrow$ comes from $\varphi : V \to V'$ being bijective.
2. A 1-1 correspondence of the edges E of (V, E) with the edges E' of $(V', E') \rightsquigarrow$ comes from the additional property in the definition of an isomorphism that $\forall a, b \in V$ with $a \neq b, ab \in E \Leftrightarrow \varphi(a)\varphi(b) \in E'$.

Definition: If there exists an isomorphism $\varphi : V \to V'$ between two graphs (V, E) and (V', E'), then (V, E) and (V', E') are called isomorphic.

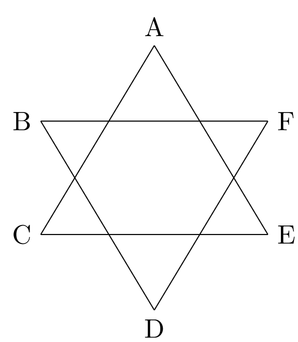
Remark: Just like an isomorphism of groups discussed earlier in the course, an isomorphism of graphs means (V, E) and (V', E') have the same "iso" form "morphē". "Being isomorphic" is an equivalence relation, so we get classes of graphs that have the same form as our equivalence classes.

9.4 Subgraphs

Task: Understand sub-objects of a graph.

Definition: Let (V, E) and (V', E') be graphs. The graph (V', E') is called a subgraph of (V, E) if $V' \subseteq V$ and $E' \subseteq E$, **i.e.** if (V', E') consists of a subset V' of the vertices of (V, E) and a subset E' of edges (V, E) between vertices in V'.

Example: Star of David on the flag of Israel
$V = \{a, b, c, d, e, f\}$
$E = \{ac, ce, ae, bf, fd, bd\}$



2 triangle subgraphs of the star of David:

$V' = \{a, c, e\}$ $E' = \{ac, ce, ae\}$

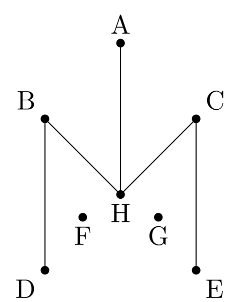
$V" = \{b, f, d\}$ $E" = \{bf, fd, bd\}$

9.5 Vertex Degrees

Task: Use numbers to understand incidence relationships.

Definition: Let (V, E) be a graph. The degree deg v of a vertex $v \in V$ is defined as the number of edges of the graph that are incident to v, **i.e.** the number of edges with v as one of their endpoints.

Example:



def f = deg $g = 0$
deg d = deg e = deg $a = 1$
deg b = deg $c = 2$
deg $h = 3$

Definition: A vertex of degree 0 is called an isolated vertex.

Definition: A vertex of degree 1 is called a pendant vertex.

Theorem: Let (V, E) be a graph. Then $\sum_{v \in V}$deg $v = 2\#(E)$, where $\sum_{v \in V}$deg v is the sum of the degrees of all the vertices of the graph, and $\#(E)$ is the number of edges of the graph.

Proof: $\sum_{v \in V}$deg v is the sum of all the entries in the adjacency matrix. Every edge $vv' \in E$ contributes 2 to the sum $\sum_{v \in V}$deg v, 1 for the vertex v and 1 for the vertex $v' \Rightarrow$ each edge must be counted twice, so $\sum_{v \in V}$deg $v = 2\#(E)$.

qed

Corollary: $\sum_{v \in V}$deg v is an even integer.

Proof: Since $\sum_{v \in V} \deg\ v = 2\#(E)$, and $\#(E) \in \mathbb{N}$, the result follows.

qed

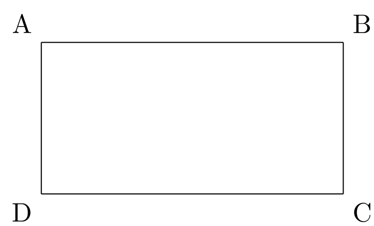
Corollary: In any graph, the number of vertices of odd degrees must be even.

Proof: Assume not, then $\sum_{v \in V} \deg\ v$ is an odd integer as $odd + even = odd \Rightarrow\Leftarrow$ to the previous corollary.

qed

Definition: A graph is called k-regular for some non-negative integer k if every vertex of the graph has degree equal to k.

Example: A rectangle is 2-regular.
$\deg\ a = \deg\ b = \deg\ c = \deg\ d = 2$.



Definition: A graph (V, E) is called regular is $\exists k \in \mathbb{N}$ s.t. (V, E) is k-regular.

Corollary: Let (V, E) be a k-regular graph. Then $k\#(V) = 2\#(E)$ where $\#(V)$ is the number of vertices and $\#(E)$ is the number of edges.

Proof: By the theorem, $\sum_{v \in V} \deg\ v = 2\#(E)$, but (V, E) is k-regular $\Rightarrow$ deg $v = k\ \forall v \in V$. Therefore $\sum_{v \in V} \deg\ v = \#(V) \times k = 2\#(E)$.

qed

Example: Consider a complete graph (V, E) with n vertices. (V, E) is $(n-1)$-regular because every vertex is adjacent to all the remaining $(n-1)$ vertices.

Corollary: A complete bipartite graph $K_{p,q}$ is regular $\Leftrightarrow p = q$

Proof: Recall that $V = V_1 \cup V_2\ V_1 \cap V_2 = \emptyset$ for a bipartite graph, where $\#(V_1) = p$ and $\#(V_2) = q$.
"$\Leftarrow$" If $p = q, \forall v \in V_1$ satisfies that deg $v = p = q$ and $\forall v \in V_2$ satisfies that deg $v = p = q$ since the graph is complete $\Rightarrow K_{p,q}$ is p-regular.
"$\Rightarrow$" $K_{p,q}$ is regular $\Rightarrow \forall v \in V_1$ and $\forall v' \in V_2$, deg $v =$ deg v', but $K_{p,q}$ is complete $\Rightarrow v$ is adjacent to all vertices in V_2, **i.e.** deg $v = \#(V_2)$ and v' is adjacent to all vertices in V_1, **i.e.** deg $v' = \#(V_1)$. Therefore, $\#(V_1) = \#(V_2)$.

qed

9.6 Walks, trails and paths

Task: Make rigorous the notion of traversing parts of a graph in order to understand its structure better.

Definition: Let (V, E) be a graph. A walk $v_0v_1v_2...v_n$ of length n in the graph from vertex a to vertex b is determined by a finite sequence $v_0, v_1, v_2, ..., v_n$ of vertices of the graph s.t. $v_0 = a, v_n = b$ and $v_{i-1}v_i$ is an edge of the graph for $i = 1, 2, ..., n$.

Definition: A walk $v_0v_1v_2...v_n$ in a graph is said to traverse the edges $v_{i-1}v_i$ and to pass through the vertices $v_0, v_1, ..., v_n$. Length of walk = # of edges traversed $\Rightarrow$ the smallest possible number is zero edges. As a result, we have the following definition:

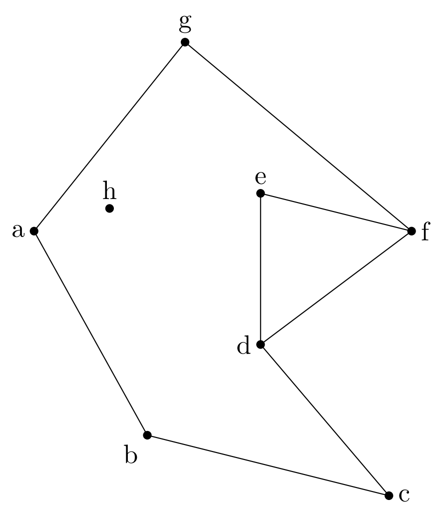
Definition: A walk that consists of a single vertex $v \in V$ and has length zero is called trivial.

Definition: Let (V, E) be a graph. A trail $v_0v_1v_2...v_n$ of length n in the graph from some vertex a to some vertex b is a walk of length n from a to b with the property that edges $v_{i-1}v_i$ are distinct for $i = 1, 2, ..., n$. In other words, a trail is a walk in the graph, which traverses edges of the graph at most once.

Definition: Let (V, E) be a graph. A path $v_0v_1v_2...v_n$ of length n in the graph from some vertex a to some vertex b is a walk of length n from a to b with the property that vertices v_0, v_1, ..., v_n are distinct. In other words, a path is a walk in the graph, which passes through the vertices of the graph at most once.

Definition: A walk, trail or path in a graph is called trivial if it is a walk of length zero consisting of a single vertex $v \in V$; otherwise, the walk, trail, or path is called non-trivial.

Example:



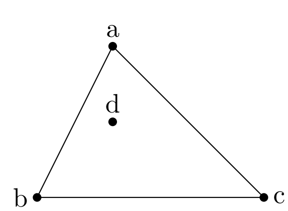
1. h is a trivial walk/trail/path
2. $defd$ is a trail, but not a path because we pass through the vertex d twice.
3. def is a path
4. $gfdefdc$ is a walk but not a trail or a path

9.7 Connected Graphs

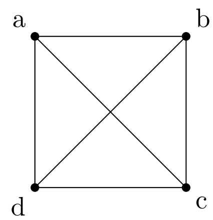
Task: Use the ideas above related to traversing parts of a graph in order to define a particularly important category of graphs.

Definition: An undirected graph (V, E) is called connected if $\forall u, v \in V$ vertices, $\exists$ path in the graph from u to v.

Examples: 1. Is not connected as d is not connected to any other vertex.



2. Is connected. $\exists$ path between any two of the vertices.



Theorem: Let (V, E) be a undirected graph, and let $u, v \in V$. $\exists$ path between u and v in the graph $\Leftrightarrow \exists$ walk in the graph between u and v.

Proof: "$\Rightarrow$" trivial: A path is a walk.
"$\Leftarrow$" $\exists$ walk between u and v. Choose the walk of least length between u and v, (**i.e.** $\nexists$ a walk of lower length than this one) and prove it is a path. Let this walk be $a_0a_1...a_n$ with $a_0 = u$ and $a_n = v$. Assume $\exists j, k$ with $0 \leq j, k \leq n$ s.t. $j < k$ and $a_j = a_k$, but then $a_0a_1...a_ja_{k+1}...a_n$ would be a walk from u to v of <u>strictly smaller</u> length than $a_0a_1...a_n$. $\Rightarrow\Leftarrow$ as we chose $a_0a_1...a_n$ to be of minimal length $\Rightarrow a_j \neq a_k \forall j, k$ s.t. $0 \leq j, k \leq n \Rightarrow a_0a_1...a_n$ is a path between u and v.

qed

Corollary: An undirected graph (V, E) is connected $\Leftrightarrow \forall u, v \in V\ \exists$ walk in the graph between u and v.

9.8 Components of a graph

Task: Divide a graph into subgraphs that are isolated from each other.

Let (V, E) be an undirected graph. We define a relation $\sim$ on the set of vertices V, where $a, b \in V$ satisfy $a \sim b$ iff $\exists$ walk in the graph from a to b.

Lemma: Let (V, E) be an undirected graph. The relation $a \sim b$ or $a, b \in V$, which holds iff $\exists$ walk in the graph between a and b is an equivalence relation.

Proof: We must show $\sim$ is reflexive, symmetric, and transitive.

Reflexive: $\forall v \in V, v \sim v$ since the trivial walk is a walk from v to itself.

Symmetric: If $a \sim b$ for $a, b \in V$, then $\exists$ walk $v_0v_1...v_n$ where $v_0 = a$ and $v_n = b$. This walk can be reversed to $v_nv_{n-1}...v_1v_0$, which now goes from $v_n = b$ to $v_0 = a$. Therefore, $b \sim a$ as needed.

Transitive: If $a \sim b$ and $b \sim c$, for $a, b, c \in V$, there $\exists$ walk $av_1v_2...v_{n-1}b$ from a to b and $\exists$ walk $bw_1w_2...w_{m-1}c$ from b to c. We put these two walks together (concatenate them) to yield the walk $av_1v_2...v_{n-1}bw_1w_2...w_{m-1}c$ from a to c. Therefore $a \sim c$.

qed

The equivalence relation $\sim$ on V partitions it into disjoint subsets $V_1, V_2, ...V_p$, where

1. $V_1 \cup V_2 \cup ... \cup V_p = V$
2. $V_i \cap V_j = \emptyset$ if $i \neq j$
3. Two vertices $a, b \in V_i \Leftrightarrow a \sim b$, **i.e.** $\exists$ walk in (V, E) from a to b

Note that an edge is a walk of length 1, so if $a, b \in V$ satisfy that $\exists\, ab \in E$, then a and b belong to the same V_i. As a result, we can partition the set of edges as follows:

$$E_i = \{ab \in E \mid a, b \in V_i\}$$

Clearly, $E_1 \cup E_2 \cup ... \cup E_p = E$ and $E_i \cap E_j = \emptyset$ if $i \neq j$. Furthermore, $(V_1, E_1), (V_2, E_2), ..., (V_p, E_p)$ are subgraphs of (V, E), and these subgraphs are disjoint since $V_i \cap V_j = \emptyset$ and $E_i \cap E_j = \emptyset$ if $i \neq j$. The subgraphs (V_i, E_i) are called the <u>components</u> (or <u>connected components</u>) of the graph (V, E).

Lemma: The vertices and edges of any walk in an undirected graph are all contained in a single component of that graph.

Proof: Let $v_0 v_1 ... v_n$ be a walk in a graph (V, E), then $v_0 v_1 ... v_r$ is a walk in (V, E) $\forall r$ $1 \leq r \leq n \Rightarrow v_0 \sim v_r$ $\forall r$ $1 \leq r \leq n \Rightarrow v_r$ belongs to the same component of the graph as v_0. The same is true for all the edges $v_{i-1} v_i$ for $1 \leq i \leq n$.

qed

Lemma: Each component of an undirected graph is connected.

Proof: Let (V, E) be a graph and let (V_i, E_i) be any component of (V, E). $\forall u, v \in V_i$, by definition $\exists$ walk in (V, E) between u and v. By previous lemma, however, all vertices and edges of this walk are in $(V_i, E_i) \Rightarrow$ the walk between u and v is a walk in (V_i, E_i), but this assertion is true $\forall u, v \in V_i \Rightarrow (V_i, E_i)$ is connected.

qed

Moral of the story Any undirected graph can be represented as a disjoint union of connected subgraphs, namely its components $\Rightarrow$ the study of undirected graphs reduces to the study of connected graphs, as components don't share either vertices or edges.

9.9 Circuits

Task: Use closed walks to understand the structure of graphs better.

Definition: Let (V, E) be a graph. A walk $v_0v_1...v_n$ in (V, E) is called closed if $v_0 = v_n$, **i.e.** if it starts and ends at the same vertex.

Definition: Let (V, E) be a graph. A circuit is a nontrivial closed trail in (V, E), **i.e.** a closed walk with no repeated edges passing through at least two vertices.

Definition: A circuit is called simple if the vertices $v_0, v_1, v_2, ...v_{n-1}$ are distinct.

NB: This is the strongest condition regarding vertices that we can impose since $v_0 = v_n$.

Alternative terminology: Some authors use cycle to denote a simple circuit, while for others cycle denotes a circuit regardless of whether it is simple or not.

Q: When does a graph have simple circuits?

A: We can give 2 criteria for the existence of simple circuits:

1. Every vertex has degree ≥ 2.
2. $\exists u, v \in V$ s.t. $\exists$ 2 distinct paths from u to v.

Theorem: If (V, E) has no isolated or pendant vertices, **i.e.** $\forall v \in V$ deg $v \geq 2$, then (V, E) contains at least one simple circuit.

Proof: Consider all paths in (V, E). The maximum length of a path is $\#(V)-1$ since a path of length p passes through $p+1$ vertices. Take a path $v_0v_1...v_m$ in (V, E) of maximum length, **i.e.** any other path in (V, E) has length $\leq m =$ length of $v_0v_1...v_m$. Now consider the vertex v_m. deg $v_m \geq 2$ by assumption. We know v_{m-1} is adjacent to v_m since the edge $v_{m-1}v_m$ is part of the path $v_0v_1...v_m$, but deg $v_m \geq 2$ means $\exists w \in V$ s.t. $wv_m \in E$. If $w \neq v_i$ for $0 \leq i \leq m-2$, then $v_0v_1...v_mw$ is a path in (V, E) longer than $v_0v_1...v_m \Rightarrow\Leftarrow$ to the fact that $v_0v_1...v_m$ was chosen of maximal length. Therefore, $w = v_i$ for some $0 \leq i \leq m-2$, but then $v_iv_{i+1}...v_mv_i$ is a simple circuit in the graph.

qed

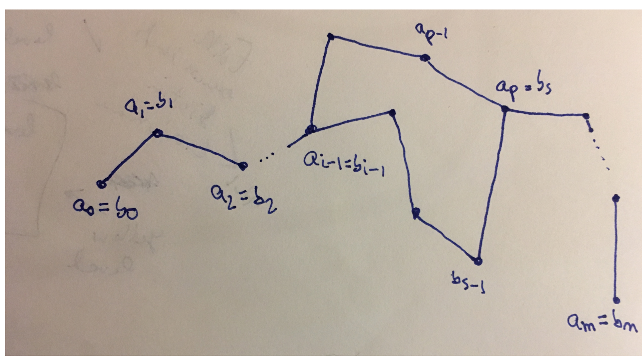
Theorem: Let (V, E) be an undirected graph and let $u, v \in V$ be vertices s.t. $u \neq v$ and $\exists$ at least two distinct paths in (V, E) from u to v. Then the graph contains at least one simple circuit.

Proof: Let $a_0a_1a_2...a_m$ and $b_0b_1...b_n$ be the two distinct paths in the graph between u and v, **i.e.** $a_0 = b_0 = u$ and $a_m = b_n = v$. WLOG let $m \leq n$. Since the paths are distinct $\exists\, i$ with $0 \leq i \leq m$ s.t. $a_i \neq b_i$. Choose the smallest i for which $a_i \neq b_i$, **i.e.** $a_0 = b_0, a_1 = b_1, ..., a_{i-1} = b_{i-1}$, but $a_i \neq b_i$. We have thus eliminated the redundancies at the beginning of the paths. We now need to eliminate redundancies at the other end of the paths. We know $a_m = b_n$ so $a_j \in \{b_k \mid i-1 < k \leq n\}$ is certainly satisfied for $j = m$, but we want to choose the smallest index for which this condition is satisfied. Let this index be $p \Rightarrow a_p \in \{b_k \mid i-1 < k \leq n\}$, **i.e.** $a_p = b_s$ for some s s.t. $i-1 < s \leq n$. Since p is the smallest index satisfying $a_p \in \{b_k \mid i-1 < k \leq n\}$,

$$a_i, a_{i+1}, ..., a_{p-1} \notin \{b_k \mid i-1 < k \leq n\}$$

$\Rightarrow \quad \underbrace{a_{i-1}a_i...a_p}_{\textit{indices running in increasing order}} \quad \underbrace{b_{s-1}...b_i}_{\textit{indices running in decreasing order}} \quad a_{i-1}$ is a simple circuit in (V,E) (recall $a_p = b_s$ and $a_{i-1} = b_{i-1}$) $\Rightarrow (V,E)$ has at least one simple circuit.

qed



9.10 Bridge lecture between Michaelmas and Hilary terms

Task: Review some definitions in order to continue the unit on graph theory.

Recall that an undirected graph consists of a finite set of points V called the vertices of the graph together with a finite set E of edges, where each edge joins two distinct vertices of the graph. More formally: Let V be a set. We denote by V_2 the set consisting of all subsets of V with exactly 2 elements. If $P(V)$ is the power set of V, i.e. the set of all subsets of V, $V_2 = \{A \in P(V) \mid \#(A) = 2\}$.

Definition: An undirected graph (V,E) consists of a finite set V together with a subset E of V_2. The elements of V are the vertices of the graph, and the elements of E are the edges of the graph.

Definition: A graph is said to be trivial if it consists of a single vertex.

Definition: If v is a vertex of some graph (V, E), if e is an edge of the graph, and $e = vw$ for some vertex w of the graph, then the vertex v is said to be incident to the edge e, and the edge e is said to be incident to the vertex v.

Definition: Two distinct vertices v and w of a graph (V, E) are said to be adjacent if and only if $vw \in E$.

Definition: Let (V, E) and (V', E') be graphs. The graph (V', E') is said to be a subgraph of (V, E) if and only if $V' \subset V$ and $E' \subset E$, i.e. if and only if the vertices and edges of (V', E') are all vertices and edges of (V, E).

Definition: Let (V, E) be a graph. The degree $\deg v$ of a vertex v of this graph is defined to be the number of edges of the graph that are incident to v, i.e. the number of edges of that graph that have v as one of their endpoints.

Definition: A vertex of degree 0 is said to be an isolated vertex.

Definition: A vertex of a graph of degree 1 is said to be a pendant vertex.

Definition: Let (V, E) be a graph. A walk $v_0v_1v_2...v_n$ of length n in the graph from vertex a to vertex b is determined by a finite sequence $v_0, v_1, v_2, ..., v_n$ of vertices of the graph s.t. $v_0 = a, v_n = b$ and $v_{i-1}v_i$ is an edge of the graph for $i = 1, 2, ..., n$. A walk $v_0v_1v_2...v_n$ in a graph is said to traverse the edges $v_{i-1}v_i$ and to pass through the vertices $v_0, v_1, ..., v_n$.

Each vertex v in a graph determines a walk of length zero in the graph consisting of the single vertex v; such a walk is said to be trivial.

Definition: Let (V, E) be a graph. A trail $v_0v_1v_2...v_n$ of length n in the graph from some vertex a to some vertex b is a walk of length n from a to b with the property that edges $v_{i-1}v_i$ are distinct for $i = 1, 2, ..., n$.

A trail in a graph is thus a walk, which traverses edges of the graph at most once.

Definition: Let (V, E) be a graph. A path $v_0v_1v_2...v_n$ of length n in the graph from some vertex a to some vertex b is a walk of length n from a to b with the property that vertices v_0, v_1, ..., v_n are distinct.

A path in a graph is thus a walk, which passes through the vertices of the graph at most once.

Definition: Let (V, E) be a graph. A circuit is a nontrivial closed trail in (V, E), **i.e.** a closed walk with no repeated edges passing through at least two vertices.

Definition: An undirected graph is said to be connected if given any two vertices v and w of the graph, there exists a path in the graph from v to w.

We relaxed this condition by proving that an undirected graph (V, E) is connected $\iff \forall\, v, w \in V\ \exists$ walk in the graph between v and w.

9.11 Eulerian trails and circuits

Task: Look at trails and circuits that traverse every edge of a graph. Derive criteria when such trails and circuits exist.

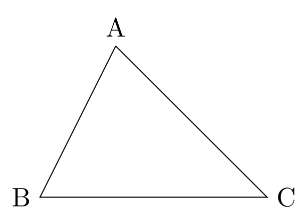
Definition: An Eulerian trail in a graph is a trail that traverses every edge of that graph. In other words, an Eulerian trail is a walk that traverses every edge of the graph exactly once.
Trail $\Rightarrow$ an edge is traversed at most once.
Eulerian $\Rightarrow$ every edge is traversed.

Definition: An Eulerian circuit is a graph is a circuit that traverses every edge of the graph.

Origin of the terminology: Eulerian comes from the Swiss mathematician Leonhard Euler (1707-1783) who solved the problem of the seven bridges of Königsberg/ Kaliningrad (then Prussia, now Russia) over the river Pregel in 1736. His negative solution is considered the beginning of graph theory as a subfield of mathematics. We will rederive Euler's results shortly. Google to see the configuration of the bridges on the river Pregel.

Examples:

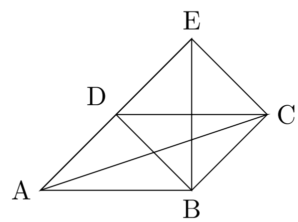
1. $ABCA$ is an Eulerian circuit. The triangle is K_3.



2. Consider K_5, the complete graph with 5 vertices.
   $EABECDBCADE$ is an Eulerian circuit.

In both cases, the degree of the vertices is even for all vertices. We'll see this property is important and derive other necessary and sufficient conditions for the existence of Eulerian trails and circuits.

Theorem: Let (V, E) be a graph, and let $v_0 v_1 ... v_m$ be a trail in (V, E). Let $v \in V$ be a vertex, then the number of edges of the trail incident to v is



even except when the trail is not closed and the trail starts or finishes at v, in which case the number of edges of the trail incident to the vertex v is odd.

Proof: Note that 0 is an even integer as $0 = 2 \times 0$.

Case 1: $v \neq v_0$ and $v \neq v_m$. If the trail does not pass through v, the number of edges incident to v belonging to the trail is 0, which is even.
If the trail passes through v, then edges of the trail incident to v are of the form $v_{i-1}v_i$ and v_iv_{i+1} with $v = v_i$ and $0 < i < m$. Therefore, the number of edges of the trail incident to v equals twice the number of integers i among $1, 2, ..., m-1 (0 < i < m)$ s.t. $v = v_i \Rightarrow$ the number is even.

Case 2: $v = v_0$ and the trail is not closed, **i.e.** $v_m \neq v_0$. The edges incident to v are v_0v_1 along with $v_{i-1}v_i$ and v_iv_{i+1} whenever $v = v_i$, hence $1 + 2 \times \#(\text{instances when } v = v_i)$, which is odd.

Case 3: $v = v_m$ and the trail is not closed, **i.e.** $v_m \neq v_0$. Repeat the argument in case 2 with $v_{m-1}v_m$ replacing v_0v_1 to get that the number of edges incident to v is odd.

Case 4: The trail is closed and $v = v_0 = v_m$. The edges incident to v are $v_0v_1, v_{m-1}v_m$ as well as $v_{i-1}v_i$ and v_iv_{i+1} for each i s.t. $v = v_i \Rightarrow$ once again, the number of edges incident to v is even.

qed

Corollary 1: Let v be a vertex of the graph. Given any circuit in the graph, the number of edges incident to v traversed by that circuit is even.

Proof: Apply the theorem to $v_0v_1...v_m$ s.t. $v_0 = v_m$. We deduce that the number of edges incident to v is even.

Corollary 2: If a graph admits an Eulerian circuit, then the degree of every vertex of that graph must be even.

Proof: Let (V, E) be the graph. $\forall v \in V$, the number of edges of any Eulerian circuit incident to v is even by the previous corollary. Since an Eulerian circuit by definition traverses every edge of the graph, every edge incident to v is an edge of the Eulerian circuit $\Rightarrow$ deg v is even $\forall v \in V$ (**NB:** deg v could be zero if v is an isolated vertex).

Example: By the previous corollary, K_4, the complete graph on four vertices, cannot have an Eulerian circuit since $\forall v$ in K_4, deg $v = 3$ (K_4 is 3-regular as we observed in a previous lecture).

Corollary 3: If a graph admits an Eulerian trail that is not a circuit, then the degrees of exactly two vertices of the graph must be odd, and the degrees of the remaining vertices must be even. The vertices with odd degrees are exactly the initial and final vertices of the Eulerian trail.

Proof: By the theorem, the initial and final vertices of the Eulerian trail have odd degree, whereas all vertices in between have even degrees.

qed

Next, prove the converse of corollary 2: A non-trivial connected graph has an Eulerian circuit if the degree of each of its vertices is even. The proof is carried out in a series of lemmas:

Lemma A: If the degree of each vertex is even, then $\exists$ circuit.

Lemma B: If the degree of each vertex is even, if $\exists$ circuit, and if $\exists$ edges not in the circuit incident to a vertex in the circuit, we can construct another circuit.

Lemma C: If we have two circuits with at least one vertex in common, we can combine them.

Lemma D: A criterion for when a trail is Eulerian in a connected graph.

Lemma A: Let vw be an edge of a graph in which the degree of every vertex is even, then $\exists$ circuit of the graph that traverses the edge vw.

Proof: We construct the circuit starting with the edge vw. Let $v_0 = v$ and $v_1 = w$. Let $v_0v_1...v_k$ be any trail of length $k \geq 1$ traversing the edge vw. Suppose $v_k \neq v = v_0$. As we proved in the previous theorem, since v_k is an endpoint of a non-closed trail, then the number of edges of the trail incident to v_k is odd, but deg v_k is even $\Rightarrow \exists$ edge of the graph incident to v_k that is not traversed by the trail $v_0v_1...v_k$. Let v_kv_{k+1} be this edge, then $v_0v_1...v_kv_{k+1}$ is a trail of length $k+1$ that starts at v and traverses vw. Since every edge of the graph is traversed at most once by a trail, the length of any trail in the graph cannot be greater than the number of edges of the graph $\#(E)$. We have shown above that if our trail is not closed, then it can be extended. By successive extensions, we will eventually have constructed a trail that cannot be extended (in at most $\#(E) - 1$ steps). Therefore, that trail must be closed. As the edge vw is traversed, this trail is nontrivial $\Rightarrow$ it is a circuit.

qed

Lemma B: Let (V, E) be a connected graph s.t. $\forall v \in V$, deg v is even, and let some circuit $v_0v_1...v_{m-1}v_0$ be given. Suppose that for some i with $0 \leq i \leq m-1$, some but not all the edges of the graph incident to v_i are traversed by $v_0v_1...v_{m-1}v_0$, then $\exists$ another circuit in (V, E) passing through v_i that does not traverse any edge traversed by $v_0v_1...v_{m-1}v_0$.

Proof: Let E' be the set of edges not traversed by $v_0v_1...v_{m-1}v_0$. (V, E') is a subgraph of (V, E). $\forall v \in V$, # of edges of $v_0v_1...v_{m-1}v_0$ incident to $v = d(v) - d'(v)$, where $d(v) = \deg(v) =$ # of edges in (V, E) incident to v and $d'(v) =$ # of edges in (V, E') incident to v. By Corollary 1, $d(v) - d'(v)$ is even, but by assumption $d(v) = \deg v$ is even $\Rightarrow d'(v)$ is even $\Rightarrow$ the degree of every vertex in the subgraph (V, E') is even. Now consider the vertex v_i in the statement of Lemma B. Some but not all edges incident to v_i are traversed by $v_0v_1...v_{m-1}v_0 \Rightarrow d'(v_i) > 0$, **i.e.** at least one edge incident to v_i is in the subgraph (V, E'). We are now exactly in the scenario described by Lemma A $\Rightarrow$ by Lemma A, $\exists$ circuit in (V, E') passing through v_i. This circuit is also a circuit in (V, E) as (V, E') is a subgraph of (V, E), and since all of its edges are in E', this other circuit does not traverse any edge traversed by $v_0v_1...v_{m-1}v_0$.

qed

Lemma C: Suppose that a graph contains a circuit of length m and a circuit of length n. Suppose also that no edge of the graph is traversed by both circuits, and that at least one vertex of the graph is common to both circuits, then the graph contains a circuit of length $m+n$.

Proof: Let v be a vertex of the graph that is common to both circuits. WLOG (without loss of generality) we assume both circuits start and finish at the vertex v. Let the first circuit be $vv_1...v_{m-1}v$, and let the second circuit be $vw_1w_2...w_{n-1}v$. We concatenate the two circuits obtaining a circuit $vv_1...v_{m-1}vw_1w_2...w_{n-1}v$ of length $m+n$.

qed

Lemma D: Let (V, E) be a connected graph, and let some trail in this graph be given. Suppose that no vertex of the graph has the property that not all the edges of the graph incident to that vertex are traversed by the trail. Then the given trail is an Eulerian trail.

Proof: Let V_1 be the set of vertices through which the trail passes, and let V_2 be the set of vertices through which the trail does not pass. $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. The conclusion of Lemma D amounts to showing $V_2 = \emptyset$. $\forall u \in V_1$, u is incident to at least one edge traversed by the trail. $\Rightarrow$ All edges incident to the vertices in V_1 are traversed by the trail, but then every vertex in V adjacent to a vertex in V_1 must belong to $V_1 \Rightarrow$ no edge can join a vertex in V_1 to a vertex in V_2. If $V_2 \neq \emptyset$, then $\exists w \in V_2$, but then w cannot be joined by a path to any vertex in $V_1 \Rightarrow V_1$ and V_2 are in different connected components of the graph $\Rightarrow\Leftarrow$ since the graph is connected $\Rightarrow$ it has only one connected component. Therefore, $V_2 = \emptyset$.

qed

Finally, we can prove Euler's theorem:

Theorem A nontrivial connected graph contains an Eulerian circuit if the degree of every vertex of the graph is even.

Proof: Let (V, E) be a non-trivial connected graph s.t. $\forall v \in V$, deg v is even. By Lemma A, (V, E) contains at least one circuit. It therefore contains a circuit of maximal length (**i.e.** at least as long as any other circuit in the graph). We seek to prove that this circuit of maximal length is indeed Eulerian.
If the graph contains some vertex v s.t. some but not all of the edges of the graph incident to v are traversed by the circuit of maximal length, and v is a vertex on the circuit of maximal length, then by Lemma B, $\exists$ a second circuit in (V, E) passing through v, which would not traverse any edge traversed by the circuit of maximal length. By Lemma C, however, we can concatenate the two circuits, obtaining a circuit of length strictly greater than the length of the circuit of maximal length $\Rightarrow\Leftarrow$ we conclude no vertex that belongs to the circuit of maximal length has the property that not all edges incident to it are traversed by the circuit of maximal length. Since (V, E) is connected, by Lemma D, the circuit of maximal length must be Eulerian.

qed

Corollary 2 along with this theorem together gives us:

Theorem: A non-trivial connected graph has an Eulerian circuit $\Leftrightarrow$ the degree of each of its vertices is even.

Corollary: Suppose a connected graph has exactly two vertices whose degree is odd. $\exists$ an Eulerian trail in the graph joining the two vertices with odd degrees.

Proof: We reduce this case to the previous one by embedding the graph (V, E) with vertices v, w that have odd degree into a graph (V', E') s.t. $V' = V \cup \{u\}$ for $u \notin V$ and $E' = E \cup \{uv, uw\}$. (V, E) is a subgraph of $(V'E')$, (V', E') is connected, and each one of its vertices has even degree by construction. By the theorem we just proved, (V', E') has an Eulerian circuit. We reorder the vertices so that the final two edges are the two added edges wu and uv. We now delete the edges wu and uv to obtain an Eulerian trail in the original graph (V, E) from v to w.

qed

9.12 Hamiltonian Paths and Circuits

Task: Look at paths and circuits that pass through every vertex of a graph.

Definition: A Hamiltonian path in a graph is a path that passes exactly once through every vertex of a graph.
Path $\Rightarrow$ we pass through a vertex at most once (no repeated vertices)
Hamiltonian $\Rightarrow$ we pass through every vertex.

Definition: A Hamiltonian circuit in a graph is a simple circuit that passes through every vertex of the graph.

Origin of the Terminology: Named after William Roman Hamilton (1805-1865) who showed in 1856 that such a circuit exists in the graph consisting of the vertices and edges of a dodecahedron (see page 88 in David Wilkins' notes for the picture of a Hamiltonian circuit on a dodecahedron). Hamilton developed a game called Hamilton's puzzle or the icosian game in 1857 whose object was to find Hamiltonian circuits in the dodecahedron (many solutions exist). This game was marketed in Europe as a pegboard with holes for each vertex of the dodecahedron.
NB: The dodecahedron is a Platonic solid, and it turns out every Platonic solid has a Hamiltonian circuit. Recall that the Platonic solids are the tetrahedron (4 faces), the cube (6 faces), the octahedron (8 faces), the dodecahedron (12 faces), and the icosahedron (20 faces). Each of these is a regular graph.

Theorem: Every complete graph K_n for $n \geq 3$ has a Hamiltonian circuit.

Proof: Let $V = \{v_1, v_2, v_3, ...v_n\}$ be the set of vertices of K_n, then $v_1 v_2 v_3 ... v_n v_1$ is a Hamiltonian circuit. All edges in this circuit are part of K_n because K_n is complete.

qed

9.13 Forests and Trees

Task: Use the notion of a circuit to define trees.

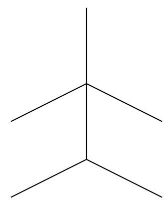
Definition: A graph is called acyclic if it contains no circuits (also known as cycles).

Definition: A forest is an acyclic graph.

Definition: A tree is a connected forest.

Examples:

1. Is a tree and a forest.



2. Is a forest with 2 connected components (**i.e.** it consists of 2 trees.)

Theorem: Every forest contains at least one isolated or pendant vertex.

Proof: Recall that when we studied circuits we proved a theorem that if (V, E) is a graph s.t. $\forall v \in V$ deg $v \geq 2$ (**i.e.** (V, E) has no isolated or pendant vertices), then (V, E) contains at least one simple circuit. The graph (V, E) is a forest, **i.e.** it contains no circuits $\Rightarrow \exists v \in V$ s.t. deg $v = 0$ or deg $v = 1$

qed

Theorem: A non-trivial tree contains at least one pendant vertex.

Proof: A non-trivial tree (V, E) must contain at least 2 vertices. Assume $\exists v \in V$ s.t. deg $v = 0$, **i.e.** v is isolated, then v forms a connected component by itself, but then (V, E) has at least 2 connected components as $\#(V) \geq 2 \Rightarrow\Leftarrow$ to the fact that a tree is by definition connected. Therefore, $\forall v \in V$, deg $v \geq 1$, but by the previous theorem $\exists v \in V$ s.t. $0 \leq$ deg $v \leq 1 \Rightarrow \exists v \in V$ s.t. deg $v = 1$ (since a tree is a forest).

qed

Theorem: Let (V, E) be a tree, then $\#(E) = \#(V) - 1$, where $\#(E)$ is the number of edges of the tree and $\#(V)$ is the number of vertices.

Proof: Use induction on $\#(V)$.

Base Case: $\#(V) = 1$. The graph is trivial $\Rightarrow \#(E) = 0$, so $0 = 1 - 1$ as needed.

Inductive Step: Suppose that every tree with m vertices ($\#(V) = m$) has $m - 1 = \#(V) - 1 = \#(E)$ edges. We seek to prove that if (V, E) is a tree with $m + 1$ vertices, then it has m edges.
By the previous theorem, (V, E) has one pendant vertex. Let that

vertex be v. Since deg $v = 1$, then there is only one edge incident to v. Let vw be that edge. w is then the only vertex of (V,E) adjacent to v. We wish to reduce to the inductive hypothesis. The most natural way is to delete v from V and vw from E. Let $V' = V\backslash\{v\}$ and $E' = E\backslash\{vw\}$. (V',E') is a subgraph of (V,E) such that $\#(V') = \#(V) - 1$ and $\#(E') = \#(E) - 1$. To use the inductive hypothesis, we must show (V',E') is a tree, **i.e.** (V',E') is connected and $(V'E')$ contains no circuits. $\forall v_1, v_2 \in V'$, since (V,E) is a tree hence connected, $\exists$ path from v_1 to v_2 in (V,E). This path cannot pass through v because deg $v = 1 \Rightarrow$ it would have to pass through w twice contradicting the fact that it is a path (all vertices are distinct) $\Rightarrow$ this path is in $(V',E') \Rightarrow (V'E')$ is connected.
(V',E') is a subgraph of (V,E), which is a tree, hence does not contain any circuits, so (V',E') contains no circuits.
(V',E') is thus a tree, $\Rightarrow$ by the inductive hypothesis, $\#(E') = \#(E) - 1 = \#(V') - 1 = \#(V) - 1 - 1 = \#(V) - 2 \Rightarrow \#(E) - 1 = \#(V) - 2 \Leftrightarrow \#(E) = \#(V) - 1$ as needed.

qed

Theorem: Let (V,E) be a tree, then $\forall\, v, w \in V$ with $v \neq w$ $\exists!$ path in (V,E) from v to w.

Proof: (V,E) is a tree $\implies$ (V,E) is connected $\implies$ $\exists$ path from v to w. Assume there exist two distinct paths from v to w. By a previous theorem, we deduce (V,E) contains a circuit (recall that one criterion for having a circuit in a graph was the existence of two distinct paths between two vertices) $\Rightarrow\Leftarrow$ (V,E) is a tree, hence it contains no circuits $\implies$ the path between v and w in (V,E) is unique.

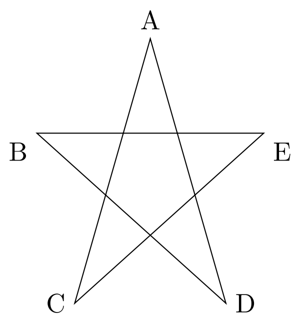
qed

9.14 Spanning Trees

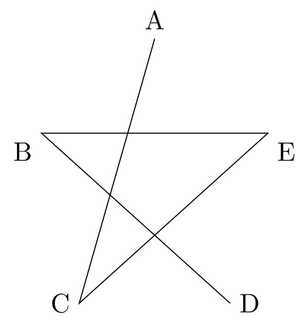
Task: For any graph, construct a subgraph containing all the vertices of the original graph such that this subgraph is a tree.

Definition: A spanning tree in a graph (V,E) is a subgraph of the graph (V,E), which is a tree and includes every vertex in V.

Example:



the pentagram has



as a spanning tree (we delete the edge AD so that there is no circuit).

Remark: A graph (V, E) may have more than one spanning tree, **i.e.** spanning trees are not unique.

Theorem: Every connected graph contains a spanning tree.

Proof: Let (V, E) be a connected graph. Let $\mathcal{C}$ be the collection of all connected subgraphs (V', E') of the graph (V, E) with $V' = V$ (**i.e.** containing all the vertices of the original graph). The original graph $(V, E) \in \mathcal{C}$, so $\mathcal{C}$ is not empty. Choose (V, E') in $\mathcal{C}$ such that the number of edges $\#(E')$ is minimal, **i.e.** (V, E') is such that $\forall (V, E'') \in \mathcal{C}$, $\#(E') \leq \#(E'')$.

Claim: (V, E') is the required spanning tree.

Proof of claim: (V, E') is connected and has the same vertices as (V, E) since it belongs to $\mathcal{C}$. We just need to show that (V, E') is a tree, **i.e.** that it contains no circuits.
We prove so indirectly, **i.e.** by contradiction. Assume (V, E') contains a circuit. Let vw be one of the edges traversed by a circuit in (V, E'). Let $E'' = E' - \{vw\}$ (we take out that edge). There still exists a walk from vertex v to vertex w via the remaining edges of the circuit. Note that since (V, E') is connected, there exists a walk from every vertex in V to v via the edges in E' and therefore to either v or w via edges in E''. Since there exists a walk from v to w via edges in E'', every vertex in V is connected to v via a walk whose edges belong to $E'' \Rightarrow (V, E'')$ is connected $\Rightarrow (V, E'') \in \mathcal{C}$, but $\#(E'') = \#(E') - 1 \Rightarrow\Leftarrow$ as (V, E') was selected to be the graph in $\mathcal{C}$ with the least number of edges $\Rightarrow (V, E')$ cannot contain a circuit $\Rightarrow (V, E')$ is the required spanning tree.

qed

Corollary: Let (V, E) be a connected graph with $\#(V)$ vertices and $\#(E)$ edges. If $\#(E) = \#(V) - 1$, then (V, E) is a tree.

Proof: By the previous theorem, every connected graph contains a spanning tree, and by a previous theorem proven during the section on trees, that tree has $\#(V)-1$ edges $\Rightarrow$ The spanning tree has the same number of edges as (V,E) and is its subgraph by definition $\Rightarrow$ (V,E) is its own spanning tree $\Rightarrow$ (V,E) is a tree.

qed

9.15 Constructing spanning trees

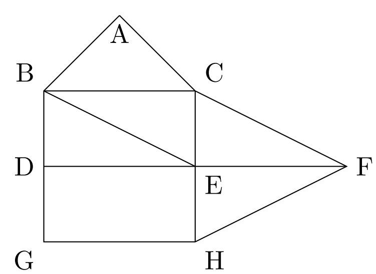
Task: Given a connected undirected graph, investigate two ways of constructing a spanning tree for it.

Let (V,E) be a connected undirected graph. We can proceed in one of two ways to construct a spanning tree for it:

1. Start with (V,E) itself. Break up all of its circuits by deleting one edge per circuit.
2. Start with an edge in E. Let this edge be vw. Add back all remaining vertices in $V-\{v,w\}$ by adding in one edge in E per vertex such that at each step the subgraph of (V,E) that we have is both connected AND a tree.

Remark: Note that algorithm 1 is akin to the proof of the theorem that every connected graph has a spanning tree.

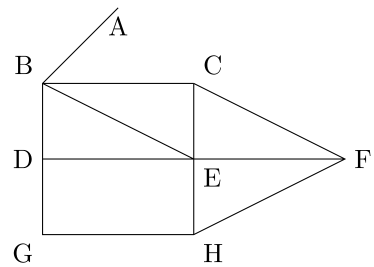
Example: Consider



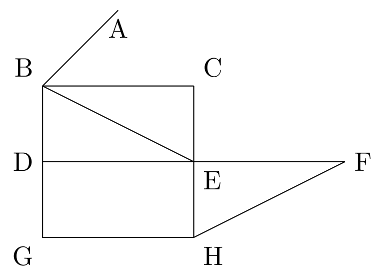
We shall illustrate both 1 and 2 on this graph.

First procedure 1:

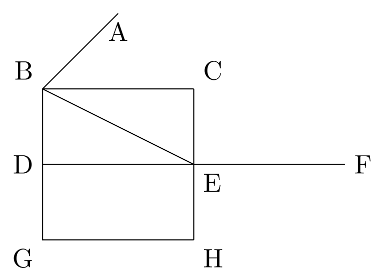
Note $ABCA$ is a circuit. We have a choice which edge to delete. Let us choose to delete AC.



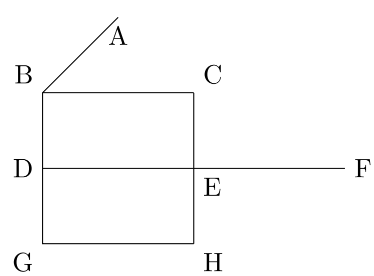
$CEFC$ is a circuit. We choose to delete CF.



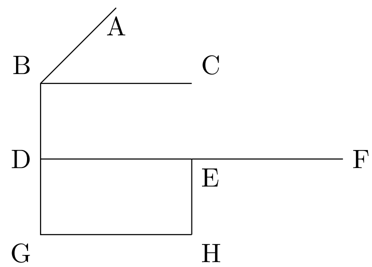
$HFEH$ is a circuit. We choose to delete FH.



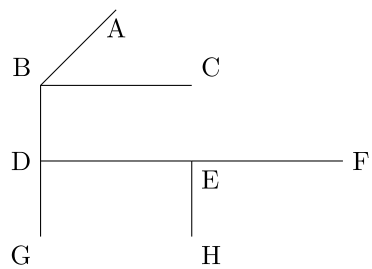
$BDEB$ is a circuit. We choose to delete BE.



$BCEDB$ is a circuit. We choose to delete CE.



$DEHGD$ is a circuit. We choose to delete GH.



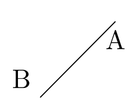
The graph that is left doesn't seem to have any circuits. We check that it is a tree using the formula we proved earlier in the course that for a tree $\#(E) = \#(V) - 1$.
$V = \{A, B, C, D, E, F, G, H\} \Rightarrow \#(V) = 8$
$E' = \{AB, BC, BD, DE, EF, DG, EH\} \Rightarrow \#(E') = 7 = \#(V) - 1$
So (V, E') that we have constructed is a tree and hence the spanning tree of the original (V, E).

Now we follow procedure 2. We start with a vertex in V, and at each step we add on an edge from E such that this edge is adjacent to a vertex already in the collection of vertices and also to a vertex that is not already in the collection. In other words, at each step, we add a vertex and an edge such that the resulting graph is connected. We stop once we capture all vertices in V.

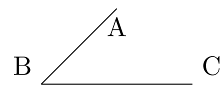
We start with vertex A.

A

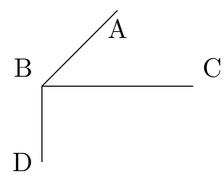
We could add vertex B and edge AB OR we could add vertex C and edge AC. We choose to add vertex B and edge AB.



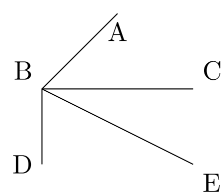
Next, we choose to add vertex C and edge BC.



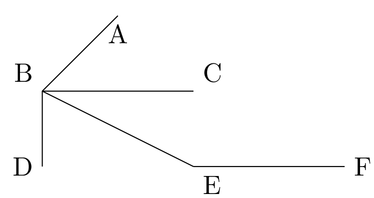
Next, we choose to add vertex D and edge BD.



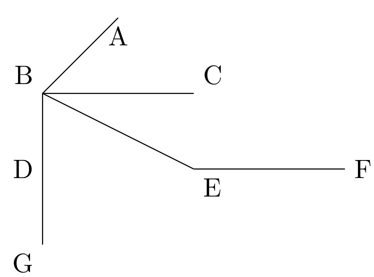
Next, we choose to add vertex E and edge BE.



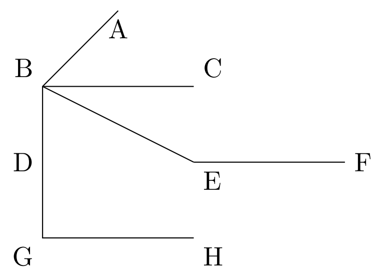
Next, we choose to add vertex F and edge EF.



Next, we choose to add vertex G and edge DG.



Next, we choose to add vertex H and edge GH.



We now have all vertices in $V = \{A, B, C, D, E, F, G, H\}$

We started with 1 vertex and 0 edges. At each step we added 1 vertex and 1 edge $\Rightarrow$ at each step i, if V_i is the set of vertices at step i and E_i is the set of edges at step i, we have that $\#(E_i) = \#(V_i) - 1$ for $i = 0, 1, ..., 7$. In other words, at each step, our subgraph (V_i, E_i) is a tree and by construction it is connected. When $V_i = V$, **i.e.** for $i = 7$, (V, E_7) is a spanning tree of the original (V, E).

NB: Procedure 1 and procedure 2 yielded DIFFERENT spanning trees of (V, E) as we had lots of choices as to which edges to delete or add respectively. We thus see that a spanning tree of a connected graph is not unique unless of course, the original graph is itself a tree.

9.16 Kruskal's Algorithm

Task: If each edge of a connected graph (V, E) comes with a particular cost, describe an algorithm that finds the spanning tree of (V, E) with minimal cost.

Definition: Let (V, E) be an undirected graph. A cost function $c : E \to \mathbb{R}$ on the set E of edges of the graph is a function that assigns to each edge e of the graph a real number $c(e)$.

Let $c : E \to \mathbb{R}$ be a cost function on the set E of edges of a graph (V, E). Given any subset $S \subset E$, we define the cost on S to be $c(S) = \sum_{e \in S} c(e)$, the sum of the costs of all elements of S.

Definition: Let (V, E) be a connected graph with cost function $c : E \to \mathbb{R}$. A spanning tree (V, E_M) is said to be minimal (with respect to the cost function) if $\forall\, (V, E_T)$ a spanning tree of (V, E), $c(E_M) \leq c(E_T)$.

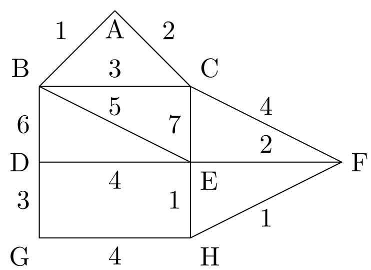
Kruskal's Algorithm for finding minimal spanning trees: Let (V, E) be a connected graph with an associated cost function $c : E \to \mathbb{R}$.

1. Start with $(V, \emptyset)$, the subgraph of (V, E) consisting of all the vertices of (V, E) and no edges.

2. List all edges in E in a queue so that the cost of the edges is non-decreasing in the queue, **i.e.** if $e, e' \in E$ and if $c(e) < c(e')$, then e precedes e' in the queue.
3. Take edges successively from the front of the queue, and determine whether or not the addition of that edge to the current subgraph will create a cycle (circuit). If a circuit is created by this addition, discard the edge; otherwise, add it to the subgraph. Continue until the queue is emptied.

We will first do an example, and after the example we will prove Kruskal's algorithm yields a spanning tree that is indeed minimal.

Example:

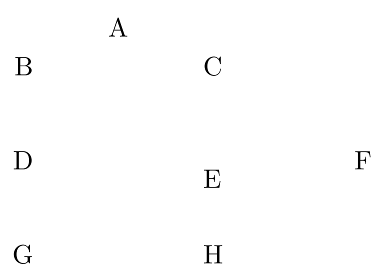


The cost function here is
$c(AB) = 1, c(AC) = 2, c(BC) = 3$
$c(BD) = 6, c(BE) = 5, c(DE) = 4$
$c(CE) = 7, c(CF) = 4, c(EF) = 2$
$c(EH) = 1, c(FH) = 1, c(DG) = 3$
$c(GH) = 4$

We can also use a table format to write down the cost function $c : E \to \mathbb{R}$

AB	AC	BC	BD	BE	DE	CE	CF	EF	EH	FH	DG	GH
1	2	3	6	5	4	7	4	2	1	1	3	4

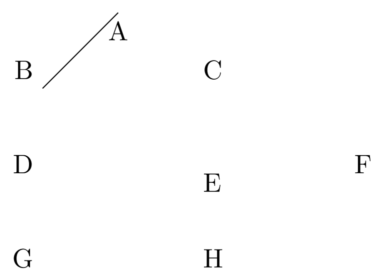
We start with $(V, \emptyset)$. This is step 0 of the algorithm (The starting state).



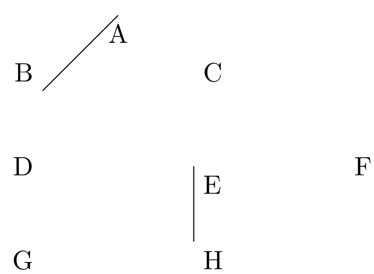
We list the edges in a queue so that the cost is non-decreasing.

AB	EH	FH	AC	EF	BC	DG	DE	CF	GH	BE	BD	CE
1	1	1	2	2	3	3	4	4	4	5	6	7

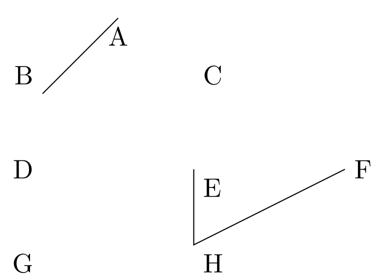
Step 1: We can add AB (no circuit is formed).



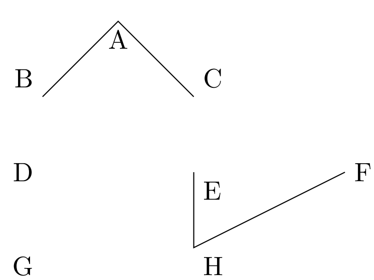
Step 2: We can add EH.



Step 3: We can add FH.



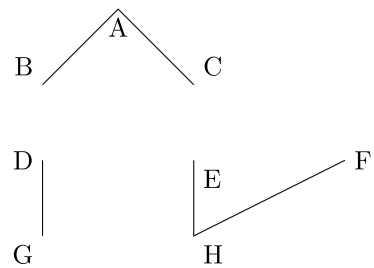
Step 4: We can add AC.



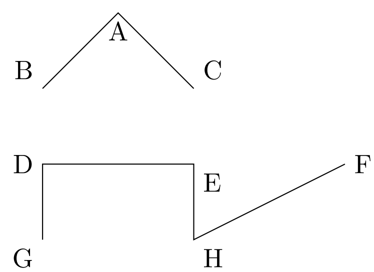
Step 5: We cannot add edge EF because we would create circuit $EFHE$, so EF gets discarded.

Step 6: We cannot add edge BC because we would create circuit $ABCA$, so BC gets discarded.

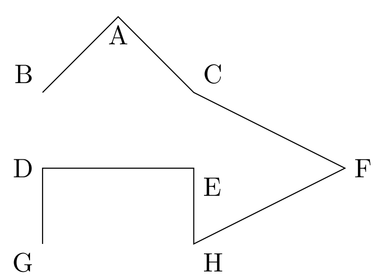
Step 7: We can add DG.



Step 8: We can add DE.



Step 9: We can add CF.



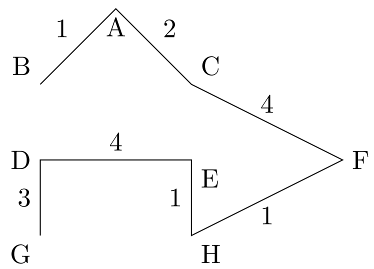
Step 10: We cannot add GH because we would create circuit $DEHGD$.

Step 11: We cannot add BE because we would create circuit $BEHFCAB$.

Step 12: We cannot add BD because we would create circuit $BDEHFCAB$.

Step 13: We cannot add CE because we would create circuit $CEHFC$.

The minimal spanning tree given by Kruskal's algorithm is thus:



with cost $c(E_T) = 1 + 2 + 4 + 1 + 1 + 4 + 3 = 16$

Now that we have some intuition about the Kruskal algorithm, let us prove that it always yields a spanning tree that is indeed minimal.

Proposition: Let (V, E) be a connected graph with associated cost function $c : E \to \mathbb{R}$. Kruskal's algorithm yields a spanning tree of (V, E).

Proof: Since an edge is added from the queue only if no circuit is formed, we conclude the subgraph (V, E') of (V, E) produced by the Kruskal algorithm must be acyclical (**i.e.** contains no circuits). To prove (V, E') is a spanning tree of (V, E), we must show (V, E') is connected. Assume not, then (V, E') has components $(V_1, E'_1), (V_2, E'_2), ..., (V_m, E'_m)$ for $m \geq 2$. (V, E) is connected, however $\Rightarrow \exists$ edge $e_{ij} \in E$ for $1 \leq i, j \leq m, i \neq j$ such that adding edge e_{ij} connects (V_i, E'_i) and (V_j, E'_j), but edge e_{ij} could not have possibly created a circuit when considered in the queue $\Rightarrow\Leftarrow$ (V, E') cannot have more than one connected component $\Rightarrow$ (V, E') is connected.

qed

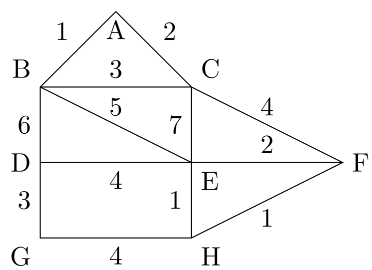
Proposition: Let (V, E) be a connected graph with associated cost function $c : E \to \mathbb{R}$. Kruskal's algorithm yields a minimal spanning tree of (V, E).

Proof: We already showed in the previous proposition that Kruskal's algorithm yields a spanning tree. Now we have to show that spanning tree is minimal with regards to $c : E \to \mathbb{R}$. Let (V, E') be the spanning tree given by the algorithm. If $(V, E') = (V, E)$, **i.e.** if the original connected graph is a tree, then there is nothing to prove, as it is the only possible spanning tree of the original connected graph. Assume $(V, E') \neq (V, E)$ **i.e.** (V, E) contains some circuit. Let all the edges of (V, E) be $e_1, e_2, ..., e_m$ that we label such that $c(e_i) \leq c(e_j)\ \forall 1 \leq i < j \leq m$. In other words, $c(e_1) \leq c(e_2) \leq ... \leq c(e_{m-1}) \leq c(e_m)$. Kruskal's algorithm chooses the lowest cost $\#(V) - 1$ edges from $e_1, e_2, ..., e_m$ such that the resulting subgraph is a spanning tree of (V, E). Therefore, if (V, E'') is any other spanning tree of (V, E), then $c(E') \leq c(E'')$.

qed

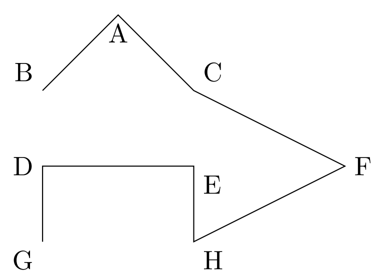
Definition: Let (V, E) be a connected graph with associated cost function $c : E \to \mathbb{R}$. Let (V, E') be the minimal spanning tree of (V, E) produced by Kruskal's algorithm. (V, E') is called the Kruskal tree.

NB: If two or more edges have the same cost, then we can reshuffle them in the queue used to determine the Kruskal tree. Therefore, the Kruskal tree might not be unique. In the example we used to illustrate Kruskal's algorithm we see this scenario at work:

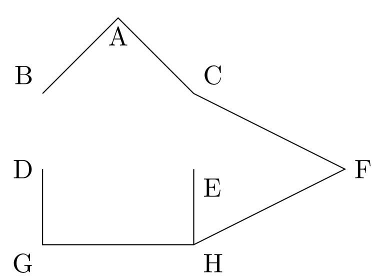


$$c(DE) = c(GH) = 4$$

We used the queue AB, EH, FH, AC, EF, BC, DG, DE, CF, GH, BE, BD, CE to produce the Kruskal tree



Whereas the queue AB, EH, FH, AC, EF, BC, DG, GH, CF, DE, BE, BD, CE would have produced the Kruskal tree



which has the same cost.

Remarks: 1. Joseph Kruskal published this algorithm that bears his name in 1956, two years after he finished his PhD at Princeton. Kruskal is known for work in computer science, combinatorics, and statistics.

2. The cost of an edge is sometimes called the weight of that edge.
3. Kruskal's algorithm starts with a disconnected graph $(V, \emptyset)$ and adds edges until the graph becomes connected and a tree, thus a spanning tree. In other words, until the last addition of an edge, the graph is disconnected.

9.17 Prim's Algorithm

Task: Describe another algorithm for constructing the minimal spanning tree, which is characterized by the fact that at each step of the algorithm, the subgraph is a tree. This algorithm is called Prim's Algorithm.

Vojtěch Jarník first discovered and published this algorithm in 1930. Robert Prim subsequently rediscovered and published it in 1957. It was once again rediscovered by Edsger Wybe Dijkstra in 1959.

Moral of the story: The idea behind this algorithm is very natural. We apply procedure 2 for constructing a spanning tree that we discussed before using the same queue of edges ordered by cost as in Kruskal's algorithm. The result at each step is a tree, and at the end we get a minimal spanning tree.

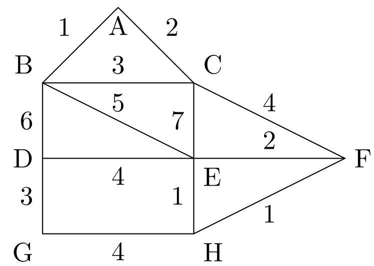
Prim's algorithm: Let (V, E) be a connected graph with an associated cost function $c : E \to \mathbb{R}$.

1. Start by choosing some vertex $v \in V$. Our starting subgraph is $(\{v\}, \emptyset)$.
2. List all edges in E in a queue so that the cost of the edges is non-decreasing in the queue, **i.e.** if $e, e' \in E$ and if $c(e) < c(e')$, then e precedes e' in the queue.
3. We identify the first edge in the queue, which has one vertex included in the current subgraph and the other vertex not included in the subgraph. We add that edge to the current subgraph as well as the vertex not already included. Since the subgraph with which we started was a tree, the resulting subgraph is a tree (we added one vertex and one edge). Continue this process until it is not possible to proceed any further, **i.e.** we have added all vertices in V.

The fact that at each stage we have a tree, and at the end that tree contains all vertices in V guarantees that Prim's Algorithm yields a spanning tree. The fact that we choose what edge to add next by following the queue of edges ordered by cost guarantees that the tree we obtain is a minimal spanning tree of the original connected graph (V, E).

Let us illustrate Prim's Algorithm on the same graph we used for Kruskal's algorithm.

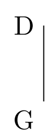
Example: Consider



We use the same queue as before - AB, EH, FH, AC, EF, BC, DG, DE, CF, GH, BE, BD, CE.

We have a choice of which vertex we take to start the algorithm. Let us choose vertex D. So at step 0, we have $(\{D\}, \emptyset)$.

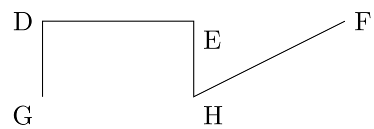
Step 1: We process the queue and find that the first edge in it incident to vertex D is DG. We add vertex G (not already in the subgraph) and edge DG.



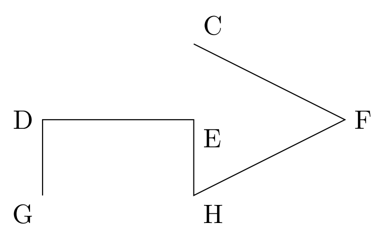
Step 2: We process the queue looking for the first edge incident to either vertex D or vertex G and find DE. We add vertex E (not already in the subgraph) and edge DE.

Step 3: We process the queue from the beginning again looking for the first edge incident to D, E or G and find EH. We add vertex H (not already in the subgraph) and edge EH.

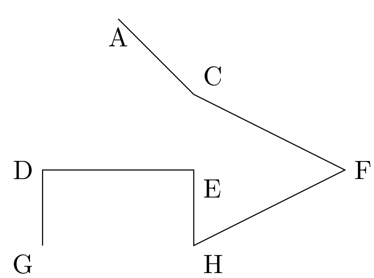
Step 4: We process the queue from the beginning again looking for the first edge incident to D, E, G or H with an endpoint not in the set $\{D, E, G, H\}$ and find FH. We add vertex F and edge FH.



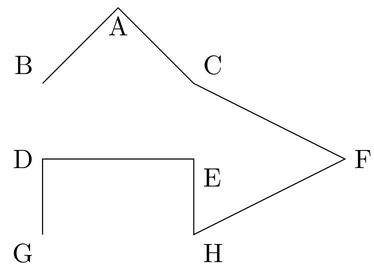
Step 5: We process the queue from the beginning looking for the first edge incident to D, E, F, G or H with the other endpoint not in $\{D, E, F, G, H\}$ and find CF. We add vertex C and edge CF.



Step 6: We process the queue from the beginning looking for the first edge incident to C, D, E, F, G or H with the other endpoint not in $\{C, D, E, F, G, H\}$ and find AC. We add vertex A and edge AC.



Step 7: We process the queue from the beginning looking for the first edge incident to A, C, D, E, F, G or H with the other endpoint not in $\{A, C, D, E, F, G, H\}$ and find AB. We add vertex B and edge AB.



We have recovered all vertices of the original graph so the algorithm ends here. Prim's Algorithm produced the same tree as Kruskal's in this case given the same queue.

Remarks:

1. Just like Kruskal's Algorithm, Prim's Algorithm produces a unique minimal spanning tree if no two edges have the same cost. If there are edges with the same cost, reshuffling them yields different queues that in turn yields different minimal spanning trees.
2. We make a choice as to which vertex kickstarts Prim's Algorithm. Different choices yield different trees at intermediate steps of the algorithm.

Definition The minimal spanning tree yielded by Prim's Algorithm is called the Prim spanning tree.

Definition Let (V_i, E_i) be the subgraph at the end of step i of Prim's Algorithm. All vertices in V_i are called visited vertices. If (V, E) is the original connected graph on which Prim's Algorithm is being applied, all vertices in $V \setminus V_i$ are called the unvisited vertices.

Applications of minimal spanning trees:

- Design of networks such as computer networks, transportation networks, telecommunication networks, water supply networks, electrical grids, etc.
- Computing minimal spanning trees appears as a subroutine in algorithms such as algorithms approximating NP-hard problems such as the travelling salesman problem.
- Minimal spanning trees can be used to describe financial markets, in particular how stocks are correlated.
- Various other problems in computer science and engineering.

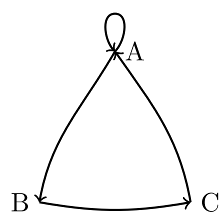
9.18 Directed Graphs

Task: Introduce a new category of graph where the edges have directions and loops are allowed.

Definition: A directed graph or digraph (V, E) consists of a finite set V together with a subset E of $V \times V$. The elements of V are the vertices of the digraph, whereas the elements of E are the edges of the digraph.

Remark: Recall that when we defined undirected graphs (V, E), the set of edges E was a subset of V_2, where V_2 was the set consisting of all subsets of V with exactly two elements. Note that $\{v, w\} = \{w, v\} \in V_2$ if $v \neq w$, whereas $(v, w) \neq (w, v) \in V \times V$. The pairs in $V \times V$ are ordered. Also, $(v, v) \in V \times V \Rightarrow$ loops are allowed as edges of a digraph, whereas they weren't allowed as edges of an undirected graph.

Definition: Let $(v, w) \in E$ be the edge of a digraph (V, E). We say that v is the initial vertex, and w is the terminal vertex of the edge. Furthermore, we say that the vertex w is adjacent from the vertex v and vertex v is adjacent to the vertex w, whereas the edge (v, w) is incident from the vertex v and incident to the vertex w.



is a digraph with $V = \{A, B, C\}$ and $E = \{(A, A), (A, B), (B, C), (C, A)\}$

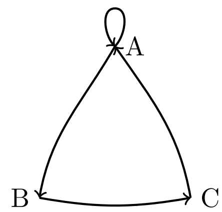
We use arrows to indicate the direction of the edges of a digraph.

Just like an undirected graph, a directed graph has an adjacency matrix associated with it. Let (V, E) be a directed graph, and let the vertices in V be ordered $v_1, v_2, ..., v_m$. The adjacency matrix of (V, E) is the $m \times m$ matrix (b_{ij})

$$\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mm} \end{pmatrix}$$

$$\text{where } b_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Example:



Let $v_1 = A, v_2 = B, v_3 = C$

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

$(A, C) \notin E$ but $(C, A) \in E$

Remark: The adjacency matrix of an undirected graph always had 0's on the diagonal, whereas the adjacency matrix of a directed graph could have some 1's on the diagonal due to the presence of loops.

9.19 Directed Graphs and Binary Relations

Task: Describe the one-to-one correspondence between directed graphs and binary relations on finite sets.

Let V be a finite set.

<u>To every relation R on V, there corresponds a directed graph</u>:

Indeed, set $E = \{(v, w) \in V \times V \mid vRw\}$, then (V, E) is a directed graph.

<u>To every directed graph (V, E), there corresponds a relation R on V</u>:

Indeed, we define the relation R on V as follows:
$\forall v, w \in V, vRw \Leftrightarrow (v, w) \in E$.

Moral of the story: We can use directed graphs to visually represent binary relations on finite sets.

10 Countability of Sets

Task: Understand what it means for a set to be countable, countably infinite, uncountably infinite. Show that the set of all languages over a finite alphabet is uncountably infinite, whereas the set of all regular languages over a finite alphabet is countably infinite.

We want to understand sizes of sets. In the unit on functions last term, when we looked at functions defined on finite sets, we wrote down a set A with n elements as $A = \{a_1, ..., a_n\}$. <u>This notation mimics the process of counting :</u> a_1 is the first element of A, a_2 is the second element of A, and so on up to a_n is the n^{th} element of A. In other words, another way of saying A is a set of n elements is that there exists a bijective function $f : A \longrightarrow \{1, 2, ..., n\}$. Let $J_n = \{1, 2, ..., n\}$.

Definition: A set A has n elements $\iff \exists f : A \longrightarrow J_n$ a bijection.

NB: This definition works $\forall n \geq 1, n \in \mathbb{N}^*$.

Notation: $\exists f : A \longrightarrow J_n$ a bijection is denoted as $A \sim J_n$.

More generally, $A \sim B$ means $\exists f : A \longrightarrow B$ a bijection, and it is a relation on sets. In fact, it is an equivalence relation (check!). $[J_n]$ is the equivalence class of all sets A of size n, i.e. $\#(A) = n$.

Definition: A set A is finite if $A \sim J_n$ for some $n \in \mathbb{N}^*$ or $A = \emptyset$.

Definition: A set A is infinite if A is not finite.

Examples: $\mathbb{N}, \mathbb{Q}, \mathbb{R}$, etc.

To understand sizes of infinite sets, generalise the construction above. Let $J = \mathbb{N}^* = \{1, 2, ...\}$.

Definition: A set A is countably infinite if $A \sim J$.

Definition: A set A is uncountably infinite if A is neither finite nor countably infinite.

In fact, we often treat together the cases A is finite or A is countably infinite since in both of these cases the counting mechanism that is so familiar to us works. Therefore, we have the following definition:

Definition: A set A is countable if A is finite ($A \sim J_n$ or $A = \emptyset$) or A is countably infinite ($A \sim J$).

There is a difference in terminology regarding countability between CS sources (textbooks, articles, etc.) and maths sources. Here is the dictionary:

CS	**Maths**
countable	at most countable
countably infinite	countable
uncountably infinite	uncountable

It's best to double check which terminology a source is using.

Goal: Characterise $[\mathbb{N}]$, the equivalence class of countably infinite sets, and $[\mathbb{R}]$, the equivalence class of uncountably infinite sets the same size as $\mathbb{R}$.

Bad News: Both $[\mathbb{N}]$ and $[\mathbb{R}]$ consist of infinite sets.

Good News: We only care about these two equivalence classes.

NB: These are uncountably infinite sets of size bigger than $\mathbb{R}$ that can be obtained from the power set construction, but it is unlikely you will see them in your CS coursework.

To characterise $[\mathbb{N}]$ we need to recall the notion of a sequence:

Definition: A <u>sequence</u> is a set of elements $\{x_1, x_2, ...\}$ indexed by J, i.e. $\exists f : J \to \{x_1, x_2, ...\}$ such that $f(n) = x_n \ \forall n \in J$.

Recall that sequences and their limits were used to define various notions in calculus (differentiation, integration, etc.) Also, calculators use sequences in order to compute with various rational and irrational numbers.

Examples

1. $\boldsymbol{\pi} \simeq 3.1415...$ i.e. instead of π we can work with the following sequence of rational numbers : $x_1 = 3$, $x_2 = 3.1$, $x_3 = 3.14$, $x_4 = 3.141$, $x_5 = 3.1415, ...$ $\lim\limits_{n\to\infty} x_n = \pi$. π is irrational, i.e., $\pi \in \mathbb{R}\backslash\mathbb{Q}$.
2. $\frac{1}{3} \simeq 0.333...$ means we can set up the sequence of rational numbers $x_1 = 0$, $x_2 = 0.3$, $x_3 = 0.33$, $x_4 = 0.333$, $x_5 = 0.3333$ etc. such that $\lim\limits_{n\to\infty} x_n = \frac{1}{3}$. Note that $\frac{1}{3} \in \mathbb{Q}$.

Restatement of the definition of countably infinite: A set A is countably infinite if its elements can be arranged in a sequence $\{x_1, x_2, ...\}$. This is another of saying A is in bijective correspondence with J, i.e $\exists f : A \longrightarrow J$ a bijection, namely $A \sim J$.

Application of the restatement: $\mathbb{Z} \sim \mathbb{N}$

Indeed, we can write $\mathbb{Z}$ as a sequence since $\mathbb{Z} = \{0, 1, -1, 2, -2, ...\}$ so $\mathbb{Z} \in [\mathbb{N}]$, $\mathbb{Z}$ is countably infinite like $\mathbb{N}$.

Big difference between finite and infinite sets: Let A, B be finite sets such that $A \subsetneqq B$, i.e. $A \subset B$ but $A \neq B$. Then $A \nsim B$ since $\#(A) < \#(B)$ and $J_n \nsim J_m$ if $n \neq m$. Let A, B be infinite sets such that $A \subsetneqq B$, $A \subset B$, but $A \neq B$. It is possible that $A \sim B$. We saw this behaviour in Hilbert's hotel problem (Hilbert's Paradox of the Grand Hotel): $\mathbb{N}^* \subsetneqq \mathbb{N}$, but $\mathbb{N} \sim \mathbb{N}^*$ via the bijection $f : \mathbb{N} \longrightarrow \mathbb{N}^*$ given by $f(n) = n + 1$, so $\{0, 1, 2, ...\} \sim \{1, 2, 3, ...\}$.

In the same vein, we get the following result:

Theorem: Every infinite subset of a countably infinite set is itself countably infinite.

Proof: Let $E \subseteq A$ be the subset in question, where E is infinite and A is countably infinite. A is countably infinite $\iff A \sim J \iff A = \{x_1, x_2, ...\}$.

To show E is countably infinite, we want to show we can represent $E = \{x_{n_1}, x_{n_2}, ...\}$. We construct this sequence of n_j's from the indices of the elements of A in the enumeration $\{x_1, x_2, ...\}$ as follows:

Let n_1 be the smallest integer in J such that $x_{n_1} \in E \subseteq A$. We construct the rest of the sequence of n_j's by induction. Say we have constructed $n_1, n_2, ..., n_{k-1} \in \mathbb{N}^*$. Let n_k be the smallest integer greater than n_{k-1} such that $x_{n_k} \in E$. By construction, $n_1 < n_2 < ...$ and $E = \{x_{n_1}, x_{n_2}, ...\}$

qed

Remark: $\{x_{n_1}, x_{n_2}, ...\}$ is called a subsequence of $\{x_1, x_2, ...\}$

Algorithmic restatement of previous proof:

Let $A = \{x_1, x_2, ...\}$ be an enumeration of A (i.e. writing the countably infinite set A as a sequence). We process $\{x_1, x_2, ...\}$ as a queue. First look at x_1. If $x_1 \in E$, keep x_1 and let $n_1 = 1$; otherwise, discard x_1. Process each x_i in turn, keeping only those that are in E. Their indices form the subsequence $\{n_j\}_{j=1,2,...}$ where $E = \{x_{n_1}, x_{n_2}, x_{n_3}, ...\}$.

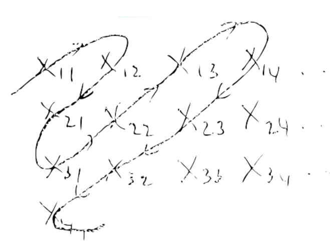
Next, we want to show $\mathbb{Q} \sim \mathbb{N}$, the set of rational numbers is countably infinite.

Notation: A sequence $\{x_1, x_2, \dots\}$ can also be denoted by $\{x_i\}_{i=1,2,...}$

Theorem: Let $\{A_n\}_{n=1,2,...}$ be a sequence of countably infinite sets. Let $S = \bigcup_{n=1}^{\infty} A_n$. Then S is countably infinite.

Proof: Each A_n is countably infinite $\iff A_n \sim J$, $\forall n \geq 1 \iff A_n = \{x_{n_k}\}_{k=1,2,...} = \{x_{n_1}, x_{n_2}, x_{n_3}, ...\}$ We use two indices like for the entries of a matrix. The first index tells us which A_n set the element belongs to, while the second index tells us where the element is in the enumeration (the counting) of A_n.

Write



$= \{x_{11}, x_{12}, x_{21}, x_{31}, x_{22}, x_{13}, x_{14}, x_{23}, x_{32}, x_{41}, ...\} = \bigcup_{n=1}^{\infty} A_n = S$ is countably infinite because even if some x_{ij}'s are the same, $A_n \subseteq S\ \forall n \geq 1$, and $A_n \sim J$.

qed

Corollary 1: Suppose an indexing set I is countable, and $\forall i \in I$, A_i is countable, then $T = \bigcup_{i \in I}$ is countable.

Proof: The biggest set we can obtain is when I is countably infinite and each A_i is countably infinite. By the previous theorem, T is countably infinite in that case. Therefore T is at most countably infinite (may be finite if I is finite and each A_i is finite), so T is countable.

qed

Corollary 2: Let A be a countably infinite set, and let $A^n = A \times ... \times A = \{(a_1, a_2, ..., a_n) \mid a_1, a_2, ..., a_n \in A\}$. Then A^n is countably infinite.

Proof: We use induction:

Base case $n = 1$ $A^1 = A \sim J \implies A^1$ is countably infinite.

Inductive step Assume A^{n-1} is countably infinite.

$$A^n = A^{n-1} \times A = \{(b, a) \mid b \in A^{n-1}, a \in A\}.$$

$\forall b \in A^{n-1}$ $S_b = \{(b, a) \in A^n \mid a \in A\} \sim J \sim A \implies S_b$ is countably infinite. $A^n = \bigcup_{b \in A^{n-1}} S_b \sim J$ by Corollary 1, so A^n is indeed countably infinite as claimed.

qed

Corollary 3: $\mathbb{N}^n$ is countably infinite $\forall n \geq 1$.

Proof: $\mathbb{N} \sim J$, so the result follows from Corollary 2.

qed

Corollary 4: $\mathbb{Z}^n$ is countably infinite $\forall n \geq 1$.

Proof: We showed $\mathbb{Z} \sim J$, so the result follows from Corollary 2.

qed

Corollary 5: $\mathbb{Q}$ is countably infinite.

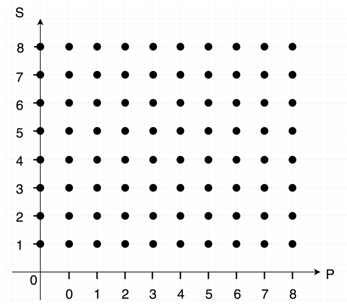
Proof: $\mathbb{Q} = \{\frac{p}{q} \mid q \neq 0, p, q \in \mathbb{Z}, (p, q) = 1 \text{ (no common factors)}\}$, but we can represent $\mathbb{Q}$ as

$$\{(p, q) \mid q \neq 0 \; p, q \in \mathbb{Z}\}/\sim \subseteq \mathbb{Z}^2,$$

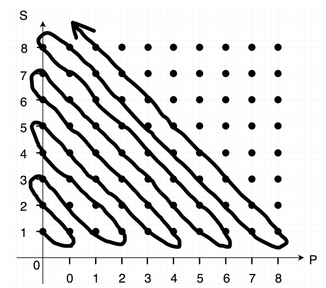
where $(p_1, q_1) \sim (p_2, q_2) \iff \frac{p_1}{q_1} = \frac{p_2}{q_2} \iff p_1 \, q_2 = p_2 \, q_1$ by cross multiplication. We also know $\mathbb{Z} \subseteq \mathbb{Q}$ (let $q = 1$). Therefore, $\mathbb{Q}$ is sandwiched between $\mathbb{Z} = \mathbb{Z}^1$ and $\mathbb{Z}^2$, both of which are countably infinite $\implies \mathbb{Q}$ is countably infinite.

qed

Remark: We can give a visual representation of the previous argument as follows:



The dots are pairs (p,q) with $q \neq 0$ $p, q \in \mathbb{Z}$, which form a lattice. We can use the snake trick from the theorem to show that the positive rational numbers $\mathbb{Q}^+ = \left\{ \frac{p}{q} \in \mathbb{Q} \;\middle|\; \frac{p}{q} > 0 \right\}$ are countably infinite.



Similarly, we can show

$$\mathbb{Q}^- = \left\{ \frac{p}{q} \in \mathbb{Q} \;\middle|\; \frac{p}{q} < 0 \right\}$$

is countably infinite.

Then $\mathbb{Q} = \mathbb{Q}^- \cup \{0\} \cup \mathbb{Q}^+$ is countably infinite by Corollary 1.

Next, show that the set of sequences of 0's and 1's is uncountably infinite. We will use this result to show other sets are uncountably infinite.

Theorem: Let A be the set of all sequences $s = \{x_1, x_2, ...\} = \{x_n\}_{n=1,2,3...}$ such that $x_n \in \{0, 1\}$ $\forall n \geq 1$. Then A is uncountably infinite.

Remark: This result is proven via a clever construction, which is due to Georg Cantor (1845 - 1918), a very famous German mathematician who invented set theory. Cantor also came up with the diagonal argument (snake trick) we used to prove a countably infinite union of countably infinite sets is countably infinite, the idea that sizes of sets should be understood via bijections ($A \sim B$ for A, B sets), as well as the notions of countably infinite and uncountably infinite.

Proof: Assume A is countably infinite $\iff$ $A = \{s_1, s_2, ...\}$,where $s_j = \{x_n^j\}_{n=1,2,...}$ for $x_n^j = 0$ or $x_n^j = 1$. We will now construct a sequence s_0 of

$0's$ and $1's$ that cannot be in the enumeration $\{s_1, s_2, ...\}$. Let s_0 be such that

$$x_j^0 = \begin{cases} 1, & \text{if } x_j^j = 0. \\ 0, & \text{if } x_j^j = 1. \end{cases}$$

In other words, s_0 differs from each s_j in the j^{th} element $\implies s_0 \notin \{s_1, s_2, \dots\}$, but s_0 is a sequence of 0's and 1's $\implies s_0 \in A \Rightarrow\Leftarrow$

qed

Corollary : The power set $\mathcal{P}(\mathbb{N})$ of $\mathbb{N}$ is uncountably infinite.

Remark: Recall our proof that if B is a set with n elements, $\#(B) = n$, then its power set $\mathcal{P}(B)$ has 2^n elements based on the "on/off" idea. For each element of B, we have the choice to include it in our subset ("on") or not to include it ("off"). Therefore, we have 2 choices for each element and $\#(B) = n$, so $\#\mathcal{P}(B) = 2^n$.

Proof: $\mathbb{N} \sim J$, so we can write $\mathbb{N} = \{x_1, x_2, \dots\}$. When we form a subset of $\mathbb{N}$, for each i, we can include x_i or leave it out. Say we represent including x_i by 1 and leaving x_i out by 0. Then each subset of $\mathbb{N}$ can be represented uniquely as a sequence of 0's and 1's. In fact, there is a one-to-one correspondence between the subsets of $\mathbb{N}$ and the sequences of 0's and 1's. Therefore $\mathcal{P}(\mathbb{N}) \sim A$, where A is the set of all sequences of 0's and 1's, but we showed in the previous theorem that A is uncountably infinite $\Rightarrow \mathcal{P}(\mathbb{N})$ is uncountably infinite.

qed

We shall also use the one-to-one correspondence with the set of sequences of 0's and 1's in order to prove $\mathbb{R}$ is uncountably infinite. The argument proceeds in two steps:

(1) We show $\mathbb{R} \sim (0, 1)$ via a cleverly chosen bijection.

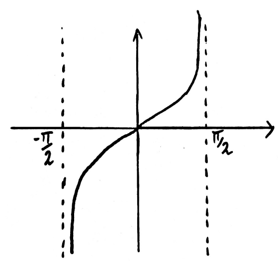
(2) We set up a correspondence between $(0, 1)$ and the set A of all sequences of 0's and 1's via a binary expansion.

Step 1 is the following proposition:

Proposition: $\mathbb{R}$ is in bijective correspondence with the interval $(0, 1)$.

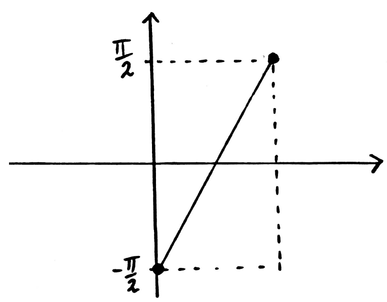
Remark: $(0, 1) \subsetneq \mathbb{R}$, but we saw infinite sets can be in one-to-one correspondence with one of their proper subsets.

Proof: STEP 1. Recall from trigonometry that $\tan : \left(-\frac{\pi}{2}, \frac{\pi}{2}\right) \to \mathbb{R}$ is a bijection. Here is the graph:



$\tan x = \frac{\sin x}{\cos x}$ and $\cos\left(-\frac{\pi}{2}\right) = \cos\left(\frac{\pi}{2}\right) = 0$. The lines $x = -\frac{\pi}{2}$ and $x = \frac{\pi}{2}$ are asymptotes of the graph.

We now use a linear function, a bijection, to show $(0,1) \sim \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$. Let $g(x) = \pi x - \frac{\pi}{2}$. Here is the graph:



The composition of two bijections is itself a bijection $\Rightarrow \tan(g(x)) = \tan\left(\pi x - \frac{\pi}{2}\right)$ is a bijection from $(0,1)$ to $\mathbb{R}$. The map we want $f : \mathbb{R} \to (0,1)$ is its inverse $f(x) = \left(\tan\left(\pi x - \frac{\pi}{2}\right)\right)^{-1}$ as the inverse of a bijection is itself a bijection.

qed

STEP 2. This step is a bit more complicated: To each $x \in (0,1)$, we want to associate $0.x_1x_2x_3\cdots$, where after the decimal $\{x_1, x_2, x_3, \dots\}$ is a sequence of 0's and 1's. In other words, we are giving a binary expansion of every $x \in (0,1)$ as

$$0.x_1x_2x_3\cdots = 0+\frac{1}{2}x_1+\frac{1}{4}x_2+\frac{1}{8}x_3+\cdots = 0+\frac{1}{2}x_1+\frac{1}{2^2}x_2+\frac{1}{2^3}x_3+\cdots = 0+\sum_{n=1}^{\infty}\frac{1}{2^n}x_n.$$

Recall that $\sum_{n=1}^{\infty} \frac{1}{2^n} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1$. This means that

$$\frac{1}{2^k} \sum_{n=1}^{\infty} \frac{1}{2^n} = \frac{1}{2^{k+1}} + \frac{1}{2^{k+2}} + \frac{1}{2^{k+3}} + \cdots = \frac{1}{2^k} \ \forall k \geq 1.$$

Thus, $0.1\underbrace{000\cdots}_{\text{all 0's}}$ and $0.0\underbrace{1111\cdots}_{\text{all 1's}}$ both represent $\frac{1}{2}$.

Similarly, any $x \in (0,1)$ that is a sum of the form $\frac{1}{2^{p_1}} + \frac{1}{2^{p_2}} + \cdots + \frac{1}{2^{p_k}}$ for $p_1, \ldots, p_k \in \mathbb{N}^*$, $p_1 < p_2 < \cdots < p_k$ has two binary representations.

Q: Can we represent $x = \frac{1}{2^{p_1}} + \frac{1}{2^{p_2}} + \cdots + \frac{1}{2^{p_k}}$ in an easier to understand form?

A: Yes, we bring the fractions to the same denominator:

$$\begin{aligned} x &= \frac{1}{2^{p_1}} + \frac{1}{2^{p_2}} + \cdots + \frac{1}{2^{p_k}} = \frac{2^{p_k - p_1}}{2^{p_k-p_1} \cdot 2^{p_1}} + \frac{2^{p_k-p_2}}{2^{p_k-p_2} \cdot 2^{p_2}} + \cdots + \frac{2^{p_k-p_{k-1}}}{2^{p_k-p_{k-1}} \cdot 2^{p_{k-1}}} + \frac{1}{2^{p_k}} \\ &= \frac{2^{p_k-p_1} + 2^{p_k-p_2} + \cdots + 2^{p_k-p_{k-1}} + 1}{2^{p_k}} = \frac{\text{odd natural number}}{\text{power of 2}} \\ &= \frac{m}{2^n} \qquad \text{for } m \in \mathbb{N} \text{ odd and } n \in \mathbb{N}^* \text{ as } p_1 < p_2 < \cdots < p_k, \end{aligned}$$

so the differences $p_k - p_1, p_k - p_2, \ldots, p_k - p_{k-1}$ are all positive integers. So the sequence in $(0,1)$ that has two decimal binary expansions is $\left\{\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}, \ldots\right\} = B$. Note that B is countably infinite as each set $B_n = \left\{0 < \frac{\text{odd}}{2^n} < 1\right\}$ is finite, $B = \bigcup_{n=1}^{\infty} B_n$ is countable by our corollary, and the countably infinite sequence $\left\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots\right\} \subseteq B$, which means the countable set B must be countably infinite. Now let us examine the binary expansions of the elements $y \in B$. $\forall y \in B$, $y = \frac{1}{2^{p_1}} + \frac{1}{2^{p_2}} + \cdots + \frac{1}{2^{p_k}}$ for $p_1, \ldots, p_k \in \mathbb{N}^*$, $p_1 < p_2 < \cdots < p_k$. The two binary expansions corresponding to $y, b_{y,1}$ and $b_{y,2}$, are of the form $0.x_1 x_2 \cdots x_{p_k-1} x_{p_k} x_{p_k+1} \cdots$ where $x_1, \ldots, x_{p_k-1}$ are common to $b_{y,1}$ and $b_{y,2}$, whereas $x_{p_k}, x_{p_k+1}, \ldots$ differ. Now

$$x_j = \begin{cases} 1 & \text{if } j = p_1, p_2, \ldots, p_{k-1} \\ 0 & \text{otherwise} \end{cases}$$

for $1 \leq j \leq p_k - 1$ is the common part corresponding to $\frac{1}{2^{p_1}} + \frac{1}{2^{p_2}} + \cdots + \frac{1}{2^{p_{k-1}}}$, whereas the difference comes from the two possible ways of representing the last term in the sum $\frac{1}{2^{p_k}}$ namely $10000\cdots$ or $01111\cdots$. Therefore, $b_{y,1}$ has $x_{p_k} = 1$ and $x_j = 0 \ \forall j > p_k$ (corresponding to $1000\cdots$), whereas $b_{y,2}$ has $x_{p_k} = 0$ and $x_j = 1 \ \forall j > p_k$ (corresponding to $0111\cdots$). Let $s_{y,1} \in A$ be the sequence corresponding to $b_{y,1}$ in A, the set of all sequences of 0's and 1's, i.e. if $b_{y,1} = 0.x_1 x_2 x_3 \cdots$ $s_{y,1} = \{x_1, x_2, x_3, \ldots\}$. Let $s_{y,2} \in A$ be the sequence corresponding to $b_{y,2}$. We now define $B_1 = \{b_{y,1} \mid y \in B\}$, $B_2 = \{b_{y,2} \mid y \in B\}$, $A_1 = \{s_{y,1} \mid y \in B\}$, $A_2 = \{s_{y,2} \mid y \in B\}$. B is in one-to-one correspondence to

B_1, B_2, A_1, A_2 by construction, so $B \sim B_1$, $B \sim B_2$, $B \sim A_1$, $B \sim A_2$, but B is countably infinite $\Rightarrow A_1, A_2, B_1, B_2$ are all countably infinite.

We have just one more observation to make regarding the correspondence between sequences of 0's and 1's in A and elements of $(0,1)$, namely that the zero sequence $\{0,0,\dots\}$ corresponds to the binary expansion $0.000\cdots = 0 \notin (0,1)$ since $(0,1) = \{x \in \mathbb{R} \mid 0 < x < 1\}$ and the one sequence $\{1,1,1,\dots\}$ corresponds to the binary expansion $0.1111\cdots = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = \sum_{n=1}^{\infty} \frac{1}{2^n} = 1 \notin (0,1)$. Now we can finally prove that $(0,1)$ is uncountably infinite.

Proposition: $(0,1)$ is uncountably infinite.

Proof: We define a map $f : (0,1) \to \{0.x_1x_2x_3\cdots \mid x_j \in \{0,1\}\ \forall j \geq 1\}$ as follows

$$f(y) = \begin{cases} b_{y,1} & \text{if } y \in B \leftarrow \text{ The first of the two possible binary expansions} \\ 0.x_1x_2x_3\cdots & \text{if } y \notin B \leftarrow \text{ The unique binary expansion.} \end{cases}$$

By our previous discussion, f is a bijection as defined

$$\Rightarrow (0,1) \sim \{0.x_1x_2x_3\cdots \mid x_j \in \{0,1\}\ \forall j \geq 1\}.$$

Also, by our previous discussion

$$\{0.x_1x_2x_3\cdots \mid x_j \in \{0,1\}\ \forall j \geq 1\} \sim A \setminus (A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\}),$$

where A is the set of all sequences of 0's and 1's , $\{0,0,\dots\}$ is the constant zero sequence and $\{1,1,\dots\}$ is the constant one sequence.

$$(0,1) \sim A \setminus (A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\})$$

since $\sim$ is transitive (it is an equivalence relation).

A_2 is countably infinite, so $A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\}$ is countably infinite (we've added two elements to A_2, so it stays countably infinite). In a previous theorem, we proved A is uncountably infinite. Thus, $A \setminus (A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\})$ is of the form $\{\text{uncountably infinite set}\} \setminus \{\text{countably infinite set}\}$. I claim $A \setminus (A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\})$ is uncountably infinite. Indeed, let $\tilde{A} = A \setminus (A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\})$. $A = \tilde{A} \cup A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\}$. Assume $\tilde{A}$ is countable, then A is the union of a countable set with a countably infinite set, hence A is countable $\Rightarrow\Leftarrow$ Therefore, $\tilde{A} = A \setminus (A_2 \cup \{0,0,\dots\} \cup \{1,1,\dots\})$ is uncountably infinite, but $\tilde{A} \sim (0,1)$ ($\sim$ is symmetric) $\Rightarrow (0,1)$ is uncountably infinite.

qed

Theorem: $\mathbb{R}$ is uncountably infinite.

Proof: By the previous proposition, $(0,1)$ is uncountably infinite. By the proposition before this one, $(0,1) \sim \mathbb{R} \Rightarrow \mathbb{R}$ is uncountably infinite.

qed

Under the equivalence relation $\sim$ of bijective correspondence, we have shown $\mathbb{N}, \mathbb{N}^*, \mathbb{N}^n\ \forall n \geq 1, \mathbb{Z}^n\ \forall n \geq 1, \mathbb{Q}^n\ \forall n \geq 1 \in [\mathbb{N}]$ all of these are countably infinite, and A (all sequences of 0's and 1's), $\mathcal{P}(\mathbb{N})$, and $[\mathbb{R}]$ are uncountably infinite.

A very natural question to ask at this point:

Q: Is there some intermediate equivalence class in size between $[\mathbb{N}]$ and $[\mathbb{R}]$?

A: The *Continuum hypothesis* (CH) gives a negative answer to this question.

The Continuum hypothesis (CH): There is no set whose cardinality is strictly between the cardinality of the integers and the cardinality of real numbers.

Cardinality means size or number of elements.

Georg Cantor stated CH in 1878, believed it was true, but could not prove it. It became one of the crucial open problems in mathematics. Hilbert stated it in 1900 first among the 23 problems that were supposed to hold the key for the advancement of mathematics. Everybody expected CH to be either true or false. The answer is that CH is independent from the standard axiomatic system used in mathematics called ZFC (Zermelo-Frankel with the Axiom of Choice). In other words, CH cannot be proven either true or false when working within the axiomatic framework of ZFC. In 1940 Kurt Gödel showed CH cannot be proven false within ZFC. In 1963 Paul Cohen showed CH cannot be proven true within ZFC and won the Fields Medal (like the Nobel Prize for mathematics) for his work.

10.1 Applications of Countability of Sets to Formal Languages

Task: Figure out the size of the set of all languages over a finite alphabet and the size of all regular languages over a finite alphabet.

Let A be a finite alphabet, i.e. $A = \{a_1, a_2, \ldots, a_n\}$. Recall that $A^* = \displaystyle\bigcup_{j=0}^{\infty} A^j$ is the set of all possible words in the alphabet A. A^j is the set of all words of length j in the alphabet A.

Q: What is $\#(A^j)$, the size (cardinality) of A^j?

A: If $j = 0$, $A^0 = \{\epsilon\}$, where ϵ is the empty word, so $\#(A^0) = 1$. In general, we have n choices of letters in the first position, n choices of letters $(a_1, \ldots, a_n)$ in the second position, and so on up to the j-th position. In total, we have $\underbrace{n \times n \times \cdots \times n}_{\text{j times}} = n^j$ possibilities.

Therefore, $\#(A^j) = n^j$. Note that when $j = 0$, $n^0 = 1 = \#(A^0) = \#(\{\epsilon\})$.

Theorem: If A is a finite alphabet, then the set of all words over A

$$A^* = \bigcup_{j=0}^{\infty} A^j$$

is countably infinite.

Proof: We showed A^j is a finite set for each j. In fact, $\#(A^j) = n^j$. $\bigcup_{j=0}^{\infty} A^j$ is therefore a countably infinite union of disjoint finite sets (note that $A^j \cap A^k = \emptyset$ if $j \neq k$ as no words of length j can be of length k if $j \neq k$). By Corollary 1 to the theorem that a countably infinite union of countably infinite sets is countably infinite, $A^* = \bigcup_{j=0}^{\infty} A^j$ is countable. Since the sets A^j are mutually disjoint and there is a countably infinite number of them, A^* cannot be finite, so A^* is countably infinite.

qed

Corollary I: If A is a finite alphabet, then the set of all languages over A is uncountably infinite.

Proof: Recall that a language L is <u>any</u> subset of words in the alphabet A, hence L is any subset of A^*. Therefore, the set of all languages over A is precisely $\mathcal{P}(A^*)$, the power set of A^*. We showed in the previous theorem that A^* is countably infinite, i.e. $A^* \sim \mathbb{N} \Rightarrow \mathcal{P}(A^*) \sim \mathcal{P}(\mathbb{N})$, but we previously proved $\mathcal{P}(\mathbb{N})$ is uncountably infinite by putting it in one-to-one correspondence with the set of all sequences of 0's and 1's $\Rightarrow \mathcal{P}(A^*)$ is uncountably infinite.

qed

Corollary II: The set of all programs in any programming language is countably infinite.

Proof: For any programming language, a program is a finite string over a finite alphabet, the set of characters allowable in that programming language. Let us call this finite alphabet A. Then the set of all programs in the given programming language is A^*. Since $A^* \sim \mathbb{N}$ as proven in the theorem, the set of all programs is countably infinite.

qed

Recall:

Theorem: A language over a finite alphabet A is regular $\Leftrightarrow$ it is given by a regular expression.

Recall the definition of a regular expression.

Definition: Let A be an alphabet

1. $\emptyset, \epsilon$, and all elements of A are regular expressions;
2. If w and w' are regular expressions, then $w \circ w'$, $w \cup w'$, and w^* are regular expressions.

Note that regular expressions sometimes have parentheses in order to change the priority of operations $*$ (Kleene star), $\circ$ (concatenation), and $\cup$ (union). Therefore, any regular expression over the alphabet A is a string over the enlarged alphabet $\tilde{A} = A \cup \{\text{“}\emptyset\text{”}, \text{“}\epsilon\text{”}, \text{“} * \text{”}, \text{“} \circ \text{”}, \text{“} \cup \text{”}, \text{“(”}, \text{“)”}\}$. I put quotation marks to denote the fact that $\emptyset, \epsilon, *, \circ, \cup, (,)$ are now viewed as letters of the enlarged alphabet $\tilde{A}$.

Theorem: The set of all regular languages over a finite alphabet A is countably infinite.

Proof: Since the alphabet A is finite, the enlarged alphabet

$$\tilde{A} = A \cup \{\text{“}\emptyset\text{”}, \text{“}\epsilon\text{”}, \text{“} * \text{”}, \text{“} \circ \text{”}, \text{“} \cup \text{”}, \text{“(”}, \text{“)”}\}$$

is also finite. By the theorem proven earlier, $\tilde{A}^*$ is therefore countably infinite. A regular language then is given by a regular expression, which is a string over the enlarged alphabet $\tilde{A}$, hence an element of $\tilde{A}^*$. Therefore, the set of all regular languages over the alphabet A is countably infinite.

qed

Moral of The Story

Given a finite alphabet A, the set of regular languages (which is countably infinite) is much smaller than the set of all languages over A (which is uncountably infinite). Therefore, regular languages constitute a special category within the set of all languages over a given alphabet.

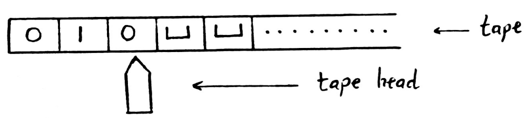
11 Turing Machines

Task: Look at a more realistic model of a computer than a finite state acceptor.

Turing machines were first proposed by Alan Turing in 1936 in order to explore the theoretical limits of computation. We shall see that certain problems cannot be solved even by a Turing machine and are thus beyond the limits of computation.

A Turing machine is similar to a finite state acceptor but has unlimited memory given by an infinite tape (we mean countably infinite here). The infinite tape is divided into cells each of which holds a character of a tape alphabet. The Turing machine is equipped with a tape head that can read and write symbols on the tape and move left (back) or right (forward) on the tape. Initially, the tape contains only the input string and is blank everywhere else. To store information, the Turing machine can write this information on the tape. To read information that it has written, the Turing machine can move its head back over it. The Turing machine continues computing until it decides to produce an output. The outputs "accepts" and "rejects" are obtained by entering accepting or rejecting states respectively. It is also possible for the Turing machine to go on forever never stopping if it does not enter either an accepting or a rejecting state.

Illustration of a Turing Machine



⊔ the blank symbol is part of the tape alphabet

Example. Let $A = \{0, 1\}$ and let $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$. We know L is not a regular language, so there is no finite state acceptor that can recognize it, but there is a Turing machine that can.

Initial state of the tape: Input string of 0's and 1's, then infinitely many blanks.

Idea of this Turing machine: Change a 0 to an x, and then a 1 to a y until either:

- all 0's and 1's have been matched, hence ACCEPT
- the 0's and 1's do not match or the string does not have the form 0^*1^*, hence REJECT.

Algorithm

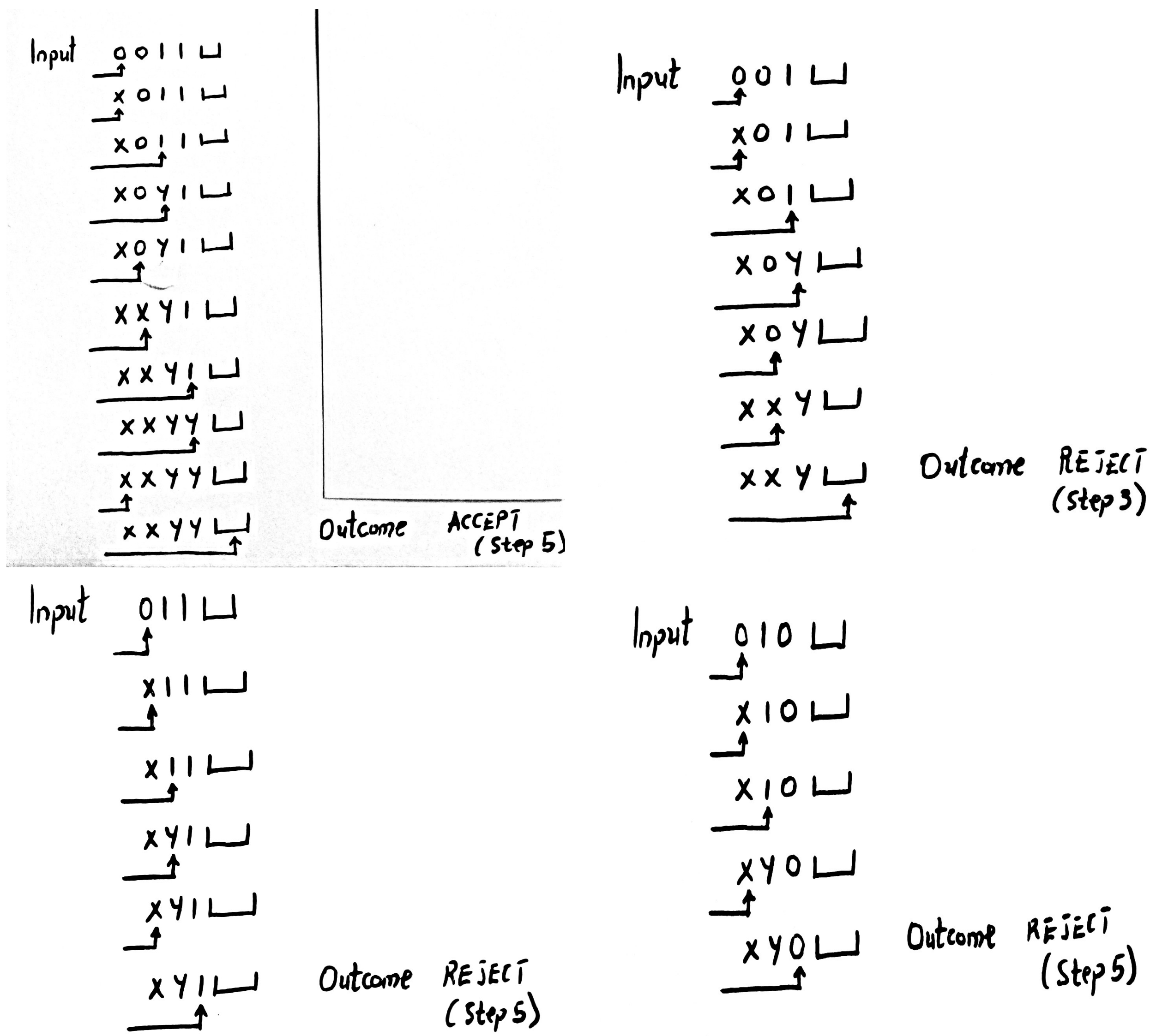
The tape head is initially positioned over the first cell.

1. If anything other than 0 is in the first cell, then REJECT.
2. If 0 is in the cell, then change 0 to x.
3. Move right to the first 1. If none, then REJECT.
4. Change 1 to y.

5. Move left to the leftmost 0. If none, move right looking for either a 0 or a 1. If either 0 or 1 is found before the first blank symbol, then REJECT; otherwise, ACCEPT.

6. Go to step 2.

Let's process some strings:



Note that we have the following:

- $A = \{0, 1\}$ the input alphabet
- $\sqcup \notin A$, where $\sqcup$ is the blank symbol.
- $\tilde{A} = \{0, 1, x, y, \sqcup\}$ is the tape alphabet.
- $\mathcal{S}$ a set of states.
- Note also that the tape head is moving right or left, so we also need to have a set $\{L, R\}$ with L for left and R for right specifying where the tape head goes.

Recall that a finite state acceptor was given by $(\mathcal{S}, A, i, t, F)$ with the transition mapping being given by $t : \mathcal{S} \times A \to \mathcal{S}$. Recall that $\mathcal{S}$ is the set of states, A is the alphabet, i is the initial state, t is the transition mapping, and F is the set of finishing states.

By contrast, for a Turing machine the transition mapping is of the form

$$t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}.$$

Having the tape alphabet $\tilde{A}$ instead of A in the codomain of the transition mapping indicates that the Turing machine can write. Having the set $\{L, R\}$ in the codomain of the transition mapping indicates that the Turing machine's head can move left or right.

Definition: A Turing machine is a 7-tuple $(\mathcal{S}, A, \tilde{A}, i, t, S_{accept}, S_{reject})$, where $\mathcal{S}$,A, $\tilde{A}$ are finite sets and

(a) $\mathcal{S}$ is the set of states.

(b) A is the input alphabet not containing the blank symbol $\sqcup$.

(c) $\tilde{A}$ is the tape alphabet, where $\sqcup \in \tilde{A}$ and $A \subseteq \tilde{A}$.

(d) $t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}$ is the transition mapping

(e) i is the initial state of the machine.

(f) $S_{accept} \in \mathcal{S}$ is the accept state.

(g) $S_{reject} \in \mathcal{S}$ is the reject state and $S_{accept} \neq S_{reject}$.

Remarks about the definition

1). Since A does not contain the blank symbol $\sqcup$, the first blank on the tape marks the end of the input string.

2). If the Turing machine is instructed to move left, and it has reached the first cell of the tape, then it stays at the first cell.

3). The Turing machine continues to compute until it enters either the accept or reject states at which point it halts. If it does not enter either, then it goes on forever.

Example (considered again): $A = \{0, 1\}$ $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$

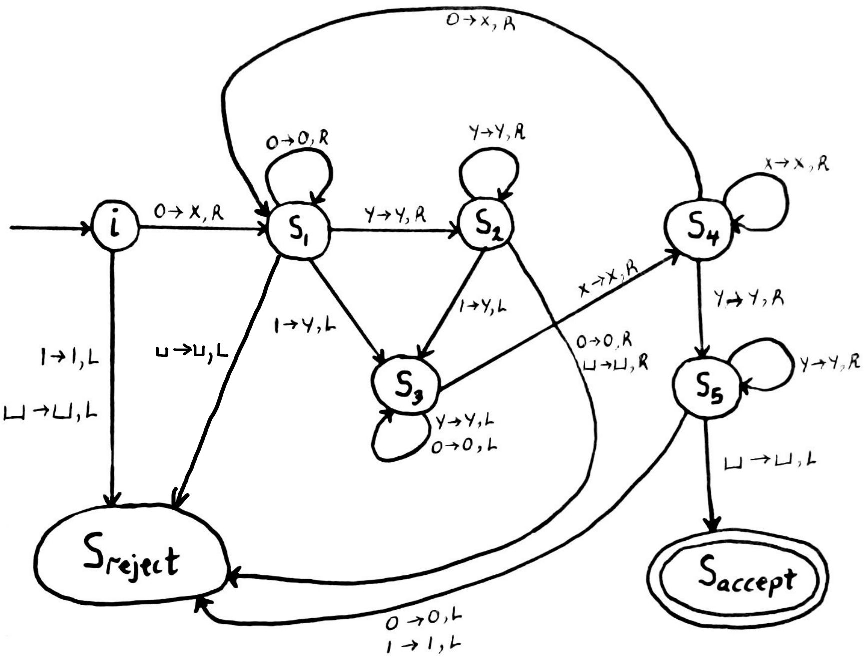
We need to be able to write down the transition mapping hence the set of states $\mathcal{S}$. Recall that what we gave was an algorithm, and using that algorithm we processed strings to convince ourselves that the corresponding Turing machine behaved correctly.

Here is the algorithm again:

The tape head is initially positioned over the first cell.

1. If anything other than 0 is in the first cell, then REJECT.
2. If 0 is in the cell, then change 0 to x.
3. Move right to the first 1. If none, then REJECT.
4. Change 1 to y.
5. Move left to the leftmost 0. If none, move right looking for either a 0 or a 1. If either 0 or 1 is found before the first blank symbol, then REJECT; otherwise, ACCEPT.
6. Go to step 2.

Before we can write down the set of states $\mathcal{S}$ or the transition mapping t, let us draw a transition diagram, which is the Turing machine equivalent to drawing a finite state acceptor when we looked at regular languages.



$i \to S_{reject}$ represents step 1 of the algorithm.

$i \to S_1$ and $S_4 \to S_1$ represent step 2 of the algorithm ($i \to S_1$ at the first pass through the string; $S_4 \to S_1$ at subsequent passes).

$S_1 \to S_1$, $S_1 \to S_2$, $S_2 \to S_2$ represent the first part of step 3.

$S_1 \to S_{reject}$ and $S_2 \to S_{reject}$ represent the second part of step 3.

$S_1 \to S_3$ and $S_2 \to S_3$ represent step 4.

$S_3 \to S_3$ and $S_3 \to S_4$ represent the first sentence in step 5.

$S_4 \to S_4$, $S_4 \to S_5$, $S_5 \to S_5$ represent the second sentence in step 5.

$S_5 \to S_{reject}$ is the first half of the third sentence in step 5.

$S_5 \to S_{accept}$ is the second half of the third sentence in step 5.

$S_4 \to S_1$ represents step 6.

We have accounted for all pieces of our algorithm, therefore, we have written down a Turing machine, where $A = \{0, 1\}$, $\tilde{A} = \{0, 1, x, y, \sqcup\}$,

$$\mathcal{S} = \{i, S_{accept}, S_{reject}, S_1, S_2, S_3, S_4, S_5\},$$

i is the initial state; $S_{accept} \in \mathcal{S}$ is the accept state; $S_{reject} \in \mathcal{S}$ is the reject state.

We just have to write down the transition mapping $t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}$:

$$t(i, 0) = (S_1, x, R)$$

$$t(i, 1) = (S_{reject}, 1, L)$$

$$t(i, \sqcup) = (S_{reject}, \sqcup, L)$$

The above transitions are the only 3 transitions possible out of state i, but $t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}$ so technically, to write down the full transition mapping, we must assign triplets in $\mathcal{S} \times \tilde{A} \times \{L, R\}$ even to input from $\tilde{A}$ that cannot occur when in i :

$$t(i, x) = (S_{reject}, x, L)$$

$$t(i, y) = (S_{reject}, y, L)$$

We assign S_{reject}, same element of $\tilde{A}$, and one of the allowable tape head directions. Technically, the Turing machine halts when it enters either an accepting state (S_{accept}) or a rejecting state (S_{reject}), so in practice we can define $\tilde{\mathcal{S}} = \{i, S_1, S_2, S_3, S_4, S_5\} = \underbrace{\mathcal{S} \setminus \{S_{accept}, S_{reject}\}}_{\text{set of nonhalting states}}$ and $t : \tilde{\mathcal{S}} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}$, so we avoid writing down the transitions from S_{accept} and S_{reject}. We only have states S_1, S_2, S_3, S_4, and S_5 left.

$$t(S_1, 0) = (S_1, 0, R)$$

$$t(S_1, y) = (S_2, y, R)$$

$$t(S_1, 1) = (S_3, y, L)$$

$$t(S_1, \sqcup) = (S_{reject}, \sqcup, R)$$

These four transitions are on the diagram.

$$t(S_1, x) = (S_{reject}, x, R)$$

This last transition is not on the diagram; cannot occur, so added for completeness.

$$t(S_2, y) = (S_2, y, R)$$

$$t(S_2, 1) = (S_3, y, L)$$

$$t(S_2, 0) = (S_{reject}, 0, R)$$

$$t(S_2, \sqcup) = (S_{reject}, \sqcup, R)$$

These four transitions are on the diagram; can occur.

$t(S_2, x) = (S_{reject}, x, R) \leftarrow$ not on the diagram; cannot occur; added for completeness

$$t(S_3, y) = (S_3, y, L)$$

$$t(S_3, 0) = (S_3, 0, L)$$

$$t(S_3, x) = (S_4, x, R)$$

on the diagram; can occur.

$$t(S_3, \sqcup) = (S_{reject}, \sqcup, R)$$

$$t(S_3, 1) = (S_{reject}, 1, R)$$

not on the diagram; cannot occur; added for completeness.

$$t(S_4, x) = (S_4, x, R)$$

$$t(S_4, y) = (S_5, y, R)$$

$$t(S_4, 0) = (S_1, x, R)$$

on the diagram; can occur.

$$t(S_4, 1) = (S_{reject}, 1, R)$$

$$t(S_4, \sqcup) = (S_{reject}, \sqcup, R)$$

not on the diagram; cannot occur; added for completeness.

$$t(S_5, y) = (S_5, y, R)$$

$$t(S_5, \sqcup) = (S_{accept}, \sqcup, L)$$

$$t(S_5, 0) = (S_{reject}, 0, L)$$

$$t(S_5, 1) = (S_{reject}, 1, L)$$

on the diagram; can occur.

$t(S_5, x) = (S_{reject}, x, L) \leftarrow$ not in the diagram; cannot occur; added for completeness

Moral of the Story

The transition mapping is a very inefficient way of specifying a Turing machine as a lot of transitions cannot occur unlike what we saw for a finite state acceptor, where the input alphabet was exactly the alphabet of the language. Here $A \subset \tilde{A}$. Therefore, we will specify a Turing machine via either an algorithm or the transition diagram only.

To figure out which languages are recognized by a Turing machine, we need to introduce the notion of a configuration. As a Turing machine goes through its computations, changes take place in

(1) the state of the machine

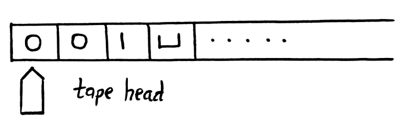
(2) the tape contents

(3) the tape head location.

A setting of these three items is called a configuration.

Representing configurations: We represent a configuration as uS_iv, where u, v are strings in the tape alphabet $\tilde{A}$ and S_i is the current state of the machine. The tape contents are then the string uv and the current location of the tape head is on the first symbol of v. The assumption here is that the tape contains only blanks after the last symbol in v.

Example: $\epsilon i 001$ is the configuration



as we start examining the string 001 in our previous example of a Turing machine.

Definition: Let C_1, C_2 be two configurations of a given Turing machine. We say that the configuration C_1 yields the configuration C_2 if the Turing machine can go from C_1 to C_2 in one step.

Example: If S_i, S_j are states, u and v are strings in the tape alphabet $\tilde{A}$, and $a, b, c \in \tilde{A}$. A configuration $C_1 = uaS_ibv$ yields a configuration $C_2 = uS_jacv$ if the transition mapping t specifies a transition $t(S_i, b) = (S_j, c, L)$. In other words, the Turing machine is in state S_i, it reads character b, writes character c in its place, enters state S_j, and its head moves left.

Types of Configurations

Initial configuration with input u is iu, which indicates that the machine is in the initial state i with its head at the leftmost position on the tape (which is the reason why this configuration has no string left of the state).

Accepting configuration $uS_{accept}v$ for $u, v \in \tilde{A}^*$ (u, v string in $\tilde{A}$), namely the machine is in the accept state.

Rejecting configuration $uS_{reject}v$ for $u, v \in \tilde{A}^*$, namely the machine is in the reject state.

Halting configurations yield no further configurations; no transitions are defined out of their states. Accepting and rejecting configurations are examples of halting configurations.

Definition: A Turing machine M accepts input $w \in A^*$ (string over the input alphabet A) if $\exists$ sequences of configurations $C_1, C_2, \ldots, C_k$ such that:

1. C_1 is the start configuration with input w.
2. Each C_i yields C_{i+1} for $i = 1, 2, \ldots, k-1$.
3. C_k is an accepting configuration.

Definition: Let M be a Turing machine. $L(M) = \{w \in A^* \mid M \text{ accepts } w\}$ is the language recognized by M.

Definition: A language $L \subset A^*$ is called Turing-recognizable if $\exists M$ a Turing machine that recognizes L, i.e. $L = L(M)$.

NB: Some textbooks use the terminology recursively enumerable language (RE language) instead of Turing-recognizable.

Turing-recognizable is not necessarily as strong a notion as we might need because a Turing machine can accept, reject, or loop.

Looping is any single or complex behaviour that does not lead to a halting state. The problem with looping is that the user does not have infinite time. It can be difficult to distinguish between looping or taking a very long time to compute. We thus prefer deciders.

Definition: A decider is a Turing machine that enters either an accept state or a reject state for every input in A^*.

Definition: A decider that recognizes some language $L \subset A^*$ is said to decide that language.

Definition: A language $L \subset A^*$ is called Turing-decidable if $\exists$ a Turing machine M that decides L.

NB: Some textbooks use the terminology recursive language instead of Turing-decidable.

Example. $L = \{0^m 1^m \mid m \in \mathbb{N}, m \geq 1\}$ is Turing-decidable because the Turing machine we built that recognized it was in fact a decider (check again to convince yourself that machine did not loop).

Turing-decidable $\Rightarrow$ Turing-recognizable, but the converse is not true: Turing-recognizable **does not imply** Turing-decidable. We will give an example of a language that is Turing-recognizable, but NOT Turing-decidable before the end of the term.

11.1 Variants of Turing machines

Task: Explore variants of the original set-up of a Turing machine and show they do not enlarge the set of Turing-recognizable languages.

(A). Add "stay put" to the list of allowable directions

Say instead of allowing just $\{L, R\}$ (the tape head moves left or right), we also allow the "stay put" option (no change in the position of the tape head). Thus, the transition mapping is defined as $t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R, N\}$, where N is for "no movement" (stay put) instead of $t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}$. We realize N is the same as $L + R$ or $R + L$ (move the tape head left by one all, then right by one all or the other way around) $\Rightarrow$ variant (A) yields no increase in computational power.

(B). Multitype Turing machine

We allow the Turing machine to have several tapes, each with its own tape head for reading and writing. Initially, the input is on tape 1, and the others are blank. The transition mapping then must allow for reading, writing, and moving the tape heads on some or all of the tapes simultaneously. If k is the number tapes, then the transition mapping is defined as $t : \mathcal{S} \times \tilde{A}^k \to \mathcal{S} \times \tilde{A}^k \times \{L, R, N\}^k$ since one of the tape heads or more might not move for some transitions, we make use of the option N ("no movement") besides left and right. Multitype Turing machines seem more powerful than ordinary (simple-tape) ones, but that is not the case.

Definition: We call two Turing machines M_1 and M_2 equivalent if $L(M_1) = L(M_2)$, namely if they recognize the same language.

Theorem. Every multitype Turing machine has an equivalent single-tape Turing machine.

Sketch of proof. Let $M^{(k)}$ be a Turing machine with k tapes. We will simulate it with a single-tape Turing machine $M^{(1)}$ constructed as follows. We add # to the tape alphabet $\tilde{A}$ and use it to separate the contents of the different tapes. $M^{(1)}$ also needs to keep track of the locations of the tape heads of $M^{(k)}$. It does so by adding a dot to the character to which a tape head is pointing. We thus only need to enlarge the tape alphabet $\tilde{A}$ by allowing a version with a dot above for every character in $\tilde{A}$ apart form # and the blank symbol $\sqcup$.

q.e.d

Corollary. A language L is Turing-recognizable $\Leftrightarrow$ some multitype Turing machine recognizes L.

Proof. " $\Rightarrow$ " A language L is Turing-recognizable if $\exists M$ a single-tape Turing machine that recognizes it. A single-tape Turing machine is a special type of a multitype Turing machine, so we are done.

" $\Leftarrow$ " follows from the previous theorem.

q.e.d

(C). A nondeterministic Turing machine

Just like a nondeterministic finite state acceptor, a nondeterministic Turing machine may proceed according to different possibilities, so its computation is a tree, where each branch corresponds a different possibility. The transition mapping of such a nondeterministic Turing machine is given by

$$t : \mathcal{S} \times \tilde{A} \to \mathcal{P}(\mathcal{S} \times \tilde{A} \times \{L, R\}),$$

where $\mathcal{P}(\mathcal{S} \times \tilde{A} \times \{L, R\})$ shows we have different possibilities on how to proceed.

Theorem. Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Idea of the proof. We construct a deterministic Turing machine that simulates the nondeterministic one by trying out all possible branches. If it finds an accept state on one of the computational branches, it accepts the input; otherwise, it loops.

Corollary. A language is Turing-recognizable $\Leftrightarrow$ some nondeterministic Turing machine recognizes it.

Proof. " $\Rightarrow''$ A deterministic Turing machine is a nondeterministic one, so this direction is obvious.

“ $\Leftarrow$ ” follows from the previous theorem.

(D). Enumerators

As we saw, a Turing-recognizable language is called in some textbooks a recursively enumerable language. The term comes from a variant of a Turing machine called an enumerator. Loosely, an enumerator is a Turing machine with an attached printer. The enumerator prints out the language L it accepts as a sequence of strings. Note that the enumerator can print out the strings of the language in any order and possibly with repetitions.

Theorem. A language L is Turing-recognizable $\Leftrightarrow$ some enumerator enumerates (outputs) L.

Proof. “ $\Leftarrow$ ” Let E be the enumerator. We construct the following Turing machine M :

$M =$ on input w

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, accept w.

Thus, M accepts exactly those strings that appear on E's list and no others, hence exactly L.

“ $\Rightarrow$ ” Let M be a Turing machine that recognizes L. We would like to construct an enumerator E that outputs L. Let A be the alphabet of L, i.e. $L \subset A^*$. In the unit on countability, we proved A^* is countably infinite (note that the alphabet A is always assumed to be finite), so A^* has an enumeration as a sequence $A^* = \{w_1, w_2, \ldots\}$.

$E =$ Ignore the input

1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input $w_1, w_2, \ldots, w_i$.
3. If any computations accept, print out the corresponding w_j.

Every string accepted by M will eventually appear on the list of E, and once it does, it will appear infinitely many times because M runs from the beginning on each string for each repetition of step 1. Note that each string accepted by M is accepted in some finite number of steps, say k steps, so this string will be printed on E's list for every $i \geq k$.

q.e.d

Moral of the Story

The single-tape Turing machine we first introduced is as powerful as any variants we can think of.

11.2 Algorithms

Task. Use Hilbert's 10th problem to give an example of something that is Turing-recognizable but not Turing-decidable.

We saw that the Continuum Hypothesis of Cantor was the 1st of Hilbert's 23 problems in 1900 at the International Congress of Mathematicians.

Hilbert's 10th Problem

Find a procedure that tests whether a polynomial in several variables with integer coefficients has integer roots.

Example: $p(x, y) = 2x^2 - xy - y^2$ is a polynomial in 2 variables (x and y) with integer coefficients $(2, -1, -1)$ that has integer roots.

$$p(1, 1) = 2 \cdot 1^2 - 1 \cdot 1 - 1^2 = 0$$

so $x = 1 = y,\ 1 \in \mathbb{Z}$ is a solution. Hilbert's problem asked how to find integer roots via a set procedure. In 1936, independently, Alonzo Church invented λ-calculus to define algorithms, while Alan Turing invented Turing machines. Church's definition was shown to be equivalent to Turing's. This equivalence says

$$\boxed{\begin{array}{l}\text{Intuitive notion}\\ \text{of algorithms}\end{array}} = \boxed{\begin{array}{l}\text{Turing machine}\\ \text{algorithms}\end{array}}$$

and is known as the Church-Turing thesis. It led to the formal definition of an algorithm and eventually to resolving in the negative Hilbert's 10th problem. Using previous work by Martin Davis, Hilary Putman, and Julia Robinson, Yuri Matijasevic proved in 1970 that there is no algorithm, which can decide whether a polynomial has integer roots. As we shall see now, Hilbert's 10th problem is an example of a problem that is Turing-recognizable but not Turing-decidable. Let $D = \{p \mid p \text{ is a polynomial with an integer root}\}$. Hilbert's 10th problem is asking whether D is decidable. Let us simplify the problem to the one variable case:

$$D_1 = \{p \mid p \text{ is a polynomial in variable } x \text{ with an integer root}\}.$$

We can easily write down a Turing machine M_1 that recognizes D_1 :

$M_1 =$ on input p, where p is a polynomial in x

1. Evaluate p with x set successively to the values $0, 1, -1, 2, -2, \ldots$ If at any value the polynomial evaluates to 0, accept.

If p does indeed have an integer root, M_1 will eventually find it and accept p. If p does not have an integer root, then M_1 will run forever.

Principle behind M_1: $\mathbb{Z} \sim \mathbb{N}$, i.e. $\mathbb{Z}$ is countably infinite, so we can write $\mathbb{Z}$ as a sequence (enumerate it)

$$\mathbb{Z} = \{s_1, s_2, \dots\} = \{s_i\}_{i=1,2,\dots} = \{0, 1, -1, 2, -2, \dots\}$$

Now, consider polynomials of n variables $p(x_1, \dots, x_n)$. We want to find $(x_1, \dots, x_n) \in \mathbb{Z}^n$ such that $p(x_1, \dots, x_n) = 0$, so in general Hilbert's 10th problem is asking us to build a decider for

$$D_n = \{p(x_1, \dots, x_n) \mid \exists (x_1, \dots, x_n) \in \mathbb{Z}^n \text{ such that } p(x_1, \dots, x_n) = 0\}.$$

We can easily build a Turing machine M_n that recognizes D_n using the principle behind M_1 : $\mathbb{Z}^n$ is countably infinite because it is the Cartesian product of a countably infinite set with itself n times. Since $\mathbb{Z}^n$ is countably infinite, we can enumerate it, namely write it as a sequence $\mathbb{Z}^n = \{c_1, c_2, \dots\}$, where $c_i = (x_1^{(i)}, \dots, x_n^{(i)})$.

Then M_n = on input p, where p is a polynomial in $x_1, x_2, \dots, x_n$

1. Evaluate p with $(x_1, \dots, x_n)$ set successively to the values $c_1, c_2, \dots$ If at any value $c_i = (x_1^{(i)}, \dots, x_n^{(i)})$, $p(x_1^{(i)}, \dots, x_n^{(i)}) = 0$, accept p.

If p has an integer root $(x_1^{(i)}, \dots, x_n^{(i)}) \in \mathbb{Z}^n$, then the Turing machine accepts; otherwise, it goes on forever (it loops) just like M_1. It turns out M_1 can be converted into a decider because if $p(x)$ of one variable has a root, then that root must fall between certain bounds, so the checking of possible values can be made to terminate when those bounds are reached. By contrast, no such bounds exist when the polynomial is of two variables or more $\Rightarrow M_n$ for $n \geq 2$ CANNOT be converted into a decider. This is what Matijasevic proved.

11.3 Decidable Languages

Task: Explore whether certain languages are decidable that come from our study of formal languages and grammars.

(1). The acceptance problem for deterministic finite state acceptors (DFA's)

Test whether a given deterministic finite state acceptor (DFA) B accepts a given string w.

We can rewrite the acceptance problem as a language:

$$L_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts inputs string } w\}$$

Theorem. L_{DFA} is a Turing decidable language.

Proof. We construct a Turing machine M that decides L_{DFA} as follows:

M = on input $\langle B, w\rangle$, where B is a DFA and w is a string

1. Simulate B on input w.

2. If the simulation ends in an accept state of B, accept $\langle B, w\rangle$. If it ends in a non-accepting state of B, reject $\langle B, w\rangle$.

We need to provide more details on the input $\langle B, w\rangle$. B is a finite state acceptor, which we defined as a 5-tuple $(\mathcal{S}, A, i, t, F)$ with $\mathcal{S}$ the set of states, A the alphabet, i the initial state, t the transition mapping $t : \mathcal{S} \times A \to \mathcal{S}$, and F the set of finishing states. The string w is over the alphabet A, so the pair $\langle B, w\rangle$ as input for our Turing machine is in fact $(\mathcal{S}, A, i, t, F, w)$. The Turing machine M starts in the configuration $\epsilon i w$. If $w = uv$, where $u \in A$ is the first character in the word w and if $t(i, u) = s$, then the next configuration of the Turing machine M is usv, i.e. the new state corresponds to the state s in which B enters from the initial state i upon receiving input character u and the tape head has moved right past u ready to examine the second character of w. Once the string w has been completely processed, then the configuration of the Turing machine is $ws_w\epsilon$. If the final state s_w where we ended up is an accepting state, i.e. $s_w \in F$, then we accept $\langle B, w\rangle$; otherwise, we reject $\langle B, w\rangle$.

q.e.d

(2). <u>The acceptance problem for nondeterministic finite state acceptors (NFA's)</u>

Test whether a given nondeterministic finite state acceptor B accepts a given string w.

Rewrite this acceptance problem as a language:

$$L_{NFA} = \{\langle B, w\rangle \mid B \text{ is a NFA that accepts input string } w\}.$$

<u>Theorem</u>. L_{NFA} is a Turing-decidable language.

<u>Proof</u>. This result is in fact a corollary to the previous theorem. As we showed in our unit on formal languages and grammars, given any NFA B, $\exists$ a deterministic finite state acceptor (DFA)B' that corresponds to it (with potentially many more states). Therefore, to any pair $\langle B, w\rangle \in L_{NFA}$, there corresponds a pair $\langle B', w\rangle \in L_{DFA}$. Since L_{DFA} is a Turing-decidable language, L_{NFA} is Turing-decidable as well.

q.e.d

(3). <u>The acceptance problem for regular expressions</u>

Test whether a regular expression R generates a string w. We rewrite this acceptance problem as the language

$$L_{REX} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}.$$

Theorem. L_{REX} is a Turing-decidable language.

Proof. Recall that a language L is regular $\Leftrightarrow$ L is accepted by a deterministic or nondeterministic finite state acceptor $\Leftrightarrow$ L is given by a regular expression. There exists an algorithm to construct a nondeterministic finite state acceptor from any given regular expression $\Rightarrow \forall \langle R, w\rangle \in L_{REX}, \exists \langle B, w\rangle \in L_{NFA}$ that corresponds to it. Since L_{NFA} is Turing-decidable, L_{REX} is Turing-decidable.

q.e.d

(4). Emptiness testing for the language of an automaton

Given a DFA B, figure out whether the language recognized by B, $L(B)$ is empty or not, i.e. whether $L(B) \neq \emptyset$ or $L(B) = \emptyset$. Rewrite the emptiness testing problem as a language:

$$E_{DFA} = \{\langle B\rangle \mid B \text{ is a DFA and } L(B) = \emptyset\}.$$

Theorem. E_{DFA} is a Turing-decidable language.

Proof. A DFA B accepts a certain string w if we are in an accepting state when the last character of w has been processed. We design a Turing machine M to test this condition as follows:

$M =$ on input $\langle B\rangle$, where B is a DFA:

1. Mark the initial state of B.
2. Repeat until no new states of B get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, then accept; otherwise, reject.

We have thus marked all states of B where we can end up given an input string. If no such state is an accepting state, then B will not accept any string, i.e. $L(B) = \emptyset$ as needed.

q.e.d

(5). Checking whether two given DFA's accept the same language

Given B_1, B_2 DFA's, test whether $L(B_1) = L(B_2)$. We rewrite this problem as the language

$$EQ_{DFA} = \{\langle B_1, B_2\rangle \mid B_1 \text{ and } B_2 \text{ DFA's and } L(B_1) = L(B_2)\}.$$

Theorem. EQ_{DFA} is a Turing-decidable language.

Proof. Given two sets Γ and Σ, $\Gamma \neq \Sigma$ if $\exists x \in \Gamma$ such that $x \notin \Sigma$ (i.e. $\Gamma \setminus \Sigma \neq \emptyset$) or $\exists x \in \Sigma$ such that $x \notin \Gamma$ (i.e. $\Sigma \setminus \Gamma \neq \emptyset$). Recall from our unit on set theory that $\Gamma \setminus \Sigma = \Gamma \cap \overline{\Sigma}$, Γ intersects the complement of Σ. Similarly, $\Sigma \setminus \Gamma = \Sigma \cap \overline{\Gamma}$. Therefore, $\Gamma \neq \Sigma \Leftrightarrow (\Gamma \cap \overline{\Sigma}) \cup (\Sigma \cap \overline{\Gamma}) \neq \emptyset$. This expression is called the symmetric difference of sets Γ and Σ in set theory. Now, returning to our problem, note that B_1 and B_2 are DFA's $\Rightarrow L(B_1)$ and $L(B_2)$ are regular languages. Furthermore, we showed the set of regular languages is closed under union, intersection, and the taking of complements $\Rightarrow (L(B_1) \cap \overline{L(B_2)}) \cup (L(B_2) \cap \overline{L(B_1)})$ is a regular language $\Rightarrow \exists C$ a DFA that recognizes the symmetric difference of $L(B_1)$ and $L(B_2)$ $(L(B_1) \cap \overline{L(B_2)}) \cup (L(B_2) \cap \overline{L(B_1)})$. $L(B_1) = L(B_2)$ if this symmetric difference is empty $\Rightarrow \forall \langle B_1, B_2 \rangle \in EQ_{DFA}\ \exists \langle C \rangle \in E_{DFA}$, the language corresponding to the emptiness testing problem. Since E_{DFA} is Turing-decidable, EQ_{DFA} is Turing-decidable.

q.e.d

Next, we look at context-free grammars (CFG's) that we studied last term.

(6).

$$L_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG and } w \text{ is a string}\}.$$

Theorem. L_{CFG} is a Turing-decidable language.

Sketch of proof. We could try to go through all possible applications of production rules allowable under G to see whether we can generate w, but infinitely many derivations may need to be tried. Therefore, if G does not generate w, our algorithm would not halt. We would thus have a Turing machine that is a recognizer but not a decider. To get a decider, we have to put G into a special form called a Chomsky normal form that takes $2n-1$ steps to generate a string w of length n. We do not need to know what a Chomsky normal form is, just that one exists in order to write down our decider M :

$M =$ on input $\langle G, w \rangle$, where G is a context-free grammar and w is a string.

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w if $n > 0$. If $n = 0$, list all derivations with one step.
3. If any of these derivations generates w, then accept; otherwise, reject.

q.e.d

(7). Emptiness testing for context-free grammars

Given a context-free grammar G, figure out whether the language it generates $L(G)$ is empty or not.

Rewrite as a language

$$E_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}.$$

Theorem. E_{CFG} is a Turing-decidable language.

Proof. We use a similar marking argument as we did to show E_{DFA} was Turing-decidable. We define the Turing machine as

$M =$ on input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variable get marked:
3. Mark any non-terminal $\langle T \rangle$ if G contains a production rule $\langle T \rangle \to u_1 \cdots u_k$, and each symbol (terminal or non-terminal) $u_1, \ldots, u_k$ has already been marked.
4. If start symbol $\langle S \rangle$ is not marked, accept; otherwise, reject.

As we can see from step 4, if $\langle S \rangle$ is marked, then the context-free grammar will end up generating at least one string as all terminals have already been marked in step 1. Therefore, $L(G) \neq \emptyset$, and we reject G.

q.e.d

(8). Equivalence problem for context-free grammars

Given two context-free grammars, G_1 and G_2, determine whether they generate the same language, i.e. $L(G_1) = L(G_2)$.

Rewrite this problem as a language:

$$EQ_{CFG} = \{\langle G_1, G_2 \rangle \mid G_1 \text{ and } G_2 \text{ are CFG's and } L(G_1) = L(G_2)\}.$$

To solve the equivalence problem for DFA's, we used the symmetric difference and the fact that the emptiness problem for DFA's is Turing-decidable. In this case, the emptiness problem for CFG's is Turing-decidable as we just proved, but the symmetric difference argument does NOT work as the set of languages produced by context-free grammars is NOT closed under complements or intersection, so the following result is true instead:

Proposition. EQ_{CFG} is not a Turing-decidable language.

This proposition is proven using a technique called reducibility. An even more general result is true, the equivalence problem for Turing machines is undecidable:

$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are Turing machines and } L(M_1) = L(M_2)\}.$

Proposition. EQ_{TM} is not a Turing-decidable language.

This proposition follows from another result, namely that the emptiness testing problem for Turing machines is undecidable:

$$E_{TM} = \{\langle M\rangle \mid M \text{ is a Turing machine and } L(M) = \emptyset\}.$$

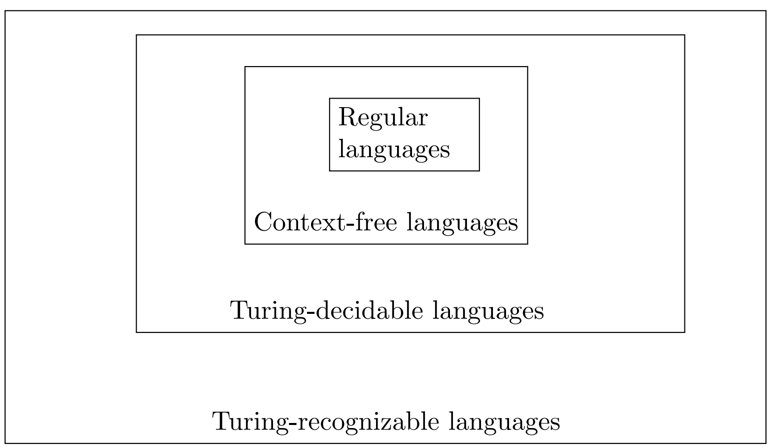
Proposition. E_{TM} is not a Turing-decidable language.

Returning to context-free grammars, we now know that L_{CFG} and E_{CFG} are Turing-decidable, but EQ_{CFG} is not. Recall that a language is called context-free if it can be generated by a context-free grammar.

Moral of the story

We now know how the main types of languages relate to each other {regular languages} $\subset$ {context-free languages} $\subset$ {Turing-decidable languages} $\subset$ {Turing-recognizable languages}.

Visually, we represent the relationship using a Venn diagram:



So Turing machines provide a very powerful computational model. What is surprising is that once we have built a Turing machine to recognize a language, we do not know whether there is a simpler computational model such as a DFA that recognizes the same language. Define

$\text{REGULAR}_{TM} = \{\langle M\rangle \mid M \text{ is a Turing machine and } L(M) \text{ is a regular language}\}.$

Theorem. REGULAR_{TM} is not a Turing-decidable language.

This theorem is proven using reducibility. In fact, even more is true:

Rice's Theorem. Any property of the languages recognized by Turing machines is not Turing-decidable.

11.4 Undecidability

Task: Understand why certain problems are algorithmically unsolvable.

Recall that a Turing machine M is defined as a 7-tuple $(\mathcal{S}, A, \tilde{A}, i, t, S_{accept}, S_{reject})$, where

(a). $\mathcal{S}$ is the set of states.

(b). A is the input alphabet not containing the blank symbol $\sqcup$.

(c). $\tilde{A}$ is the tape alphabet, where $\sqcup \in \tilde{A}$ and $A \subseteq \tilde{A}$.

(d). i is the initial state of the machine.

(e). $t : \mathcal{S} \times \tilde{A} \to \mathcal{S} \times \tilde{A} \times \{L, R\}$ is the transition mapping.

(f). $S_{accept} \in \mathcal{S}$ is the accept state.

(g). $S_{reject} \in \mathcal{S}$ is the reject state and $S_{accept} \neq S_{reject}$.

Definition: An encoding $\langle M \rangle$ of a Turing machine M refers to the 7-tuple

$$(\mathcal{S}, A, \tilde{A}, i, t, S_{accept}, S_{reject})$$

that defines M and is therefore a finite string.

Recall that earlier in the module we proved the following results:

Theorem. If A is a finite alphabet, then the set of all words over A

$$A^* = \bigcup_{j=0}^{\infty} A^j$$

is countably infinite.

Corollary I. If A is a finite alphabet, then the set of all languages over A is uncountably infinite.

Corollary II. The set of all programs in any programming language is countably infinite.

Recall that we proved Corollary II by realizing that for any programming language, a program is a finite string over the finite alphabet of all allowable characters in that programming language.

Corollary III. Given a finite alphabet A, the set of all Turing-recognizable languages over A is countably infinite.

Proof. An encoding $\langle M \rangle$ of a Turing machine M is the 7-tuple

$$(\mathcal{S}, A, \tilde{A}, i, t, S_{accept}, S_{reject}),$$

which is a finite string over a language B that contains A and is finite. By the theorem, $B^* = \bigcup_{j=0}^{\infty} B^j$ is countably infinite. Since $\langle M \rangle \in B^*$, there are at most countably infinitely many Turing machines M that recognize languages over A $\Rightarrow$ there are at most countably infinitely many Turing-recognizable languages over A. We know we can build Turing machines with as large a set of states $\mathcal{S}$ as we want $\Rightarrow$ the set of Turing machines that recognize languages over A cannot be finite $\Rightarrow$ it is countably infinite.

q.e.d

Proposition. Let A be a finite alphabet. Not all languages over A are Turing-recognizable.

Proof. By Corollary I, the set of all languages over A is uncountably infinite. By Corollary III, the set of all Turing-recognizable languages over A is countably infinite $\Rightarrow$ there are many more languages over A than can be recognized by a Turing machine.

q.e.d

Remark. This result makes a lot of sense because while we normally look at simpler, well-structured problems where there is a pattern, most languages over A have no pattern to them.

To understand more on the set of all Turing machines, we define the language

$$L_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing machine and } M \text{ accepts } w\}.$$

Here w is a string over the input alphabet A.

We will prove that L_{TM} is a Turing-recognizable language, but L_{TM} is NOT Turing-decidable.

Proposition. L_{TM} is a Turing-recognizable language.

Proof. We define a Turing machine U that recognizes L_{TM} :

$U =$ on input $\langle M, w \rangle$, where M is a Turing machine and w is a string

1. Simulate M on string w.
2. If M ever enters its accept state, then accept; if M ever enters its reject state, then reject.

U loops on input $\langle M, w \rangle$ if M loops on $w \Rightarrow U$ is a recognizer but not a decider.

q.e.d

Remark. The Turing machine U is an example of the universal Turing machine first proposed by Turing in 1936. It is called universal because it simulates any other Turing machine. This idea of a universal Turing machine led to the development of stored-program computers.

NB: Philosophically, the universal Turing machine we just constructed runs into the following big issues:

1. U itself is a Turing machine. What happens when U is given an input $\langle U, w \rangle$?
2. The encoding of a Turing machine is a string, what happens when we input $\langle M, \langle M \rangle \rangle$ or even worse $\langle U, \langle U \rangle \rangle$

We are getting very close to Russell's paradox, the set $\Gamma = \{D \mid D \notin D\}$ which showed the axioms of naive set theory were inconsistent and led to more complicated axioms.

In our case, these issues lead to showing the language L_{TM} cannot possibly be Turing-decidable.

Proposition. L_{TM} is not Turing-decidable.

Proof. Assume L_{TM} is Turing-decidable and obtain a contradiction. If L_{TM} is Turing-decidable, then $\exists$ decider H for L_{TM}. Given input $\langle M, w \rangle$, the decider H

- accepts if M accepts w.
- rejects if M does not accept w.

We now construct another Turing machine D with H as a subroutine, which behaves like the set Γ defined by Russell:

$D =$ on input $\langle M \rangle$, where M is a Turing machine

1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. Output the opposite of what H outputs. If H accepts, then reject; if H rejects, then accept.

Now, let us run D on its own encoding $\langle D \rangle$:

D on input $\langle D \rangle$

- accepts if D does not accept $\langle D \rangle$.
- rejects if D accepts $\langle D \rangle$.

$\Rightarrow\Leftarrow$ D cannot exist, hence H cannot exist. The language L_{TM} has no decider.

q.e.d

11.5 Example of a language that is not Turing-recognizable

Task: Use what we know about L_{TM} to build an example of a language that is not Turing-recognizable.

Definition: Given an alphabet A that is finite, $A^* = \bigcup_{j=0}^{\infty} A^j$, and then a language $L \subset A^*$, we define the complement $\bar{L}$ of L as $\bar{L} = A^* \setminus L$, i.e. all words over A that are not in L.

Definition: A language L is called co-Turing-recognizable if its complement $\bar{L}$ is Turing-recognizable.

Theorem. A language L is decidable $\Leftrightarrow$ L is Turing-recognizable and co-Turing-recognizable.

" $\Rightarrow$ " If L is decidable $\Rightarrow$ L is Turing-recognizable. Note that if L is decidable $\Rightarrow$ $\exists$ a Turing machine M that decides L. Build a Turing machine $\tilde{M}$ that reverses the output of M, i.e. if M accepts a string w, then $\tilde{M}$ rejects the same string w. If M rejects w, then $\tilde{M}$ accepts w. $\tilde{M}$ is therefore a decider for $\bar{L} \Rightarrow \bar{L}$ is Turing-decidable $\Rightarrow$ $\bar{L}$ is Turing-recognizable, so L is Turing-recognizable and co-Turing recognizable.

" $\Leftarrow$ " If both L and $\bar{L}$ are Turing-recognizable $\Rightarrow \exists M_1$ that recognizes L and $\exists M_2$ that recognizes $\bar{L}$. We use Turing machines M_1 and M_2 to build a decider M for L as follows:

$M =$ on input w, where w is a string:

1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, accept; if M_2 accepts, then reject.

Running M_1 an M_2 in parallel simply means that M has two tapes, one for simulating M_1 and one for simulating M_2.

Note that for any string w, either $w \in L$ or $w \in \bar{L}$, which means either M_1 or M_2 accepts $w \Rightarrow M$ either accepts or rejects any string. In fact, M accepts $w \Leftrightarrow w \in L$ by construction $\Rightarrow M$ is a decider for $L \Rightarrow L$ is Turing-decidable.

q.e.d

Corollary. $\bar{L}_{TM}$ is not Turing-recognizable.

Proof. We proved L_{TM} is Turing-recognizable. If $\bar{L}_{TM}$ were Turing-recognizable, then L_{TM} would be both Turing-recognizable and co-Turing recognizable $\Rightarrow$ by the previous theorem, L_{TM} would be Turing-decidable $\Rightarrow\Leftarrow$ as we proved the contrary $\Rightarrow \bar{L}_{TM}$ is not Turing-recognizable, and we have constructed our example of a non Turing-recognizable language.

q.e.d