Our next model of computation is the Turing machine. There are a few key differences between Turing machines (TMs) and DFAs, which make a TM more powerful:

1. A TM has an infinite tape.
2. A TM goes forward (to the right) and backward (to the left).
3. A TM can both read and write.
4. A TM has one accept state and one reject state. Once it reaches either of these states, it stays there.

Transitions between states of a TM contain more information than those of a DFA. For example, the following diagram illustrates a transition from state $q_1$ to $q_2$ that involves reading a symbol $a$, overwriting it with $b$, and moving the tape head to the right.

![Transition Diagram](image)

**Topics Covered**

1. Example of a Turing Machine
2. Formal Definition
3. Computation Histories
4. Turing Machine Variations

**1 Example of a Turing Machine**

Recall that the language $L = \{a^n b^n c^n\}$ is neither regular nor context-free. Nonetheless, we can construct a Turing machine that accepts $L$. On input $aaabbbccc$, the tape of our TM is initialized to look like:

```
| a | a | a | b | b | b | c | c | c | . . . |
```
The overall algorithm for our TM will work as follows:

1. If the input is $\ldots$, accept.

2. If the input starts with $b$ or $c$, reject.

3. Make sure that the input is of the form $a^+b^+c^+$, where $a$, $b$, and $c$ are each repeated a nonzero number of times (perhaps not all the same amount) and occur in the correct order. If not, reject. Otherwise, return to the beginning of the input.
   (a) While doing this, mark the first $a$ as $\dot{a}$, the first $b$ as $\dot{b}$, and the first $c$ as $\dot{c}$.

4. Skip over any crossed-out $a$, $\theta$.

5. If see $a$:
   (a) Cross out the $a$, $\theta$.
   (b) Skip over $a$s.
   (c) Skip over $b$.
   (d) Skip over crossed-out $bs$, $\theta$.
   (e) If see $b$:
      i. Cross out the $b$, $\theta$.
      ii. Skip over $bs$.
      iii. Skip over $\dot{c}$.
      iv. Skip over crossed-out $cs$, $\theta$.
      v. If see $c$:
         A. Cross out the $c$, $\theta$.
         B. Go to the beginning of the input.
      vi. If don’t see $c$, reject.
   (f) If don’t see $b$, reject.

6. If don’t see $a$, move forward. If see $b$ or $c$, reject. If see $\ldots$, accept.

In an intermediate step of our computation, the tape of the Turing machine might look like:

\[
\dot{a} \theta a b \dot{b} b \dot{c} \varepsilon c \ldots
\]
Before we construct the full Turing machine, let’s consider the single step of going to the beginning of the input from some point in the tape. In this machine, the start of the tape and the start of the input are the same. The state transitions are of the form:

\[
\begin{align*}
    a &\rightarrow a, L \\
b &\rightarrow b, L \\
c &\rightarrow c, L
\end{align*}
\]

We use the notation \(\ast\) to label a transition whose inputs include all inputs not otherwise listed for a given state. The \textit{back} node is as we previously constructed. All in all, here is a Turing machine to accept \(L\):

\[
\begin{align*}
    q_0 &\rightarrow \hat{a}, R \\
    q_1 &\rightarrow b, R \\
    q_2 &\rightarrow c, R \\
    q_3 &\rightarrow \epsilon, R \\
    q_4 &\rightarrow a, R \\
    q_5 &\rightarrow a, R \\
    q_6 &\rightarrow b, R \\
    q_7 &\rightarrow c, R \\
    q_r &\rightarrow \epsilon, R
\end{align*}
\]
For this Turing machine, we can intuitively think about what each state represents. For example, \( q_1 \) corresponds to reading through \( a \)s in the third step of our algorithm, while \( q_2 \) and \( q_3 \) correspond to reading through \( b \)s and \( c \)s, respectively. The state \( q_4 \) represents looking for the next \( a \), \( q_5 \) represents looking for the next \( b \), and \( q_6 \) represents looking for the next \( c \). Finally, \( q_7 \) is when there are no more \( a \)s, and we need to verify that there are no more \( b \)s and \( c \)s.

2 Formal Definition

- Formally, we define a Turing machine (TM) as a seven-tuple \((Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})\) where:
  1. \( Q \) is a finite set of states
  2. \( \Sigma \) is a finite input alphabet where \( \_ \) is not in \( \Sigma \)
  3. \( \Gamma \) is a finite tape alphabet; that is, a finite set such that \( \Sigma \subset \Gamma \) and \( \_ \) is in \( \Gamma \)
  4. \( \delta : (Q - q_{accept} - q_{reject}) \times \Gamma \to \Gamma \times \{L, R\} \times Q \) is a transition function
  5. \( q_0 \in Q \) is the start state
  6. \( q_{accept} \in Q \) is the designated accept state
  7. \( q_{reject} \in Q \) is the designated reject state

In our previous example of a TM, the seven components are:

1. \( Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, \text{back}, q_a, q_r\} \)
2. \( \Sigma = \{a, b, c\} \)
3. \( \Gamma = \{a, b, c, \_a, \_b, \_c, a, b, c, \_\} \)
4. \( \delta \), where one transition is \( \delta(q_4, \_b) = (\_b, R, q_7) \)
5. \( q_0 = q_0 \)
6. \( q_{accept} = q_a \)
7. \( q_{reject} = q_r \)

3 Computation Histories

- The configuration of a TM \( M \) consists of a state \( q \in Q \), a location (i.e. the index on the tape where the head is pointing), and the current contents of the tape.
The starting configuration of a TM $M$ on input $aaabbbccc$ can be illustrated as:

$$q_0 \quad a \quad a \quad a \quad b \quad b \quad b \quad c \quad c \quad c \quad \ldots$$

After the first step of our algorithm, the configuration is:

$$\dot{a} \quad q_1 \quad a \quad a \quad b \quad b \quad b \quad c \quad c \quad c \quad \ldots$$

A few of the immediately subsequent configurations are:

$$\dot{a} \quad a \quad q_1 \quad a \quad b \quad b \quad b \quad c \quad c \quad c \quad \ldots$$

$$\dot{a} \quad a \quad a \quad q_1 \quad b \quad b \quad b \quad c \quad c \quad c \quad \ldots$$

$$\dot{a} \quad a \quad a \quad b \quad q_2 \quad b \quad b \quad c \quad c \quad c \quad \ldots$$

$$\dot{a} \quad a \quad a \quad b \quad b \quad q_2 \quad b \quad c \quad c \quad c \quad \ldots$$

- A **start configuration** of a TM $M$ on input $w$ is $q_0w$ where $q_0$ is $M$’s start state. In general, we can denote the configuration of a Turing machine as a string rather than a table. For example, $aqa_1aabbeccc$ denotes a configuration.

- An **accepting configuration** is any configuration where the state is $q_{accept}$.

- A **rejecting configuration** is any configuration where the state is $q_{reject}$.

- For two configurations $C$ and $C'$ of a TM $M$, we say that $C'$ follows from $C$ if $C = (q, i, w)$ and $C' = (q', i', w')$ such that $\delta(q, w_i) = (w'_i, i' - i, q')$. We consider $L$ (a leftward step) to be $-1$ and $R$ (a rightward step) to be $1$. We only have $i' - i = 0$ if we are at the leftmost end of the tape and trying to move left.

- A **computation history** (CH) of a TM $M$ on input $w$ is a sequence, possibly infinite, of configurations $C_0, \ldots, C_i, \ldots$ such that $C_0$ is the start configuration of $M$ on input $w$, and for all $i$, $C_i$ follows from $C_{i-1}$.

- $M$ accepts $w$ if the CH of $M$ on input $w$ is finite and ends in an accepting configuration. $M$ rejects $w$ if the CH is finite and ends in a rejecting computation. In general, the language of $M$ is $L(M) = \{ w \mid M$ accepts $w \}$.

- $L$ is **Turing-recognizable** if there exists a Turing machine $M$ such that $L(M) = L$.

- $L$ is **decidable** if there exists a Turing machine $M$ such that $L(M) = L$ and $M$ always halts.
In the coming weeks, we will consider Turing-recognizability and decidability in much greater detail. One of the themes will be determining which languages are not computable. Turing machines turn out to be an incredibly helpful model of computation for this. In particular, an equivalent Turing machine can be constructed for any possible program, in whatever programming language you like. By equivalent, we mean that they accept the same language. The Church-Turing thesis tells us that we can construct a TM for any program or real-world computation.

4 Turing Machine Variations

For something that can model any algorithm, our definition of a Turing machine may seem somewhat arbitrary. Can there be a TM that stays put on a turn, or one that has a doubly infinite tape? While these variations of Turing machines may be more suited to solving certain problems, a regular Turing machine can model any of them, and vice versa.

Two-Tape Turing Machine Suppose we have a TM with an input tape (as before) and a second tape, which is initially blank. Transitions for this TM are of the form \( \delta(q, a_1, a_2) = (b_1, b_2, L \text{ or } R, L \text{ or } R, q') \).

A two-tape TM is useful for recognizing the language \( L = \{w\#w\} \), where \# is a special symbol denoting the middle of the string. With two tapes, we can first read \( w \) and copy it onto the second tape until we reach \#. Then, we match the contents of the two tapes to verify whether or not the string \( w \) before \# is the same as the string following \#. While it is still possible to recognize this language with a single-tape TM, the algorithm is more straightforward with two tapes.

Nondeterministic Turing Machine Turing machines can also be nondeterministic, meaning that there are multiple possible transitions from a given configuration. The transition function is of the form \( \delta(q, a) = \{(b, L, q'), (c, R, q''), (d, R, q)\} \), where elements of the set correspond to potential following configurations. In this case, we consider a configuration to follow from another if there exists a transition between them. Just like there is an equivalent DFA for any NFA, you can simulate a nondeterministic TM with a regular TM.