Finite Automata and Turing Machines

Bryant Villamil and Nikhil Sardana

November 2019

1 Computation

What *is* a computer, exactly? Is it your phone? A desktop? A powerful server sitting in some cluster? What about PowerPoint? Maybe, you say, the first three are computers, but the last one—that's just a piece of software. It isn't a computer, it runs *on* computers. Computers, you might define, are machines that answer questions. They *solve problems*.

I might ask you then: *Well, what type of problems can a computer solve?*

If I ask you to calculate the 50^{th} root of 100 trillion, why, your phone could have the answer in half a second. But a computer from the 1960s, well, it might take millennia to find the answer. And yet, these are both generally defined as "computers." So, as our processing power has grown, has the set of "computable" problems grown over time?

A nice counterargument would be: Well, both a 1960s computer and an iPhone XS can compute the same problems, it will just take the old IBM mainframe a lot more time.

And this is the way we'll be looking about computibility today. We won't be asking "Which problems can we hope to compute *efficiently*?" (That's a topic for another day.) Rather, we want to ask: "Which problems can we hope to compute *at all*?"

You might say: Given enough time and Moore's Law, every problem is computable. And it turns out, this isn't true. There are some problems which we simply cannot solve.

And in this case, it would be really helpful for us to have a model for computation. We'd like a simple model, for which we can say: If this model can solve this problem, then we can compute it using our real-life computers, given enough time. And if this model can't solve a problem, then our computers haven't got a chance, no matter how much time we give them.

We can't just choose the latest, biggest, fastest supercomputer, and make that our model for computation, because the fastest computer, along with the problems it can solve, change every year. We need a more abstract model of computation.

Today, we'll be looking at two possible models of computation. One, **finite automata**, have a finite amount of memory. The other, **Turing machines**, have infinite memory.

We're first going to discuss automata. After all, our real-world computers have finite memory. Shouldn't our models have finite memory too?

2 Languages

Before we talk about models of computation, we need to ask ourselves: what is a *problem*?

We'll, that's a vague question. A problem could be "Is 1331 a perfect cube?" or "Do lemurs live in Peru?" A problem has a set of correct answers, and a set of incorrect answers.

For today, we're going to define all yes or no problems as set membership problems. For example, the question "Is x a perfect cube?" is really the same thing as "Is x in the set $\{1, 8, 27, \dots\} = \{n^3 \mid n \in \mathbb{N}\}$?" We're going to encode all our set elements as finite strings, with each string made up of certain characters limited to a finite alphabet.

Remember, we can reframe *any* yes/no question as a set membership problem, even ones that we don't think of as "computation". How would we change "Do lemurs live in Peru?" into a set membership problem? Well, there are finitely many species in the world. Assign a different binary string to each creature (101 = lemur, 000 = panda, 110 = ocelot, etc.). Encode each country's fauna as one long string, which concatenates all strings for species living in the country. The question "Do lemurs live in Peru?" now becomes: Does Peru's fauna string p contain 101? In other words, is p in the set of all binary strings that contain 101?

Formally, this way of framing problems is called *language membership*, and there's a few definitions we need to get out of the way before we dive in.

- **Alphabets** are finite, nonempty sets of characters. For example, $\Sigma = \{a, b, c\}$.
- **Strings** are finite sequences of characters.
- **Languages** are sets of strings.
- ϵ denotes the empty string.
- A **Language over the alphabet** Σ is a set of strings made of characters from Σ.
- Σ^* is the set of all strings made from characters in Σ (this includes ϵ).

Example 2.1. Let $\Sigma = \{a\}$ be an alphabet. The following are some languages over Σ: $\{\epsilon\}$, $\{aaa\}$, $\{a, a\}$, $\{aa, aaa, a\}$. Here, $\Sigma^* = \{\epsilon, a, aa, aaa, aaaa, aaaaa, \dots\}$.

Example 2.2. Let $\Sigma = \{0, 1\}$ be an alphabet. The following are some languages over Σ: $\{\epsilon\}$, $\{00, 1\}$, $\{101, 0\}$, $\{1, 0, 11\}$. Here, $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$.

Any yes or no computational problem can be framed as a language membership problem. So, if we want a model of computation, all we need is a model that takes in an input string, and checks if that string is in a language.

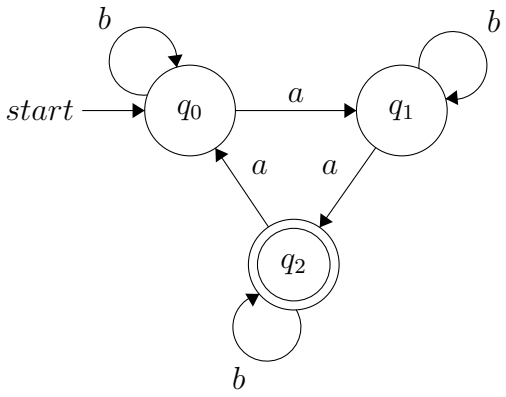
For example, let's say you're writing a JavaScript program for a website to check if a string is a valid e-mail address. You might probably check for an `@` sign along with a `.com` at the very end of the input. If the input string contains these two parameters, then you accept it as a valid email address.

Alternatively, we can reframe this program as a problem involving languages. We have a language $L = \{x | x$ is a string and x contains an @ and a .com$\}$. Our email address program is the same thing as asking: for some string s, is $s \in L$?

We need a model that is simple enough for us to understand and work with, yet powerful enough to tell us if a string s is in a language L or not, for some very complicated languages (more complex than the one above). The first such model we'll explore is the **deterministic finite automata**.

3 Deterministic Finite Automata

A **finite automaton** is a model used to determine whether a string is in a language. Finite automata consist of **states**, which are locations in an automaton (represented as circles), and **transitions**, paths between states (represented as arrows).



An automata reads a string over some alphabet ($\Sigma = \{a, b\}$) beginning at the **start state** (q_0). The automata reads the next character in the string and follows the corresponding transition. Once the automata reads all the characters in the string, it accepts the string if it's in an **accepting state** (q_2) or rejects it otherwise.

For example, in the above DFA, on the input string bba, we start at q_0, read the first b, transition to q_0, read the second b, transition to q_0 again, read the a, transition to q_1, and then reject.

If every state has a transition for every character in the alphabet Σ, it is *deterministic* and called a **deterministic finite automaton** (DFA).

The **language of a DFA** is the set of all strings the DFA accepts. We say a DFA D *accepts* or *recognizes* a language L if L is the language of the DFA.

Example 3.1. Which of the following strings does the DFA above accept?

(a) aabbaa

(b) baba

(c) aaabaaabaa

(d) aa

(e) aaaaa

Solution. The language of the above DFA is the set of all strings made of a's and b's where the amount of a's mod 3 equals 2. In set builder notation it would look like $\{w \in \{a, b\}^* \mid \text{number of a's in } w \mod 3 = 2\}$. □

DFAs are a compelling model for language membership problems because they are simple to construct and reason about. A DFA D accepting a language L is essentially a very limited computer that solves the language membership problem for L. For any string $s \in \Sigma^*$, if $s \in L$, D accepts s. Otherwise, D rejects s.

3.1 Practice Questions

1. What language does the following DFA accept?

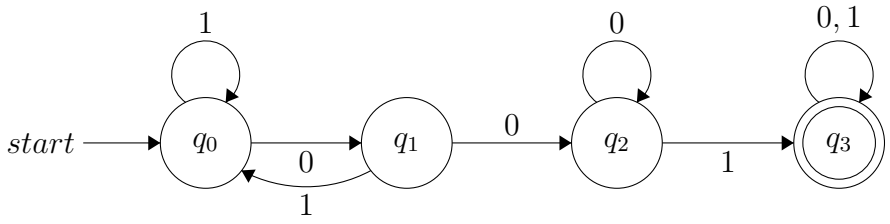


Figure 1: A Mystery DFA.

2. Draw a DFA that accepts the language consisting of all binary strings modulo 6: $\{s \mid s \in \{0,1\}^* \text{ and } s \mod 6 = 0\}$.

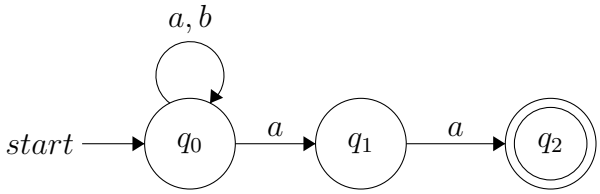
3.2 Limitations

However, you might have qualms about the state-transition model of a DFA. It seems somewhat limiting. How powerful can a DFA be when we can only have finitely many states? In other words, how complex of a language can a DFA recognize? Are there languages that no DFA can recognize?

Yes! There are many languages DFAs cannot recognize. In fact, DFAs can only recognize a small class of languages, called **regular languages**. We will discuss this further in Section 5. But first, let's relax one requirement of the DFA: determinism. Can we get a more powerful model if we allow ourselves more freedom with our transitions?

4 Nondeterministic Finite Automata

An automaton is **nondeterministic** if at any state there are zero or more transitions for a character in the alphabet.



The automaton accepts a string if *any* series of valid transitions leads to an accepting state. If a character *ch* is read on a state that has no transition for *ch*, the automata rejects the string. For example, in the above NFA, the strings *aa*, *abaa*, *abbaa* are accepted, but *ab* and *aba* are rejected.

Exercise 4.1. What is the language of accepted strings for the above NFA?

Exercise 4.2. What is the language of strings accepted by the following NFA?

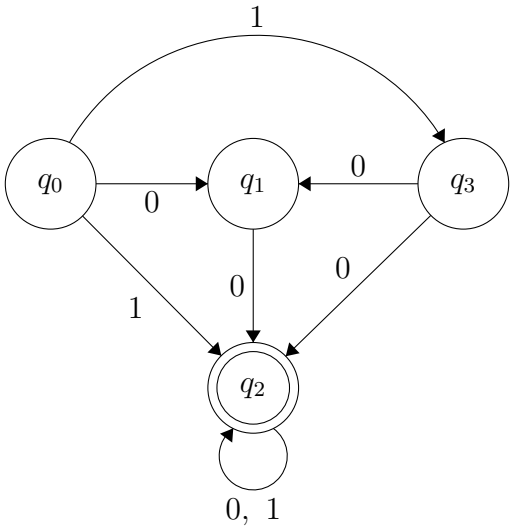
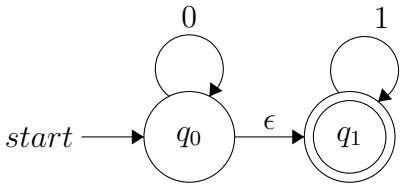


Figure 2: A more complex NFA. Or is it?

NFAs can also have special transitions, called **ϵ-transitions**, which can be optionally taken and consume no input. For example, in the NFA below, the string 0 is accepted, because we can first take the transition from $q_0 \to q_0$, and then the ϵ-transition from q_0 to q_1, where we end.



Example 4.1. Which of the following strings does the NFA above accept?

(a) 0001

(b) 0101

(c) 1

(d) ϵ

Solution. The language of the above NFA is the set of all strings made of 0's and 1's that does not contain 01 as a substring. In set builder notation it would look like $\{w \in \{0,1\}^* \mid$ w does not contain 01 as a substring$\}$. □

4.1 Practice Questions

1. What language does the following NFA accept?

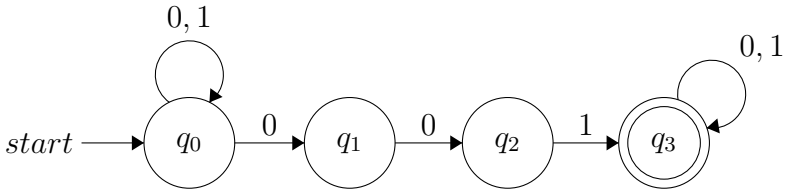


Figure 3: A Familiar NFA.

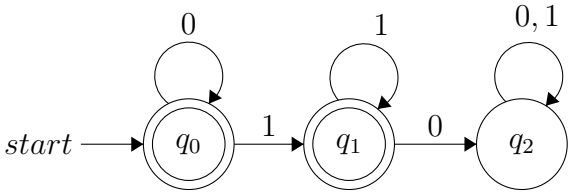
2. Draw a NFA that accepts the language consisting of all binary strings that end in an even number of zeros (00, 0000, 000000, etc.)

4.2 Equivalence of DFAs and NFAs

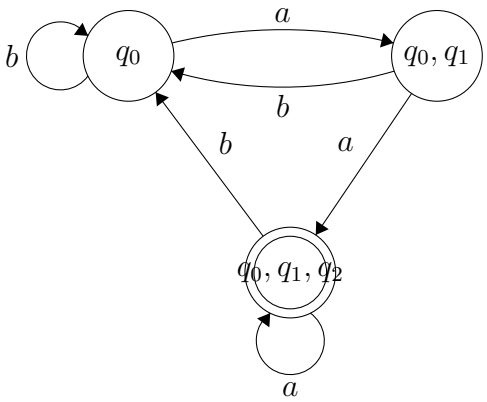
Since NFAs have fewer restrictions than DFAs, you might think that NFAs are more powerful—that there are languages that NFAs can recognize that no DFA can. However, NFAs and DFAs are equivalent. NFAs are simply a more convenient way of drawing automata. We won't formally prove the equivalence, but we'll give some intuition.

One direction of the bijection is trivial. DFAs are just NFAs with more restrictions, so every DFA is an NFA. Lets try to see why every NFA has an equivalent DFA.

The NFA in Example 4.3 can be converted into the following DFA.

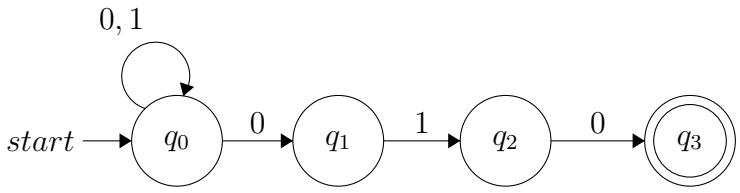


The NFA in Example 4.1 can be converted into the following DFA using a procedure called the subset construction.



The **subset construction** is a procedure of converting a NFA to a DFA. The above DFA was created by first creating a set of states that the start state q_0 transitions to when reading the character a. This set becomes the state $\{q_0, q_1\}$ in the DFA that q_0 transitions to when reading the character a. When the second character a is read, $\{q_0, q_1\}$ transitions a state which represents the set of all states that q_0 and q_1 collectively transition to in the NFA when reading a. In this case, $\{q_0, q_1\}$ transitions to $\{q_0, q_1, q_2\}$ when reading a. Following the procedure, $\{q_0, q_1, q_2\}$ transitions to itself. The same is done for the character b. Since the only state in the NFA that has a transition for b is q_0 which transitions to q_0, all the states in the DFA (including q_0, $\{q_0, q_1\}$, and $\{q_0, q_1, q_2\}$) transition to q_0.

Exercise 4.3. Make the following NFA into a DFA using the subset contruction.



Recall that DFAs and NFAs are models used to determine whether a string is in a language. Since a DFA is an NFA and an NFA can be converted into a DFA using the subset construction, DFAs and NFAs can recognize the same languages and thus are equally powerful!

5 Regular Languages

A **regular language** L is a language for which there exists a DFA that accepts only strings in L.

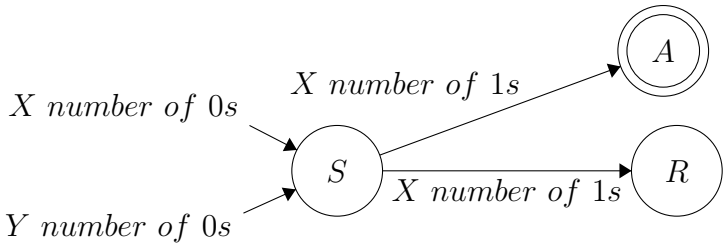
For example, the language $\{x \mid x \in \{0,1\}^* \text{ and } x \text{ contains } 001 \text{ as a substring}\}$ is regular, because the DFA from Figure 1 above decides this language.

A language L is **non-regular** if there is no DFA that recognizes L. This corresponds to a language that would require infinite memory to recognize.

For example, let $\Sigma = \{0,1\}$ and consider the language $L = \{0^n 1^n \mid n \in \mathbb{N}\} = \{\epsilon, 01, 0011, 000111, \dots\}$. Since $\mathbb{N}$ is infinite and every DFA must have a finite number of states, there must be some common state which the DFA ends up at after reading m 0s and n 0s, $m \neq n$. Otherwise, if each k 0s gets its own state, our DFA would have infinitely many states.

Consider the state S that is reached after reading X 0s and Y 0s where $X \neq Y$. Now, consider two input strings, $s_1 = 0^X 1^X$ and $s_2 = 0^Y 1^X$. After reading the first X characters of s_1, the remaining characters in s_1 consist of X 1s, so the DFA must end at an accept state (A) because there are X 0s followed by X 1s.

However, after the first Y characters of s_2, the DFA is also at state S. Reading the next X characters, the DFA must end at a reject state (R) and reject s_2, because $X \neq Y$, so $0^Y 1^X \notin L$. A picture illustrates this below.



However, the automata is deterministic! From state S, if we read the same remaining X characters we must end up at the same state, so $A = R$. But since we cannot have a state simultaneously accept and reject, $A \neq R$. Thus, we have a contradiction.

Therefore, we cannot double count a state! If the largest DFA has N states, and we give it a string that begins with $N+1$ 0s, the DFA has no remaining states that can keep track of the last 0! Therefore, any DFA needs infinitely many states to recognize $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. So, L is non-regular.

6 Turing Machines

Now that we've seen that DFAs and NFAs aren't powerful enough to decide non-regular languages (and the vast majority of languages we care about are non-regular), we need a more powerful model.

A Turing machine consists of a **tape**, which is infinitely long in one direction (we start at the left end) and a tape **head**. Initially, our input is written on the tape, followed by infinitely many blank tape cells, denoted by $\square$. Our input is a finite string, made up of characters from the **input alphabet** Σ.

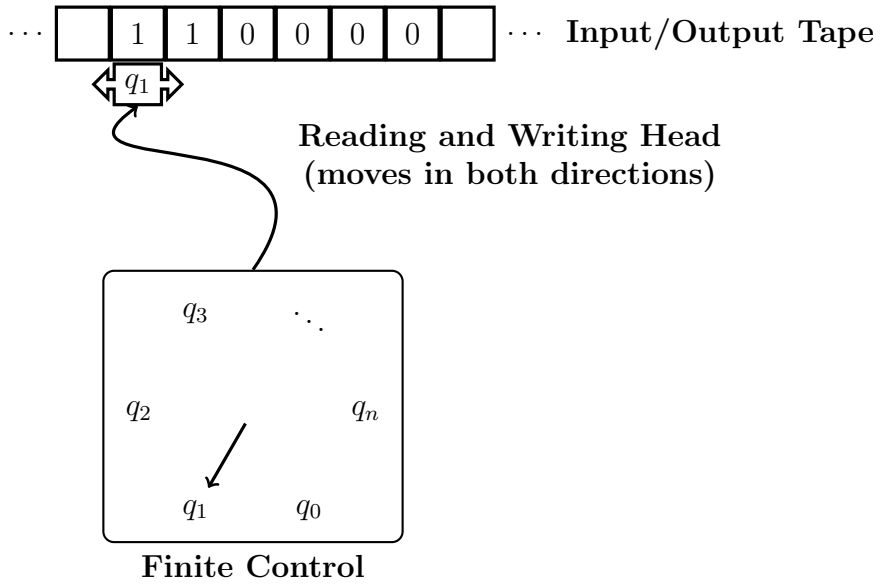


Figure 4: A Turing Machine with input alphabet $\Sigma = \{0, 1\}$ and head states $\{q_0, \ldots, q_n\}$.

The head of the Turing machine has finitely many states (just like the states of the DFA).

Our head starts at the left-most tape cell. At each step of our computation, the tape head reads the contents of a single cell, and changes state based on the cell's contents. Then, the head writes a character to the cell. Finally, the head moves either right or left.

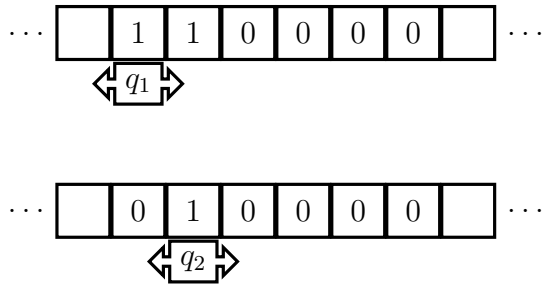


Figure 5: One step of the Turing machine. Between the first and second diagrams, the head read 1, changed state to q_2, wrote 0, and then moved right.

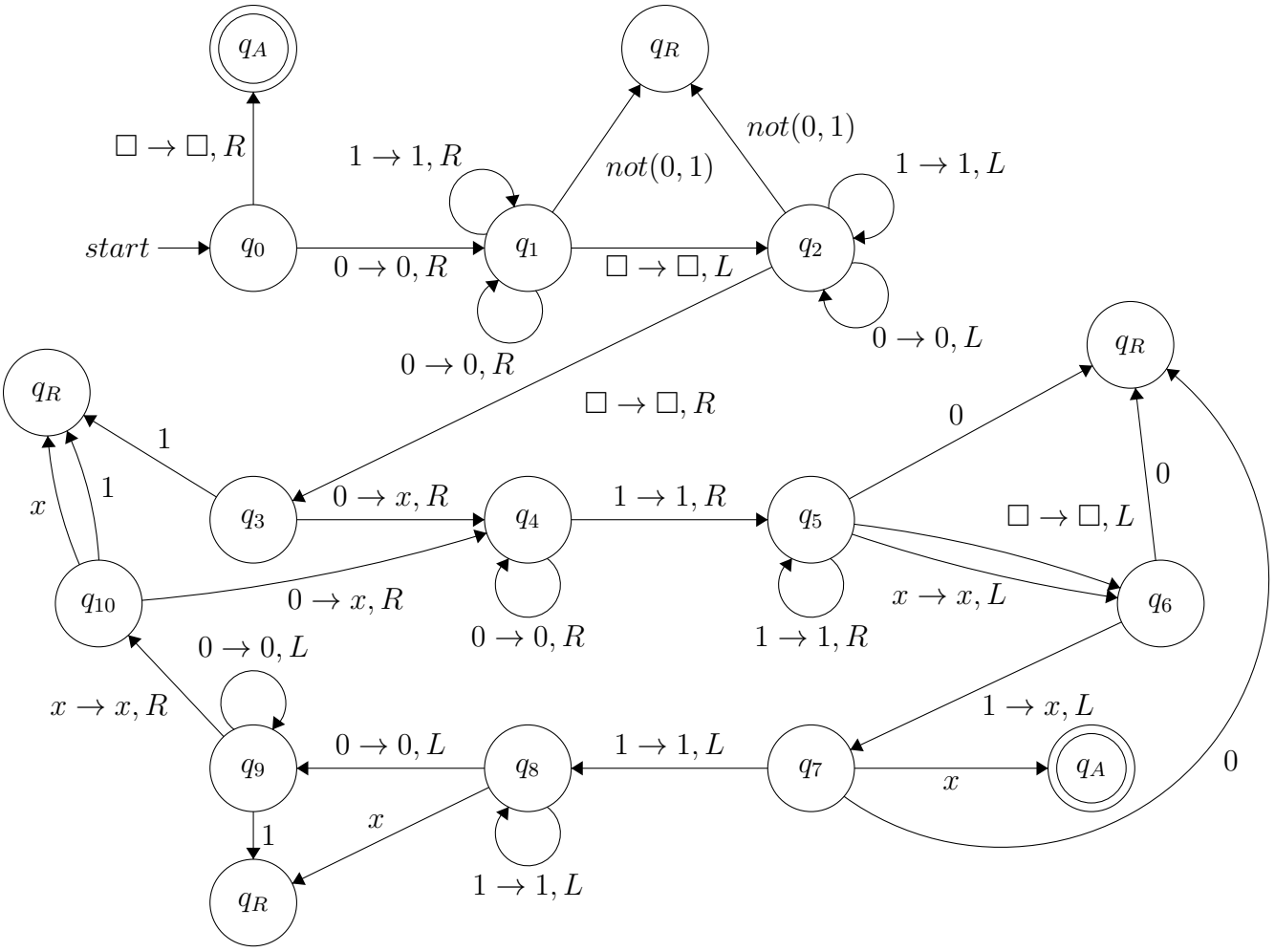
If our head ever reaches an accepting state q_a, then we immediately accept and our computation terminates. (This is unlike a DFA, in which we must finish our input before we accept.) Similarly, if we reach a rejecting state, then we immediately reject. If a Turing machine accepts or rejects an input, we say it **halts** or terminates. However, it is possible we never reach an accepting or rejecting state. Since we have to move right or left at every step, it's possible for a Turing machine to compute forever. We say a Turing machine **loops** when it fails to terminate.

Note: the head can write any character in the **tape alphabet** Γ, which contains Σ but can contain characters outside Σ. This will allow us to write special symbols that we know don't occur naturally in the input, and let us know we've already seen a certain cell and performed computation.

We can immediately see that Turing machines are more powerful than DFAs and NFAs. Because they have infinite memory, Turing machines can decide some non-regular languages.

Example 6.1. There exists a Turing machine M which can accept every input string in $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ and reject all inputs outside of L.

Solution. Construct a Turing machine M with tape alphabet $\Gamma = x \cup \Sigma = \{0, 1, x\}$, with the following transitions.



Holy moly! That looks extraordinarily complicated! It sort of looks like a DFA, but with weird transitions. Let's break it down.

The above diagram dictates our entire computation—in other words, how the tape head writes and moves depending on what it reads and what state it's currently in. We start off with the tape head at q_0 and read the very first character of the input. At each step of the computation, we take a different transition depending on the character we read.

Each transition is of the form $(A \to B, L/R)$, meaning at that state, if the head reads A, it writes B, and moves left or right. For transitions where there is only a single character listed (e.g. $x, 0, 1$ or $not(0,1)$), this is just shorthand for reading $x, 0, 1$ or not 0 or 1 and signifies that the character we write and direction we move don't matter.

As soon as we reach an accept state (q_A) we accept, and as soon as we reach a reject state (q_R) we reject.

Now that we understand how to read the transition diagram, let's look at what the algorithm actually does.

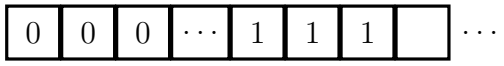


Figure 6: A non-regular input.

1. First, check over the entire input. If we ever reach a character that is not 0 or 1, reject. If the tape is blank, accept (computed in states q_0, q_1 and q_2).

2. Starting at the leftmost character on the tape that is not an x. If we read 0, replace 0 with x. Otherwise, reject. (q_3)

3. Travel to the rightmost character on the tape that is not an x. If we do not read 1, reject (q_4, q_5, q_6). Otherwise, replace 1 with x ($q_6 \to q_7$).

4. Repeat steps 2 and 3 until there are no more 0s or 1s (q_8, q_9, q_{10} travels back to the beginning, and connects back to q_3). If, after repeating step 3, there are no zeros but there are 1s, then we know that our input was of the form 0^m1^n, $n \geq m$. So, we reject if we reach this state ($q_8 \to q_R$). If we reach a point where there are no 1s but there are 0s, then we know that our input was of the form 0^m1^n, $n \leq m$. So, we reject if we reach this state ($q_6 \to q_R$). If there are no zeros and no ones, then we know that our input was of the form $0^m1^m, m \in \mathbb{N}$. Thus, we accept ($q_7 \to q_A$).

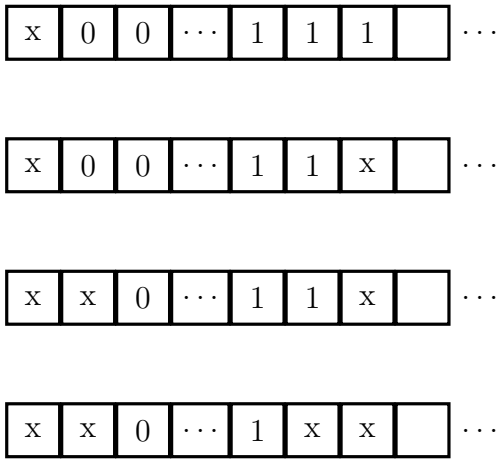


Figure 7: A few steps of the algorithm deciding $L = \{0^n1^n \mid n \in \mathbb{N}\}$.

Using the above algorithm, the Turing machine M accepts all strings in L and rejects all strings outside of L. □

In the future, we won't draw out the transition diagram. It's sufficient just describe the algorithm in word form, and imagine that a corresponding transition function exists.

6.1 Practice Questions

1. Prove that there exists a Turing Machine which accepts all strings in the language $L = \{ww \mid w \in \{0,1\}^*\}$ and rejects all strings not in L. Is there a DFA which accepts L?

2. For every regular language R, is there a Turing machine which accepts strings if they are in R and rejects them if they are not?

6.2 Decidable vs. Recognizable

A language L is **decidable** if there exists a Turing machine M such that M accepts every string in L and M rejects every string not in L. Here, we say M *decides* L.

A language L is **recognizable** if there exists a Turing machine M such that M accepts every string in L and M rejects **or loops** on every string not in L. Here, we say M *recognizes* L.

We see that if and only if L and $\neg L$ ($\neg L$ is the language of all strings not in L) are both recognizable, then L is decidable.

We have already seen 0^n1^n is decidable. Now, we will give examples of undecidable, recognizable, and unrecognizable languages.

Consider the following language

$$L = \{(M, w) \mid M \text{ is a Turing machine and } w \text{ is a string}\}$$

One first glance, this looks quite odd. Languages consist of strings. However, this language consists of (M, w) tuples, where M is a Turing machine. However, every Turing machine can be encoded as a string. We can always write down the finitely many transitions between states, and the alphabet and all the parameters, encoding it somehow.

Consider the following, somewhat similar language.

$$A_{TM} = \{(M, w) \mid M \text{ is a Turing machine and } w \text{ is a string and } M \text{ accepts } w\}$$

Theorem 1. *A_{TM} is recognizable.*

Proof. In order to decide the language, we construct a Turing machine, which, given an input (M, w), simulates M on w. Such a Turing machine is called the Universal Turing machine, and is denoted U. U takes in (M, w), simulates M on w, and if M accepts w, then U accepts, and if M rejects w, then U rejects. Consequently, if M loops on w, then U loops on (M, w). So, A_{TM} is recognizable. □

Theorem 2. *A_{TM} is undecidable.*

Proof. Assume there is a Turing machine T which decides A_{TM}. In other words, T accepts (M, w) if $(M, w) \in A_{TM}$ and rejects (M, w) if $(M, w) \notin A_{TM}$.

$$T((M, w)) = \begin{cases} T \text{ accepts } (M, w) & \text{if } (M, w) \in A_{TM} \ \ (M \text{ accepts } w) \\ T \text{ rejects } (M, w) & \text{if } (M, w) \notin A_{TM} \ \ (M \text{ does not accept } w) \end{cases}$$

We will show this leads to a contradiction.

Now, remember, w is any string, and any Turing machine can be encoded as a string. What happens if we run T on (Turing machine, Turing machine) tuples, where the input strings w are Turing machines? We might get a table that looks something like this:

	M_1	M_2	M_3	$\dots$
M_1	accept	reject	accept	$\dots$
M_2	reject	accept	accept	$\dots$
M_3	accept	reject	reject	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

where the the cell in the ith row and jth column represents the output of $T(M_i, M_j)$, and $M_1, M_2, \ldots$ is some ordering of Turing machines. We don't actually know what the values are for the cells in the above table, but it turns out that won't matter.

Now, consider the following Turing machine D:

$$D(M) = \begin{cases} D \text{ accepts } M & \text{if } T \text{ rejects } (M, M) \\ D \text{ rejects } M & \text{if } T \text{ accepts } (M, M) \end{cases}$$

Once, more consider the table for T on tuples where both elements are Turing machines:

	M_1	M_2	M_3	$\ldots$	D
M_1	accept	reject	accept	$\ldots$	accept
M_2	reject	accept	accept	$\ldots$	reject
M_3	accept	reject	reject	$\ldots$	accept
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
D	reject	accept	accept	$\ldots$	reject

We see that the output of D simply reverses the diagonal, because $D(M_1)$ is the opposite of $T(M_1, M_1)$, and $D(M_2)$ is the opposite of $T(M_2, M_2)$, and so forth. The bold values (flipped diagonal) in the table below are the outputs of D.

	M_1	M_2	M_3	$\ldots$	D
M_1	**reject**	reject	accept	$\ldots$	accept
M_2	reject	**reject**	accept	$\ldots$	reject
M_3	accept	reject	**accept**	$\ldots$	accept
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
D	reject	accept	accept	$\ldots$	**???**

However, what happens at $D(D)$? By construction, D rejects D if T accepts (D, D), but T accepts (D, D) when D accepts D. Also by construction, D accepts D if T rejects (D, D), but this only happens when D does not accept D.

Therefore, we have

$$D(D) = \begin{cases} D \text{ accepts } D & \text{if } D \text{ does not accept } D \\ D \text{ rejects } D & \text{if } D \text{ accepts } D \end{cases}$$

But, this is a contradiction! So, D cannot exist, so T cannot exist, so there can be no Turing machine T that decides A_{TM}. Thus, A_{TM} is undecidable. □

6.3 Practice Questions

1. Prove $\neg A_{TM}$ is unrecognizable.
2. Prove the Halting problem $HALT = \{(M, w) \mid M \text{ is a Turing machine and } M \text{ halts on } w\}$ is recognizable and undecidable. What can you conclude about $\neg HALT$?

6.4 Final Thoughts

There's still quite a bit we haven't talked about regarding Turing machines. First, it's not too difficult to show multi-tape Turing machines are equivalent to single-tape Turing machines. Second, we can use Turing machines to talk about computational complexity, and whether certain algorithms run in polynomial time, nondeterministic polynomial (NP) time, or are NP-hard/NP-complete (and beyond). We will cover these topics in future lectures.

We leave you with the Church-Turing thesis, which is an unproven conjecture, but should hopefully spur some thoughts about the power of this fairly simple model.

Thesis 6.1 (*Church-Turing*)**.** Any real-world computation can be computed on a Turing machine.

If we accept the Church-Turing thesis, then in some sense, no machine we build can have greater capability than any Turing-complete system. (A system is *Turing-complete* if it can simulate any Turing machine.) However, the bar for Turing-completeness is fairly low. Of course, the instruction sets for your computers (x86) and phones (ARM) are Turing-complete, as well as most general-purpose programming languages (Java, Python, etcetera). But so are many systems you might not expect, which were never intended for computation, such as Minecraft, Magic: The Gathering, and even PowerPoint (see Resource #3).

And so, we return to our question from the very beginning: What *is* a computer? In a practical sense, your iPhone, a mainframe from the 1960s, and PowerPoint have very different capabilities. But in theory, they can all solve the same problems. Perhaps they're all computers after all.

7 Resources

- Some definitions from `http://web.stanford.edu/class/cs103/`.
- Some practice questions from CS 154: `https://cs154.stanford.edu`
- Fun video: "On the Turing Completeness of PowerPoint", by Tom Wildenhaim `https://youtu.be/uNjxe8ShM-8`
- The CS 103 and CS 154 websites are good places for supplementary material to this lecture.
- Automata drawing software: `http://madebyevan.com/fsm`
- Turing Machine diagram: `http://www.texample.net/tikz/examples/turing-machine-2/`