The Three Hour Tour Through Automata Theory

Analyzing Problems as Opposed to Algorithms

In CS336 you learned to analyze ***algorithms***. This semester, we're going to analyze ***problems***.

We're going to see that there is a hierarchy of problems: easy, hard, impossible.

For each of the first two, we'll see that for any problem, there are infinitely many programs to solve the problem.

> By the way, this is trivial. Take one program and add any number of junk steps. We'll formalize this later once we have some formalisms to work with.

Some may be better than others. But in some cases, we can show some sort of boundary on how good an algorithm we can attempt to find for a particular problem.

Let's Look at Some Problems

Let's look at a collection of problems, all which could arise in considering one piece of C++ code. **[slide - Let's Look at Some Problems]**

As we move from problem 1 to problem 5, things get harder in two key ways. The first is that it seems we'll need more complicated, harder to write and design programs. In other words, it's going to take more time to write the programs. The second is that the programs are going to take a lot longer to run. As we get to problems 4 and 5, it's not even clear there is a program that will do the job. In fact, in general for problem 4 there isn't. For problem 5, in general we can't even get a formal statement of the problem, much less an algorithmic solution.

Languages

Characterizing Problems as Language Recognition Tasks

In order to create a formal theory of problems (as opposed to algorithms), we need a single, relatively straightforward framework that we can use to describe any kind of possibly computable function. The one we'll use is language recognition.

What is a Language?

A language is a set of strings over an alphabet, which can be any *finite* collection of symbols. **[Slide - Languages]**

Defining a Problem as a Language Recognition Task

We can define any problem as a language recognition task. In other words, we can output just a boolean, True or False. Some problems seem naturally to be described as recognition tasks. For

example, accept grammatical English sentences and reject bad ones. (Although the truth is that English is so squishy it's nearly impossible to formalize this. So let's pick another example -- accept the syntactically valid C programs and reject the others.)

Problems that you think of more naturally as functions can also be described this way. We define the set of input strings to consist of strings that are formed by concatenating an input to an output. Then we only accept strings that have the correct output concatenated to each input. **[Slide - Encoding Output]**

Branching Out -- Allowing for Actual Output

Although it is simpler to characterize problems simply as recognition problems and it is possible to reformulate functional problems as recognition problems, we will see that we can augment the formalisms we'll develop to allow for output as well.

Defining Languages Using Grammars

Now what we need is a general mechanism for defining languages. Of course, if we have a finite language, we can just enumerate all the strings in it. But most interesting languages are infinite. What we need is a *finite* mechanism for specifying *infinite* languages. Grammars can do this.

The standard way to write a grammar is as a production system, composed of rules with a left hand side and a right hand side. Each side contains a sequence of symbols. Some of these symbols are terminal symbols, i.e., symbols in the language we're defining. Others are drawn from a finite set of nonterminal symbols, which are internal symbols that we use just to help us define the language. Of these nonterminal symbols, one is special -- we'll call it the start symbol. **[Slide - Grammars 1]**

If there is a grammar that defines a language, then there is an infinite number of such grammars. Some may be better, from various points of view than others. Consider the grammar for odd integers. What different grammars could we write? One thing we could do would be to introduce the idea of odd and even digits. **[Slide - Grammars 2]**

Sometimes we use single characters, disjoint from the characters of the target language, in our rules. But sometimes we need more symbols. Then we often use < and > to mark multiple character nonterminal symbols. **[Slide - Grammars 3]**

Notice that we've also introduced a notation for OR so that we don't have to write as many separate rules. By the way, there are lots of ways of writing a grammar of arithmetic expressions. This one is simple but it's not very good. It doesn't help us at all to determine the precedence of operators. Later we'll see other grammars that do that.

Grammars as Generators and as Acceptors

So far, we've defined problems as language recognition tasks. But when you look at the grammars we've considered, you see that there's a sense in which they seem more naturally to be generators than recognizers. If you start with S, you can generate all the strings in the language

defined by the grammar. We'll see later that we'll use the idea of a grammar as a generator (or an enumerator) as one way to define some interesting classes of languages.

But you can also use grammars as acceptors, as we've suggested. There are two ways to do that. One is *top-down*. By that we mean that you start with S, and apply rules. **[work this out for a simple expression for the Language of Simple Arithmetic Expressions]** At some point, you'll generate a string without any nonterminals (i.e., a string in the language). Check and see if it's the one you want. If so accept. If not, try again. If you do this systematically, then if the string is in the language, you'll eventually generate it. If it isn't, you may or may not know when you should give up. More on that later.

The other approach is bottom up. In this approach, we simply apply the rules sort of backwards, i.e., we run them from right to left, matching the string to the right hand sides of the rules and continuing until we generate S and nothing else. **[work this out for a simple expression for the Language of Simple Arithmetic Expressions]** Again, there are lots of possibilities to consider and there's no guarantee that you'll know when to stop if the string isn't in the language. Actually, for this simple grammar there is, but we can't assure that for all kinds of grammars.

The Language Hierarchy

Remember that our whole goal in this exercise is to describe classes of problems, characterize them as easy or hard, and define computational mechanisms for solving them. Since we've decided to characterize problems as languages to be recognized, what we need to do is to create a language hierarchy, in which we start with very simple languages and move toward more complex ones.

Regular Languages

Regular languages are very simple languages that can be defined by a very restricted kind of grammar. In these grammars, the left side of every rule is a single nonterminal and the right side is a single terminal optionally followed by a single nonterminal. **[slide - Regular Grammars]** If you look at what's going on with these simple grammars, you can see that as you apply rules, starting with S, you generate a terminal symbol and (optionally) have a new nonterminal to work with. But you can never end up with multiple nonterminals at once. (Recall our first grammar for Odd Integers [**slide - Grammars 1**]).

Of course, we also had another grammar for that same language that didn't satisfy this restriction. But that's okay. If it is possible to define the language using the restricted formalism, then it falls into the restricted class. The fact that there are other, less restricted ways to define it doesn't matter.

It turns out that there is an equivalent, often useful, way to describe this same class of languages, using regular expressions. **[Slide Regular Expressions and Languages]** Regular expressions don't look like grammars, in the sense that there are no production rules, but they can be used to define exactly the same set of languages that the restricted class of regular grammars can define. Here's a regular expression for the language that consists of odd integers, and one for the

language of identifiers. We can try to write a regular expression for the language of matched parenthesis, but we won't succeed.

Intuitively, regular languages are ones that can be defined without keeping track of more than a finite number of things at once. So, looking back at some of our example languages **[slide Languages]**, the first is regular and none of the others is.

Context Free Languages

To get more power in how we define languages, we need to return to the more general production rule structure.

Suppose we allow rules where the left hand side is composed of a single symbol and the right hand side can be anything. We then have the class of context-free grammars. We define the class of context-free languages to include any language that can be generated by a context-free grammar.

The context-free grammar formalism allows us to define many useful languages, including the languages of matched parentheses and of equal numbers of parentheses but in any order **[slide - Context-Free Grammars]**. We can also describe the language of simple arithmetic expressions **[slide - Grammars 3]**.

Although this system is a lot more powerful (and useful) than regular languages are, it is not adequate for everything. We'll see some quite simple artificial language it won't work for in a minute. But it's also inadequate for things like ordinary English. [**slide - English Isn't Context-Free]**.

Recursively Enumerable Languages

Now suppose we remove all restrictions from the form of our grammars. Any combination of symbols can appear on the left hand side and any combination of symbols can appear on the right. The only real restriction is that there can be only a finite number of rules. For example, we can write a grammar for the language that contains strings of the form $a^n b^n c^n$. **[slide - Unrestricted Grammars]**

Once we remove all restrictions, we clearly have the largest set of languages that can be generated by any finite grammar. We'll call the languages that can be generated in this way the class of recursively enumerable languages. This means that, for any recursively enumerable language, it is possible, using the associated grammar, to generate all the strings in the language. Of course, it may take an infinite amount of time if the language contains an infinite number of strings. But any given string, if it is enumerated at all, will be enumerated in a finite amount of time. So I guess we could sit and wait. Unfortunately, of course, we don't know how long to wait, which is a problem if we're trying to decide whether a string is in the language by generating all the strings and seeing if the one we care about shows up.

Recursive Languages

There is one remaining set of languages that it is useful to consider. What about the recursively enumerable languages where we could guarantee that, after a finite amount of time, either a given string would be generated or we would know that it isn't going to be. For example, if we could generate all the strings of length 1, then all the strings of length 2, and so forth, we'd either generate the string we want or we'd just wait until we'd gone past the length we cared about and then report failure. From a practical point of view, this class is very useful since we like to deal with solutions to problems that are guaranteed to halt. We'll call this class of languages the recursive languages. This means that we can not only generate the strings in the language, we can actually, via some algorithm, decide whether a string is in the language and halt, with an answer, either way.

Clearly the class of recursive languages is a subset of the class of recursively enumerable ones. But, unfortunately, this time we're not going to be able to define our new class by placing syntactic restrictions on the form of the grammars we use. There are some useful languages, such, as $a^n b^n c^n$, that are recursive. There are some others, unfortunately, that are not.

The Whole Picture

[Slide - The Language Hierarchy]

Computational Devices

Formal Models of Computational Devices

If we want to make formal statements about the kinds of computing power required to solve various kinds of problems, then we need simple, precise models of computation.

We're looking for models that make it easy to talk about what can be computed -- we're not worrying about efficiency at this point.

When we described languages and grammars, we saw that we could introduce several different structures, each with different computational power. We can do the same thing with machines. Let's start with really simple devices and see what they can do. When we find limitations, we can expand their power.

Finite State Machines

The only memory consists of the ability to be in one of a finite number of states. The machine operates by reading an input symbol and moving to a new state that is determined solely by the state it is in and the input that it reads. There is a unique start state and one or more final states. If the input is exhausted and the machine is in a final state, then it accepts the input. Otherwise it rejects it.

Example: An FSM to accept odd integers. **[Slide - Finite State Machines 1]**

Example: An FSM to accept valid identifiers. **[Slide - Finite State Machines 2**]

Example: How about an FSM to accept strings with balanced parentheses?

Notice one nice feature of every finite state machine -- it will always halt and it will always provide an answer, one way or another. As we'll see later, not all computational systems offer these guarantees.

But we've got this at a price. There is only a finite amount of memory. So, for example, we can't count anything. We need a stronger device.

Push Down Automata

Deterministic PDAs

Add a single stack to an FSM. Now the action of the machine is a function of its state, the input it reads, and the values at the top of the stack.

Example: A PDA to accept strings with balanced parentheses. **[Slide - Push Down Automata]** Notice that this really simple machine only has one state. It's not using the states to remember anything. All its memory is in the stack.

Example: A PDA to accept strings of the form $w\#w^R$, where $w \in \{a,b\}^*$ **[slide - Pushdown Automaton 2]**.

Example: How about a PDA to accept strings with some number of a's, followed by the same number of b's, followed by the same number of c's? **[slide - PDA 3]** It turns out that this isn't possible. The problem is that we could count the a's on the stack, then pop for b's. But how could we tell if there is the right number of c's. We'll see in a bit that we shouldn't be too surprised about this result. We can create PDA's to accept precisely those languages that we can generate with CFGs. And remember that we had to use an unrestricted grammar to generate this language.

Nondeterministic PDAs

Example: How about a PDA to accept strings of the form $w\ w^R$? We can do this one if we expand our notion of a PDA to allow it to be nondeterministic. The problem is that we don't know when to imagine that the reversal starts. What we need to do is to guess. In particular, we need to try it at every point. We can do this by adding an epsilon transition from the start state (in which we're pushing w) to the final state in which we're popping as we read w^R.**[slide - A Nondeterministic PDA]** Adding this kind of nondeterminism actually adds power to the PDA notion. And actually, it is the class of nondeterministic PDA's that is equivalent to the class of context-free languages. No surprise, since we were able to write a context free grammar for this language

By the way, it also makes sense to talk about nondeterministic finite state machines. But it turns out that adding nondeterminism to finite state machines doesn't increase the class of things they can compute. It just makes it easier to describe some machines. Intuitively, the reason that nondeterminism doesn't buy you anything with finite state machines is that we can simulate a nondeterministic machine with a deterministic machine. We just make states that represent sets of states in the nondeterministic machine. So in essence, we follow all paths. If one of them accepts, we accept.

Then why can't we do that with PDA's? For finite state machines, there must be a finite number of states. DAH. So there is a finite number of subsets of states and we can just make them the states of our new machine. Clunky but finite. Once we add the stack, however, there is no longer a finite number of states of the total machine. So there is not a finite number of subsets of states. So we can't simulate being in several states at once just using states. And we only have one stack. Which branch would get it? That's why adding nondeterminism actually adds power for PDAs.

Turing Machines

Clearly there are still some things we cannot do with PDAs. All we have is a single stack. We can count one thing. If we need to count more than one thing (such as a's and b's in the case of languages defined by $a^n b^n c^n$), we're in trouble.

So we need to define a more powerful computing device. The formalism we'll use is called the Turing Machine, after its inventor, Alan Turing. There are many different (and equivalent) ways to write descriptions of Turing Machines, but the basic idea is the same for all of them **[slide - Turing Machines]**. In this new formalism, we allow our machines to write onto the input tape. They can write on top of the input. They can also write past the input. This makes it easier to define computation that actually outputs something besides yes or no if we want to. But, most importantly, because we view the tape as being of infinite length, all limitations of finiteness or limited storage have been removed, even though we continue to retain the core idea of a finite number of states in the controller itself.

Notice, though, that Turing Machines are not guaranteed to halt. Our example one always does. But we could certainly build one that scans right until it finds a blank (writing nothing) and then scans left until it finds the start symbol and then scans right again and so forth. That's a legal (if stupid) Turing Machine. Unfortunately, (see below) it's not always possible to tell, given a Turning Machine, whether it is guaranteed to halt. This is the biggest difference between Turing Machines and the FSMs and PDAs, both of which will always halt.

Extensions to Turing Machines

You may be thinking, wow, this Turing Machine idea sure is restrictive. For example, suppose we want to accept all strings in the simple language $\{w \# w^R\}$. We saw that this was easy to do in one pass with a pushdown automaton. But to do this with the sort of Turing Machine we've got so far would be really clunky. **[work this out on a slide]** We'd have to start at the left of the string, mark a character, move all the way to the right to find the corresponding character, mark

it, scan back left, do it again, and so forth. We've just transformed a linear process into an n^2 one.

But suppose we had a Turing Machine with 2 tapes. The first thing we'll do is to copy the input onto the second tape. Now start the read head of the first tape at the left end of the input and the read head of the second tape at the right end. At each step in the operation of the machine, we check to make sure that the characters being read on the two tapes are the same. And we move the head on tape 1 right and the head on tape 2 to the left. We run out of input on both machines at the same time, we accept. **[slide -A Two Head Turing Machine]**

The big question now is, "Have we created a new notational device, one that makes it easier to describe how a machine will operate, or have we actually created a new kind of device with more power than the old one? The answer is the former. We can prove that by showing that we can simulate a Turing Machine with any finite number of tapes by a machine that computes the same thing but only has one tape. **[slide - Simulating k Heads with One]** The key idea here is to use the one tape but to think of it has having some larger number of tracks. Since there is a finite tape alphabet, we know that we can encode any finite number of symbols in a finite (but larger) symbol alphabet. For example, to simulate our two headed machine with a tape alphabet of 3 symbols plus start and blank, we will need 2*2*5*5 or 100 tape symbols. So to do this simulation, we must do two main things: Encode all the information from the old, multi-tape machine on the new, single tape machine and redesign the finite state controller so that it simulates, in several moves, each move of the old machine.

It turns out that any "reasonable" addition you can think of to our idea of a Turing Machine is implementable with the simple machine we already have. For example, any nondeterministic Turing Machine can be simulated by a deterministic one. This is really significant. In this, in some ways trivial, machine, we have captured the idea of computability.

Okay, so our Turing Machines can do everything any other machine can do. It also goes the other way. We can propose alternative structures that can do everything our Turing Machines can do. For example, we can simulate any Turing Machine with a deterministic PDA that has two stacks rather than one. What this machine will do is read its input tape once, copying onto the first stack all the nonblank symbols. Then it will pop all those symbols off, one at a time, and move them to the second stack. Now it can move along its simulated tape by transfering symbols from one stack to the other. **[slide - Simulating a Turing Machine with Two Stacks]**

The Universal Turing Machine

So now, having shown that we can simulate anything on a simple Turing Machine, it should come as no surprise that we can design a Turing Machine that takes as its input the definition of another Turing Machine, along with an input for that machine. What our machine does is to simulate the behavior of the machine it is given, on the given input.

Remember that to simulate a k-tape machine by a 1 tape machine we had first to state how to encode the multiple tapes. Then we had to state how the machine would operate on the encoding. We have to do the same thing here. First we need to decide how to encode the states

and the tape symbols of the input machine, which we'll call M. There's no upper bound on how many states or tape symbols there will be. So we can't encode them with single symbols. Instead we'll encode states as strings that start with a "q" and then have a binary encoding of the state number (with enough leading zeros so all such encodings take the same number of digits). We'll encode tape symbols as an "a" followed by a binary encoding of the count of the symbol. And we'll encode "move left" as 10, "move right" as 01, and stay put as 00. We'll use # as a delimiter between transitions. **[slide - Encoding States, Symbols, and Transitions]**

Next, we need a way to encode the simulation of the operation of M. We'll use a three tape machine as our Universal Turing Machine. (Remember, we can always implement it on a one tape machine, but this is a lot easier to describe.) We'll use one tape to encode the tape of M, the second tape contains the encoding of M, and the third tape encodes the current state of M during the simulation. **[slide - The Universal Turing Machine]**

A Hierarchy of Computational Devices

These various machines that we have just defined, fall into an inclusion hierarchy, in the sense that the simpler machines can always be simulate by the more powerful ones. **[Slide - A Machine Hierarchy]**

The Equivalence of the Language Hierarchy and the Computational Hierarchy

Okay, this probably comes as no surprise. The machine hierarchy we've just examined exactly mirrors the language hierarchy. **[Slide - Languages and Machines]**

Actually, this is an amazing result. It seems to suggest that there's something quite natural about these categories.

Church's Thesis

If we really want to talk about naturalness, can we say anything about whether we've captured what it means to be computable? Church's Thesis (also sometimes called the Church-Turing Thesis) asserts that the precise concept of the Turing Machine that halts on all inputs corresponds to the intuitive notion of an algorithm. Think about it. Clearly a Turing Machine that halts defines an algorithm. But what about the other way around? Could there be something that is computable by some kind of algorithm that is not computable by a Turing Machine that halts? From what we've seen so far, it may seem unlikely, since every extension we can propose to the Turing Machine model turns out possibly to make things more convenient, but it never extends the formal power. It turns out that people have proposed various other formalisms over the last 50 years or so, and they also turn out to be no more powerful than the Turing Machine. Of course, something could turn up, but it seems unlikely.

Techniques for Showing that a Problem (or Language) Is Not in a Particular Circle in the Hierarchy

Counting

$a^n b^n$ is not regular.

Closure Properties

L = { $a^n b^m c^p$: $m \neq n$ or $m \neq p$ } is not deterministic context-free. **[slide - Using Closure Properties]** Notice that L' contains all strings that violate at least one of the requirements for L. So they may be strings that aren't composed of a string of a's, followed by a string of b's, followed by a string of c's. Or they may have that property but they violate the rule that m, n, and p cannot all be the same. In other words, they are all the same. So if we intersect L' with the regular expression a*b*c*, we throw away everything that isn't a string of a's then b's then c's, and we're left with strings of n a's, followed, by n b's, followed by n c's.

Diagonalization

Remember the proof that the power set of the integers isn't countable. If it were, there would be a way of enumerating the sets, thus setting them in one to one correspondence with the integers. But suppose there is such a way **[slide - Diagonalization]**. Then we could represent it in a table where element (i, j) is 1 precisely in case the number j is present in the set i. But now construct a new set, represented as a new row of the table. In this new row, element i will be 1 if element (i,i) of the original table was 0, and vice versa. This row represents a new set that couldn't have been in the previous enumeration. Thus we get a contradiction and the power set of the integers must not be countable.

We can use this technique for perhaps the most important result in the theory of computing.

The Unsolvability of the Halting Problem

There are recursively enumerable languages that are not recursive. In other words, there are sets that can be enumerated, but there is no decision procedure for them. Any program that attempts to decide whether a string is in the language may not halt.

One of the most interesting such sets is the following. Consider sets of ordered pairs where the first element is a description of a Turing Machine. The second element is an input to the machine. We want to include only those ordered pairs where the machine halts on the input. This set is not recursively enumerable. In other words, there's no way to write an algorithm that, given a machine and an input, determines whether or not the machine halts on the input. **[slide - The Unsolvability of the Halting Problem]**

Suppose there were such a machine. Let's call it HALTS. HALTS(M,x) returns true if Turing machine M halts on input x. Otherwise it returns false. Now we write a Turing Machine program that implements the TROUBLE algorithm. Now what happens if we invoke HALTS(TROUBLE,TROUBLE)? If HALTS says true, namely that TROUBLE will halt on

itself, then TROUBLE loops (i.e., it doesn't halt, thus contradicting our assumption that HALTS could do the job). But if HALTS says FALSE, namely that TROUBLE will not halt on itself, then TROUBLE promptly halts, thus again proving our supposed oracle HALTS wrong. Thus HALTS cannot exist.

We've used a sort of stripped down version of diagonalization here **[slide - Viewing the Halting Problem as Diagonalization]** in which we don't care about the whole row of the item that creates the contradiction. We're only invoking HALTS with two identical inputs. It's just the single element that we care about and that causes the problem.

Let's Revisit Some Problems

Let's look again at the collection of problems that we started this whole process with. **[slide - Let's Revisit Some Problems]**

Problem 1 can be solved with a finite state machine.

Problem 2 can be solved with a PDA.

Problem 3 can be solved with a Turing Machine.

Problem 4 can be semi solved with a Turning Machine, but it isn't guaranteed to halt.

Problem 5 can't even be stated.

So What's Left?