Theory of Computation
Professor Subrahmanyam Kalyanasundaaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Time Complexity - Part 1

(Refer Slide Time: 00:17)

Hello and welcome to lecture number 45 of the course Theory of Computation. In the previous lectures, we concluded the topic of computability theory, where we saw what is computability? What are languages? What is decidability? What is Turing recognizability? What are some languages that can be decided and that cannot be decided? What are some languages that cannot even be recognized?

And various techniques to understand this including reductions, we saw all of this. And now, we are entering the third part of the course. So, the first part of the course was automata theory where we saw NFAs, DFAs, PDAs etc. Then the second part was computability theory. And now, we are entering to the third part of the course, which is complexity theory.

So, in the remaining time that we have in this course, which is roughly something between 3 and 4 weeks that we have, we will see a brief introduction to the area of complexity theory. So, what is complexity theory? It is an area of computer science, where we try to understand computational problems in terms of how much resource we need to solve them.

And, this leads to definitions of complexity classes, so, complexity classes are classes of computational problems. So, many computational problems are classified together based on how much of certain resources needed. So, something needs a significant amount of space or significant amount of time, we classify into certain complexity class and if it needs more space, another class and so on.

And there are several models of computation. So, we saw many in our course itself, but there are even more models of computation and many resources are also there in these models. So, even in the models that we saw two of the resources that stand are the time and space, how much memory is required to do a certain computation and how much time is required. There are also other parameters that are studied such as randomness.

There is a circuit model of computation, where we try to study the circuit parameters, how much they are needed, there are also like parallel computation models where we will worry about how much parallelism is there. There are models of computation where there are multiple computers interact in a distributed setting, and the amount of interaction required is considered as a resource.

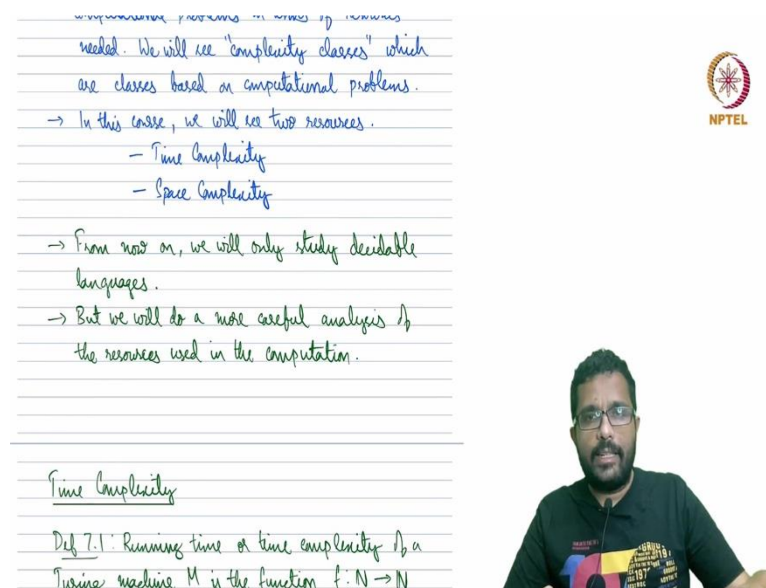
So, what I am saying is that, in general, there are several resources that are considered for computation. And the goal of complexity theory is to understand computational problems in terms of how much of a certain resource needed. And the most common two resources are time and space. So, even when we run programs on our computer, we see how much time it takes sometimes, it takes more time, and sometimes it does not take that much time.

And we also see how much space it takes. So, time and space is something that everybody knows about. And these are two resources that we will consider in this course, time complexity

and space complexity, complexity in terms of how much time is needed and how much space is needed.

What we will do in the remaining time is we will try to cover some of the basic aspects of time complexity as well as space complexity. And hopefully this will also serve as an introduction to learning more on complexity theory. If you get interested, then of course there are further resources that you can look up to beyond this course. So, this is roughly what we are going to see from now.

(Refer Slide Time: 04:36)



And from now on we will only be studying decidable languages. So, we have studied computability theory and we know what is computable or decidable problems and we know how to identify languages that are not decidable. So, now we are not going to be bothered about that.

That is for now, we are only going to be seeing decidable languages and we will in these languages, we will try to see how much time or how much of space we need. Will do a more careful analysis of resources needed. Because so far, what we have bothered about is mostly can this model of computation recognize a certain language or decide a certain language.

Can DFAs recognize this, can NFAs recognize this, can Turing machines recognize this? We never bothered how many steps of computation is needed, how many states are needed or anything like that, we would say two machines are equal and if they recognize the same

language. We did not really bother about how much time it takes, or how much space it takes. From now on, we are going to be more careful about these kinds of things.

Because only then can we say, this takes this much time, and this does not take this much time. So, that is going to be the tone from now on, we will only be studying decidable. So, will not be even bothered to say that this is not decidable, or should this be decidable, we need to convince ourselves of that, nothing like that, we just go and study the languages assuming that they are decidable. And we will look at the time needed, space needed et cetera more carefully.

(Refer Slide Time: 06:27)

→ But we will do a more careful analysis of the resources used in the computation.

Time Complexity

Def 7.1: Running time or time complexity of a Turing machine M is the function $f: N \to N$ where $f(n)$ is the maximum time taken by M to accept/reject an input of length n.

Def 7.2: Landau's O-notation.

If $f, g: N \to R^+$, we say that

$f(n) = O(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$ for some constant c.

So, now without spending much more time, let us delve into the seventh chapter of the book Sipser. This chapter is on time complexity. So, what is time complexity? So, running time or time complexity, both these terms terminologies are used, of a Turing machine, given a Turing machine, let us say which is deciding some language. This running time is given by a function called $f(n)$, where, what is $f(n)$? $f(n)$ is a function of n.

And what is n? n is the length of the input. So, the running time is specified in terms of the length of the input, a longer input may take a longer time to decide. Shorter input may take lesser time to decide. That is the rationale for viewing it as a function on the length of the input. So, running time is the function from N to N where f is the maximum time taken by the Turing machine to decide an input of length n.

So, f is the maximum time taken by the Turing machine to decide on an input of length n. So, different inputs of the same length may take different time, so if that is the case, we take the

maximum, if they are all the same, then the maximum is also the same. So, given an input a certain length, we want to know what the maximum time it takes. And that is our time complexity. So, first, we will be mostly looking at deterministic Turing machines. Later, we will define the same similar things for non-deterministic Turing machines as well.

(Refer Slide Time: 08:18)

Then, let me just also introduce to you the O notation, perhaps you are familiar with this, but just for the sake of completeness, this is called O notation. This is also called Landau's O notation. So, if you are given f and g two functions from integers or natural numbers to real numbers, we say that $f(n)$ is big O of $g(n)$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$, for some constant c.

We say $f(n)$ is little o of $g(n)$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. So, if the limit is at most, some constant, we say it is big O of $g(n)$, and if the limit is 0, we say it is little o of $g(n)$. So, for instance, if $f(n) = 3n^2 + 2n + 5$, then we say that f is $O(n^2)$, because if you divide f by n^2, you get a constant, which is at most 3, as n tends to infinity.

So, f is also little o of n^3. So, it is a smaller o right? So, big O means f by g is upper bounded by a constant. And little o means the limit of f by g as n tends to infinity is 0.

And these two are used to usually upper bound, meaning we want to say f is at most something, then we say f is a big O of something or little o of something. There is also another notation called Ω (omega). This is used to lower bound things. We say $f(n)$ is $\Omega(g(n))$ if, $\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$, for some constant c . So, it is kind of the opposite of O for some constant c.

So, f is Ω $(g(n))$, if f by g is at least some constant. So, in some way, we are saying that f is at least some factor times g. And we say $f(n) = \theta(g(n))$, if $f(n)$ is both big O and Ω of a certain function $g(n)$. So, for example, if you consider, let us say $f(n) = 2n^3 + 5n^4 + 10$. You can see that f is equal to $\theta(n^4)$, you can both upper bound this by n^4 and lower bound is by n^4 so, that I will leave it as an exercise.

So, when the same function acts as the upper bound as well as the lower bound, we say f is $\theta(g(n))$. So, this happens when f is $O(g(n))$ as well as $\Omega(g(n))$. So, f is$\theta(g(n))$, if and only if f is $O(g(n))$ and f is $\Omega(g(n))$. So, both these things are met. So, this is an example, here f is $\theta(n^4)$. So, there is only a constant factor difference between f and g in the asymptotic settings, that is what this means.

Here the term $5n^4$ is the dominating one, when n tends to infinity, this term grows much faster than the other terms $3n^3$ and 10.

So, this dictates how f grows. So, that is why, the way f grows will be very similar to how n^4 grows. So, this is the theta notation. So, this is there in the book. And it is also fairly standard. If you take any book on algorithms or something. So, maybe you can make yourself familiar with these notations, with these symbols.

So, we have big O, little o and omega, and sometimes for analyzing algorithms in terms of this much time, this much space, et cetera. These notations turn out to be helpful because usually

we do not care that much about these coefficients, like the constants that are there, what we mostly care about is the rate of growth. Rate of growth as n changes. So, these things do not really matter so much. The coefficients 5 or 3 or something.

(Refer Slide Time: 14:13)

So, the next definition is $DTIME(t(n))$. $DTIME\big(t(n)\big)$ I think in the book in Sipser, it is defined as $TIME(t(n))$, is the set of all languages that are decided in time order $t(n)$, in big $O(t(n))$. So, $DTIME(t(n))$ is a complexity class. So, this is the first complexity class that we are seeing, it is a set of all languages that can be decided in time t(n) by a deterministic Turing Machine. This is also important by a DTM not an NTM. And where is a time?

We have not yet defined time even here the word time came in. So, when I say time what I mean is, I just maybe highlight here by time I mean the number of computational steps, because we have only been dealing with a Turing machine so, there is no measure or clock that is there. So, it is, like, the time means the number of steps needed. So, we have how many steps are needed to get to the decision that is what I mean by time here.

So, running time means we are measuring it in terms of number of steps. Now, how that number of steps relate to the actual time, itself depends on how the Turing machine is run, but Turing machine being an abstract model, we do not have anything like that. But when we translate this to an actual computer or actual processor, these things can be translated you understand that. You can look at how many computations are performed in a certain second. So, you have machines that run in Giga hertz and so on.

So, based on that you can actually translate into an actual measure of time, some how many microseconds or nanoseconds or whatever. So, when I say time, it means number of steps in terms of Turing machine. So, again, I will highlight that here as well. So, $DTIME(t(n))$ is the class of languages that can be decided by a deterministic Turing machine in time t(n), time order t(n) so, time $O(t(n))$.

Sometimes I end up saying order t(n), when I mean, $O(t(n))$. So, $O(t(n))$, in that much time, what are the class of languages that we can decide. And it is important, the D, the deterministic Turing machine is important. That is why I decided I slightly deviated from the terminology used in the book I am saying DTIME instead of just time. So, that the D here, I am using it to stress that we are dealing only with deterministic Turing machines.

So, Turing machine is deterministic, and could be multitape, I am fine with it being multitape. So, being multitape actually allows the machine to be faster, meaning it can do more computations in a multitape setting than it can do in a single tape setting. However, so depending on the function $t(n)$, the way we define it, is a deterministic Turing machine and it can be multitape.

And this is ok because even though sometimes we look at specific functions here for t(n), t(n) can be n^2 or n^3, usually, what we mostly care is if t(n) is a polynomial or not. So, we care maybe about n^2 versus 2^n , we usually care less about n^2 versus n^3, we care about polynomial versus not polynomial. So, that is why we are going to be assuming that it is deterministic and it could be multitape.

(Refer Slide Time: 18:24)

And finally, we come to or not finally, but we now come to the one of the most important definitions which is that of P. So, the complexity class P is defined to be, the union of $DTIME(n^k)$, where k varies from 1 to infinity.

$$P = \bigcup_{k=1}^{\infty} DTIME(n^k)$$

So, what are the class of problems or languages that can be decided in $DTIME(n)$, $DTIME(n^2)$, $DTIME(n^3)$, $DTIME(n^4)$ and so on. So, this and it is an infinite union that we gather together and call it P. So, the reason for calling it P is that we have n, n^2, n^3, et cetera. So, these are all polynomial functions. So, anything of the form n^k is considered to be polynomial.

That is why this is called P and P stands for polynomial something that can be decided in polynomial time by a deterministic Turing Machine. So, P stands for the class of languages that can be decided by a deterministic Turing machine in polynomial time. And this is one of the most important and most fundamental complexity classes that we will see.

So, they are going to be hearing P much more in the rest of the rest of the time we are going to be discussing time complexity and even outside this course, if you are reading algorithms or some other branch of computing itself, you may encounter this complexity class P. That is because it is a very significant and important complexity class because it stands for the all the efficient and practical algorithms.

So, P stands for all the languages that have efficient and practical algorithms. So, any language if you want to decide, we say it is, it turns out that if you think we can decide it quickly or efficiently, it turns out that it has an algorithm that classifies it to be in P. And the next thing is, it is a robust class in the sense that, if you have a slightly different model of computation maybe without multitape, even then the definition does not change. So, small variations in the model of computation, it does not really change the way things are set up et cetera So, in that sense, P is a fairly robust complexity class. And finally, I want to stress the difference between exponential and polynomial.

(Refer Slide Time: 21:38)

So, terms like this 2^n, 3^n et cetera, these are exponential whereas polynomials are n^2, n^3, n^{10}, et cetera, these are polynomial, thing is that, whenever we have an exponential function such as 2^n, and whenever we have a polynomial function such as n^{10} the exponential function is going to grow much faster.

So, maybe for some small values of n you will see that 2^n is small and n^{10} is large. So, but then once n crosses the threshold, it will look something like this. So, for small values of n maybe n^{10} may grow something like this, but 2^n we will keep low in the beginning and then suddenly shoot up sorry, I almost went backwards.

So, what I want to say is that this may keep low for initial values, but then immediately it will shoot up. So, this 2^n, this is n^{10}, but whenever we have any polynomial growing function and exponentially growing function, the exponentially growing function may grow slowly at the beginning.

But then very soon it shoots up because you can see here I am just multiplying by another n or not even multiplying. So, when n goes from 100 to 101, it just becomes 100^{10} to 101^{10}, but when here it is doubling when n goes from 100 to 101 it is like is multiplying by another factor of 2.

So, exponential functions grew much faster. It does not matter what the base of the exponent is and what the exponent is in the case of polynomial. So, whenever we have something like

2^n versus n^{10}, the 2^n will grow faster whenever we have something 3^n with n^{10}, the 3^n will grow faster. Assuming that the base of the exponent is something greater than 1.

And since exponential grows faster and polynomial is somethings more slowly growing. Polynomial time or polynomial functions are considered to be more manageable. So, if a program or if an algorithm has a running time that is growing polynomially on n so, if the running time is f(n) where let us say $f(n) = n^6 + 5n^2$, this is polynomial and this is considered usually considered an efficient algorithm because n^6 is polynomially growing.

Whereas, if the running time for something was something like $3^n + n^2 + 5n^3$ or something, the dominating term here is 3^n. So, this is exponential. So, in this case, we say it is exponentially growing. So, whenever we have something that is polynomially growing, that is more preferred in terms of running time of an algorithm.

And P indicates the class of all languages that have efficient algorithms by efficient I really mean polynomial algorithms. The running time of that is polynomial.

(Refer Slide Time: 25:41)

So, just a small bit of history. There was this paper that came out in the 60s, I think 1963 I think, called Paths, Trees and Flowers. This is a paper by Jack Edmonds, who is a computer scientist or mathematician. And so the reason I want to like I have kind of pasted a couple of pages from his paper or one page some aspects from the paper. So, this is the title of the paper paths, trees and flowers.

And he was trying to find something called a matching in a graph, so I will not get into what is a matching et cetera, because it is not that complex a definition, but it is not very important to what I am going to say next. So, he was trying to find an algorithm for determining the largest matching or the maximum matching in a graph. So, I have indicated here for the maximum. So, he calls it maximum cardinality matching.

However, so, the thing is that most of the composite theory really, kind of the definitions that are given and things took shape in the 70s. So, in 1963, there was no formal thing, there was no formal theory that stated that polynomial algorithms are better than exponential algorithms. So, there was no clear so, that the complexity class P itself was not defined in 1963.

So, the thing is that he, but then many people had this thought that this is supposed to be efficient, et cetera. So, that is a problem that he faced in the sense that he had an algorithm for a certain problem, which is considered to be significant and important, and he presented that in the paper, now, he wants to sell his result. So, he wants to say this runs very quickly, this is very efficient.

So, now, it is easy. Now, these days, you can say this is an efficient algorithm, this is a polynomial time algorithm, but then there was no notion that polynomial time is better. So, he had to kind of set the stage for that. So, he had to kind of motivate why P polynomial time is better, but at a time that this theory was not formally kind of put forth by anybody, but at the same time, he also did not want to kind of spend too much time so, he just added a small section in this paper, and he called it digression.

So, he says, I have to explain what I mean by efficient algorithm. So, and he says, it is just an algorithm and not a formalized like, it is not a program, he says, the competition details are vital. My purpose is only to show as attractively as I can that there is an efficient algorithm and he goes through the dictionary definition of what is efficient, it is efficient means adequate in operational performance.

This is roughly the meaning I want in the sense that it is conceivable for a maximum matching to have no efficient algorithm. And what he wants to say is that he wants to put forth a result that there is a good algorithm for finding the maximum matching. And he says there is an obvious finite algorithm. So, what does he mean by this? So, what he means is it by obvious finite meaning something that runs in bounded time.

So, the time bounded algorithm that he calls it obvious is the brute force algorithm. So, maybe there is only so many things that you can try, and then you try out all possible options, and then you find what is the maximum matching. But that is usually the brute force approach usually, the running time is quite bad, and it is usually exponential. And that is exactly what he observes also.

But that algorithm increases in difficulty exponentially with the size of the graph. So, he wants to do something better than the brute force or something better than exponential. And he wants to know if there is an algorithm , whose difficulty only increases algebraically with the size of the graph. So, by algebraically, he means polynomially, is just another term.

And he wants to contrast between the exponential and polynomial thing. That is what he is doing here. He says brute force is usually exponential. And what I am presenting is much more efficient than that which is polynomial. Then he says the following, which is just very interesting, and very kind of precise if you want to call it that, the mathematical significance of this paper largely rests on the assumption that the two preceding sentences have mathematical meaning.

So, he wants to say that, obviously, algebraic or polynomial is better than exponential. And assuming that he wants to go to his algorithm, but then unfortunately, the theory was not there. So, he just by putting forth a small section called digression, he just goes on to explain. So, let me just read out the rest. I am not prepared to set up the machinery necessary to give them formal meaning, nor is the present context appropriate for doing this.

But I should explain, I should like to explain the idea a little further informally, it may be that since one is customarily concerned with the existence, convergence, finiteness and so forth, one is not inclined to take seriously the question of the existence of better than finite algorithms. And then I think the section digression does not end here, it goes on further, what he is saying is, usually people just look for a finite algorithms or brute force algorithms, because they care about existence, et cetera.

But this is one of the papers where he actually says this better than the brute force algorithm. So, that is why he had to kind of motivate this part. Unfortunately, for him, the theory was not there, then. And the theory would come only like around 10 years later. So, this was a paper in 60s, which actually kind of shows that people had this thought in 60s, but did not have, because

his result was not about setting up the theory, it was about solving the problem of finding a matching or the maximum matching.

So, he did not want to spend effort putting forth a theory, but it was clear that he had an idea of what he was doing So, this is interesting, because it is a paper that preceded all this, but actually touched upon this notion of what is polynomial, what is exponential, et cetera. So, in that sense, it is perhaps considered to be one of the first papers, which actually even attempt this sort of a discussion.

So, you can search for this paper, and if you are somebody who is in the field of algorithms, et cetera, this is something that you may be reading or you may be interested in reading anyway, even outside the scope of computations theory. So, paper in graph algorithms.