Computability and Algorithms - Course Notes

Universality - (Udacity)

Introduction - (Udacity, Youtube)

In 1936, when Alan Turing wrote his famous paper "On Computable Numbers", he not only created the Turing machine but had a number of other major insights on the nature of computation. Turing realized that the computer program itself could also be considered as part of the input. There really is no difference between the program and data. We all take that for a given today: we create computer code in a file that gets stored on the computer no differently than any other type of file. And data files often have computer instructions embedded in them. Even modern computer fonts are basically small programs that generate readable characters at arbitrary sizes.

Once he took the view of program code as data, Turing had the beautiful idea that a Turing machine could simulate that code. There is some fixed Turing machine, a universal Turing machine, that can simulate the code of any other Turing machine. Again, this is an idea that we take for granted today, as we have interpreters and compilers that can run the code of any programming language. But, back in Turing's day, this idea of a universal machine was the major breakthrough that allowed Turing, and will allow us, to develop problems that Turing machines or any other computer cannot solve.

Encoding a Turing Machine - (Udacity, Youtube)

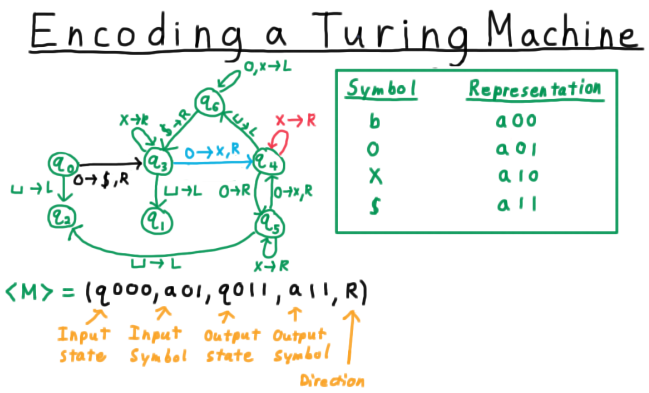
Before we can simulate or interpret a Turing machine, we first have to represent it using a string. Notice that this presents an immediate challenge: our universal Turing machine must use a fixed alphabet for its input and have a fixed number of states, but it must be able to simulate other Turing machines with arbitrarily large alphabets and an arbitrary numbers of states. As we'll see, one solution is essentially to enumerate all the symbols and states and represent them in binary. There are lots of ways to do this: the way we're going to do it is a compromise of readability and efficiency.

Let M be a Turing machine with states $Q = \{q_0, \ldots q_{n-1}\}$ and tape alphabet $\Gamma = \{a_1, \ldots a_m\}$. Define i and j so that 2^i is at least the number of states and 2^j is at least the number of tape symbols. Then we can encode a state q_k as the concatenation of the symbol q with the string w, where w is the binary representation of k. For example, if there are 6 states, then we need three bits to encode all the states. The state q_3 would be encoded as the string q011. By convention, we make

- q_0 the initial state,
- q_1 the accept state,
- q_2 the reject state.

We use an analogous strategy for the symbols, encoding a_k as the symbol a followed by the string w, where w is the binary representation of k. For example, if there are 10 symbols, then we need four bits to represent them all. If a_5 is the star symbol, we would be encode that symbol as a0101.

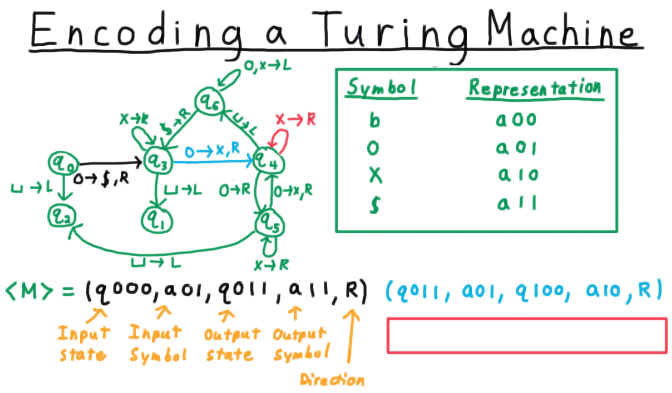
Let's see an encoding for an example.



This example decides whether the input consists of a number of zeros that is a power of two. To encode the Turing machine as a whole we really just need to encode its transition function. We'll start by encoding the black edge from the diagram. We are going from state zero, seeing the symbol zero, and we go to state three, we write symbol 0 and we move the head to the right. Remember that the order here is input state, input symbol. Then output state, output symbol, and finally direction.

Encoding Quiz - (Udacity)

Now, I'm not going to write out all the rest of the transitions, but I think it would be a good idea for you to do one more. So use this red box here to encode this red transition.



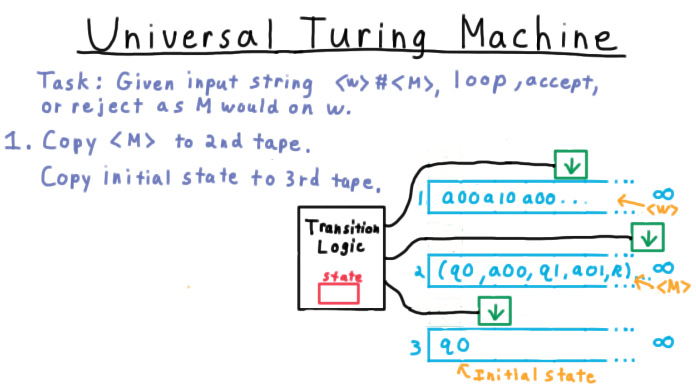
Building a Universal Turing Machine - (Udacity, Youtube)

Now, we are ready to describe how to build this all-powerful universal Turing machine. As input to the universal machine, we will give the encoding of the input and the encoding of M, separated by a hash. We write this as

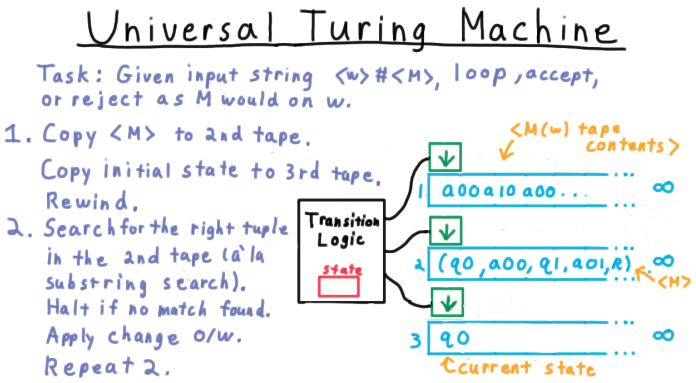
<w>#<M>

The goal is to similate M's execution when given the input w, halting in an accept or reject state, or not halting at all, and ultimately outputing the encoding of the output of M on w when M does halt. We'll describe a 3-tape Turing machine that achieves this goal of simulating M on w.

The input comes in on the first tape. First, we'll copy the description of the machine to the second tape and copy the initial state to the third tape. For example, the tape contents might end up like this.



The we rewind all the heads and begin the second phase. Here, we search for the appropriate tuple in the description of the machine. The first element has to match the current state stored on tape three, and the symbol part has to match the encoding on the tape 1. If no match is found, then we halt the simulation and put the universal machine in an accepting or rejecting state according to the current state of the machine being interpreted. If there is a match, however, then we apply the changes to the first tape and repeat.



Actually, interpreting a Turing machine description is surprisingly easy. We've just seen how Turing machines are indeed reprogrammable just like real-world computers. This lends further support to the Church-Turing thesis, but it also has significance beyond that. Since the input to a Turing machine can be interpreted as a Turing machine, this suggest that programs are a type of data. But arbitrary data can also be interpretted as a (possibly invalid) Turing machine. So is there any difference between data and program? Perhaps, we can leave this question for the philosophers.

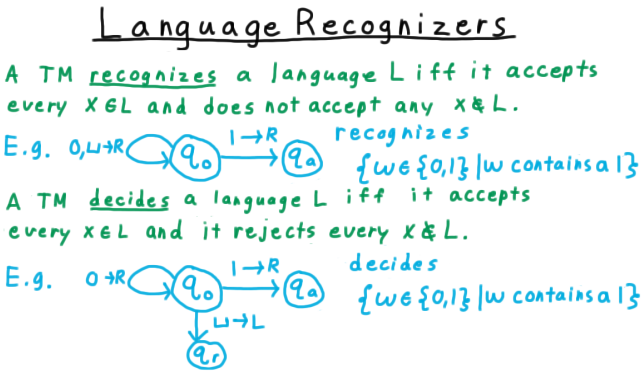
Abstraction - (Udacity, Youtube)

At this point, the character of our discussion of computability is going to change signficantly. We've established the key properties of Turing machines: that they can do anything we mean by computation and that we can pass a description of a Turing machine to a Turing machine as input for it to simulate. With these points established, we won't need to talk about the specifics of Turing machines much anymore. There will be little about tapes or states, transitions or head positions. Instead, we will think about computation at a very high level, trusting that if we really had to, we could write out the Turing machine to do it. If we need to write out code, we will do so only in pseudocode or with pictures.

What is there left to talk about? Well, remember from the very first lesson that we argued that not all functions are computable, or as we later said, not all languages can be decided by Turing machines. We're now in a good position to talk about some of these undecidable languages. We had to wait until we established the universality of Turing machines because these languages are going to consist of strings that encode Turing machines. The rest of the lesson will review the definitions of recognizability and decidability, and then we'll talk about the positive side of the story: the languages about Turing machines that we CAN decide or recognize. As we'll see, there are plenty that we can't.

Language Recognizers - (Udacity, Youtube)

Recall that a Turing machine recognizes a language if it accepts every string in the language and does not accept anything that is not in the language. It could either reject or loop. This Turing machine here recognized the language of binary strings containing a 1, but it looped on those that don't contain a 1.



In order to decide a language, the Turing machine must not only accept every string in the language but it must also explicitly reject every string that is not in the language. This machine achieves that by not looping on the blanks.

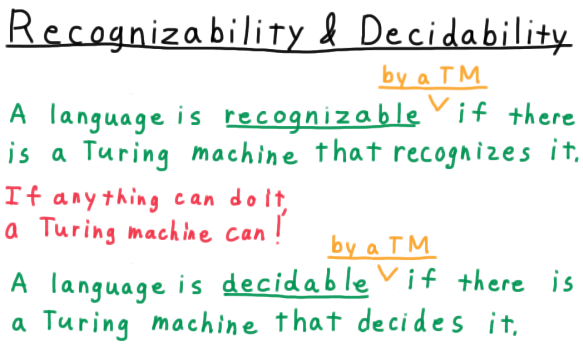
Recognizability and Decidability - (Udacity, Youtube)

Ultimately, however, we are not interested in whether a particular Turing machine recognizes or decides a language; rather we are interested in whether there *is* a Turing machine that can recognize or decides the language. Therefore, we say that a language is recognizable if there is a Turing machine that recognizes it, and we say that a language is decidable if there is a Turing machine that decides it.

> *A language is **recognizable** if there is a Turing machine that recognizes it.*
> *A language is **decidable** if there is a Turing machine that decides it.*

Now, looking at this someone might object, "Shouldn't we say 'recognizable by a Turing machine' and 'decidable by a Turing machine'?" Of course, we could and the statements would still be true. But we don't, the reason being that we strongly believe that if anything can do it a Turing machine can! That's the Church-Turing thesis.

In an absolute sense, we believe that a language is only recognizable by anything if a Turing machine can recognize it and a language is only decidable by anything if a Turing machine can decide it, and we use terms that reflect that belief.



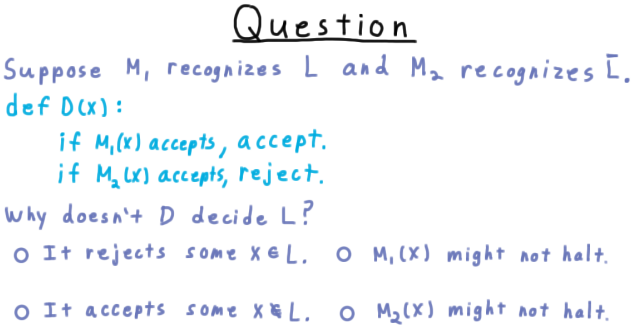
Now other terms are sometimes used instead of recognizable and decidable. Some say that Turing machines compute languages, so to go along with that they say that languages are **computable** if they there is a Turing machine that computes it. Another equivalent term for decidable is **recursive**. Mathematicians often prefer this word.

And those who use that term will refer to recognizable languages as **recursively enumerable**. Some also call these languages **Turing-acceptable** and **semi or partially decidable**.

We should also make clear the relationship between these two terms. Clearly, if a language is decidable, then it is also recognizable; the same Turing machine works for both. It feels like it should also be true that if a language is recognizable and its complement is also recognizable, then the language is decideable. This is true, but there is a potential pitfall here that we need to make sure to avoid.

Decidability Exercise - (Udacity)

Suppose that we are given one machine M_1 that recognizes a language L and another M_2 that recognizes $\overline{L}$. If we were to ask your average programmer to use these machines to decide the language, his first guess might go something like this.

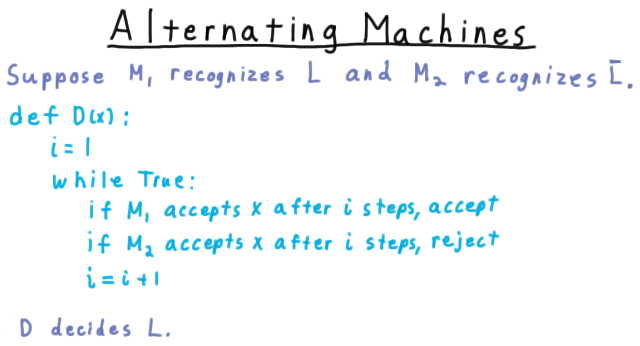


This program will not decide L, however, and I want you to tell me why. Check the best answer.

Alternating Machines - (Udacity, Youtube)

Here is the alternating trick in some more detail. Suppose that M_1 recognizes a language and M_2 recognizes its complement. We want to decide the language.

In psuedocode, the alternating strategy might look like this.



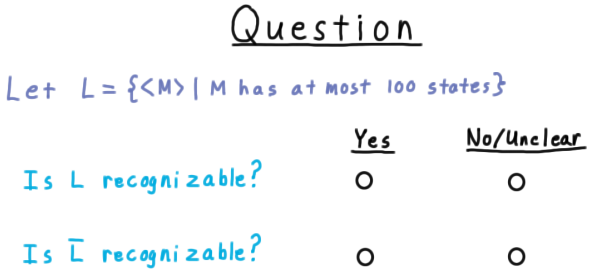
In every step, we execute both machines one more step than in the previous iteration. Note that it doesn't matter if we save the machines configuration and start where we left off or start over. The question is whether we get the right answer, not how fast. The string has to be either in L or in $\overline{L}$, so one of these has to halt after some finite number of steps, and when i hits that value, this program will give the right answer.

Overall then, we have the following theorem.

> *A language L is decidable if and only if L and its complement are both recognizable.*

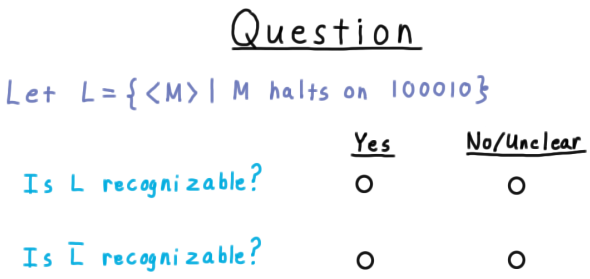
Counting States - (Udacity)

Now we're going to go through a series of languages and try to figure out if they and their complements are recognizable. First, let's examine the set of strings that describe a Turing machine that has at most 100 states. You can assume the particular encoding for Turing machines that we used, but any encoding will serve the same purpose. Indicate whether you think that L is recognizable and whether L complement is recognizable. We don't have a way proving that a language is not recognizable yet, so I've labeled the "No" option as unclear.



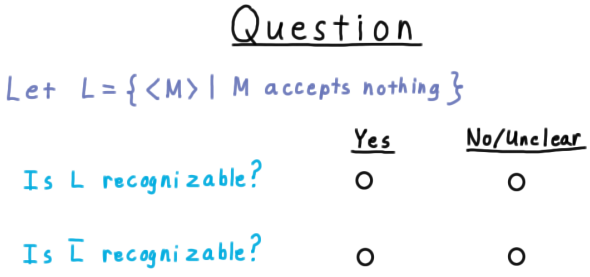
Halting on 34 - (Udacity)

Next, we consider the set of Turing machines that halt on the number 34 written in binary. Indicate whether L and L complement are recognizable.



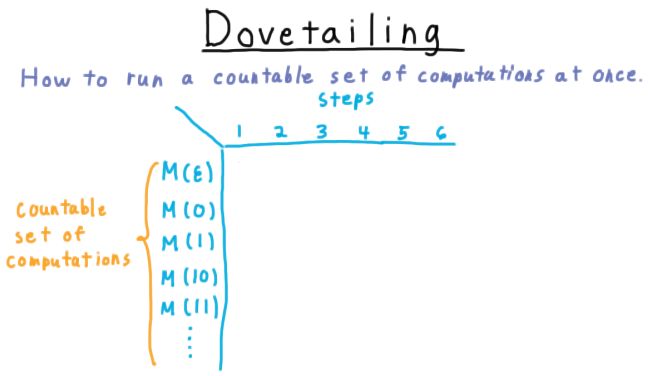
Accepting Nothing - (Udacity)

Let's consider another language, this time the set of Turing machine descriptions where the Turing machine accepts nothing. Tell me: are either L or L complement recognizable?



Dovetailing - (Udacity, Youtube)

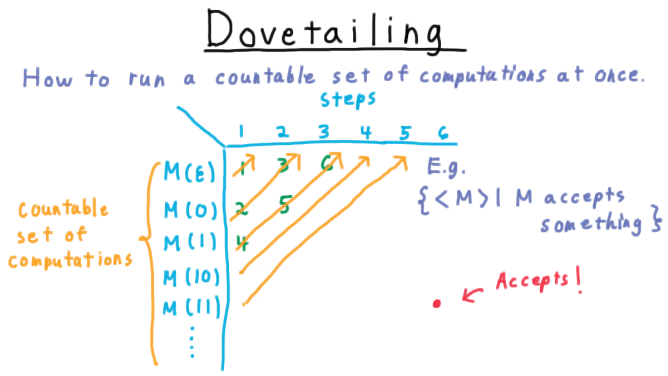
Here is the dovetailing trick, which let's you run a countable set of computations all at once. We'll illustrate the technique for the case where we are simulating a machine M on all binary strings with this table here.



Every row in the table corresponds to a computation or the sequence of configurations the machine goes through for the given input. Simulating all of these computation means hitting every entry in this table. Note that we can't just simulate M on the empty string first or we might just keep going forever, filling out the first row and never getting to the second. This is the same problem that we encountered when trying to show that a countable union of countable sets is countable, or that the set of rational numbers is countable.

And the solution is the same too. We go diagonal by diagonal, first simulating the first computation for one step. Then the second computation for one step and the first computation for two steps, etc. Eventually every

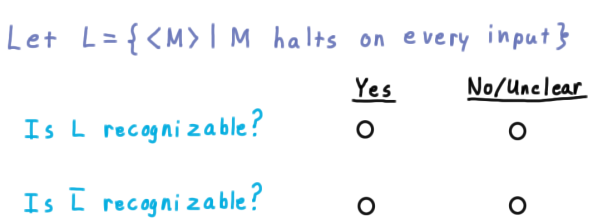
configuration in the table is reached.



Thus, if we are trying to recognize the language of Turing machine descriptions where the Turing machine accepts something, then a Turing machine in the language must accept some string after finite number of steps. This will correspond to some some entry in the table, so we eventually reach it and accept.

Always Halting - (Udacity)

Let's consider one last language, the set of descriptions of Turing machines that halt on every input. Think carefully, and indicate whether you think that either L or L complement is recognizable.



Conclusion - (Udacity, Youtube)

In this and the previous lessons, we've developed a set of ideas and definitions that sets the stage for understanding what we can and cannot solve on a computer.

- We've seen the Turing machine, an amazingly simple model that can only move a tape back and forth while reading and writing on that tape.
- We've seen that despite it's simplicity this model captures the full power of computers now and forever.
- We've seen how to consider Turing machine programs as data themselves and create a universal Turing machine that can simulate those programs.
- And at the end of this lesson, we saw some languages defined using program as data that don't seem to be easily decidable.

In the next lecture we will show how to prove many languages cannot be solved by a Turing machine, including the most famous one, the halting problem.