Lecture 25

Review of the Course Material

Congratulations on making it to the end of the course! Here is a review of the material we've covered.

The course can be split into 4 units:

1. The first unit covered regular languages and their associated models of computation and equivalent formulations: DFAs, NFAs, regular expressions, and the Myhill-Nerode theorem.

2. The second unit covered context-free languages and their equivalent formulations: context free grammars and (non-deterministic) pushdown automata.

3. The third unit covered computability theory. This includes Turing machines, decidable and recognizable languages, computable functions, enumerators, and topics such as the Busy Beaver function, Kolmogorov complexity, the Recursion theorem, and Rice's theorem.

4. The fourth and final unit covered complexity theory. This includes complexity classes such as P, NP, coNP, PSPACE, and EXP, and the relationships between them.

We can think of the first two units as covering models of computation that are *too weak*: the regular and context-free languages are important classes of languages, but they don't characterize everything that can be done on a computer. For example, it is easy to write a computer program that checks whether the input is formatted like $0^n1^n2^n$, and yet such a program recognizes a language which is not regular and not context-free.

We can think of the third unit as covering models of computation that are *too strong*, or at least, models of computation that capture the power of computers without taking into account the fact that in practice, the amount of time and space available to us is limited. If a language is decidable, then it can indeed be decided by a computer program, but we don't have any bound on how many resources such a program might require. This is not particularly realistic, as we live in a finite universe. For instance, there are fewer than 10^{100} atoms in the observable universe; if we had a program that decides a language A but uses exponential space – say 2^n space where n is the input size – then it may not be possible to run such a program on inputs of size 400, even if we were to turn the whole universe into a computer.

(Time is an equally important resource: it is not physically possible to do a step of computation in less than Planck time, which means that it's not possible to do more than around 10^{50} sequential steps of computation in a year, no matter how we might design a futuristic computer. Since we have less than 10^{50} years until the death of the universe, we can fit at most 10^{100} sequential steps of computation in our finite universe, so an exponential-time algorithm – say, one running in 2^n time, where n is the input size – cannot be implemented on inputs of size 400 or larger.)

The fourth unit introduced resource-bounded complexity classes. In particular, the class P is finally something we might call a reasonable definition of the problems efficiently solvable by a computer. We also introduced other classes, such as NP, and we showed that some problems are NP-complete – meaning that they are as hard as any other language in NP. In practice, many problems we are interested in solving turn out to be NP-complete, meaning that they are in NP but that there is no polynomial-time algorithm for solving them unless P = NP.

There are other complexity classes that we did not cover in this course. One important class is the class BQP, of languages that can be decided in polynomial time on a *quantum* Turing machine. This class is meant to model the set of languages that a quantum computer might be able to solve. We don't know whether P = BQP, but we do know of some languages that are in BQP that we don't know how to solve efficiently using classical computers; for this reason, many people conjecture that BQP $\neq$ P. If so, and if it is possible to build quantum computers in real life, it would mean that the computational power of the universe isn't captured by P, but is actually stronger. The class P would still be a good model for what can be done on ordinary computers, however. I'll also mention that even quantum computers don't seem to be able to solve NP-complete problems; we believe that NP $\not\subseteq$ BQP. This means that even with quantum computers, the NP-complete problems will still be intractable.

We will now briefly review each of these units in turn.

25.1 Regular languages

The regular languages are those that can be computed by a machine that has a finite, constant amount of memory (not even enough to store the input), and receives the symbols of the input string one at a time.

We've seen many different characterizations of the regular languages:

1. The regular languages are the languages that can be recognized by a DFA.
2. The regular languages are the languages that can be recognized by an NFA.
3. The regular languages are the languages that can be represented by a regular expression.
4. The regular languages are those that have finitely many equivalence classes for their associated equivalence relation, as defined in the Myhill-Nerode theorem.

We also showed that the regular languages are closed under many different operations:

1. Complement, union, and intersection
2. concatenation and star
3. Change of alphabet (e.g. replacing the symbol 0 with the symbol 1 and vice versa in all strings)
4. Other operations, such as reversing the strings in a language, or taking all prefixes, all suffixes, or all substrings.

Roughly speaking, the expressive power of regular languages (or DFAs) is that they can look for a pattern (e.g. the set of all strings containing 00110 as a substring is regular) and they can count modulo some constant (e.g. the set of all strings in $\{0,1\}^*$ which contain an odd number of 1s is regular, as is the set of all strings whose length is equivalent to 2 modulo 5). However, they cannot count; the language $\{0^n1^m : m \geq n\}$ is not regular.

We saw several ways to show that a language is not regular:

1. The pumping lemma for regular languages can be used to show that a language is not regular. (though it cannot be used to show that a language is regular).
2. One could also use closure properties to show that a language is not regular; if B is known to be regular, then one can show that A is not regular by showing that $A \cap B$ is not regular, for example.
3. Finally, the Myhill-Nerode theorem can be used to show that a language is not regular. To do so, one would need to find a sequence of infinitely many strings, no two of which are equivalent under the equivalence relation for the language; this shows the language has infinitely many equivalence classes, and hence is not regular.

25.2 Context-free languages

The context-free languages are those that can be generated by a context-free grammar. This class is somewhat less natural than the class of regular languages, but still shows up in many contexts, including in compiler design and in linguistics. We saw several equivalent characterizations of context-free languages:

1. The context-free languages are those that can be generated by a context-free grammar.
2. The context-free languages are those that can be generated by a context-free grammar in Chomsky Normal Form.
3. The context-free languages are those that can be recognized by a non-deterministic pushdown automaton.

The characterization in terms of grammars in Chomsky Normal Form can be surprisingly useful; for example, it's how we proved the pumping lemma for context-free languages.

We saw that the context-free languages are closed under various operations:

1. union, but not intersection or complement
2. intersection with a regular language (i.e. if A is context-free and B is regular, then $A \cap B$ is context-free)
3. concatenation and star
4. change of alphabet (as for regular languages)
5. reverse, prefix, suffix, and substring (as for regular languages).

The context-free languages are a superset of the regular languages, and have more expressive power. Languages like $\{0^n 1^n : n \in \mathbb{N}\}$ are context-free but not regular. However, intuitively context-free languages can only remember in a limited, stack-like format: they can count well enough to recognize $\{0^n 1^n : n \in \mathbb{N}\}$, but they cannot recognize $\{0^n 1^n 2^n : n \in \mathbb{N}\}$. Intuitively, this is because because matching the count of 0s with the count of 1s already requires a pushdown automaton to clear its stack, after which it must forget the number n because the stack has been cleared.

We've seen two main ways of showing that languages are not context-free:

1. The pumping lemma for context-free languages.
2. Closure properties. A particularly useful one is intersection with a regular language: to show that A is not context-free, it's enough to show that $A \cap B$ is not context-free, where B is any regular language of our choice.

25.3 Computability theory

In this unit, we actually defined two classes of languages: the decidable languages and the recognizable languages. Separately, we also introduced the notion of computable functions. All three notions have a characterization in terms of Turing machines. We also argued that the definitions of decidable and recognizable languages, and of computable functions, don't change if we switch models from Turing machines to multi-tape Turing machines or if we change the size of the tape alphabet; we also briefly argued that other computational models, such as computer programs and deterministic stack machines (from Watrous's notes) also have the same computational power.

We gave a few equivalent characterizations of recognizable languages:

1. The recognizable languages are those that can be recognized by a Turing machine (meaning there is a machine that accepts exactly the strings in the language, though it may loop on strings not in the language)

2. The recognizable languages are those that are *recursively enumerable*, meaning they can be printed out by an enumerator

3. The (non-empty) recognizable languages are those that can be represented as the range of a computable function.

We also gave two characterizations of decidable languages:

1. The decidable languages are those that can be decided by a Turing machine (meaning that there is a Turing machine that accepts strings in the language and rejects strings not in the language, without ever halting)

2. The decidable languages are those that are both recognizable and co-recognizable (a language is co-recognizable if its complement is recognizable).

One interesting thing we saw about Turing machines is the existence of a *universal* Turing machine, which takes as input the description of another machine and its input, and simulates that machine on that input. In effect, universal Turing machines are interpreters which take in the description of a program and run that program. Their existence is what allows our modern computers to exist: the modern computers are fully programmable, and one can install programs on them as software without attaching additional hardware that runs each new program.

We also showed some closure properties for decidable and recognizable languages:

1. The recognizable languages are closed under union and intersection, though not complement; the decidable languages are closed under union, intersection, and complement

2. The decidable and recognizable languages are closed under concatenation and star

(We did not investigate the closure of the decidable and recognizable languages under reverse, prefix, suffix, and substring operations; feel free to investigate those yourselves.)

The decidable languages represent the problems that can be solved by a computer (with no limit on the running time or memory used by it). The recognizable languages are the languages that may be *printed out* by a computer (via the enumerator characterization), but such an enumeration does not necessarily give a way to decide the language. These are very powerful language classes that include almost all decision problems one might encounter in everyday life. However, not all languages are decidable or even recognizable. We've seen four main ways of showing that a language is not decidable or recognizable:

1. Diagonalization arguments, which start with the assumption that a language is decidable and reach a self-referential paradox. These can show that a language A is not decidable, which also means that either A or $\overline{A}$ is not recognizable (it's usually easy to tell which of the two is not recognizable, simply by showing that the other one is recognizable).

2. Mapping reductions can be used to start with one undecidable or unrecognizable language and show that another language is also undecidable or unrecognizable. We've also seen reduction-style arguments that were not structured like formal mapping reductions.

3. The recursion theorem, which is a powerful tool that essentially lets you diagonalize but with less hassle. The recursion theorem lets you assume a Turing machine has access to its own source code.

4. Rice's theorem, which says that it is not possible to decide whether the language a given Turing machine has some non-trivial property.

We've also seen a few ways to show that a function is not computable. One way was by showing that it grows too fast (like with the busy beaver function). We saw different type of argument when we argued that Kolmogorov complexity is not computable. Additionally, the range of a computable function is recognizable, so one could show that a function is not computable by showing that its range is not recognizable. Finally, one might show that if a function were computable, it would give a mapping reduction from an unrecognizable language to a recognizable one, which is impossible.

25.4 Complexity Theory

In the last unit of the course, we examined resource-bounded classes of languages, which are usually called complexity classes. We introduced quite a few of these classes: P, NP, coNP, PSPACE, EXP, and NEXP. For a list of a few hundred more, check out the Complexity Zoo. We also defined the notion of hardness and completeness: a language A is *hard* for a class if $B \leq_m^p A$ for every language B in that class. We say A is *complete* for a class if it is both hard for that class and also a member of that class. We've primarily looked at NP-hard and NP-complete languages, as those are the most important cases of hardness and completeness.

The most important complexity class for us was the class P of languages which can be decided in polynomial time. This class has several nice properties: first, because different computational models can all simulate each other with small overhead (certainly at most polynomial, and even less in some cases), the class P does not depend on the exact computational model; we get the exact same class whether we define it in terms of Turing machines, stack machines, or some other mathematical formalization of programming languages. Second, the class P seems to be a reasonable mathematical formalization of the notion of "easy" or at least "tractable" computational problems; most of the problems in P turn out to be tractable in practice, and most of the problems not known to be in P turn out to be intractable in practice (there are some exceptions, for example problems in P whose best-known algorithm runs in time n^{100} or something, but they seem to be very rare).

The class P has nice closure properties: it is closed under union, intersection, complement, concatenation, and star, among other operations.

We don't have good ways of showing that a language is *not* in P. It turns out to be very difficult to show that there is no fast algorithm for something; developing methods to do so is one of the main open questions in theoretical computer science. One thing we do know, however, is that more time always gives you more computational power, via the Time Hierarchy Theorem. This means that we know there are languages in EXP which are not in P.

Instead of actually proving that a language A is not in P, what people usually do instead (if they believe A is intractable) is to show that A is NP-hard. This immediately means that A is not in P under the assumption that $\mathsf{P} \neq \mathsf{NP}$.

To show that a language A is NP-hard, all we need to do is to show that $B \leq_m^p A$, where B is a language we already know is NP-hard. We've shown that SAT and 3-SAT are NP-hard (in fact, they are NP-complete, meaning they are both NP-hard and also are in NP). This was originally shown by Cook and Levin in the early 1970s. Since then, there have been thousands of other languages that were shown to be NP-hard or NP-complete, all by a trail of reductions tracing back to SAT or a variant of it. Modernly, to show that a language is NP-hard, you have a variety of choices of which other NP-hard language you want to reduce from. We didn't study such reductions much in this course; if you are interested in learning more, take CS 341 (and perhaps the followup advanced algorithms course, CS 466).

We saw, by Ladner's theorem, that not *every* problem outside of P can be NP-hard (assuming $\mathsf{P} \neq \mathsf{NP}$), so the idea of showing a language is intractable by showing that it is NP-hard cannot work for all intractable languages. However, in practice it works for many of them.

We also saw some classes larger than NP, including PSPACE and EXP. There exist PSPACE-complete and EXP-complete problems as well, and these are considered even harder than the NP-complete problems; however, we did not cover them in this course.

(Interestingly, the PSPACE-complete problems tend to look like games, such as "the set of all strings which encode a winning Go position on an $n \times n$ board"; such languages are usually PSPACE-hard, and if we add a bound on the number of turns in the game – say, by considering the game a draw if more than n^{10} turns were played, where n is the board size – then these languages are typically in PSPACE as well, making them PSPACE-complete).

There were several recurrent tricks regarding these complexity classes. For NP, we saw it has a useful characterization in terms of verifiers (instead of non-deterministic Turing machines). Padding arguments showed up several times, such as for showing that $\mathsf{P} = \mathsf{PSPACE}$ implies that $\mathsf{EXP} = \mathsf{EXPSPACE}$. The fact that a computation which takes only S space can take at most $2^{O(S)}$ time showed up when proving that $\mathsf{PSPACE} \subseteq \mathsf{EXP}$. Finally, Savitch's theorem showed that non-deterministic computations can be simulated by deterministic computations using only quadratically more space (though possibly exponentially more time).

25.5 Final remarks

In this course, we've looked at various ways of mathematically defining the intuitive concept of computation. Such formalizations allowed us to rigorously prove various interesting things about computations, such as the existence of undecidable languages, or the existence of NP-complete problems.

In the first half of the course, we examined weak computational models (corresponding to the regular languages and the context-free languages), and showed that they have many elegant properties. In particular, the equivalence of DFAs and regular expressions was tricky to prove, and may seem surprising at first; for context-free languages, there was a similar equivalence between context-free grammars and PDAs.

In the second half of the course, we looked at stronger computational models; we argued that all realistic but sufficiently-strong computational models are essentially equivalent to a Turing machine (this was called the Church-Turing thesis). We then saw that even this stronger computational model can't solve every task: there are natural problems, such as whether a given program will halt, which are undecidable. In fact, one of the main takeaways from the second part of the course

was that predicting the behavior a given computer program (faster than just running it) is generally not possible. For time-bounded computations, the Time Hierarchy Theorem was one formalization of this intuition. Another formalization is the conjecture that $\mathsf{P} \neq \mathsf{NP}$, which is one of the most important open problems in complexity theory.