MIT 6.045J - Automata, Computability, Complexity

MIT 6.045J - Automata, Computability, Complexity

Taught by Prof. Scott Aaronson

Course Description

This course provides a challenging introduction to some of the central ideas of theoretical computer science. It attempts to present a vision of "computer science beyond computers": that is, CS as a set of mathematical tools for understanding complex systems such as universes and minds. Beginning in antiquity, with Euclid's algorithm and other ancient examples of computational thinking, the course will progress rapidly through finite automata, Turing machines and computability, decision trees and other concrete computational models, efficient algorithms and reducibility, NP-completeness, the P versus NP problem, the power of randomness, cryptography and one-way functions, computational learning theory, interactive proofs, and quantum computing and the physical limits of computation. Class participation is important, as the class will include discussion and debate about many of these topics.

Course website
Lecture Videos (YouTube)

Lectures

1. Introduction
2. Logic, Circuits, and Gates
3. Finite State Automata, Regular Languages
4. Regular Expressions, Context-Free Grammars
5. Turing Machines
6. Decidability
7. Turing Recognizability, Oracles
8. More Reducibility, Dovetailing, Godel

Lecture 1 - Introduction

What is Computer Science?

This course justifies why computer science deserves to be called a "science". It doesn't give you any skills that you can apply in your daily job, or your projects, but it introduces you to some Great Ideas in Computer Science.

It starts with a set of rules that we assume to be true, just as we assume axioms to be true in mathematics, and shows what we can and cannot do with this set of rules. It shows us the limits of what is possible and what is definitely impossible.

Factoring is Hard

The first "idea" we look at is that factoring a composite number into its prime factors is hard, at least with our current computational model. Factoring will be easy with a quantum computer. For now, the best we can hope to do is spend exponential time finding the factors, and the best algorithm we know takes $ O(2^{cN/3}) $ time to find the factors, where N is the number of digits in the number.

However, checking if a number is prime or composite is much easier - possible to do in polynomial time. This is called primality testing, and we have primality testing algorithms that run in around $ O(n^3) $ time.

These two ideas are the foundation of today's cryptographic systems, and if we are able to build a true quantum computer, that will basically be the end of all privacy.

Euclidean Geometry

Compass and straightedge geometry can be considered the first model of computation in theoretical computer science. The course explores this model as an example of what we will be doing further along with other, more modern models.

The compass and straightedge geometry model says that given any two points on a plane, we can perform the following operations on them -

1. Draw a line through them
2. Bisect the line
3. Extend the line arbitrarily
4. Draw a circle with the line segment as a diameter

Furthermore, you can mark the intersections of any two curves as a new point, and use that point for further constructions.

Using this model of computation, we can reduce it to the following model -

Starting with the numbers 1 and 0, perform the following operations on them -, +, x, ÷, and come up with new numbers. The limits of this model are then seen in

the numbers (or types of numbers) that we can't come up with. For example, we can come up with 2 by using the pythagorean theorem in the plane. However, we can't come up with 32 using just 1s, 0s, and the above operations. The proof of this requires Galois theory, but it is a proven limitation.

This example is more mathematical than computational, but this illustrates the type of reasoning we perform in this course. We start with a set of well defined rules, and then extend them to find out exactly what we can and cannot do, as well as give reasons as to why we definitely cannot do something, regardless of how clever we are or how much computational power we have.

Lecture 2 - Logic, Circuits, and Gates

Gates and Universality

This lecture introduces the fundamental logic gates - AND, OR, and NOT, and then talks about their universality.

Universality in this case means that we can construct any boolean function using a combination of these gates. In computer science, universality is the norm, and most sets are usually universal. You have to work a little to get a set that is not universal. Some examples for non-universal sets in logic gates are - {AND, OR}, and {XOR, NOT}.

This is because AND gates and OR gates are monotonic functions. That is, changing one of the inputs from a 0 to a 1 can only change the output from a 0 to a 1, or leave it unaffected. Whereas to represent a NOT gate, we would need to change the output from a 1 to a 0 when the input went from 0 to 1, which is not possible to do with monotonic gates.

An example of a universal set of gates is {AND, OR, NOT}. To prove that this set is universal, we need to prove that we can use these gates to produce an arbitrary boolean expression of any number of bits. We can do this using a truth table.

For any arbitrary boolean expression, draw a truth table or all possible input combinations. Then, only consider the input combinations which output a 1. For each such input combination, build a circuit that would output a 1. That is, if one input combination that outputs a 1 for 3 inputs is $ {X, Y, Z} = {0, 0, 1} $, then we would build a circuit for this combination as $\neg X \wedge \neg Y \wedge Z$.

Once we have circuits for each such row (that outputs a 1 in the final boolean expression), we just OR all of those circuits to get our expression.

This is the same technique I learned in college. However, this is pretty inefficient, because it produces a circuit of size $O(n \cdot 2^n)$. The size of a circuit is the same as the number of gates in the circuit. That's pretty bad, so we need to improve our approach, which we will do shortly.

Another set of universal gates we can use is {AND, NOT}. This is because we can simulate the OR gate using an AND gate and a NOT gate. Another set of universal gates is {NAND} . We can use the NAND gate to simulate any other gate, hence it is universal (so is the NOR gate). But these sets don't help us either, because using the same approach as above with these sets of universal gates would only affect the size of the circuit by a constant factor.

Gates and P vs NP

Now you might wonder if there is a bound on the number of gates we will need to represent any arbitrary boolean expression of n bits. That is, if we have any arbitrary boolean function of n bits, what is the smallest size of the circuit we can build to represent it? Using the approach above, we definitely know that it can be done in at most $O(n \cdot 2^n)$ bits, but can we come up with a better bound?

Proving that a specific circuit definitely needs an exponential number of gates and it cannot be represented in any fewer is an open problem. If you think about it, it's the same as P vs NP. If you build a circuit to represent an NP-complete problem, you can't prove that it needs an exponential number of gates (at least no one has proven it yet), and proving it would be the same as proving $P \neq NP$.

However, we can prove that a majority of boolean expressions need a circuit of exponential size. Sure, there are some boolean expressions for which we can have as small circuit, but most require a large circuit. This was proven by Claude Shannon in 1949 using a counting argument.

The gist of the argument is as follows - we count the number of boolean expressions which take n bits as input, we count the number of circuits that we can build using n inputs of size S, and we compare the two quantities to realize how large S has to be to equal the number of boolean expressions with n inputs.

The number of boolean expressions that take n bits as input is 2^{2^n}. This is because for an n bit boolean expression, the truth table has 2^n rows. Now for each row, the output value could be 0 or 1, so the number of possible truth tables is 2^{2^n}.

Now, let's say we want to build a circuit of size S, which takes n bits as input. The number of circuits of size 1 which take n bits as input is just $nC2$, since any 2 input bits could be used as the input to a gate. Similarly, the number of circuits of size 2 is $\binom{n}{2} \cdot \binom{(n+1)}{2}$, since adding the first gate creates a new input to which we can attach the second gate. If we keep going, we find that the number of circuits of size S using n inputs is $\binom{n}{2} \cdot \binom{(n+1)}{2} \cdot ... \cdot \binom{(n+S-1)}{2}$. We can approximate this to be at most $(n+S)^{2S}$.

Now we want

$ (n+S)^{2S} = 2^{2^{n}} $

Which gives us

$S = \Theta(\frac{2^n}{n})$

This means that to represent a boolean expression with n inputs, most circuits will have to be of size $S = \Theta(\frac{2^n}{n})$.

Lecture 3 - Finite State Automata & Regular Languages

Finite Automata

A finite automaton is the same as a finite state machine. It has a finite set of states that it can be in, and it takes input from a set called its alphabet. It has a transition function that takes one alphabet and the current state as input and transitions to the next state.

The notation we will use for these quantities is the following -

Q | Finite set of states |
Σ | Finite alphabet |
$\delta : Q \times \Sigma \to Q$ | Transition function |
q_0 | Start state |
$F \subset Q$ | Accepted states |

We call

$$L = 0, 1^*$$

a *language*. In this case, a language is the set of all binary strings of length 0 or more. A language is just any set of strings. We say that a language is *accepted* or *recognized* by an automaton if the automaton ends up in an accepting state for all strings in the language, and we denote that as $L(M)$, meaning L is a language recognized by the finite automaton M. We say that L is a *regular* language if we can design an automaton to accept that language.

Regular languages have the following properties - 1. The intersection of regular languages is regular 2. Complement of a regular language is regular 3. Regular languages are closed under union, intersection and complement 4. All finite languages are regular

An example of a language that is not regular is the language of all palindromes. We can prove this using the pigeonhole principle. A finite state automaton has a finite number of states, and this language can have strings of infinite length. Since the amount of "memory" a finite automaton has is essentially equal to the number of states it has, it cannot store inputs whose length is more than the number of states it has. Therefore, we cannot design a finite state automaton for this language, so it is not regular. This was just a rough proof, but the lecture gives a detailed proof.

Nondeterministic Finite State Automata

A non-deterministic finite state automaton is an extension of DFAs. It has the property that it can have any number of ways to transition from one state to another, even when it reads the same symbol. That means that if it is in a state q_i, it can

have any number of paths to follow when it reads the symbol 0, for example, whereas a DFA only has one path per symbol.

For example, we have the NDFA that accepts the language where the 3rd to last bit is a 1, which can be drawn as given below -

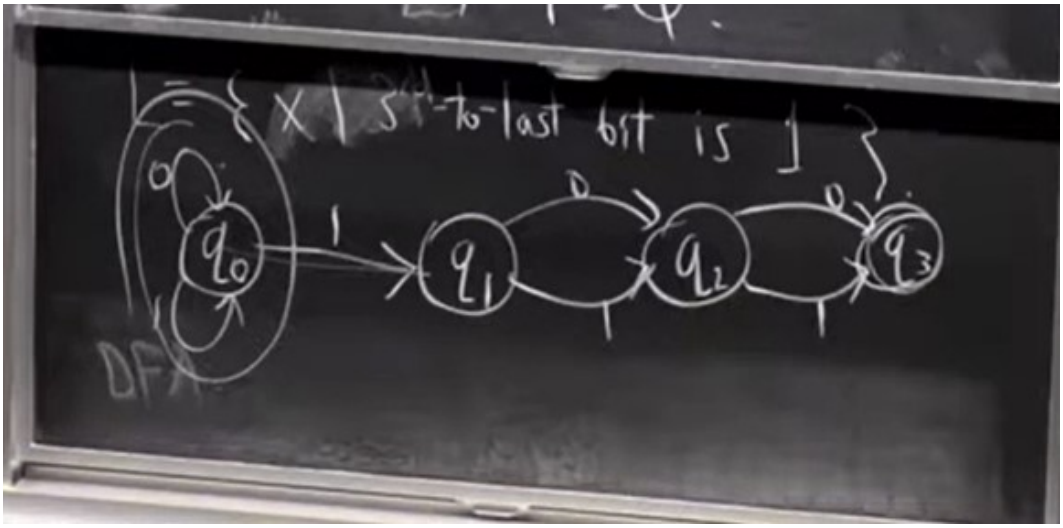


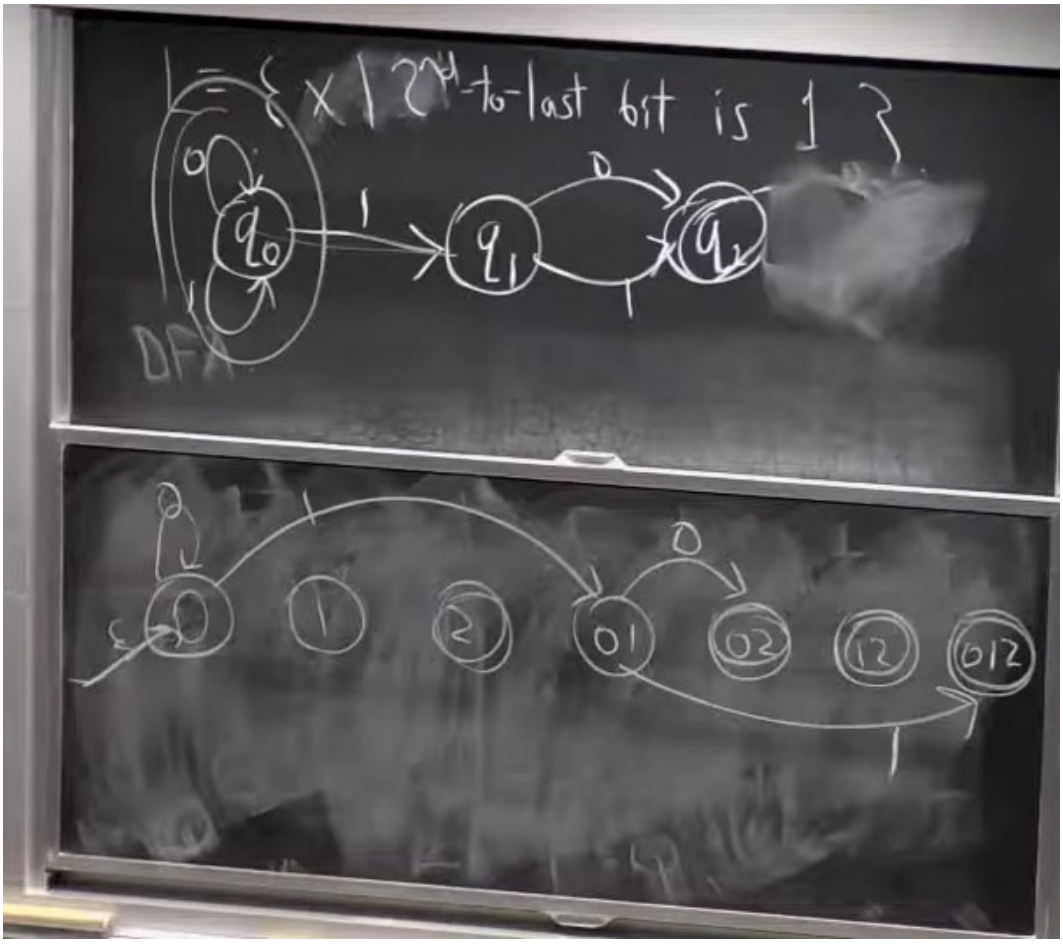
Figure 1: NDFA accepting language where 3rd to last bit is 1

It turns out that NDFAs are essentially equivalent to DFAs. For every NDFA that accepts a language L, we can construct a DFA that accepts the same language, and for every DFA that accepts a language L, we can construct an NDFA that accepts the same language.

We do this by power set construction. First of all, converting from a DFA to an NDFA is trivial. All DFAs are NDFAs. To convert from an NDFA to a DFA, we construct a power set of the states the NDFA can be in. For example, if the NDFA can be in states {0, 1, 2}, then we construct a power set {0, 1, 2, 01, 02, 12, 012} (we exclude the null element in the power set). We can then convert from an NDFA to a DFA by following transitions from the start state of the NDFA, and whenever we read a character that can transition to multiple states, we jump to that element in the power set that has all of those states.

For example, if we consider the same NDFA given above, and we start at the start state and read a 1, we can end up in either the start state itself (state 0) or state 1. So in the DFA we transition to state 01.

An example conversion is shown below for the NDFA that accepts languages where the second to last bit is a 1 to its corresponding DFA.


L = { x | 2nd-to-last bit is 1 }
0
1
q0
q1
q2
DFA
0
1
2
01
02
12
012

Lecture 4 - Regular Expressions, Context-Free Grammars

Regular Expressions to DFAs

Regular expressions are basically equivalent to DFAs. To show this, we have to show that, given a regular expression, we can design a DFA that accepts the same language as the regular expression, and given a DFA, we can write a regular expression that accepts the same language as the DFA.

To prove the first part of the implication, regular expression $\rightarrow$ DFA, we first show that we can design an NFA to represent a regular expression. We then know how to convert from an NFA to a DFA.

Regular expressions have three important operations - union, concatenation, and the * operation (zero or more). Suppose we have a regular expression without any of these operations, then this regular expression can be represented by a simple NFA that looks like a simple path. If the regular expression is `0110`, for example, the NFA would just be a start node, followed by 4 other nodes, which it reaches if it reads the characters `0110` in sequence, the last of which is the accepting state.

So we can easily design an NFA for a regular expression that doesn't use any operations. Now we show that if any two regular expressions are combined with an operation, then we can combine the corresponding NFAs for the two expressions using a simple method. The methods are shown in the photo below -

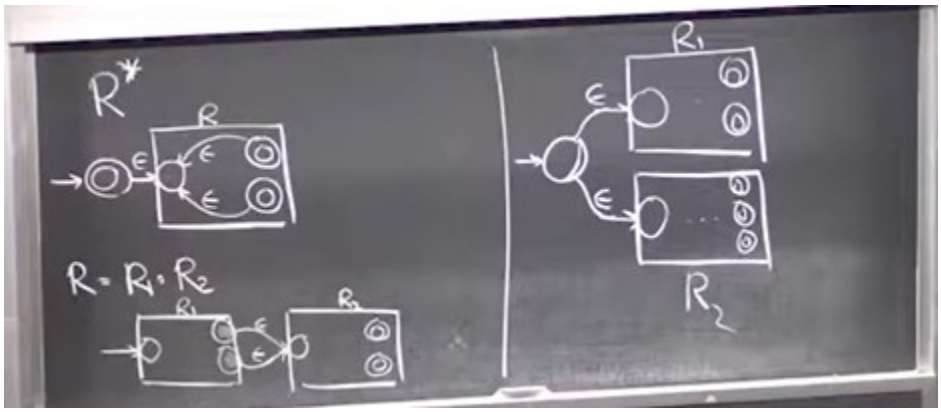


Figure 2: Converting from regular expressions to NFAs

If the expressions are combined using the union operation - `01|10`, then we just take the two NFAs for the two expressions and put a common start state before both of them with null transitions to the start states of the two NFAs. Think of this as similar to an OR gate.

If the expressions are concatenated, think of this as similar to an AND gate.

If an expression has the * operation, we add transitions from the accepting state back to the start state with null transitions, as shown above.

DFAs to Regular Expressions

To convert from a DFA to a regular expression, we first convert from a DFA to a GNFA. A GNFA stands for General NFA. A general NFA is just like an NFA, but its transitions function operates on regular expressions instead of an alphabet. So you can have regular expressions on the arrows between states.

For our purposes, we will make GNFAs satisfy the constraint that they only have one start state with only outgoing edges, and only one accepting state with only incoming edges. Once we convert from a DFA to a GNFA with some number of states k, we reduce the k states down to 2 (as shown in the lecture, pretty straightforward). Once we are left with 2 states in our GNFA, we can write it as a regular expression.

Context-Free Grammars

A context-free grammar is a set of rules for generating any string in a language. It can be defined by the following tuple - (V, Σ, R, S). V is the set of variables for the grammar, Σ is the alphabet of the language we are going to generate, R is the set of rules for generating the language, and S is the start variable from which we start generation.

An example of a context-free grammar is

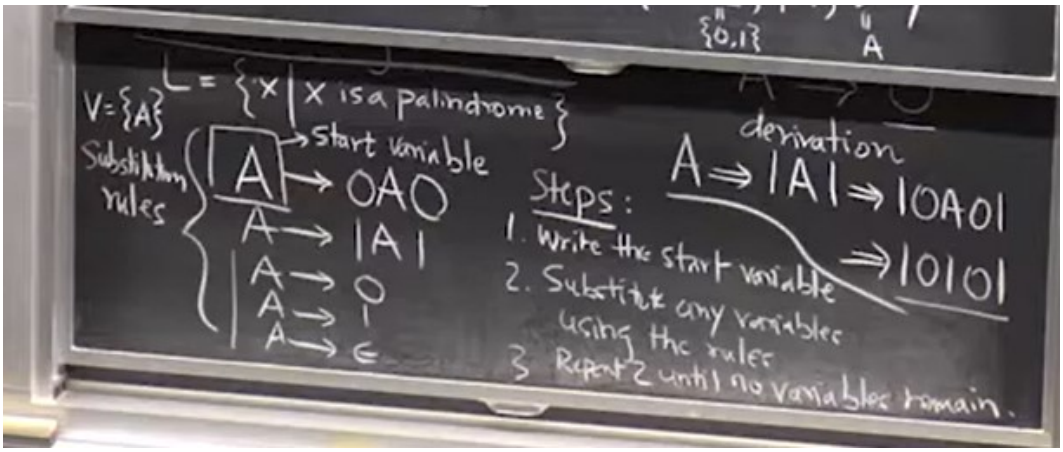


Figure 3: Context-free grammar

Languages that are generated from a context-free grammar are called context-free languages. As you might expect, since CFG is a more powerful model of computation as compared to regular expressions or DFAs, regular languages are a subset of

context-free languages. That is, every regular language is a context-free language (i.e. can be expressed with a CFG).

Pushdown Automata

Pushdown automata (PDAs) are a more powerful variation of NFAs. Think of them as basically NFAs with access to a stack onto which they can push items and pop items. They allow us to express a larger set of languages.

Even though NFAs are equivalent to DFAs, PDAs are not equivalent to their deterministic counterparts. That is, DPDAs $\neq$ PDAs.

Lecture 5 - Turing Machines

Alan Turing, in 1936, proposed a more powerful model of the finite automata, called the Turing machine (On Computable Numbers, A.M. Turing, 1936).

A Turing machine is a finite automata which has the ability to read a symbol and transition to another state, just like a DFA, but it can also write a symbol on the same paper it is reading from, and it can further move the paper it is reading from either to the right or to the left. It turns out that this model is powerful enough to be able to make arbitrary finite computations.

For example, we can have a Turing machine that accepts the language of palindromes. Let's say we have an infinitely long tape with an input. The start and end of the input are marked with special symbols, say `#`, so we know that the input has started or ended. Then we can design a Turing machine to accept the input only if it is a palindrome by the following (rough) algorithm -

1. Start in a some start state.
2. If you read a one in the start state, overwrite it with a `#` and keep moving to the right till you reach the end of the input.
3. If the last symbol read is also 1, replace it with a `#` and come back to the start of the input (the 1 that we overwrote with a #), and repeat.

Church-Turing Hypothesis

Now that we have such a powerful model of computation, you may wonder why not keep making it even more powerful. What if we have random access in our model of computation, or multiple tapes to read from and write to instead of just one. It turns out that all of these models of computation are no more powerful than a Turing machine, because they can be simulated by a Turing machine.

This is called the Church-Turing thesis. It is a hypothesis and doesn't have a proof, and could be disproved some time in the future, but for now we haven't found a counterexample.

The Church-Turing thesis says that any model of computation you can conceive that follows the physical laws of our universe can be simulated by a regular Turing machine.

The lecture gives two examples of such "reductions". A Turing machine that has a one-way infinite tape instead of a two-way infinite tape is equivalent to a two-way infinite tape Turing machine. A Turing machine that can read from two tapes simultaneously is equivalent to a regular Turing machine.

Computable Language

Finally, we define a language that a Turing machine can accept. Just as a language that was accepted by a DFA was called a regular language, and a language that was accepted by a context-free grammar was called a context-free language, a language that is accepted by a Turing machine is called a computable language.

Lecture 6 - Decidability

An important theorem regarding Turing machines is that a Turing machine can simulate other Turing machines, which can potentially be more complicated than itself. That is, a Turing machine A can act as an *interpreter* for Turing machine B, and B can be much more complicated than A.

Alan Turing actually gave such an example of an interpreter in his 1936 paper, but it turned out to have a mistake, so he had to publish a correction in 1937. Regardless, it can be done.

Halting Problem

A question you might have now is what can Turing machines not do? One famous example of a problem Turing machines can't solve is the halting problem - given a description of a Turing machine, return whether the machine will halt on an empty input.

For some Turing machines (and some computer programs), we can easily tell if the machine will halt or get stuck in an infinite loop. However, we cannot know that for sure for all Turing machines.

If we did know how to solve the halting problem, we would also be able to solve many unsolved problems in mathematics, like the Riemann hypothesis, or Goldbach's conjecture.

The proof that we can't solve the halting problem is given by contradiction as follows. Say you have a Turing machine P that takes the description of a Turing machine as input and tells you whether the input will halt on an empty input. This is not a special case, as we can always convert a Turing machine that takes some non-empty input into another Turing machine that takes an empty input but performs the same operations as the first Turing machine on the specific non-empty input.

Now, say we have another machine Q that takes the description of a Turing machine M, runs P on it, and goes into an infinite loop if M halts, and halts if M loops. If we pass the description of Q into Q itself, we get a paradox. Therefore, we contradict ourselves, and we cannot have a Turing machine P.

Busy Beaver & Infinities

Busy Beaver is a sequence that grows faster than any other countable sequence. Scott introduces it in the lecture as a way to introduce countability, sets of infinities, and prove that most languages are not countable.

The Busy Beaver sequence is defined as follows, `BB(n)` is the maximum number of steps an `n` state Turing Machine can take on an empty input before halting. Of

course, the maximum number of steps an `n` state Turing Machine can take is infinity, since it can just go into an infinite loop, so we restrict our discussion only to those Turing machines that halt on an empty input.

The first few values of the `BB` sequence are as follows -

`BB(1)` | 1 |
`BB(2)` | 6 |
`BB(3)` | 21 |
`BB(4)` | 107 |
`BB(5)` | >= 47,176,870 |
`BB(6)` | $>3 \cdot 10^{1730}$ |

We are not yet sure about the values of `BB(5)` and above. It turns out that this sequence grows faster than any computable sequence. We prove this by contradiction. Let's say that the Busy Beaver sequence was a computable sequence. Then we could design a Turing Machine to take an integer N as input and output `BB(N)`. However, if we knew `BB(N)`, we could use that fact to solve the halting problem. We could just run a Turing Machine (with N states), and if it took more than `BB(N)` steps, then we could be sure that the Turing Machine never halted.

Since we *can't* solve the halting problem, this is a contradiction, and the Busy Beaver sequence is not countable.

Scott then introduces the different types of infinities - countable and uncountable.

A countable infinity is any set that can be shown to have a one-to-one mapping with the set of integers. For example, the set of all even numbers can be shown to have a one-to-one mapping with the set of integers. This means that the set of all even numbers is countably infinite (interestingly, it also means that there are as many even numbers as integers). Similarly, the rational numbers are countably infinite.

Also, the set of all Turing Machines is countable. This is because each Turing machine can be represented as a series of transitions from each of its states, and all of these transitions can be encoded as a binary string. Since the set of all binary strings is just the set of all integers, the set of all Turing Machines is countable.

An important observation is that the set of all Languages is not countable. The lecture has a simple proof. This means that if we choose a language at random, the probability that a Turing Machine recognizes that language is 0%.

The set of countable infinity is called $\aleph$ (Aleph), and the set of uncountable infinities is called $2^{\aleph}$. In general, there is no largest set of infinity. You can always keep constructing sets that have more elements than the largest infinity you have by just taking the set of all subsets of that infinity.

Lecture 7 - Turing Recognizability, Oracles

Turing Recognizability

We now define a new kind of language, a Turing Recognizable language - very similar to a decidable language, with one key difference.

A Turing Recognizable language is a language for which we can construct a Turing Machine such that, for all inputs in the language, the Turing Machine halts and accepts, and for all inputs not in the language, the Turing Machine either halts and rejects or goes into an infinite loop.

The only difference between a decidable and recognizable language is the very last clause, which allows the machine to go into an infinite loop. This means that the halting problem can be phrased in terms of a Turing Recognizable language.

$$HALT = \{< M >, x : \text{M halts on x}\}$$

We can clearly see that for all strings in the language, we can construct a Turing machine to either halt and accept (just run M on x), or go into an infinite loop (just run M on x again, and it will go into an infinite loop if x is not in the language).

We have now seen a few classes of languages of increasing generality, shown in Figure 1.

As you would expect, the halting problem is in the set of recognizable languages but not in the set of decidable languages. Now if we define a new language, $\overline{HALT}$, which is the *complement* of $HALT$, it lies even outside the class of recognizable languages.

Reductions

Reductions are a way of solving a problem that you don't know how to solve using a solution to a problem that you do know how to solve. Reductions were seen before in 6.006 and 6.046. Reductions are seen everywhere in computer science, and are especially useful while proving the NP-completeness (or incompleteness) of some problem. Similarly, you can also use them to show that some problems are decidable or undecidable.

Formally, we represent a reduction as $A \leq_T B$, where the T stands for Turing reducibility. This denotes that problem A can be reduced to problem B, where A is the problem that we want to solve, and B is the problem that we know how to solve.

We can also write this as A is decided by M^B, where we call B an "oracle", because we get to assume that the Turing Machine M has access to Turing Machine B, which already knows how to solve its own problem

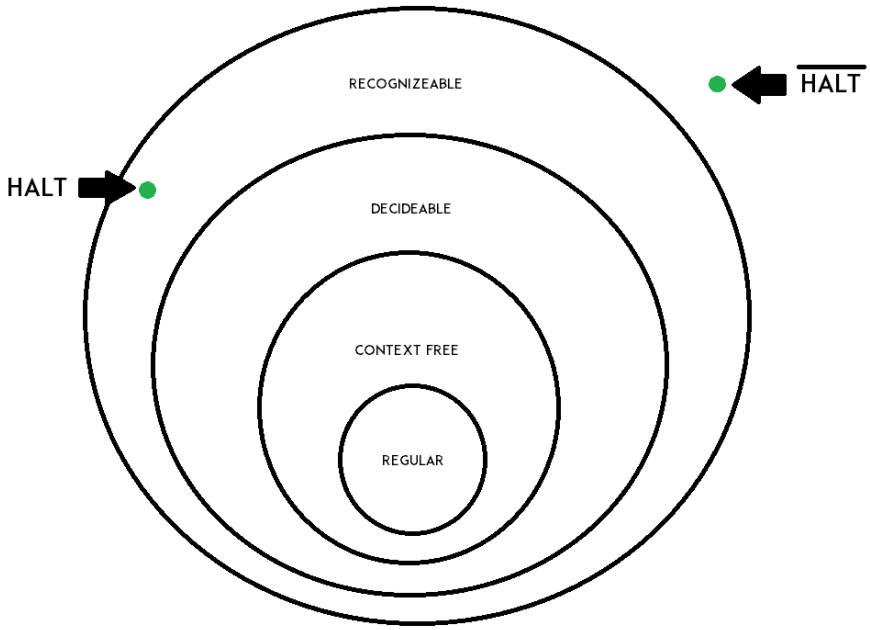


Figure 4: Classes of languages

In a Turing reduction, we can call the oracle any number of times, even zero. We are not forced to use the oracle. This is not the case for all kinds of reductions, for example in case of mapping reductions, you must call the oracle exactly once, and you must output the result of the oracle.

Mapping reductions can be a little more helpful that Turing reductions, because a mapping reduction between two problems means that the two problems are in the same class of languages, which is not guaranteed by a Turing reduction.

For example, you can come up with a Turing reduction to reduce $HALT$ to $\overline{HALT}$, but they are in complementary classes. However, we can come up with a mapping reduction from the halting problem with no input to the halting problem that takes any general input, which means that the two problems are in the same class of languages.

Fermat's last theorem was proved this way around 1995, when it was shown that that halting problem can be reduced to Fermat's last theorem using a mapping reduction. In other words, if you had an oracle that solved Fermat's last theorem, you could solve the halting problem.

Turing Degrees

A Turing degree is a set of problems that have the same difficulty. For example, the set of all decidable languages forms a Turing degree. The set of all undecidable problems also forms a Turing degree.

Each Turing degree has $\aleph$ languages. This is because each pair of languages in a degree can be reduced to each other, and you need a Turing machine to do reduce any problem to another. And since there are only $\aleph$ number of Turing machines, there can be at most $\aleph_0$ number of problems in a degree.

Another interesting fact is that there are problems even harder than the halting problem. To come up with a problem harder than the halting problem, consider a Turing machine that has access to an oracle that solves the halting problem. Now we ask whether this new Turing machine will halt on an empty input.

Using the same proof that we used to prove that the halting problem was undecidable (i.e, not solvable by a Turing machine), we can prove that this new halting problem is not solvable by our augmented Turing machine that has access to a halting problem oracle. This creates a new class of languages that form another Turing degree.

You can see that this means there are an infinite number of Turing degrees, since we can always come up with a harder halting problem.

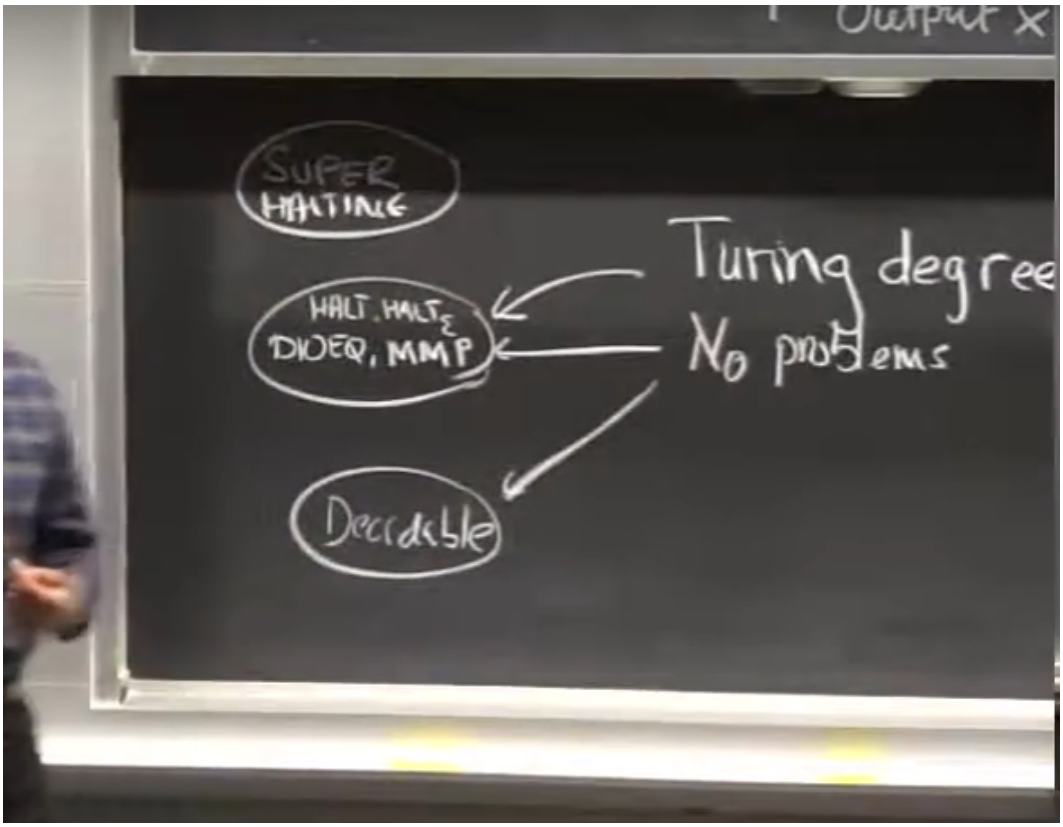


Figure 5: Turing degrees

Lecture 8 - Turing Degrees, Dovetailing, Godel

In the last lecture we defined Turing degrees, which are basically sets of languages. A Turing degree consists of a set of languages that can all be reduced to another language in the same set using a Turing reduction.

The set of computable languages is called degree 0. The set of languages equivalent to the halting problem is just above degree 0. A natural question to now ask is are there degrees between these two?

It turns out the answer is yes. There are infinitely many Turing degrees between degree 0 and the halting degree. What it means to be "between" these two degrees is that if we have a language L in this intermediate degree, the following relation holds -

$$L \leq_T HALT$$

$$L \not\leq_T \text{degree0}$$

$$HALT \not\leq_T L$$

This means that the language L is reducible to the halting problem. That is, you can build a Turing machine to recognize L given an oracle for the halting problem. However, the language is not reducible to any language in degree 0, and the halting problem is not reducible to language L.