Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

W12L68_Summary and Concluding Remarks

Hello and welcome to lecture 61 of the course Theory of Computation. This is also the last lecture of the course. So, we have already completed and summarized space complexity in the previous lecture, and now this lecture is just intended to be a summary of the entire course and some concluding remarks. So, at the beginning of the course, we asked the following question: What are the fundamental limits and capabilities of computers? What is computation? What can computation do? Is there something it cannot do? How much can it do? So these are the things that we wanted to understand. Our goal was to understand if there is something that is not computable or what is computable, what is not computable, etc.

So that was our goal. We started our journey with automata theory. These are simple computation models that recognize a certain class of questions, languages, or problems. We saw regular languages, which were recognized by multiple kinds of devices or structures.

Regular languages are recognized by DFAs, NFAs, regular expressions, right? All of them, we showed that they were equivalent. Then we understood the crucial properties. We tried to understand what is not regular, right? We used the pumping lemma. We learned the Myhill-Nerode theorem, etc., to determine languages that are not regular as well. Then we went to context-free languages, which were supposed to be a superclass of regular languages and were more powerful.

These were recognized by context-free grammars and pushdown automata. So again, there were two equivalent models—two completely different models, but it turned out to be equivalent in power. Then we saw grammars like Chomsky normal form. We saw the concept of ambiguity, where there is a string that is derived in two different ways. We also saw languages that are not context-free using the pumping lemma.

We also saw the CYK algorithm, which was an algorithm that can very efficiently tell us whether a given string was generated from a given grammar if the given grammar is in Chomsky normal form. So, we understood these structures at this automata in yielding the regular and context-free languages, and this was, in its own right, interesting that you have regular languages and context-free languages. But this also served as a stepping stone to understand computability theory and Turing machines. So, Turing machines are this kind of abstract model of modern-day computers, right. So, we initially started out with the

single-tape deterministic Turing machine, then we saw variants like the multi-tape Turing machine, we saw variants like the non-deterministic Turing machine, and we saw that all of them are equivalent.

Then we said that Turing machines are equivalent to algorithms, which is also known as the Church-Turing thesis, right. Then we defined what is decidable, what is not decidable. So, decidable languages are those that have some Turing machine that can decide them, meaning it should be able to take the string as input and should be able to definitively tell whether it is in the language or not. So, it should accept or reject; no looping or infinite loop is permitted. So, we saw languages such as ADFA, ANFA, EDFA, EQDFA; all of them were decidable, like ACFG.

But ATM is the first language that we saw to be undecidable. We saw that ATM is undecidable, right? And in order to show that this is undecidable, we had to do a lot of work. We had to start from first principles; we had to define what countable sets are, what uncountable sets are, and using that, first we had to establish that the number of Turing machines is countable, whereas the number of languages is uncountable, so there have to be languages that are not Turing recognizable. Later, we took up ATM and then, with this complex proof of diagonalization, we showed that ATM is undecidable. And then we learned reductions. Reductions are a way of transforming one problem or one language into another language.

So, if we have a decider for the second language, this will yield a decider for the first language. Reductions can also be used to show undecidability. Because if a hard problem can be reduced to another problem, it follows that the second problem is also hard. Because if the second problem is easy, then the first problem also has to be easy. So, using the fact that ATM is undecidable and using reductions, we saw other languages being undecidable, such as HALT TM and REGULAR TM.

So, HALT TM is like does M halt on a string W. REGULAR TM is does the Turing machine M recognize a regular language. Then we saw Rice's theorem, which was a general theorem about Turing machines, which actually encompasses a lot of languages and shows that all of them are undecidable. Basically, given a Turing machine, any non-trivial question about the language it recognizes is going to be undecidable, right? Then we saw the technique of computation histories, and then we saw PCP (post correspondence problem), which is also undecidable. This is a very simple, easy-to-describe problem that happens to be undecidable, right. So, that completed computability theory and that was kind of some responses or some answers to our quest of what is computation, what is computable, what is not computable, right.

So, the next question, the natural question, is: okay, so we have an understanding of what is computable and what is not computable. But how much investment do we need to

compute something? How many resources do we need? How much time do we need? How much space do we need? So, two most important resources for computation are time and space. On the basis of the time and space required, we started to study the languages and classify them into so-called complexity classes based on how much of each resource is required. In time complexity, we saw complexity classes such as P, NP, we saw what the P versus NP question is, we saw the verifier model for NP, we saw polynomial-time reductions, NP-completeness. We saw that SAT is NP-complete, which is the first language we saw to be NP-complete.

This was using the Cook-Levin theorem. Then we saw other NP-complete problems, many other NP-complete problems. It's subset sum, Hamiltonian path, integer linear programming, and so on. All of them were NP-complete. Then we turned our attention to another resource, space.

So we saw the basic model of space complexity. We saw relations between time and space complexity, NL-completeness, classes L, NL, P-space, then Savitch's theorem, all of that we saw. Then we saw a couple of other results also, NL = co-NL and P-space completeness. And that kind of summarizes what all we have seen in terms of complexity theory. So, based on time and space, we were summarizing languages or problems into complexity classes.

And this also completes the summary of the entire course: first automata theory, then computability theory, and then complexity theory. And so this is what we saw over the entire course. Of course, each one of them—some of them you could actually have spent more time on. But then this course is for a fixed duration.

And there are many, hopefully, we have covered enough breadth that helps you to pursue a specific tangent or direction that you find interesting on your own. Sipser itself has excellent problems and also other chapters that we have not really touched. So we didn't touch chapter 6, then chapters 9 and 10 we did not touch. And of course, there are other books on similar topics where some other topics are also going to be covered. So, if you are interested in this kind of stuff, there are several ideas and directions that could be pursued.

One of the most important directions that I would suggest is to learn complexity theory, computational complexity theory. So, some starting points are there in Sipser itself. You can try to read chapters 9 and 10. So, there are things like hierarchy theorems, time hierarchy, and space hierarchy theorem. So, basically, it says that if you have, let us say, n^3 time and n^2 time, there are languages that need n^3 time cannot be decided in n^2 time.

And similarly for space, there are languages that can be decided in n cube space, but cannot be decided in n square space. Basically, if there are two functions, one of which is faster

growing and one of which is slower growing, there is something that you can do with a faster growing function but cannot with a slower growing function. This kind of result is called hierarchy theorems. Then there are oracles, relativization, and something called the polynomial hierarchy, which we did not cover.

Then there are other models of computation that we have not covered. We just talked about two different things like time and space complexity. There are other models of computation. So, the interesting thing is that these models of computation come with different types of resources, not just time and space. So, there is circuit complexity where we look at circuits, meaning Boolean circuits.

So, it could be and it could have the components of these circuits will be AND gate, OR gate, NOT gate, like Boolean logic gates. And these are used to build functions, which can be used to answer questions, like the same questions we consider, like subset sum or three-set or whatever. We can build circuits to answer them, and we can try to understand these languages in terms of what types of circuits are required. So, you have complexity classes based on what types of circuits can decide them. So, what types, meaning how many gates does it have, what is the depth of it, what is the size of it, and so on. So, there are other resources that come into play.

Another interesting area is randomized algorithms or randomized computation. So, again, in randomized computation, we use, in addition to the input, we also rely on random bits or random coins. So, we want to do something. So, you take the input, and when there are steps where we will just toss a coin, and if the coin says heads, we will do something; if the coin says tails, we will do something else. These kinds of algorithms are called randomized algorithms.

And there is a lot of scope for, there are complexity classes that are specifically designed for randomized computation. So, there are courses in many universities where you just learn randomized computation or randomized algorithms. Another one is counting complexity. So, we talked mostly about decision problems, yes/no problems. There are problems where you can ask to count something, like how many paths does this graph have from a certain vertex to a certain vertex? How many matchings does this graph have, perfect matchings? How many spanning trees does this graph have? So, these kinds of questions where you count something.

So, this is different. Again, there are complexity classes for addressing these types of questions. Another one is interactive proofs where we actually answer questions using interaction between two parties. So, all of them have their own computation model and have their own sets of complexity classes where we classify problems based on the resources needed in these kinds of situations. So, these are all very interesting. And one place where you can find some of these directions kind of addressed is my own, I have a

course on computational complexity, which has been offered some, like August, July, August semesters over NPTEL platform.

But if you're just curious to learn, you don't need to actually wait for the offering. You can just, the videos are available in the public domain; you could just go to the portal or YouTube and watch these videos. So, this is one place where you can learn some of these things. I think most of these things. But there are also courses on computational complexity by even other great professors from many other big universities. So, you could actually try to watch them as well. So, I am just saying there is a lot of material available on the internet for these things.

Available for free as well. So, if you are interested in these kinds of things, some of these topics, you should try to think about or try to read up, try to explore on your own. That will be a lot of fun. Perhaps you will be interested to learn more, and that is my hope too. So, as I said, my hope is that you learn more and get interested in these topics and continue to maintain interest in these topics. So, we will have a lot of people who are experts in these kinds of areas.

So, if you have any feedback, please let me know any comments or suggestions on the lectures or how it was structured or any way it could have been improved. So, in case I teach this course again in the future, I could take this into account. If there is something that you liked or something that you did not like, feel free to let me know. I'll be happy to hear the comments and feedback. I think NPTEL also offers or NPTEL also has this proper feedback system at the end of the course, so please give your feedback in the NPTEL portal as well. My email ID is given here; it should also not be hard or difficult to find. If you just search for my name, Subramanyam Kalyan Sundaram, or search for my name, IIT Hyderabad, or something, you should easily be able to locate my homepage and my email ID. So, if you have any suggestions, feel free to write to me.

And as I said, hope you enjoyed learning and hope you will continue to be interested. I hope it excited you and that you will continue your interest. And maybe, hope at some point we will run into each other in the real world as well. And that's all. So, hope you had fun learning this course and thanks a lot for listening in, tuning in, and that's also big. Perhaps we'll meet in some other course, some other location sometime. Thank you.