$M = (Q, \Sigma, \delta, s)$

$Q$: finite set of states; $s \in Q$

$\Sigma$: finite set of symbols; $\triangleright, \square \in \Sigma$

$\delta: Q \times \Sigma \to (Q \cup \{h\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$

| s | □ | 1 | 1 | 0 | 1 | □ | □ | ... |

TM’s are exactly like DFA’s, except

- They may move **either way** on their tape
- They may **change** tape contents
- They have **unlimited extra memory** on the right end of the tape
Giving a DFA some but not all of these capabilities gives some intermediate models of computation:

- The **two-way DFA** can still only decide regular languages, though perhaps with many fewer states than the corresponding ordinary DFA. Proving this is a good exercise in the use of the Myhill-Nerode Theorem.

- The **linear-bounded automaton** can change its tape, but must stay within the bounds of the original input. It recognizes the class we’ll later call \( \text{DSPACE}(n) \) and has a corresponding grammar definition.
<table>
<thead>
<tr>
<th>mvRt.tm</th>
<th>s</th>
<th>q</th>
<th>q₀</th>
<th>q₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s, 0,</td>
<td>q₀, ⊥,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s, 1,</td>
<td>q₁, ⊥,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>⊥</td>
<td>q, ⊥,</td>
<td>s, 0,</td>
<td>s, 1,</td>
<td></td>
</tr>
<tr>
<td>⊢</td>
<td>s, ⊢,</td>
<td>h, ⊢,</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example TM**

| comment | find ⊥ | memorize & erase | change ⊥ to 0 | change ⊥ to 1 |

```
s  □  1 1 0 1  □  □  ...  
s  □  1 1 0 1  □  □  ...  
s  □  1 1 0 1  □  □  ...  
s  □  1 1 0 1  □  □  ...  
s  □  1 1 0 1  □  □  ...  
s  □  1 1 0 1  □  □  ...  
q  □  1 1 0 1  □  □  ...  
q₁ □  1 1 0 1  □  □  ...  
s  □  1 1 0 1  □  1  □  ...
```
\[
\begin{array}{c|c|c|c|c}
\text{mvRt.tm} & s & q & q_0 & q_1 \\
\hline
0 & s, 0, \rightarrow & q_0, \Box, \rightarrow & & \\
\hline
1 & s, 1, \rightarrow & q_1, \Box, \rightarrow & & \\
\hline
\Box & q, \Box, \leftarrow & s, 0, \leftarrow & s, 1, \leftarrow & \\
\hline
\triangleright & s, \triangleright, \rightarrow & h, \triangleright, \leftarrow & & \\
\end{array}
\]

\[
\begin{align*}
s & \quad \begin{array}{c|c|c|c|c} \triangleright & 1 & 1 & 0 & 1 & \Box & \Box & \ldots \\
\end{array} \\
\vdots & \quad \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
q_1 & \quad \begin{array}{c|c|c|c|c} \triangleright & 1 & 1 & 0 & \Box & \Box & \Box & \Box & \ldots \\
\end{array} \\
s & \quad \begin{array}{c|c|c|c|c} \triangleright & 1 & 1 & 0 & \Box & 1 & \Box & \Box & \ldots \\
\end{array} \\
q & \quad \begin{array}{c|c|c|c|c} \triangleright & 1 & 1 & 0 & \Box & 1 & \Box & \Box & \ldots \\
\end{array} \\
q_0 & \quad \begin{array}{c|c|c|c|c} \triangleright & 1 & 1 & \Box & \Box & 1 & \Box & \Box & \ldots \\
\end{array} \\
s & \quad \begin{array}{c|c|c|c|c} \triangleright & 1 & 1 & \Box & 0 & 1 & \Box & \Box & \ldots \\
\end{array} \\
\vdots & \quad \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
q & \quad \begin{array}{c|c|c|c|c} \Box & 1 & 1 & 0 & 1 & \Box & \Box & \ldots \\
\end{array} \\
h & \quad \begin{array}{c|c|c|c|c} \Box & 1 & 1 & 0 & 1 & \Box & \Box & \ldots \\
\end{array}
\end{align*}
\]
Ancient Greece: Axiomatization of Geometry

Early 19th Century: Non-Euclidean Geometry, Independence of Parallel Postulate (Gauss, Bolyai, Lobachevsky)

Later 19th Century: Rigorous Foundation of Calculus, Real Analysis

1901: Hilbert proposes complete axiomatization of all mathematics, which would reduce all proof to mechanical procedure

1930’s: Active interest in the question of what exactly a “mechanical procedure” might be
Formal Models for Mechanical Procedures:
Church: Lambda calculus
Gödel: Recursive function
Kleene: Formal system
Markov: Markov algorithm
Post: Post machine
Turing: Turing machine

Theorem: If $A$ and $B$ are any two of the systems above, and $f$ is a function (say, from bit strings to bits), then $f$ is formalizable in $A$ iff $f$ is formalizable in $B$.

Church-Turing Thesis: The intuitive idea of “effectively computable” is captured by the precise mathematical definition of “computable” in any of the above models.
“Why is a Turing machine as powerful as any other computational model?”

Intuitive answer: Imagine any computational device. It has:

- Finitely many states
- Ability to scan limited amount per step: one page at a time
- Ability to print limited amount per step: one page at a time
- Next state determined by current state and page currently being read (but what about randomization?)

**Note:** Without the potentially infinite supply of tape cells, paper, extra disks, extra tapes, etc. we have just a (potentially huge) **finite state machine**.

The PC on your desk, with 20 GB of hard disk is a finite state machine with over $2^{160,000,000,000}$ states!

This is better modeled as a TM with a bounded number of states, and an “infinite tape”, actually meaning a finite memory that expands whenever necessary.
We have so far defined the behavior of a Turing machine – what it will do on a particular input. Now we must define its semantics – the way we assign meaning to its behavior.

A Turing machine, once started, may or may not eventually halt. It could fail to halt in a number of ways: run off the left end of the tape, enter a loop of repeated identical configurations, or keep expanding the area of tape it uses forever. If it does halt, we want to define what its completed computation means.

One semantics dating back to Turing’s original work is to say that the Turing machine accepts its input if it halts, and rejects its input if it doesn’t halt. The language of the machine is then defined to be the set of strings that it accepts.

While simple and useful for some purposes, this semantics doesn’t allow us to distinguish among always-halting computations, which after all are our main area of interest.
We will design our Turing machines to have understandable behavior. In particular, we will design them to compute functions from strings to strings in a particular format:

\[
M(w) \equiv \begin{cases} 
y & \text{if } M \text{ on input } \triangleright w \sqsubseteq \text{ eventually halts with output } \triangleright y \sqsubseteq \\
\rightarrow & \text{otherwise}
\end{cases}
\]

\[\Sigma_0 \equiv \Sigma - \{\triangleright, \sqsubseteq\}\]

Usually, \(\Sigma_0 = \{0, 1\}\)

**Definition 7.1** Let \(f : \Sigma_0^* \rightarrow \Sigma_0^*\) be a total or partial function. We say that \(f\) is recursive iff \(\exists\ TM M, f = M(\cdot)\), i.e.,

\[
(\forall w \in \Sigma_0^*) \quad f(w) = M(w) .
\]
Remark 7.2 There is an easy to compute, 1:1 and onto map between \( \{0, 1\}^* \) and \( \mathbb{N} \). Thus we can think of the contents of a TM tape as a natural number and talk about \( f : \mathbb{N} \to \mathbb{N} \) being recursive. (We may visit this issue in HW#3.)

A partial function \( f : \mathbb{N} \to \mathbb{N} \) is a total function \( f : D \to \mathbb{N} \) where \( D \subseteq \mathbb{N} \). A partial function that is not total is called strictly partial. If \( n \in \mathbb{N} - D \), \( f(n) = \uparrow \).
Definition 7.3  Let $S \subseteq \Sigma_0^*$ or $S \subseteq \mathbb{N}$.

$S$ is a recursive set iff the function $\chi_S$ is a (total) recursive function,

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

$S$ is a recursively enumerable set ($S$ is r.e.) iff the function $p_S$ is a (partial) recursive function,

$$p_S(x) = \begin{cases} 1 & \text{if } x \in S \\ \uparrow & \text{otherwise} \end{cases}$$
There is also a common alternate terminology for these two concepts:

- Recursive sets are called **Turing decidable** because an always-halting TM can be designed to output 1 for inputs in the set and 0 for inputs not in it.

- Recursively enumerable sets are called **Turing acceptable** because of the semantics mentioned above – a TM can be designed to halt on inputs in the set and not halt on inputs not in it.

- The word **enumerable** is from another semantics – a set is r.e. iff a TM can be designed that will list all the elements of the set, running forever if the set is infinite.
Proposition 7.4 If $S$ is recursive then $S$ is r.e.

Proof: Suppose $S$ is recursive and let $M$ be the TM computing $\chi_S$.

Build $M'$ simulating $M$ but diverging if $M(x) = 0$. Thus $M'$ computes $p_S$.

We will see that the converse of this proposition is false, as there are sets that are r.e. without being recursive.
Some Recursive Functions

Proposition 7.5 The following functions are recursive. They are all total except for \( p_{even} \).

\[
\begin{align*}
copy(w) &= ww \\
\sigma(n) &= n + 1 \\
\text{plus}(n, m) &= n + m \\
\text{times}(n, m) &= n \times m \\
\exp(n, m) &= n^m \\
\chi_{even}(n) &= \begin{cases} 
1 & \text{if } n \text{ is even} \\
0 & \text{otherwise} 
\end{cases} \\
p_{even}(n) &= \begin{cases} 
1 & \text{if } n \text{ is even} \\
\text{otherwise} & 
\end{cases}
\end{align*}
\]

Proof: Exercise: please convince yourself that you can build TMs to compute all of these functions! ♠
If $\mathcal{C}$ is any class of sets, define co-$\mathcal{C}$ to be the class of sets whose complements are in $\mathcal{C}$,

$$\text{co-}\mathcal{C} = \{ S \mid \overline{S} \in \mathcal{C} \}$$

**Theorem 7.6** $S$ is recursive iff $S$ and $\overline{S}$ are both r.e.

**Thus,** $\text{Recursive} = \text{r.e.} \cap \text{co-r.e.}$

**Proof:**

(\subseteq \text{ direction})

If $S \in \text{Recursive}$ then $\chi_S$ is a recursive function by the definition.

Therefore $\chi_{\overline{S}}(x) = 1 - \chi_S(x)$ is also a recursive function.

Thus, $S$ and $\overline{S}$ are both recursive and thus both are r.e.
(other direction)

Suppose $S \in \text{r.e.} \cap \text{co-r.e.}$.

By the definition two machines $M$ and $M'$ exist, such that for all inputs $x$, $p_S(x) = M(x)$ and $p_{\overline{S}}(x) = M'(x)$.

We define a new machine $T$ that runs $M$ and $M'$ in parallel. On input $x$, $T$ does:

1. **for** $n := 1$ to $\infty$ {
   2. run $M(x)$ for $n$ steps.
   3. if $M(x) = 1$ in $n$ steps then **return**$(1)$
   4. run $M'(x)$ for $n$ steps.
   5. if $M'(x) = 1$ in $n$ steps then **return**$(0)$
}

Thus, $T(x) = \chi_S(x)$, $\chi_S$ is a recursive function, and thus $S \in \text{Recursive}$.

\[\spadesuit\]