A **Turing Machine** is yet another computational abstraction. Here are the key differences between a Turing Machine and previously discussed automata:

1. We can read and write to the tape
2. The read-write head can move left or right
3. The tape is infinite
4. Accept and reject take effect immediately

Formally, a Turing Machine is a 7-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

1. Q is the set of states
2. Σ is the input alphabet.
   - We also have a blank symbol $\sqcup$
3. Γ is the tape alphabet
   - $\sqcup \in \Gamma$
   - $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
5. $q_0 \in Q$ is the start state
6. q_{accept} is the accept state
7. q_{reject} is the reject state

There are two key terms we will need:

- A language is said to be **Turing Recognizable** if some Turing Machine recognizes it. That is, there exists some Turing Machine that can hit the accept state for exactly this set of input. This is sometimes known as *recursively enumerable languages*.

Question 1. How could a Turing Machine **fail** to recognize a string?

- A Turing Machine that never enters an infinite loop is called a *decider*. A decider that recognizes a language is said to *decide* it.

Question 2. Give a Turing Machine that recognizes $L_1 = \{a^{2^n} : n \geq 0\}$.

Question 3. Give a Turing Machine that recognizes $L_2 = \{w\#w : w \in \{a, b\}^*\}$

Question 4. Give a Turing Machine that recognizes $L_3 = \{a^i b^j c^k : i \times j == k \text{ and } i, j, k \geq 1\}$

The following is approximately what would be one lecture, but one that I think works better in this format, given our time constraints. More detail about this topic can be found in chapter three of the Sipser textbook, among other places. Of course, feel free to ask about the topics on this page and the next, or any other questions you have, on Ed Discussion.

Question 5. Consider a machine definition that is exactly like a Turing Machine, except upon transition, in addition to the options to move the read/write head left or right, we also have the option to keep it where it is ("stay put"). How does the set of languages recognizable by this machine differ from those recognized by a traditional Turing Machine?

Think about what you think is the answer before reading the next two paragraphs.

Answer: Nothing new is gained by this. Note that the "old" style Turing Machines fit into this already: just because we can stay put doesn't mean we have to ever use that option. Therefore, anything the old-style can do, this new style can do also.

Furthermore, nothing new is gained. Suppose I have one of these machines with one or more "stay put" instructions. I can replace, one by one, each with a transition that moves. Instead of staying put, that transition now moves right, and instead of its previous destination state, it goes to one newly created for this purpose. From that newly created state, without affecting the tape, and regardless of what is on it, move left. This has the same effect as a stay put and fits within the model of the original Turing Machines.

Question 6. Consider a machine definition that is exactly like a Turing Machine, except instead of one tape with one read-write head, it can have multiple. How does the set of languages recognizable by this machine differ from those recognized by a traditional Turing Machine?

This one also is not more powerful. Our current TMs fit within this, as we could choose to create a machine that has just one. Therefore, anything the old-style can do, this new style can do also (we didn't lose anything).

But did we gain anything? No. Suppose I want a Multi-tape TM (MTM) and have a TM. I can simulate the contents by partitioning the tape with a special separator character. If I ever need to expand, I can move everything down by one, similar to how is done in Insertion Sort. For each tape, we can use a special symbol to track the R/W head (and what is underneath it, so a different "R/W head is here with an a underneath it" from "R/W head is here with a b underneath it."

This might sound inefficient to you. You might be correct. We will discuss efficiency later; for now, the question is what can and cannot be computed.

Question 7. Consider a machine definition that is exactly like a Turing Machine, except transitions can be non-deterministic. With finite state automata, DFAs and NFAs recognized the same set of languages, while non-determinism allowed PDAs to recognize a larger set of languages than deterministic PDAs. How does the set of languages recognizable by a non-deterministic Turing Machine differ from those recognized by a traditional Turing Machine?

It might be a little surprising, but every nondeterministic TM has an equivalent deterministic TM. The proof idea is to imagine the execution tree produced by a non-deterministic TM, with a branch on every opportunity for a non-deterministic choice. We can use a multi-tape TM (or equivalent) to simulate non-determinism and do a BFS of all non-deterministic choices.

A Turing Machine (including the variants we discussed) is not the only model of computation. Among the others are:

- **Enumerators**, which are like a Turing Machine with a printer. These generate and print all the strings in a given language.
- Abacus Machines
- Lambda (λ) calculus.

Many other equivalents exist. As a general rule, the equivalent ones can be described by us as doing only finite amounts of work per step. There are some that are more powerful than Turing Machines, but have unreasonable descriptions, such as being able to do infinitely much work in a single step, known as a "supertask." Just like algorithms can be implemented in different programming languages, so too with computational models.

This brings me to the **Church-Turing Thesis**, due to Alonzo Church (via lambda calculus) and Alan Turing (Turing Machines). The Wikipedia page has an interesting section on this, including the history and philosophical implications. The Church-Turing thesis essentially says that what you and I (and many other people) think of as algorithms is exactly what can be computed by Turing Machines and equivalents.

It might surprise you that the definition of algorithm was not finalized until the 20th century, despite what we think of as algorithms pre-dating this. When I teach CompSci 161, for example, one of the algorithms I present dates to the 9th century. The Euclidean Algorithm for gcd, which you probably saw in Discrete Math, is over 2300 years old.

One final item for your consideration: think about why it is worth studying unsolvable problems. I do not mean this in the context of "Professor Shindler gives really hard homework questions." I mean problems that a computer cannot solve, such as testing whether or not a polynomial has integer roots.