We have seen several abstract models of computing devices:

*Deterministic Finite Automata, Nondeterministic Finite Automata, Nondeterministic Finite Automata with $\epsilon$-Transitions, Pushdown Automata, and Deterministic Pushdown Automata.*

However, none of the above “seem to be” as powerful as a real computer, right?

We now turn our attention to a much more powerful abstract model of a computing device: a **Turing machine**. This model is believed to do everything that a real computer can do.
A Turing machine is somewhat similar to a finite automaton, but there are important differences:

1. A Turing machine can both write on the tape and read from it.

2. The read-write head can move both to the left and to the right.

3. The tape is infinite.

4. The special states for rejecting and accepting take immediate effect.
Turing Machines: An Introduction

Formally, a Turing machine is a 7-tuple,

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where $Q$ is the (finite) set of states, $\Sigma$ is the input alphabet such that $\Sigma$ does not contain the special blank symbol $\sqcup$, $\Gamma$ is the tape alphabet such that $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

is the transition function, $q_0 \in Q$ is the start state, $q_{\text{accept}} \in Q$ is the accept state, and $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{accept}} \neq q_{\text{reject}}$.

The heart of a Turing machine is its transition function $\delta$, as it tells us how the machine gets from one configuration to another. A Turing machine configuration is entirely described by its current state, the current tape contents, and the current head location.
For a state $q$ and two strings $u$ and $v$ over the tape alphabet $\Gamma$, we write $u \ q \ v$ for the configuration where the current state is $q$, the current tape contents is $uv$, and the current head location is the first symbol of $v$. For instance,

$$1011 \ q_7 \ 0111$$

represents the configuration when the tape is $10110111$, the current state is $q_7$, and the head is currently on the second $0$ of $10110111$ from left to right.

We say that configuration $C_1$ yields configuration $C_2$ if the Turing machine can legally go from $C_1$ to $C_2$ is a single step. This notion can be formalized via the transition function.
Let $a$, $b$, and $c$ be symbols of $\Gamma$, let $u$ and $v$ be strings over $\Gamma$, and let $q_i$ and $q_j$ be any two states (not necessarily distinct) in $Q$. Then, we say that

$$ua \, q_i \, bv \quad \text{yields} \quad u \, q_j \, acv$$

if in the transition function $\delta(q_i, b) = (q_j, c, L)$. That handles the case where the Turing machine moves leftward. For a rightward move, we say that

$$ua \, q_i \, bv \quad \text{yields} \quad uac \, q_j \, v$$

if in the transition function $\delta(q_i, b) = (q_j, c, R)$. 
A special case occurs when the head is at the left-hand end of the configuration.

For the left-hand end, the configuration $q_i \ b v$ yields $q_j \ c v$ if the transition is left moving (i.e., the head does not move past the left-hand end), and it yields $c \ q_j \ v$ for the right moving transition.

For the right-hand end, the configuration $u a \ q_i$ is equivalent to $u a \ q_i \ \square$, as we assume that blanks follow the part of the tape represented in the configuration.

The **start configuration** of a Turing machine on input $w$ is the configuration $q_0 \ w$, which indicates that the machine is in the start state $q_0$ with its head at the leftmost position on the tape.
Turing Machines: An Introduction

An accepting configuration is a configuration in which the state is the accept state.

A rejecting configuration is a configuration in which the state is the reject state.

Accepting and rejecting configurations are halting configurations and accordingly do not yield further configurations.

A Turing machine $M$ accepts input $w$ if a sequence $C_1, C_2, \ldots, C_k$ of configurations exists such that $C_1$ is the start configuration of $M$, $C_i$ yields $C_{i+1}$, for all $1 \leq i \leq k - 1$, and $C_k$ is an accepting configuration. The set of strings that $M$ accepts is the language of $M$, denoted $L(M)$. 
Turing Machines: An Introduction

Let us use the blackboard to design a Turing machine that accepts the language

\[ L = \{ wcw \mid w \in \{0, 1\}^* \}. \]

Informally, our machine accepts \( L \) by executing the following algorithm:

1. Scan the tape from left to right to make sure the input string has exactly one symbol \( c \).
2. Zig-zag across the tape to corresponding positions on either side of \( c \) to check on whether these positions contain the same symbol. If they do not, reject. Cross off symbols as they are checked to keep track of which symbols correspond. If they do not, reject. (3) When all symbols to the left of \( c \) have been crossed off, check for any remaining symbols to the right of \( c \). If any symbols remain, reject; otherwise accept.
Turing Machines: An Introduction

Our Turing machine has 16 states, one of which is the accept state and another one is the reject state.

The input alphabet is \( \Sigma = \{0, 1, c\} \).

The tape alphabet is \( \Gamma = \{0, 1, c, X, \square\} \).

We write the symbol \( X \) on the tape to represent the action of “crossing off” an input symbol.

Let us use the black board to draw a diagram for our Turing machine.
We say that a language is **recursively enumerable** if it is accepted by some Turing machine.

For instance, the language \( L = \{wcw \mid w \in \{0, 1\}^*\} \) is recursively enumerable.

When a Turing machine starts, there are three possible outcomes: The machine may accept, reject, or *loop*. By loop, we mean that the machine simply does not halt.

A Turing machine can fail to accept an input string by entering the \( q_{\text{reject}} \) state and rejecting, or by looping. Sometimes, distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that always halts, i.e., never loops.
Turing Machines: An Introduction

Turing machines that halts on all input strings are called deciders, as they always make a decision to accept or reject.

A decider that recognizes some language is said to decide that language.

We say that a language is recursive if there exists a Turing machine to decide the language. For instance, the language $L = \{wcw \mid w \in \{0, 1\}^*\}$ is recursive.

Note that every recursive language is also recursively enumerable, but a recursively enumerable language is not necessarily a recursive language.
The following languages are all recursive languages (go ahead and try to design deciders for them):

- \{a^n \mid n \text{ is a perfect square}\}

- \{ww \mid w \in \{0, 1\}^*\}

- \{a^n \mid n \text{ is a power of 2}\}

- \{x \in \{a, b, c\}^* \mid |x|_a = |x|_b = |x|_c\}
Turing Machines: An Introduction

Recall that all the languages in the previous slides are not context-free languages. You can use the contrapositive of the Pumping Lemma to show this fact. Actually, we have the following:

Every regular language is a context-free language, but a context-free language may not be regular.

Every context-free language is a recursive language, but a recursive language may not be context-free.

Every recursive language is a recursively enumerable language, but a recursively enumerable language may not be recursive.
In 1900, mathematician David Hilbert enumerated 23 mathematical problems he considered challenges for the coming century. The tenth problem is his list concerned *algorithms*:

*Given a polynomial, find an “algorithm” for determining whether the polynomial has an integral root.*

In the way Hilbert phrased the above problem, he explicitly asked that an algorithm be “devised”.

As we now know, no algorithm exists for this task if the polynomial is defined on several variables. That is, the above problem is algorithmically unsolvable. However, mathematicians of that period could not come to this conclusion with their intuitive notion of algorithm. Why?
Turing Machines: An Introduction

Although our intuitive notion of an algorithm is enough to “devise” solutions for certain problems (the ones a computer can solve), it is useless for showing that no algorithm exists for a particular task, as our intuitive notion is not a formal one.

So, proving that an algorithm does not exist required having a clear definition of algorithm.

Such a definition was simultaneously given in 1936 by Alonzo Church and Alan Turing.

Church used a notational system called the $\lambda$-calculus to precisely define algorithms.

Turing did it with his “machines”.

©Marcelo Siqueira — Spring 2005
This connection between the informal notion of algorithm and the precise definition is known as the **Church-Turing thesis**. More precisely, Turing proposed to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to our intuitive notion of an “algorithm”.

Note that the Church-Turing thesis is not a theorem, but a “thesis”, as it asserts that a certain informal concept (algorithm) corresponds to a certain mathematical object (Turing machine). Not being a mathematical statement, the Church-Turing thesis cannot be proved! What are the implications of this fact?

Well, it is in principle possible that the Church-Turing thesis can be disproved.
To disprove the Church-Turing thesis, it suffices to come up with an alternative model of computation that is provably capable of carrying out computations that cannot be carried out by any Turing machine. It turns out that no one believes this is likely.

Having said that, let us assume that the Church-Turing thesis holds. We are not the only ones, I guarantee!

Adopting such a precise notion of an algorithm enables us to formally prove that certain problems *cannot* be solved by *any* algorithm. Such problems are said to be **undecidable** or **unsolvable**.

The tenth problem proposed by Hilbert was shown to be undecidable by Yuri Matijasevič in 1970.
To be more precise and consistent with our introductory lecture, let $D$ be the language

$$D = \{ p \mid p \text{ is a polynomial in several variables with an integral root} \}.$$ 

Here, we view $D$ as the language consisting of all polynomials in several variables with an integral root. So, the decision problem of determining whether a given polynomial $p$ with several variables has an integral root (a decision problem) is equivalent to the problem of determining whether $p$ belongs to $D$.

Matijasevič showed that there is no decider for $D$. That is, the language $D$ is not a recursive language. In constrast, we can show that $D$ is recursively enumerable. That is, we can devise a Turing machine $M$ that accepts $D$, but $M$ may never halt on an input $p$ that does not belong to $D$. 

©Marcelo Siqueira — Spring 2005
Since Turing machines can only deal with decision problems, you may find very strange the fact that a Turing machine is a formal counterpart of our intuitive notion of an algorithm, as many computational problems are not decision problems.

Well, keep in mind that every computational problem is either equivalent to a decision problem or is at least as “hard” as some decision problem. So, the restriction to decision problems is not a problem.

Besides helping us understand what we can or cannot do with computers, Turing machines can also be used to formally define the complexity of an algorithm.

You can find much more about computability and complexity theory in graduate courses at CIS (see CIS 511 and CIS 502).