1 Mathematical Notions and Terminology

Here, we will review several topics from discrete mathematics.

1.1 Sets and Subsets

Throughout this class, we will be using the concept of sets, tuples, and more. We begin with sets.

1.1.1 Sets

Definition 1.1: Set

A **set** is a collection of objects represented as a unit.

Sets may contain any type of objects, including numbers, symbols, and even other sets. The objects in a set are called its **elements** or **members**. For example, consider the set:

$$S = \{7, 21, 57\}$$

The set S contains 3 elements. To indicate set membership or nonmembership, we use $\in$ and $\notin$, respectively. So, $7 \in S$ means that 7 *is in* S while $8 \notin S$ means that 8 *is not in* S.

Additionally, we note that the order of describing a set doesn't matter, nor does reptition of its members. So, the set $A = \{1, 2, 3\}$ and $Z = \{2, 1, 3\}$ are exactly the same; that is $A = Z$. A generalization of a set that does allow for duplicates does exist, though (see multisets).

Several other common examples of sets are:

- **Singleton sets**, or sets that contain one element. For example, $\{5\}$ is a singleton set.
- **Empty set**, or sets that contain no elements. These are written $\emptyset$. It should be noted that $\{\emptyset\} \neq \emptyset$!
- **Unordered pair**, or sets that contain two elements. For example, $\{5, 2\}$ is an unordered pair because $\{5, 2\} = \{2, 5\}$.

1.1.2 Defining Sets

There are two main ways to define a set.

- Explicitly writing out the elements. For example, the set S of all positive integers between 1 and 5 can be written like so:
$$S = \{2, 4\}$$
- Through a rule. Generally, to describe a set containing elements that adhere to a rule, we use:
$$\{n \mid \text{Rule about } n\}$$
So, for example, the example used in the previous bullet point could have been written like so:
$$S = \{n \mid 1 < n < 5 \text{ and } n = 2m \text{ for some } m \in \mathbb{Z}\}$$

1.1.3 Subsets

Definition 1.2: Subset

A set A is called a **subset** of a set B, written $A \subseteq B$, if every member of A is also a member of B.

For example, the set $A = \{7\}$ is a *subset* of S since $7 \in A$ and $7 \in S$. However, the set $B = \{7, 8\}$ is *not a subset* of S since $7 \in B$ and $7 \in S$ *but* $8 \in B$ while $8 \notin S$.

Definition 1.3: Proper Subset

A set A is called a **proper subset** of a set B, written $A \subset B$ or $A \subsetneq B$, if A is a subset of B and is not equal to B.

So, for example, the set $A = \{7, 21\}$ is a proper subset of S, written $A \subsetneq S$, since every element in A is in S but both aren't equal. However, the set $A = \{7, 21, 57\}$ is not a proper subset of S.

1.1.4 Multisets

Definition 1.4: Multiset

A **multiset** is a generalization of a set that allows for multiple instances for each of its elements.

For example, define $X = \{1, 1, 2, 3\}$ and $Z = \{1, 2, 3\}$. Then, X and Z are precisely the same as *sets* but are different as *multisets*. However, note that order still doesn't matter; for example, $\{1, 1, 2, 3\}$ and $\{1, 2, 1, 3\}$ are still the same *multisets*.

1.1.5 Infinite Sets

Definition 1.5: Infinite Set

An **infinite set** is a set that contains infinitely many elements.

Several common examples of infinite sets are $\mathbb{Z}$ (the set of all integers), $\mathbb{N}$ (the set of all positive integers), $\mathbb{R}$ (the set of all real numbers).

1.1.6 Union, Intersections, and Complements

Definition 1.6: Union

The **union** of two sets A and B, written $A \cup B$, is the set we get by combining all of the elements in A and B into a single set. Mathematically, we write this as:

$$A \cup B = \{n \mid n \in A \text{ or } n \in B\}$$

For example, given $A = \{1, 2\}$ and $B = \{2, 3\}$, $A \cup B = \{1, 2, 3\}$.

Definition 1.7: Intersection

The **intersection** of two sets A and B, written $A \cap B$, is the set we get by combining all *common* elements of A and B into a single set. Mathematically, we write this as:

$$A \cap B = \{n \mid n \in A \text{ and } n \in B\}$$

For example, given the same two sets A and B used in the union example, $A \cap B = \{2\}$.

Definition 1.8: Complement

The **complement** of a set A is the set of all elements that are not in A.

For example, consider the set E of all even integers. Then, the complement of E is the set of all odd integers.

1.1.7 Power Set

Definition 1.9: Power Set

The **power set** of A, denoted $\mathcal{P}(A)$ is the set of all subsets of A.

For example, if A is the set $\{0, 1\}$, then $\mathcal{P}(A)$ would be:

$$\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$$

1.2 Sequences and Tuples

Here, we will review sequences and tuples (kind of like ordered sets).

1.2.1 Sequences

Definition 1.10: Sequence

A **sequence** of objects is a list of these objects in some order.

We usually write a sequence by writing the objects within parentheses. For example, the sequence 7, 21, 57 would be written:

$$(7, 21, 57)$$

As implied, the order matters in a sequence. In other words, the sequence $(7, 21, 57)$ is not the same as $(57, 7, 21)$. Additionally, repetition matters in a sequence. So, $(7, 21, 57)$ is not the same sequence as $(7, 21, 21, 57)$.

1.2.2 Tuples

Definition 1.11: Tuple

A finite sequence is often called a **tuple**. A sequence with k elements is a **k-tuple**.

So, $(7, 21, 57)$ is a 3-tuple. As a side note, a 2-tuple is also called an **ordered pair**.

1.2.3 Cartesian Product

Definition 1.12: Cartesian Product

The **Cartesian product** of two sets A and B, written $A \times B$, is the set of all ordered pairs wherein the first element is a member of A and the second element is a member of B.

If $A = \{1, 2\}$ and $B = \{x, y\}$, then:

$$A \times B = \{(1, x), (1, y), (2, x), (2, y)\}$$

Definition 1.13: Extended Cartesian Product

The **Cartesian product** of k sets, $A_1, A_2, \ldots, A_k$, written $A_1 \times A_2 \times \cdots \times A_k$, is the set of all k-tuples $(a_1, a_2, \ldots, a_k)$ where $a_i \in A_i$ for $1 \leq i \leq k$.

Using the definition of A and B from the previous definition, it follows that:

$$A \times B \times A = \{(1, x, 1), (1, x, 2), (1, y, 1), (1, y, 2), (2, x, 1), \dots\}$$

If we have the Cartesian product of a set A with itself, we can use the shorthand:

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

Another (familiar) example is $\mathbb{R} \times \mathbb{R}$, or the set of all real points. We can define this set like so:

$$\{(i, j) \mid i, j \in R\}$$

Some example and non-example of elements are:

$$(1, 5) \in \mathbb{R} \times \mathbb{R}$$

$$\left(\frac{1}{2}, \sqrt{2}\right) \in \mathbb{R} \times \mathbb{R}$$

$$(-0.8213, -92.3) \in \mathbb{R} \times \mathbb{R}$$

$$(i, 3 + 2i) \notin \mathbb{R} \times \mathbb{R}$$

1.3 Functions

A function is an object that sets up an input-output relationship. In other words, a function takes an input and produces an output. In every function, the same input always produces the same output. So, if f is a function whose output value is b when the input value is a, we write:

$$f(a) = b$$

Alternatively, a function is called a **mapping**. In other words, for the above function definition, we say that f maps a to b.

Definition 1.14: Function Definitions

For a function $f : A \mapsto B$:

- The **domain** A is the set of possible inputs to the function.
- The **codomain** B is the set of possible output values[a].
- The **range** is the set of actual output values. This is a subset of B.

[a] The book states that the range is the set of output values, but later states that the function may not use all the elements in the range.

Consider the function abs : $\mathbb{R} \mapsto \mathbb{R}$ defined by:

$$\text{abs}(x) = |x|$$

Here, the domain is the set of all real numbers. The codomain is the set of all real numbers as well. However, the range is the set of all *non-negative* real numbers, that is, $\mathbb{R}_{\geq 0}$.

1.3.1 Cartesian Product Input

The domain of a function f can be the Cartesian product of multiple sets $A_1 \times \cdots \times A_k$ for some sets $A_1, \ldots, A_k$. In this case, f's input is a k-tuple $(a_1, \ldots, a_k)$ where a_i are the **arguments** to f. In this case, the function with k-arguments is called a **k-ary function** and k is called the **arity** of the function. Note that:

- If $k = 1$, then f has a single argument and f is called a **unary function**.
- If $k = 2$, f is a **binary function**.

Certain binary functions can be written in a special **infix notation**; for example, the function:

$$\text{add} : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$$

Is often written as $a + b$ (with the $+$ symbol between the two arguments) instead of the **prefix notation** (with symbol preceding) $\text{add}(a, b)$.

1.3.2 Predicates

Definition 1.15: Predicate

A **predicate** is a function whose range is $\{\text{true}, \text{false}\}$.

For example, the function:

$$\text{even} : \mathbb{Z} \mapsto \{\text{true}, \text{false}\}$$

Takes in an integer and outputs `true` if the integer is even and `false` otherwise. So:

$$\text{even}(10) = \text{true}$$

$$\text{even}(11) = \text{false}$$

A predicate whose domain is a set of k-tuples $A \times \cdots \times A$ is called a **relation**, a **k-ary** relation, or a **k-ary relation on** A. A common case is a 2-ary relation, called a **binary relation**, for which we often use infix notation. For example, several very common binary relations are the "less than" and "equals" relation.

If R is a binary relation, then the statement aRb means that aRb is `true`. For example, if R is the binary relation $<$, then $3 < 5$ is `true`. Similarly, if R is a K-ary relation, the statement $R(a_1, \ldots, a_k)$ means that $R(a_1, \ldots, a_k)$ is `true`.

1.3.3 Equivalence Relations

Suppose R is a relation over a set S. Then:

- R is called **reflexive** if $\forall x \in S$, xRx. That is, every $x \in S$ is related to itself.
- R is called **symmetric** if $\forall x, y \in S$, $xRy \implies yRx$. In other words, if x is related to y, is y related to x?
- R is called **transitive** if $\forall x, y, z \in S$, xRy and yRz implies that xRz.

For example, define the equivalence relation on $\mathbb{N}$, written $\equiv_7$. For any $i, j, k \in \mathbb{N}$, say $i \equiv_7 j$ if $i - j$ is a multiple of 7. This is an equivalence relation because:

- It is reflexive as $i - i = 0$ and 0 is a multiple of 7.
- It is symmetric as $i - j$ is a multiple of 7 if $j - i$ is a multiple of 7.
- It is transitive as whenever $i - j$ and $j - k$ are both multiples of 7, then $i - k = (i - j) + (j - k)$ is the sum of two multiples of 7, and hence a multiple of 7.

1.4 Undirected Graphs

Definition 1.16: Undirected Graph

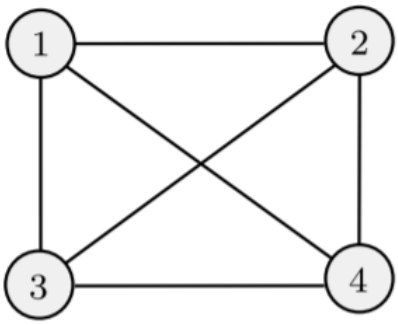
An **undirected graph**, or simply a **graph** is a pair $G = (V, E)$, where:

- V is a set whose elements are called *nodes* or *vertices*.
- E is a set of paired nodes, whose elements are called *edges*.

Informally, an undirected graph is a set of points with lines connecting some of these points. The points are called nodes or vertices and the lines are called edges.

In an undirected graph G that contains nodes i and j, the pair (i, j) or (j, i) represents the edge that connects i and j; in other words, (i, j) and (j, i) are the same. So, it follows that the order in which we put i and j in the tuple doesn't matter. Because order doesn't matter, we may also represent (i, j) as $\{i, j\}$.

For example, the following graph is an undirected graph:



Definition 1.17: Degree of a Node

The number of edges at a particular node is the **degree** of that node.

We note that no more than one edge is allowed between any two nodes; however, we may allow an edge from a node to itself (called a *self-loop*).

Looking at the graph above, we can define $G = (V, E)$ where[1]:

- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$

Additionally, each node has degree 3 because each node has 3 edges.

1.4.1 Labeled Graph

A graph is a **labeled graph** if the nodes and/or edges of a graph are labeled. For example, we might represent each node as a city and each edge as the number of miles from node A to node B.

[1] Remember, because we're working with an undirected graph, an edge like $(1, 2)$ and $(2, 1)$ refer to the same edge!

1.4.2 Subgraph

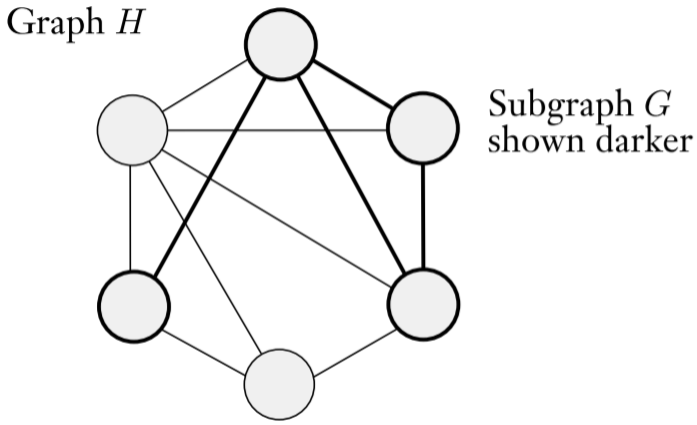
Definition 1.18: Subgraph

A graph G is a **subgraph** of graph H if the nodes of G are a subset of the nodes of H *and* the edges of G are the edges of H on the corresponding nodes.

In other words, if we had $G = (V', E')$ and $H = (V, E)$, we must have:

- $V' \subseteq V$
- $E' \subseteq E$

The following figure shows a graph H and a subgraph G:

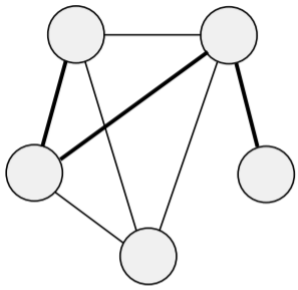


1.4.3 Path

Definition 1.19: Path

A **path** in a graph is a sequence of nodes connected by edges.

For example, a path might look like:



Definition 1.20: Simple Path

A **simple path** is a path that doesn't repeat any nodes.

1.4.4 Connections Between Nodes

Definition 1.21: Connected Graph

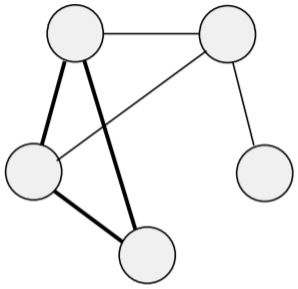
A graph is **connected** if every two nodes have a path between them.

So, the graph in the above example is a connected graph.

Definition 1.22: Cycles

A path is a **cycle** if it starts and ends in the same node.

For example, a cycle might look like:



Definition 1.23: Simple Cycle

A **simple cycle** is a cycle that contains at least three nodes and repeats only the first and last node.

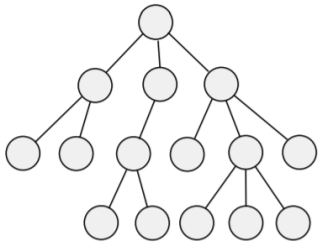
1.4.5 Trees

Definition 1.24: Tree

A graph is a **tree** if it is connected and has no simple cycles.

A tree can contain a specially designated node called the *root*, which is usually a singular node at the "top" that branches out to every other node. In this case, the nodes of degree 1 in a tree, other than the root, are called the *leaves* of the tree.

A tree might look like:



1.5 Directed Graphs

Definition 1.25: Directed Graph

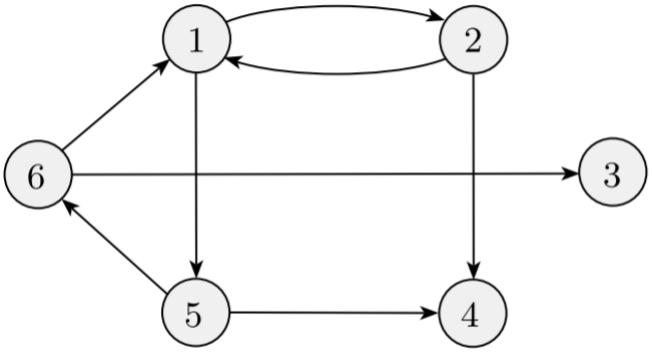
A **directed graph** is a pair $G = (V, E)$ where:

- V is a set whose elements are called *nodes* or *vertices*.
- E is a set of paired nodes, whose elements are called *directed edges*.

Informally, a directed graph is a set of points with arrows instead of lines connecting some of these points.

In a directed graph that contains nodes i and j, the pair (i, j) and (j, i) **do not represent** the same edges. In particular, (i, j) means that there is an arrow pointing *from* i to j; then, (j, i) means that there is an arrow pointing from j to i. In other words, the order **does** matter.

For example, the following graph is a directed graph:



We can define $G = (V, E)$ as:

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 2), (2, 1), (6, 1), (6, 3), (1, 5), (2, 4), (5, 4), (5, 6)\}$

1.5.1 Directed Path

Definition 1.26: Directed Path

A path in which all the arrows point in the same direction as its steps is called a **directed path**.

1.5.2 Connections

Definition 1.27

A directed graph is **strongly connected** if a directed path connects every two nodes.

1.5.3 Relationship to Binary Relations

We can use directed graphs to depict binary relations. In particular, if R is a binary relation whose domain is $D \times D$, a labeled graph $G = (D, E)$ represents R, where $E = \{(x, y) \mid xRy\}$. For example, for $D = \{1, 2, 3\}$ and binary relation $<$, the edges would be:

$$E = \{(1, 2), (2, 3), (1, 3)\}$$

2 Boolean Logic

Boolean logic is a mathematical system built around `true` and `false`, which are often represented as `1` and `0`, respectively. We can manipulate boolean values with boolean operations. The operations are as follows:

Negation: The negation, or **not**, operation is designated with the symbol $\neg$. The negation of a Boolean value is simply the opposite value. Therefore:

a	$\neg a$
0	1
1	0

Conjunction: The conjunction, or **and**, operation is designated with the symbol $\wedge$. The conjunction of two Boolean values is 1 if both values are 1 and 0 otherwise.

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Disjunction: The disjunction, or **or**, operation is designated with the symbol $\vee$. The disjunction of two Boolean values is 1 of either of those values are 1 and 0 otherwise.

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

There are several other operators that may occasionally appear; they are:

Exclusive OR: The *exclusive or*, or **XOR**, is designated by $\oplus$; it is 1 if either, but not both, of the operands is 1.

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Equality: The *equality* operator is designated by $\longleftrightarrow$; it is 1 if both operands have the same value.

a	b	$a \longleftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

Implication: The *Implication* operator is designated by $\longrightarrow$; it is 0 if its first operand is 1 and its second operand is 0 and 1 otherwise.

a	b	$a \longrightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

Finally, the **distributive law** for `AND` and `OR` is useful when manipulating Boolean expressions. That is:

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

3 Strings and Languages

Definition 3.1: Alphabet

An **alphabet** is any nonempty finite set. Generally, we use Σ and Γ to designate alphabets.

Definition 3.2: Symbols

The members of the alphabet are the **symbols** of the alphabet.

Some example of alphabets are:

$$\Sigma_1 = \{\texttt{0}, \texttt{1}\}$$

$$\Sigma_2 = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l}, \texttt{m}, \texttt{n}, \texttt{o}, \texttt{p}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}, \texttt{u}, \texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z}\}$$

$$\Gamma = \{\texttt{0}, \texttt{1}, \texttt{x}, \texttt{y}, \texttt{z}\}$$

Definition 3.3: String

A **string** over an alphabet is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas.

For example, if we use Σ_1 as our alphabet, then `01001` is a **string** over Σ_1. Likewise, if we use Σ_2 as our alphabet, then `something` is a string over Σ_2.

The set of all finite strings over Σ (any general alphabet) is denoted by Σ^*. Here, this includes:

- The empty string ϵ.
- Any **finite** combination of the symbols in this alphabet.

This does not include infinite sequences of symbols. It does have infinitely many elements.

So, for example, Σ_1^* would have strings like (and keep in mind that these are just examples):

$$\epsilon \in \Sigma_1^* \qquad \texttt{0} \in \Sigma_1^* \qquad \texttt{1} \in \Sigma_1^*$$

$$\texttt{010101} \in \Sigma_1^* \qquad \texttt{1111} \in \Sigma_1^* \qquad \texttt{0000} \in \Sigma_1^*$$

Definition 3.4: Length

If w is a string over Σ, then the **length** of w, written $|w|$, is the number of symbols that it contains.

Remarks:

- The string of length zero is called the **empty string** and is written ϵ. The empty string plays the role of 0 (like an identity) in a number system.
- If w has length n, then we can write $w = w_1 w_2 \dots w_n$ where each $w_i \in \Sigma$.

Definition 3.5: Reverse

The **reverse** of a string w, written $w^{\mathcal{R}}$, is the string obtained by writing w in the opposite order.

Remark: In other words, for a string w, we can write the reverse of w as $w^{\mathcal{R}} = w_n w_{n-1} \dots w_2 w_1$.

Definition 3.6: Substring

A string z is a **substring** of a string w if z appears consecutively within w.

For example, if we look at the string $w = \texttt{something}$, then $z_1 = \texttt{some}$ and $z_2 = \texttt{thing}$ are both substrings of w.

Definition 3.7: Concatenation

For a string x of length m and a string y of length n, the **concatenation** of x and y, written xy, is the string obtained by appending y to the end of x, as in:

$$x_1 \dots x_m y_1 \dots y_n$$

If we want to concatenate a string x with itself many times, we use the superscript notation x^k to mean:

$$\underbrace{xx \dots x}_{k}$$

If we consider the two substrings z_1 and z_2 in the previous example, then $z_1 z_2 = \texttt{something}$.

Definition 3.8: Prefix

A string x is a **prefix** of a string y if a string z exists where $xz = y$.

For example, if we look at the two substrings z_1 and z_2 in the previous example yet again, we say that z_1 is a prefix of w since $z_1 z_2 = w$.

Definition 3.9: Proper Prefix

A string x is a proper prefix of a string y if, in addition to x being a prefix of y, $x \neq y$.

So, `some` is a proper prefix of `something` while `something` is not a proper prefix of `something`.

Definition 3.10: Language

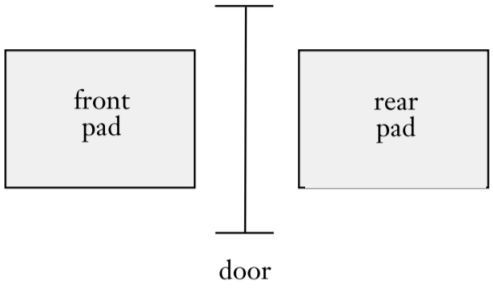
A **language** is a set of strings.

Definition 3.11: Prefix-Free

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

4 Finite Automata (1.1)

Consider the controller for an automatic one-way door.



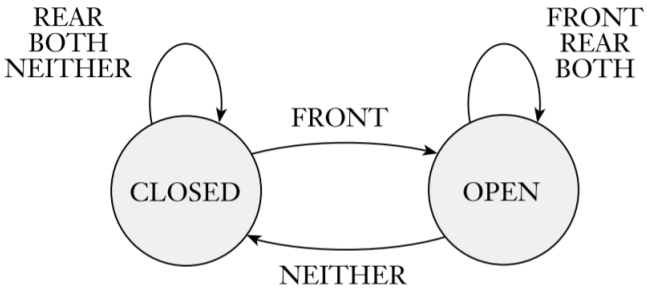
Here:

- The front pad is there is to detect the presence of a person who is about to walk through the doorway.
- The rear pad is there so that the controller can hold the door open long enough for the person to pass all the way through while also ensuring that no one behind door is hit by the door.

The controller is in either of two states: `OPEN` or `CLOSED`. This represents the condition of the door. There are also *four* possible input conditions:

- `FRONT`: A person is standing on the pad in front of the doorway (the front pad).
- `REAR`: A person is standing on the pad to the rear of the doorway (the rear pad).
- `BOTH`: People are standing on both pads.
- `NEITHER`: No one is standing on either pad.

The corresponding state diagram is:



And the corresponding transition table:

	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

The controller moves from state to state depending on what input it receives. For example:

- When it starts off in the `CLOSED` state and receives input `NEITHER` or `REAR`, it remains in the `CLOSED` state. In the state diagram, if we start at the `CLOSED` circle (state), both `NEITHER` and `REAR` loop back to `CLOSED`.
- Again, when the controller is in the `CLOSED` state and it receives the `BOTH` input, then it stays in the `CLOSED` state because opening the door may knock someone over on the rear pad (as the door opens towards the rear side).

- If the controller is in the `OPEN` state, then receiving the inputs `FRONT`, `REAR`, or `BOTH` will result in the controller remaining `OPEN`. However, if it receives the `NEITHER` input, then it goes to a `CLOSED` state.

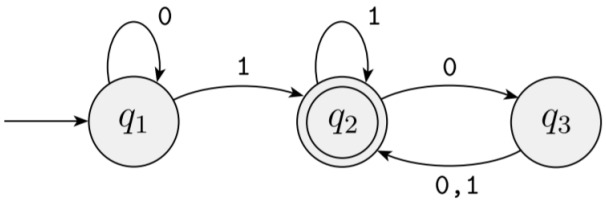
Essentially, **for the state diagram**, start at the initial state (circle) and follow the arrow depending on what input signals are received. **For the transition table**, look at the row corresponding to the initial state and the column corresponding to the input; the resulting cell will be the new state of the controller.

The figures used above (the state diagram and transition table) are both standard ways of representating a finite automaton. While this door may be very simple (due to the fact that it only really needs to store an extremely small amount of memory), in reality, we may be dealing with other devices with somewhat more memory.

Both finite automata and their probablistic counterpart **Markov chains** are useful tools when we want to attempt to recognize patterns in data.

4.1 From a Mathematical Perspective

Consider the following figure, which depicts a finite automaton called M_1:



There are a few things to note here:

- The above figure for M_1 is called the **state diagram** of M_1.
- M_1 has three **states**, labeled q_1, q_2, and q_3.
- The **start state** is the state indicated by the arrow pointing at it from nowhere. In the case of the above state diagram, this would be q_1.
- The **accept state** is the state with a double circle. In the case of the above state diagram, this would be q_2.
- The **transitions** are the arrows going from one state to another.

For a given input string, this automaton processes that string and produces an output that is either `ACCEPT` or `REJECT`. For this automaton, the processing works like so:

1. Here, the processing begins in M_1's start state.
2. Then, the automaton receives the symbols from the input string one by one from left to right.
3. After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label.
4. When it reads the last symbol, M_1 produces its output. The output is `ACCEPT` if M_1 is now in an accept state and `REJECT` if it is not.

As an example, suppose we give M_1 the input string `1101`. Then, the processing proceeds as follows:

- Start in state q_1.
- Read `1`. Transition from q_1 to q_2.

- Read `1`. Transition from q_2 to q_2.
- Read `0`. Transition from q_2 to q_3.
- Read `1`. Transition from q_3 to q_2.
- `ACCEPT` because M_1 is in an accept state q_2 at the end of the input.

So, really, what matters is that we *end up* at the accept state.

4.2 Formal Definition of a Finite Automaton

A finite automaton has several parts.

- It has a set of states and rules for going from one state to another, depending on the input symbol.
- It has an input alphabet that indicates the allowed input symbols.
- It has a start state and a set of accept states.

We use something called a **transition function**, often denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol `1`, that means that if the automaton is in state x when it reads a `1`, it then moves to state y. We can indicate the same thing with the transition function by saying that:

$$\delta(x, \texttt{1}) = y$$

All of this leads to the formal definition:

Definition 4.1: Finite Automaton

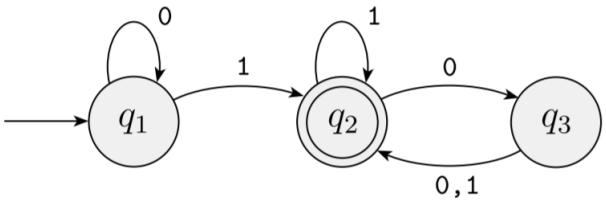
A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

1. Q is a finite set called the **states**.
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma \mapsto Q$ is the **transition function**.
4. $q_0 \in Q$ is the **start state**.
5. $F \subseteq Q$ is the **set of accept states** (sometimes also called *final states*).

Remark: F can be the empty set $\emptyset$, which means that there are 0 accept states.

4.3 Applying the Definition

Consider again M_1:



Using the formal definition above, we can describe M_1 formally be writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where:

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{\texttt{0, 1}\}$

- δ is defined as:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- q_1 is the start state.
- $F = \{q_2\}$.

4.4 Machine and Language

If A is the set of all strings (i.e. language) that machine M accepts, we say that A is the **language of machine** M, write $L(M) = A$, and say that M recognizes A.

A machine may accept *several strings*, but it always recognizes one language. A machine can accept no strings; in this case, it still recognizes the empty language $\emptyset$.

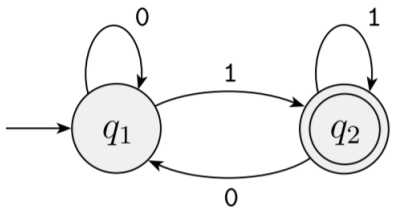
If we consider our example automaton M_1, then define:

$$A = \{w \mid w \text{ contains at least one } \texttt{1} \text{ or an even number of } \texttt{0}\text{s follow the last } \texttt{1}\}$$

Which means that $L(M_1) = A$, of equivalently, M_1 recognizes A.

4.4.1 Example 1: Simple Finite Automaton

Consider the following state diagram for the finite automaton M_2:



Here, the formal description of M_2 is as follows:

$$M_2 = (\{q_1, q_2\}, \{\texttt{0, 1}\}, \delta, q_1, \{q_2\})$$

Where δ is:

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

To figure out what A is, we try a few different strings.

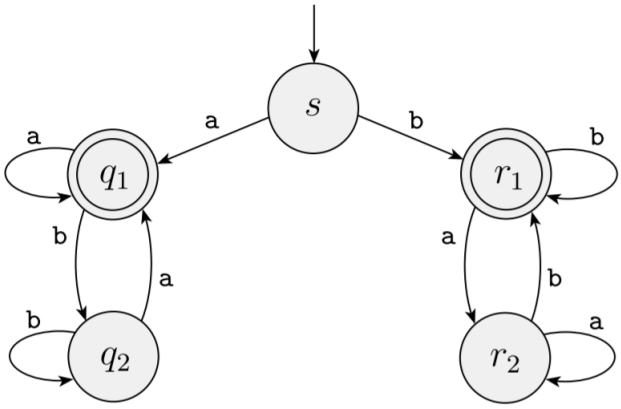
String Input	Output
ϵ	REJECT
1	ACCEPT
0	REJECT
01	ACCEPT
10	REJECT
11	ACCEPT

It's quite clear that A is simply the set of all strings that end with 1. So:

$$A = \{w \mid w \text{ ends with } \texttt{1}.\}$$

4.4.2 Example 2: Finite Automaton

Consider the following state diagram for the finite automaton M_3:



Here, the formal description of M_3 is as follows:

$$M_3 = (\{s, q_1, q_2, r_1, r_2\}, \{\texttt{a, b}\}, \delta, s, \{q_1, r_1\})$$

Where δ is:

	a	b
s	q_1	r_1
q_1	q_1	q_2
q_2	q_1	q_2
r_1	r_2	r_1
r_2	r_2	r_1

Here, we note that we cannot end at the start state. In other words, when we start with a, we take the left branch to q_1. In the left branch, notice how when we end with a, we will always end up at q_1, the accept state. So, it follows that a string like the one below is acceptable:

$$\texttt{a}w_2w_3\ldots w_{n-1}\texttt{a} \qquad w_i \in \{\texttt{a, b}\}$$

Likewise, if we start with b, we take the right branch to r_1. In the right branch, if our string ends with b, we will always end up at r_1. So, it follows that a string like the one below is also acceptable:

$$\texttt{b}w_2w_3\ldots w_{n-1}\texttt{b} \qquad w_i \in \{\texttt{a, b}\}$$

In other words, for this automaton, a string that starts and ends with the same symbol is accepted. That is:

$$A = \{w \mid w \text{ starts and ends with the same symbol.}\}$$

4.4.3 Example 3: Complicated Finite Automaton

Sometimes, it is hard to describe a finite automaton by state diagram. In this case, we may end up using a formal description to specify the machine. Consider the following example with the alphabet:

$$\Sigma = \{\texttt{RESET, 0, 1, 2}\}$$

Where RESET is treated as one symbol. For each $i \geq 1$, define A_i to be the language of all strings where the sum of the numbers is a multiple of i, except that the sum is reset to 0 whenever the symbol RESET appears. For each A_i, we have a finite automaton B_i which recognizes A_i. We define B_i formally like so:

$$B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$$

Where $Q_i = \{q_0, q_1, q_2, \ldots, q_{i-1}\}$ and the transition function δ_i is defined so that for each j, if B_i is in q_j (i.e. B_i is in state q_j), the running sum is j modulo i. In other words, for each q_j define:

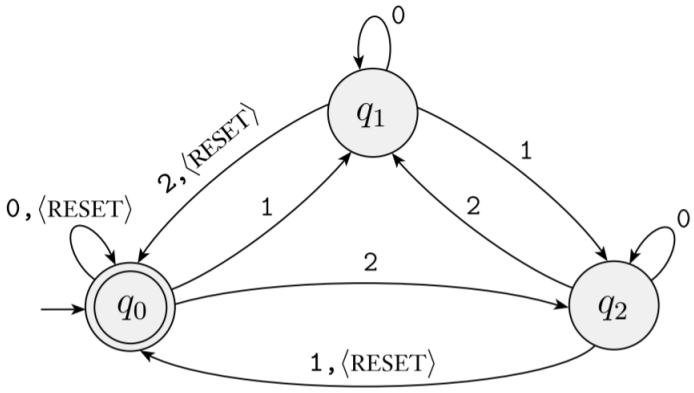
$$\delta_i(q_j, \texttt{0}) = q_j$$

$$\delta_i(q_j, \texttt{1}) = q_k \text{ where } k = j + 1 \text{ modulo } i$$

$$\delta_i(q_j, \texttt{2}) = q_k \text{ where } k = j + 2 \text{ modulo } i$$

$$\delta_i(q_j, \texttt{RESET}) = q_0$$

For example, suppose we have the following state machine B_3 which uses the same alphabet described above:



The formal description of B_3 is as follows:

$$B_3 = (\{q_0, q_1, q_2\}, \{\texttt{RESET, 0, 1, 2}\}, \delta, q_0, \{q_0\})$$

Where δ is defined by:

	RESET	0	1	2
q_0	q_0	q_0	q_1	q_2
q_1	q_0	q_1	q_2	q_0
q_2	q_0	q_2	q_0	q_1

So, as an example, let's suppose we have the string `01212`. The sum of these numbers is:

$$0 + 1 + 2 + 1 + 2 = 6 \implies 6 \equiv \boxed{0} \mod 3$$

We expect the automaton to finish at the accept state as 6 is a multiple of 3. Running through the automaton, we have:

- Input: `0`. Start at q_0, end at q_0.
- Input: `1`. Start at q_0, end at q_1. So, our automaton is at state q_1.
- Input: `2`. Start at q_1, end at q_0. So, our automaton is at state q_0.
- Input: `1`. Start at q_0, end at q_1. So, our automaton is at state q_1.
- Input: `2`. Start at q_1, end at q_0. So, our automaton is at state q_0.

Therefore, we are at an accept state as our string `01212` sums up to a multiple of 3. Of course, if there are any `RESET`s in our string, we can disregard everything up to and including the *last* `RESET` as `RESET` puts us back at the start. That is, for instance, the string `0121 RESET 21011 RESET 01212` will put the state machine in the same state as `01212`.

So, it follows that our state machine recognizes the set A_3, which consists of all strings where all digits sum up to 0 modulo 3. That is:

$$A_3 = \left\{ w \mid \sum_{\mathtt{d} \in w} d = 0 \pmod 3 \right\}$$

Note: If any `RESET`s are in the string, we can omit everything in the string *up to and including* the last `RESET`.