

# Theory of Computation

## Calendar

Monday	Tuesday	Wednesday	Thursday	Friday
<u>Proof techniques</u> Induction, contradiction, and construction Common pitfalls  <u>Sets and counting</u> Notation and operations Permutations and factorials	<u>Infinities</u> Set magnitude Cantorian diagonalization  <u>Powers and products</u> Numerical rules Set rules and notation	<u>Inductively defined sets</u> Inductive definitions Transfinite numbers  <u>Relations and functions</u> Ordered pairs, etc.	<u>Formal systems</u> Formal syntax Binary arithmetic Roman numerals  <u>String manipulation</u> Constructing formal systems	<u>Well-formed formulas</u> Propositional logic
<u>Regular expressions</u> Definitions and syntax  <u>Introduction to Scheme</u> First computer time	<u>DFAs</u> Introduction to automata	<u>NFAs</u> Nondeterminism Epsilon transitions	<u>NFA/DFA equivalence</u> Formal FA notation Equivalence proof	<u>RE/DFA equivalence</u> Formal RE notation Equivalence proof  <u>Regular languages</u> Chomsky hierarchy
<u>PDAs and CFGs</u> Push-down automata Context-free grammars  <u>Scheme project</u> Begin parsing project	<u>Turing machines</u> Syntax and examples Usefulness and importance	<u>Decidability</u> Undecidable languages  <u>Halting problem</u> Proof of existence	<u>Runtime complexity</u> P and NP	<u>Wrap-up</u>

## Monday #1

### Morning

- 10 (Activity) Class Introductions  
Learn everybody's name. Usually works best starting with the standard circle activity - sit in a circle, start to teacher's left, go around clockwise repeating back everybody's name before you. Then (or) have each student pick the next person rather than going in order. It's a little scary, but it works.
- 5 (Activity) CTY Paperwork
- 5 (Handout 00, Handout 01) Syllabus  
Overview of theory of computer science, course topics, layout of the class, students, etc. This includes introducing the instructor/TA to the students, getting student information, giving out syllabus and policy information, and doing a general overview of theory and algorithms subfields. Make sure to mention why this stuff matters, in the sense that some problems are hard, some are easy, some are impossible, and some have to be approximated. This is all still currently relevant, both in industry and research!
- 20 (Problems 00) Pretest  
Gather student information, generally through a little fun questionnaire.
- 80 (Handout 02) Proof techniques
- Introduction to the Theory of Computation, p21-25
- Ways to prove things and why they're important. Cover the basics of induction, proof by contradiction, and inference (or construction). Also cover bad proof methods - pull out the horse examples, etc.

### Afternoon

- 55 (Handout 03) Sets and counting
- Introduction to the Theory of Computation, p3-9
  - Discrete and Combinatorial Mathematics, 4th Edition, p1-11
- Start with set theory, rapidly - most students have probably seen the basic operations before. Get them used to thinking of sets and their operators as abstractions over symbolic objects; we don't care what sets are, just how they behave. Counting is even more fun; it's easy to count things in real life, but it gets harder the more abstract it is - and everything is abstract to a computer. Go over addition and multiplication rules, and use that to move into permutations. Relate these back to set operators. Mention the non-additivity of ethanol and water as an example; it's a shame this isn't a lab course... no demonstrations. Mention the pigeonhole principle as a way to think about proofs.
- 55 (Interjection 01) Induction problems

### Evening

- 120 (Problems 01) Proofs and sets

#### *Stop signs*

Remember to distribute some sort of question signals to the students so they don't have to raise their hands. Emphasize the need to keep working on different problems while waiting for a response!

## Tuesday #1

### Morning

- 60 (Handout 04) Introduction to combinatorics
- Discrete and Combinatorial Mathematics, 4th Edition, p11-33
- Move into more advanced counting and combinations. Start from permutations and explain why this breaks down for unordered sets (if they haven't noticed already) or for repeats. Introduce the reason for combinations, the basic formula, and the notation. Make sure to emphasize interchangeability of choice from sets and distribution into buckets. Tie this back to meaningless syntax with the binomial theorem and the fact that this applies to formulaic coefficients (although make sure you explain why this happens, too).
- 60 (Interjection 02) Pascal's triangle
- Demonstrate Pascal's triangle construction for those who haven't seen it. Create notation for a particular entry - say, row 3 entry 2 is  $r(3,2)$  or some such. Then we can easily write  $r(3,2)=r(2,1)+r(2,2)$ . What does this really mean? What do these numbers really mean (mention the exponential as a number that's more important for its properties than its value)? Suppose that our  $r$ 's are really just another way to write binomial coefficients. What does this mean? Can you prove it?

### Afternoon

- 60 (Handout 05) Infinity
- Discrete and Combinatorial Mathematics, 4th Edition, p151-156
- Start with magnitude notation - that's easy. Give the intuitive examples of how  $|A+B|$  and  $|A*B|$  work. Now introduce numbers as sets. The integers are easy, which gives us modifications such as the positive integers, the nonnegative integers (natural numbers), and the negative integers. Show the blackboard bold symbols - and mention that you can never keep them straight. Show using intuitive definitions that they're all closed over addition, that the integers are closed over subtraction, and that the integers or the naturals are closed over multiplication. Note why these are interrelated, but make them take a number theory or algebra course to get more in depth. Then get into division to introduce the rationals, where all of these operators are closed. Show that the integers are a strict subset of the rationals (easy), but they're still the same size (hard). Then introduce the reals, mention that they're also closed (should be obvious), show that they're a strict superset of the rationals (easy), and use diagonalization to show that they're not the same size. This gives us an interesting definition of infinity, when a strict subset may or may not be smaller in magnitude. Also mention that integer sets have a smallest element, while rationals and reals don't (and how important this is for analytical proofs), and do the neat demonstrations of the number of rationals and reals between 0 and 1.

867-5309

When doing Cantorian diagonalization, make sure the numbers on the diagonal end up being 8675309. At the appropriate point, arrange to burst into song with the TA. It seems lame, but it worked for Mark and Matt, and I still remember it ten years later.

- 60 (Interjection 03) The triangle problem
- Introduce the triangles problem as a challenge (I still don't know how to solve it). Start with  $n$  rows of dots, with the  $i$ th row containing  $i$  dots. How many triangles can you draw containing all of the dots (easy)? How many equilateral triangles can you draw within the dots (not too hard)? How many isosceles triangles? How many horizontal triangles, i.e. those with the topmost point at row  $j$  and both

bottom points at some row  $k > j$ ? How many vertical triangles, i.e. those with the topmost point at least one row above the bottommost points? How many arbitrary triangles?

*Evening*

120 (Problems 02) Pascal's triangle and infinity

**Wednesday #1**

*Morning*

60 (Handout 06) Powers and products

Get started on the notation needed to suppose infinity proofs, relations, and functions. Just the notation today! Explain power set in terms of previous discussion, and Cartesian product in terms of simple ordered pairings.

60 (Handout 07) Transfinite numbers

Show that induction can be used as a definition technique as well as a proof technique.

*Afternoon*

30 (Activity) Set

50 (Interjection 04) Set

30 (Handout 08) Relations and functions

- Discrete and Combinatorial Mathematics, 4th Edition, p217-228

Suppose we define a special kind of set as a pairing of elements from two other sets. Each element is a tuple, and we can extend this to 3-tuples (triples), etc. If we take two (or more) whole sets and do this, we get a special set called a relation, which is their cross (or Cartesian) product. Ok, this isn't too exciting, but it's another way to define pretty much any relation we've seen before (take equality, greater/less than, etc.) This works for arbitrary sets, though, and isn't always necessarily intuitive. We can make an even more specialized case of this in which each element from the first set appears exactly once - in which case we have a function. And yes, you can redefine pretty much any function you've ever seen this way, too. The domain and range sets can be the same (say for real addition) or different (the floor function maps reals to integers). Do a few examples, say with color (the "add red" function, perhaps?) and with something abstract, just to show that any old pairings can be a function. Mention closure (mapping into the same set), injections (each range element appears at most once), surjections (each range element appears at least once), bijections (both), and one-to-one mappings (function is invertible), but not in detail - kick them back towards modern algebra classes again.

*Evening*

120 (Problems 03) Inductively defined sets

**Thursday #1**

*Morning*

120 (Handout 09) CTY and the AB system

- Introduction to the Theory of Computation, p13-15
- Discrete and Combinatorial Mathematics, 4th Edition, p47-57

Learn to think about symbol pushing as a meaningless operation. Start with the getting-to-CTY exercise for old time's sake and discuss why it's tricky: solving it (easily) requires thinking about the rules in an outside context rather than thinking about the problem in the context of the rules. Then go over binary addition as symbol pushing - start with the rules, have the students do a few problems with them on the board, and then see if they can figure out what's happening. Emphasize that this isn't just important in data processing, it's everything in computer science. Programs and data are both just symbols that get manipulated in particular ways, even if it's only particularly obvious for certain types of data (mention natural and controlled language grammars, arithmetic, algebra, etc.)

#### *Getting to CTY*

Use the symbol pushing rules to manipulate strings into the sequence "CTY". The usual. Do the same for binary addition, which I, at least, find a great deal more interesting.

#### *Afternoon*

80 (Interjection 05) Formal systems

30 Nines proof, sets as formal systems

#### *Evening*

120 (Problems 04) Bases and GEB

### **Friday #1**

#### *Morning*

120 (Handout 10) WFFs

- Introduction to the Theory of Computation, p15
- Discrete and Combinatorial Mathematics, 4th Edition, p47-54

Having discussed sets operators and symbol manipulation, start WFFs as a completely abstract system. Make up nonsense symbols for zero, one, and, or, and not. Define truth tables in terms of these symbols and see if you can find some equivalences. Finally, ask what they are - and note that both boolean operators or set operators are legitimate answers. Demonstrate this. Go over truth tables with real numbers and symbols, and talk about xor, implication, and double equivalence. Show that you can define the other operators in terms of not, and, and or, and emphasize what this means for implication - it makes a good way to demonstrate that something intuitive can have a purely symbolic representation, too.

- Discrete and Combinatorial Mathematics, 4th Edition, p57-86

Go over the introduction and elimination rules for the WFF operators. Do a few sample problems, interchanging real formulas with nonsense symbols to show how tricky it can be. Make sure to do a few in natural language, too, to demonstrate that way of thinking about it. See if you can get a student to think of something intuitive that lends itself to a good proof (hopefully not something from the exercises...) Relate all of this back to truth tables - show that you can solve problems like this in that way as well, although they get big! Note that using this reduction, you can get even the basic operators back to not, and, and or again. (Ask why you can't do it with just two of those three - and see if they remember it for later in cases where you're generating groups.) Also show that you can get these back to set operators, too, using membership tables.

*Afternoon*

55 (Problems 05) WFFs

55 (Activity) Gameshow

**Sunday #1**

*Evening*

120 Review and redo

**Monday #2**

*Morning*

120 (Handout 11) REs

- Introduction to the Theory of Computation, p63-83

Start with regular expressions as people may have seen them - in grep, perl, Java, .NET, whatever. Explain how nifty and useful they are - this is a good opportunity to throw something eminently practical at the class in the midst of all of this theoretical stuff. Now mention that regular expressions actually have a very deep theoretical side, too, and limit them down to certain formal operations: a two-character alphabet (although you could use just one!), concatenation, union or choice, and Kleene closure or star. Creep up to the empty string and empty language carefully - start by explaining that regular expressions really specify a set of strings. Demonstrate this! Emphasize that a RE is exactly the same as a set of strings, so it's not a problem for the empty set to be an RE. And it follows that you can manipulate the empty set just like any other RE - concatenation is a good example. Use the empty-set-star example to get at the empty string; save as much as possible for DFAs, though, since it makes way more sense there. Remember, zero is a valid number, too! If you use the notation right, you should be able to emphasize that REs and WFFs aren't really all that different - they both describe formal languages for manipulating characters. In one case, the characters represent sets, and in another case, they represent set membership or truth values, but they're both formal, syntactic systems that happen to have nifty semantic properties as well.

*Afternoon*

55 (Scheme 00) Introduction

Cover the basics of Scheme - atoms, strings, booleans, and numbers, how to represent operators (numerics, booleans, etc.), how to write it all down, keywords and control flow (if, else, when, cond, eq, etc.)

55 (Activity) Computer time

*Evening*

120 (Problems 06) REs

## Tuesday #2

### Morning

120 (Handout 12) DFAs

- Introduction to the Theory of Computation, p31-47

Assuming you can interrelate DFAs and REs pretty easily, start by generating some simple DFAs that are more easily graspable than their RE equivalents. Two good, immediate examples are the DFA that accepts with no transitions (the empty string) and a DFA with no accept states (the empty set). Emphasize that while they're interchangeable, there are things that are easier to think about as DFAs and things that are easier to think of as REs. Describe things a little more formally then; just as REs are an alphabet and operations, DFAs are an alphabet, a set of states, one start state, a set of final states, and a transition function. Focus on the transition function - note that it's a set of pairs, just like our definition of functions from earlier on, and that it takes in a state and a character to output a state. Show the notation for this. Draw out the whole five-tuple for a few simple DFAs. Then start with some abstract five-tuple and show how it has some very clear meaning. DFAs are just one way to represent things, and it's ok to assign very concrete meaning to the states if it helps a particular problem. Just remember that they're not always that way, and when you get right down to it, they again have a very purely syntactic definition

### Afternoon

55 (Interjection 06) DFAs

55 (Activity) Sidewalk DFAs

Draw automata on the sidewalk! I love this activity.

### Evening

120 (Problems 07) DFAs

## Wednesday #2

### Morning

120 (Handout 13) NFAs and e-NFAs

- Introduction to the Theory of Computation, p44-54

If necessary, either go back to some example or mention that, using regular expressions, the empty string had to be included as a valid "character". And since every FA transition is on one (or more) characters, what happens if we try to transition on the empty string? Show this and introduce nondeterminism as the concept of being in, effectively, more than one place at the same time. You can think of things as happening in parallel and "registering" their successes and failures, too - as long as you have arbitrarily many parallel tasks. Anyhow, emphasize that only one acceptance is needed and the number of failures doesn't matter. Draw out the state tree for some small NFA and input. If there's time, show how the formal definition is nearly identical to that of a DFA. Show how easy it is to map NFAs back to the RE operators (concatenation, union, and closure). And then close with the big question: NFAs seem like DFAs, and DFAs seem like REs, and NFAs seem like REs, but NFAs seem better than DFAs. Is there any difference?

### *The Kwisatz Haderach*

See if anyone's read Dune and recognizes the concept of the Kwisatz Haderach as "the one who can be in many places at once." It makes an entertainingly goofy parallel. Also, if anyone's read Dune and seen the last Matrix movie, see what they think of the similarity. It's one of those things that's tangentially related that'll keep their attention; it's hard to find someone in a CS class who doesn't like Dune and the Matrix.

### *Afternoon*

55 (Scheme 01) Primitives

55 (Activity) Computer time

### *Evening*

120 (Problems 08) NFAs

## **Thursday #2**

### *Morning*

120 (Handout 14) NFA/DFA equivalence

- Introduction to the Theory of Computation, p54-63

Once you have the idea of being in many places at once down, proving equivalence isn't so hard - because there's a finite number of places you can simultaneously be. Start with a very small NFA and enumerate them. Make sure there's at least one you can't reach to show that they don't all matter. Then use this to do the equivalence proof for NFAs and DFAs. Then do a very quick run through of the equivalence proof for NFAs and REs: each unit RE can easily be made into an NFA, and each operation can be performed, so you can build any RE. You can show it backwards, too, if you want (the top-down approach). So... going back to the start, we said that each RE (and thus FA) was just a set of strings. Are there any sets of strings that are not, in turn, equivalent to some RE/FA?

### *Afternoon*

55 (Scheme 02) Lambdas

Start by doing into a little more depth on what define means and how you're binding a variable to a value. Expand this with the concept of a local environment with let - the local and global environment have the same structure but contain different data. Then look more closely at what lambda's really doing - it's creating a partial environment and packaging it up to be used later. Once a lambda is applied, that environment is unwrapped, filled in, and evaluated. Emphasize that you can use it without the define, and that there's really not that much difference between define, let, and lambda. At the end, bring up the concept of first class functions and show how you can pass a lambda into another lambda.

55 (Activity) Computer time

### *Evening*

120 (Problems 09) NFA/DFA conversion

## **Friday #2**

### *Morning*

60 (Scheme 03) Recursion

60 (Activity) Computer time

### *Afternoon*

55 Review and redo

55 (Activity) Game show

## **Sunday #2**

### *Evening*

120 (Activity) National Plumbers Association

## **Monday #3**

### *Morning*

120 (Handout 15) PDAs and CFGs

### *Afternoon*

55 (Scheme 04) Parsing and project introduction

55 (Activity) Computer time

### *Evening*

120 (Problems 10) PDAs and CFGs

## **Tuesday #3**

### *Morning*

135 (Handout 16) Turing machines

### *Afternoon*

65 (Scheme 05) Syntax

70 (Activity) Computer time

*Evening*

120 (Problems 11) Turing machines

**Wednesday #3**

*Morning*

135 (Handout 17) Decidability

*Afternoon*

65 (Scheme 06) Evaluation

70 (Activity) Computer time

*Evening*

120 (Problems 12) Decidability

**Thursday #3**

*Morning*

135 (Handout 18) Runtime

*Afternoon*

135 (Activity) Computer time

*Evening*

60 (Problems 13) Runtime

60 (Activity) Sprouts

**Friday #3**

*Morning*

60 (Activity) Certificates and paperwork

75 (Activity) System's Twilight

## Extras

### Automata and Grammars

#### *Regular Expression Implementation*

If you're careful, you should be able to get the class to derive the existence and behavior of DFAs by themselves. Ask them how they would implement a regular expression manipulator - how they think a computer would process a regular expression. Make sure to do it with something simple, say, three ones followed by three zeros. Do something with a plus or a star. Then something with real letters, just for fun. By now, you should have several things that look a lot like DFAs on the board, making the transition pretty easy.

#### *Two-Level Morphology*

Show that placing pairs (or, for that matter, arbitrary tuples) of characters on the transition doesn't break anything (it's no different from concatenation, syntactically, just the semantics are different). Then show that by treating the second character as "output" rather than input, you can use acceptance of strings into the first language to create a mapping into a second language. This should also (gently) introduce the idea of transitions on the empty string, which will be useful with NFAs. See if any of the students know enough of some foreign language (preferably Spanish) to do a good example there. English has such lame morphology.

### Infinity and Counting

#### *Cantor Sets and Sierpinski Triangles*

Draw a Sierpinski triangle; see if anyone has seen it before. Explain how it behaves, and use it to set up the Cantor set. How many points are in the Cantor set? How "long" is it on the number line? So how much area is taken up by a Sierpinski triangle?

### First Order Logic

#### Well-formed Formulas - Quantification

- Discrete and Combinatorial Mathematics, 4th Edition, p89-103

Start with quantification in WFFs to lead into this semi-logically. First note that you can combine notations: literals in WFFs are just statements that can be true or false, which, of course, includes set membership statements. So you can say things like " $x$  in  $A$  and  $A$  subset  $B$  implies  $x$  in  $B$ ". But now you have problems with expressiveness, since you can say " $x$  is an integer implies  $x + 5$  is an integer" for some particular element  $x$ , but you can't say it for every element  $x$  simultaneously (at least not in a finite way). Introduce forall as a piece of notation to take care of this problem. Then introduce exists as its inverse; "there is no integer  $x$  for which  $x/0$  is an integer" is a good way to do it, since it has such a nice, understandable inverse using forall. Explain that this is one of the rules for manipulating quantifiers symbolically, just like manipulating other WFF operators. Demonstrate that ordering among like quantifiers doesn't matter, but ordering among unlike quantifiers does ("For all things that I say  $p$ , there exists a meaning  $q$  such that  $p$  implies  $q$ ," versus, "There exists a meaning  $q$  such that for all things I say  $p$ ,  $p$  implies  $q$ .") Mention rules like distributivity, but leave their demonstration/proof for exercises. Note that this still has expressiveness issues - you can't say, for example, "There are exactly three integers  $x$  such that  $4/x$  is an integer." And this is a whole different can of worms.

## Groups and Cwatsets

### *Campfire*

Shred red and orange construction paper for camp songs.

### *Riddles*

Give the catch phrase, allow yes or no questions. Look these up online.

### *Frisbee Models*

Given a circle, a starting point for the frisbee, and a rule for each student (e.g. left/right then reverse after a throw), will the entire circle be tiled. This is good for showing a real-life model of something a computer could do. Bring it back to something like cellular automata.

## Scheme

### Scheme - Java and Object Orientation

Using what we already know about Scheme, give a whirlwind introduction to the programming style of Java or C++ (or C#, if anyone's familiar with it). Then contrast this with Scheme - iteration versus recursion (mention the hopelessly screwy way of doing iteration in Scheme, but leave it hanging), highlight tail recursion (which is nothing complicated, except that other languages don't have it), and bring up the lack of object orientation. Lead into the idea of lambdas as objects based on the lambdas as environment talk - lambdas package up an environment, which contains data, the same way that objects in Java package up data. First show a very simple example with one method and no data. Then call out the fact that this has a class and can be used to create multiple objects, at which point it's useful to have data as well - which you can do with a let or a lambda (since they're the same!) Play around with this concept a while so people can get used to it - it's a neat trick, but it's complicated. Think of variations on it, etc.

### Scheme - Inheritance and Polymorphism

Since we've seen classes, explain more how they apply to real life. Think of some dumb examples, and then think of one dumb enough to actually model in Scheme. Introduce the concept of passing a message back up to a parent lambda/class and demonstrate how it's done. Write out the full model - simply! - so that everyone can see it. Then add in some polymorphism, explaining the word (and griping about how people don't just say multiple inheritance). Mention the idea of interfaces, which don't really work in Scheme, and explain how they can be used in Java or for even more complicated things like COM's binary compatibility (don't say COM!)