

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

(Refer Slide Time: 00:15)



What is an algorithm?

We saw many models of computation. TM, Variants of TM, Enumerator etc. They are all equivalent in power. They all have **unrestricted access to unlimited memory**.

This is similar to programming languages. What we can compute with C, we can compute with PASCAL, and can compute with LISP etc.

Algorithms = What is decidable. Even though the computational models vary, they are of the same power.

Informally, Algorithm = Step-by-step instruction or "recipe".

In 1900, David Hilbert in ICM listed 23 problems. These are now known as Hilbert's problems. Many of these are important problems.



Hilbert's tenth problem } To devise an algorithm to decide if a given polynomial has integer roots.

Given $P(x) = 7x^6 - 8x^4 + 13x^3 - x + 10$.

Is there an integer solution to $P(x) = 0$?

One approach: Check if $P(0) = 0, P(1) = 0, P(-1) = 0, P(2) = 0, P(-2) = 0, P(3) = 0, P(-3) = 0 \dots$

$6x^6 + 7x^5 + 3x^4 - x^3 - 10 = 0$

roots.

← 2 -1 0 1 2 →

Given $P(x) = 7x^6 - 8x^4 + 13x^3 - x + 10$.

Is there an integer solution to $P(x) = 0$?

One approach: Check if $P(0)=0, P(1)=0, P(-1)=0,$
 $P(2)=0, P(-2)=0$
 $P(3)=0, P(-3)=0 \dots$

$6^2y^2 + 3xy^4 - x^2 = 0$
 $x=9, y=5, z=0$

This is a recognizer, not a decider. But can we convert this into a decider?

For a single variable polynomial $P(x)$, we can get an upper bound for x and convert it into a decider. (Problem 3.21) But not in the multivariate case.

Matijasevich (1970)

Matijasevich (1970)
 (Building on Davis, Putnam and Robinson)

→ No algorithm exists for the 10th Problem.

In other words: Hilbert's 10th is undecidable.

To show this, the basis was provided by Turing's work on computability theory.

Church-Turing thesis:

Intuitive notion of Algorithms = { Turing machine Algorithms }

How do we specify inputs?



Hello and welcome to Lecture number 32 of the course Theory of Computation. So, we have seen Turing machines and we have seen various variants of Turing machines which are multi-tape Turing machines, non-deterministic Turing machines, we saw the enumerator. So, one thing that I said towards the end of the previous lecture was that all of these models have differences, but they share one property. The property is that they all have unrestricted access to unlimited memory.

So, this is something that DFAs for instance did not have, this is something that PDAs did not have, PDAs had unlimited memory in terms of the stack, but there was only restricted access to it. But Turing machines or whatever be the precise model that is used whether it is multi-tape or non-deterministic, as long as we have unrestricted access to the unlimited memory, then

they are all equivalent in power. Equivalent in terms of what are the languages that can be recognized by this Turing machine.

So, in that sense, they are equivalent. So, the amount of time taken or amount of space required may vary from the specifics of the model used. So, perhaps non-deterministic Turing machines can do things a bit faster. But we are not talking about the resources like time and space, we are just talking about what is the class of languages that you can recognize with a certain Turing machine model.

So, the good thing is that the model is robust as long as we have unrestricted access to the unlimited memory, it does not matter what exact detail, what exact details we have for the model. All the models recognize the same class of languages and decide the same class of languages. So, we do not have to specify things like this language is recognized by a single tape Turing machine but cannot be recognized by multi-tape. No, we do not have to get into such things. Because a language that is recognized by single tape Turing machines is also recognized by multi-tape Turing machines. It is also recognized by non-deterministic Turing machines. In a way that is similar to programming languages.

So, whatever you can program with the C language, if you want to do a certain computation task with C language, you can do it in other languages also, like Java or Lisp or other languages too. So, now, the goal of this lecture is to kind of motivate you to what an algorithm is. So, so far, we have only been saying things at a higher level. What we want to refer to as an algorithm is exactly those set of things that are decidable.

And as I already mentioned, the details of the computation model, the details of a Turing machine- single tape, multi-tape deterministic, non-deterministic all that may vary, but it does not matter because all of them capture the same class of languages. So, something is decidable means it is decidable across these different models of Turing machines. So, by an algorithm, I mean, something that is decidable on a Turing Machine model.

So, this is the first step towards formalizing the concept of algorithms. What is an algorithm? We are now formalizing it. So, we know algorithms intuitively. Like you have a step by step set of instructions. Or if you are cooking something, it is like a recipe. So, you start with this, then you do this and then you check how much it has cooked and then you add the other ingredients and so on.

So, similarly, it is also a step by step set of instructions. But more formally, you want to call algorithms as the Turing Machine algorithms. Meaning, like a certain language is decidable, so how that language is decided by a Turing machine. So, algorithms are exactly the class of decidable languages and how the Turing Machine decides this. So, now, before proceeding, a small detour about some history.

So, now, we might be taking this for granted as to what an algorithm is. So, we know what this is, we have this definition. But there was a time when this was not formally captured by any clear definition or anything like that. So, as I have said in the introductory lecture of this course, this formalization happened around the second world war around the '30s, mid to late '30s, by Alan Turing.

But then computation was something that people used to do even from the ancient ages. So, people had to build stuff, people had to transact. Even in the BC periods people had to build stuff, transact, so people used to do computations. But the formalization of what is a computation was only done in the 1930s by Alan Turing. So, one of the historical things that happened was in 1900.

So, there is this four-year conference that still happens, called International Congress of Mathematicians, or International Congress of Mathematics. This happens once in every four years. So, in 2022, there was one. And so, this is a very prestigious meeting, where mathematicians meet and discuss various problems, which are of fundamental nature, and which are relevant to society as well as mathematics.

So, in 1900, David Hilbert, who is a very famous mathematician, went to this meeting, and he listed 23 problems. So, now these problems are called Hilbert's problems because he posed questions about these problems. And many of these problems are now very important, very significant. So, one of his problems what we are going to talk about, is the 10th problem that he listed. It is known as Hilbert's 10th problem.

So, you can Google it, and there is a Wikipedia page just for this problem as well. So, the problem is very simple. You can even ask it to a school going child. So, given a polynomial, you are asking whether the polynomial has any integer roots. So, for instance, here I have a polynomial. The question is, is there an integer x for which this entire thing becomes zero?

$$P(x) = 7x^6 - 8x^4 + 13x^3 - x + 10$$

So, this particular polynomial, I have not verified it. Perhaps there is. So, you can try some simple things. If you set x equal to 0, then it evaluates to 10, so it does not work. For x equal to 1, $7 - 8 + 13 - 1 + 10 = 21$. So, 0 is not a root, 1 is not a root. I am not sure if there is an integer root, but you can try playing around. This is the question, given a polynomial does it have an integer root?

So just to give another example, there is one other polynomial that I have written here in the red box, $6x^3yz^2 + 3xy^2 - x^3 - 10$. So, here, one possible set of integers solution for this, I have checked is $x = 5$, $y = 3$ and $z = 0$.

This evaluates the polynomial to 0 because the first term $6x^3yz^2$ becomes 0 because z is 0. $3xy^2$ becomes $3 \times 5 \times 9$, which is $15 \times 9 = 135$, $-x^3$ which is 125. So, $135 - 125$ is 10, minus 10 which is 0. So, this is an integer solution to this polynomial. So, this polynomial is over three variables, whereas the earlier polynomial $P(x)$ was over one variable.

But the question can be asked for single variable polynomials, multiple variable polynomials, and so on. So, one possible approach, let us say for single variable polynomials is like we checked already. If you are coming back to this $P(x)$, we checked already, 0 and 1. So, you check 0, we saw it is not a root, we saw 1 is not a root, -1 can be checked.

$$P(-1) = 7(-1)^6 - 8(-1)^4 + 13(-1)^3 - (-1) + 10$$

$$P(-1) = 7 - 8 - 13 + 1 + 10 = -3$$

So, $7x^6$ is 7 and $-8x^4$ is -8 , so -1 . Plus $13x^3$ that will be -13 , so you get -14 . Plus $-x$, which is 1, so it becomes -13 . Plus 10 gives -3 . So even this is not a root. So, we have verified that 0, 1 and -1 are not root. And then you check 2, then you check -2 , then you check 3, -3 and so on, so you keep checking.

So, you check whether 0 is a root, 1 is a root, -1 is a root, 2 is a root, -2 is a root. So, like that, basically what we are doing is you take the real line, you check 0, you check 1, check -1, then you check 2, then you check -2 and so on. So, basically, you are covering all the integers in the line by going back and forth 3, -3, 4, -4 and so on.

So, all the things are covered. So, now, you may be able to figure out that this is an algorithm or this is an approach. I do not want to say algorithm. Suppose there is some integer solution. Suppose for instance 50 is a root. So, then when we do 1, -1, 2, -2, 3, -3, up to when you reach 50, it says it is a root and then we can say yes, 50 is a root, there is an integer solution.

But if there are no integer solutions at all, this process never ends. You check 1 is root, 2 is a root, -2 is a root. All that you go back and forth and you never end. You are just cycling through all these infinite integers and you will never stop. You will stop only if there is a root. Therefore, this is a recognizer and not a decider. So, if the answer is yes, if there is an integer root, it will stop and say this is the integer root.

If there is no integer root, it will not accept. It will not say yes. If the answer is no, it will not say yes. But it will not say no also. There is no way for it to stop, because this continuous computation. In that sense, it is not a decider. A decider has to always halt. If it has an integer solution, we want to say yes. If it does not have an integer solution, we want it to say no and stop the computation. But this does not stop. So, it continues computation.

And it never says yes, but it never says no, also. Now, can we convert this to a decider? Maybe I will just erase this. But this approach, can we convert it to a decider? That is a question. Can we convert this to a decider? Let us come to a single variable. If there was some bound. Let us say you only have to check if there is a root, given it is going to be less than 100. Let us say somebody tells you.

Then that is a decider because we check 0, 1, -1, 2, -2 and so on and up to 100, -100. Let us say then, if there is a root it is going to be of size or absolute value less than 100. So up to 100, -100 we do and then if you do not find a root, we say that this does not have an integer root. So, if we have a bound, then that bound can tell us this is the place where you can stop, you do not have to check the infinite possibilities.

So, if there is a bound, that is a way to convert it to a decider. So, this works for a single variable polynomial, which is what $P(x)$ is above. For a single variable polynomial, we can get a bound. And this bound can help us make it into a decider. So, in fact, this is a problem in the textbook 3.21. So, basically, it asks you to find a bound for a given polynomial.

So, given a polynomial, which is a single variable, or a univariate polynomial, it asks to find a bound. So, it asks to say that if x is a root, then x is at most something or the absolute value of x is at most something. However, this bound only works for the single variable case, and for the multivariable case, there is no such bound. So, because there are multiple variables involved, how do you bound it?

Or maybe single bound works for all. No, but we do not have the same bound. In fact, it was shown by Matiyasevich in 1970. So, building on the works of others by Davis, Putnam,

Robinson and so on, the people worked on this problem before. So, but the final piece of the puzzle was provided by Matiyasevich, and sometimes it is attributed to all four of them Matiyasevich, Robinson, Davis and Putnam that there is no decider for this problem in the general case.

General is multivariable case there is no decider meaning you cannot write an algorithm for this question. So, this is a recognizer, a recognizer is not an algorithm. You always want to halt. A recognizer does not halt or may not halt. But what we know is that there is no algorithm that exists for the 10th problem. Or in other words, the Hilbert's 10th problem is undecidable in general. Meaning there is no algorithm for it.

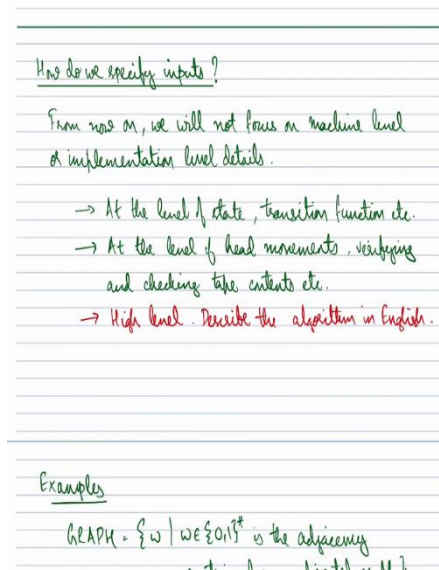
So, this approach that I highlighted above, this is a recognizer not a decider. What Matiyasevich showed building on the work of others is that Hilbert's 10th problem is undecidable. So, this was a big deal like something was asked in 1900 and finally in 1970, it was shown. So Hilbert's question was posed like is there a procedure by which we can find this, but the theory of computability had to be built in order to answer this.

And this theory was built by Alan Turing in the 1930s. There was also a parallel body of work by another mathematician called Alonzo Church. So, together, not together, but independently they build two separate notions, which actually kind of coincide. So, therefore, without getting into all the details, the point is this. What they say is that the intuitive notion of algorithms is like a step by step approach, and we want a solution, we want it to halt etcetera.

This is identical to the Turing Machine deciders. So, the intuitive notion of algorithm is the same as Turing Machine decider. So, if you want to formally show something has an algorithm or does not have an algorithm you have to show it is undecidable. So, this is what we mean by Church-Turing thesis. So, it was posed by two mathematicians, Alonzo church and Alan Turing.

So, it says that the intuitive notion of algorithms is the same as Turing Machine algorithms. So, this history was to kind of motivate this Church-Turing thesis. Such a fundamental problem. To answer this, the theory of computability and the Church-Turing thesis had to be formulated. So, what we said is what an algorithm is and we talked about Hilbert's 10th problem in order to motivate that.

(Refer Slide Time: 17:19)



The next thing is about some details. Now that we have come to an understanding of what exactly constitutes an algorithm. So, it is what a Turing Machine can decide. So, one question is how do we decide? Or how do we give something as an input? So, sometimes the problems are itself not clear. Like let us say you have a graph, now, how do we give the graph as an input?

Or suppose sometimes you can ask questions like let us say I have a Turing machine. And let us say I have a PDA. I want to ask whether this Turing machine and this PDA recognize the same language. So, how do we specify the Turing machine as an input and the PDA as an input? So, these kinds of things like how to specify inputs. I want to just talk about that and some more details.

So, one point is that so far, when we talk about Turing machines, we could go into various depths of detail. So, the most detailed is that I can talk about the transition functions δ , states q_i , individual transitions, when it goes one step right and writes a symbol B and so on or one step left and so on. So, this is at the level of state and transition function, we will not be doing this level. The other one step higher level is about head movements.

So, you scan the input, go to the rightmost end, and then you put a symbol there, then you come back. That kind of detail where you specify what the heads do and how you plan to change the tape content etcetera, or how you plan to verify the tape contents. Even this level we are not going to be talking about from now on. From now on our discussion is going to be at a high level where we describe the algorithm in just English. So, we will say that you take the input and then you check something.

(Refer Slide Time: 19:16)

Examples

GRAPH = $\{w \mid w \in \{0,1\}^*$ is the adjacency matrix of an undirected graph $\}$
= $\{w \mid |w|$ is a perfect square and $w_{ij} = w_{ji}\}$

CONNECTED-GRAPH = $\{w \mid w \in \{0,1\}^*$ is the adj. matrix of a connected undirected graph $\}$

For this, we can use BFS or DFS. We reject if the input is not in proper form or if it is not connected.

PRIMES = $\{w \mid w$ is a prime number $\}$

CONNECTED-GRAPH = $\{w \mid w$ is the adj. matrix of a connected undirected graph $\}$

For this, we can use BFS or DFS. We reject if the input is not in proper form or if it is not connected.

PRIMES = $\{w \mid w$ is a prime number $\}$

We can also ask questions about DFA/PDA/TM's etc. The entire description of the DFA or TM has to be encoded, and given as input.

One way to encode:

$\hookrightarrow 0$ $q_{acc} = 0$

From now on, we will not focus on machine level or implementation level details.

x \rightarrow At the level of state, transition function etc.

x \rightarrow At the level of head movements, verifying and checking tape contents etc.

✓ \rightarrow High level. Describe the algorithm in English.

Examples

GRAPH = $\{w \mid w \in \{0,1\}^*$ is the adjacency matrix of an undirected graph $\}$
= $\{w \mid |w|$ is a perfect square and $w_{ij} = w_{ji}\}$



So, for instance, given a graph. How do we specify a graph as an input. So, you may have seen in data structures, and even in one of our previous lectures, we talked about this way to specify a graph called the adjacency matrix. So, given an input string, we want to verify it is a proper adjacency matrix. So, it is simple, we can just check whether it is a square so the number of entries is a perfect square.

So, the matrix will be entered as let us say all the entries of the matrix. So, if it is an $n \times n$ matrix, all the n^2 entries will be provided serially. So, the number of entries is a perfect square and you also have to check that there is symmetry of the matrix. So, the adjacency matrix of an undirected graph is going to be symmetric. This also I think I mentioned two lectures back.

So, you also have to check whether the ij^{th} entry is equal to the ji^{th} entry. So, this is how you would verify whether a given input string corresponds to an adjacency matrix of a graph. So, this is how we will deal with things. So, what we are going to do is to not talk in the level of the machine or at the level of like head movements etcetera. We are only going to be talking about high level unless we have to go into the machine level detail.

So, for instance, another thing that you can ask is given an input string, which corresponds to a graph. Does this correspond to a connected graph? Is this graph connected? So, we know graph algorithms from the algorithms course. We can do traversals like breadth first search or depth first search. So, of course, we can only do it if the thing that is given to us is actually corresponding to a graph.

So, we first check whether the input corresponds to a graph. If it is not even of the proper form, if it is not a 0 1 adjacency matrix then we immediately reject it. Let us say if it is a 0 1 string, but it does not correspond to an adjacency matrix then also we reject it. If it corresponds to a graph now, we know what graph it is and then we can execute the breadth first search or depth first search.

And we accept if it is connected, reject if it is not connected. Another problem like given an integer, is it a prime number. So, these are all questions that we can ask. And like I said before, we can also ask questions like given a DFA does it satisfy some property, given a Turing machine does it satisfy some property. So, we will have to encode the given DFA. We will have to provide the DFA as an input or provide the Turing machine as an input.

(Refer Slide Time: 22:16)

etc. The entire description of the DFA or TM has to be encoded, and given as input.

One way to encode:

$L = 0$	$q_{ini} = 0$
$a_1 = 00$	$q_{maj} = 00$
$a_2 = 000$	$q_0 = 000$
$a_3 = 0000$	$q_1 = 0000$

And use 1's as delimiters $L=0$
 $R=00$
 $S=000$

$\delta(q_i, a_j) = (q_p, a_s, R)$
 $10^{i+3} 10^{j+1} 10^{i+3} 10^{s+1} 1001$

In this manner, we can encode the whole TM.

$10^{i+3} 10^{j+1} 10^{i+3} 10^{s+1} 1001$

In this manner, we can encode the whole TM, using 0's and 1's. Similarly, we can encode DFA's or PDA's, and then ask questions about them.

- Given TM M : Does M accept only palindromes?
- Given TM M : Does M accept any palindromes?
- Given TM M and w : Does M accept w ?

→ $A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$

→ Given TM M and w : Does M accept w ?
 ATM ↗

→ $A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$

→ $A_{NFA} = \{ \langle A \rangle \mid A \text{ is a NFA that accepts } w \}$

→ $E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA, } L(A) = \emptyset \}$

→ $EQ_{DFA} = \{ \langle A, B \rangle \mid A, B \text{ are DFA's, and } L(A) = L(B) \}$

→ $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$



So, how do we encode? You could encode in many ways, but one way is you can have this encoding with zeros for the input symbols. So, this corresponds to the encoding of the input symbol. So, blank is given the encoding 0. First symbol, let us say a_1 , is given the encoding 00, a_2 is given 000, a_3 is given 0000 and so on. So, how many of our input symbols are there we could have that.

And similarly, accept state q_{acc} is given 0, reject state q_{rej} is given 00, start state q_s is given 000, then q_1 is given 0000, q_2 is given 00000 and so on. And so, zeros are used to encode symbols as well as states. Now, if you want to encode a transition function, let us say like this, $\delta(q_i, a_j) = (q_p, a_s, R)$. Meaning when you read a_j from the tape.

And when you are in state q_i , then you write the symbol a_s onto the tape. You go to state q_p and then move one step right. So, this one we can encode it like this. So, q_i is denoted by 0^{i+3} , $i + 3$ zeros because if you see q_1 has four zeros, so q_2 will have five zeros and q_i will have $i + 3$ zeros. a_j is denoted by 0^{j+1} . So, a_1 has two zeros, a_2 has three zeros, so a_j will have $j + 1$ zeros.

And then q_p will have $p + 3$ zeros, a_s will have $s + 1$ zeros. And I am using 1's as delimiters. So, I am explaining how to encode this transition function. So, these 1's are delimiters. And finally, we have the right movement. Right is denoted by two zeros. So, I have denoted left by 0 right by 00 etcetera. So, this entire string denotes this rule that is highlighted, the transition function of that Turing machine.

$$\delta(q_i, a_j) = (q_p, a_s, R)$$

$$| 0^{i+3} | 0^{j+1} | 0^{p+3} | 0^{s+1} | 00 |$$

So, like that, this is just one snippet, one part of the Turing machine. So, the entire Turing machine set of rules can be encoded in this manner. So, the point here is to convey that this can be done, not really get into all the details, but this can be done and we can do it. So, from now on, we will just assume it can be done. Similarly, we can do so even for deterministic finite automata or pushdown automata. We can encode in the similar way and we can ask questions.

So, I can ask like given a Turing Machine M, so I can give it as input, I can ask does this Turing Machine accept only palindromes. So, it is a yes, no question. Does the Turing machine only accept palindromes? Or I can ask, is there any palindrome that is accepted by this Turing

machine? Or I can give a Turing Machine M and string w . Is this string w accepted by this Turing machine?

I can ask these questions. I could ask questions of DFAs, NFAs etcetera. So, some languages have some names. We will see more of them in the upcoming lectures in the upcoming weeks. So, A_{DFA} is a language where you are given a DFA and a string and you are asking whether this DFA accepts the string. So, it is the set of all pairs $\langle B, w \rangle$, where B is a DFA that accepts w . So, this is called A_{DFA} .

$$A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$$

$$A_{NFA} = \{ \langle N, w \rangle \mid N \text{ is a NFA that accepts } w \}$$

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM that accepts } w \}$$

In fact, the language that I described earlier here. Given a Turing Machine M , does M accept w this is denoted as A_{TM} . It is the acceptance problem of a Turing machine. So, this A_{DFA} is acceptance problem of a DFA. Similarly, we can ask A_{NFA} , does a given NFA accept a given string. I could ask E_{DFA} , it is like the emptiness problem. E_{DFA} , given a DFA is the language accepted by this DFA the empty language.

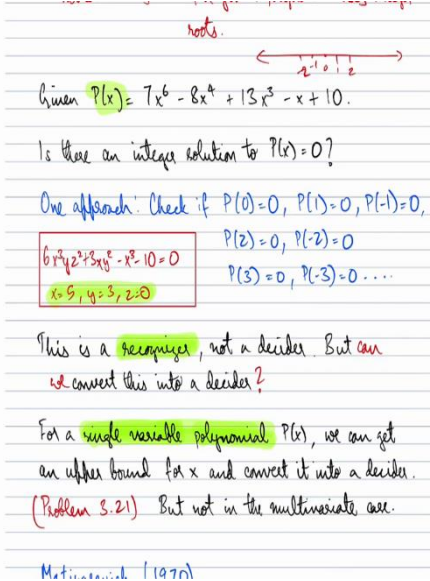
$$E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA, } L(A) = \phi \}$$

$$EQ_{DFA} = \{ \langle A, B \rangle \mid A, B \text{ are DFAs, } L(A) = L(B) \}$$

I could ask EQ_{DFA} the equivalence problem, given two DFAs are they both equivalent. Are the language recognized by them the same. So, EQ_{DFA} . I could ask the acceptance problem for a context free grammar. Given a grammar G and a string w is a string w generated by this grammar. So, I can ask all these kinds of questions. So, the point that I want to say is that for almost any question we can encode and post it to the Turing machine.

And a Turing machine can do some processing or some algorithm, and if such an algorithm is possible, it may be able to provide such an answer. So, in computability theory, we are going to be asking questions of this type, can we, so we started by asking this Hilbert's 10th problem.

(Refer Slide Time: 27:47)



roots.

Given $P(x) = 7x^6 - 8x^4 + 13x^3 - x + 10$.

Is there an integer solution to $P(x) = 0$?



One approach: Check if $P(0) = 0, P(1) = 0, P(-1) = 0,$
 $P(2) = 0, P(-2) = 0$
 $P(3) = 0, P(-3) = 0 \dots$

$6x^2 + 3x + 5x^4 - x^2 - 10 = 0$
 $x=5, y=3, z=0$

This is a recognizer, not a decider. But can we convert this into a decider?

For a single variable polynomial $P(x)$, we can get an upper bound for x and convert it into a decider. (Problem 3.21) But not in the multivariate case.

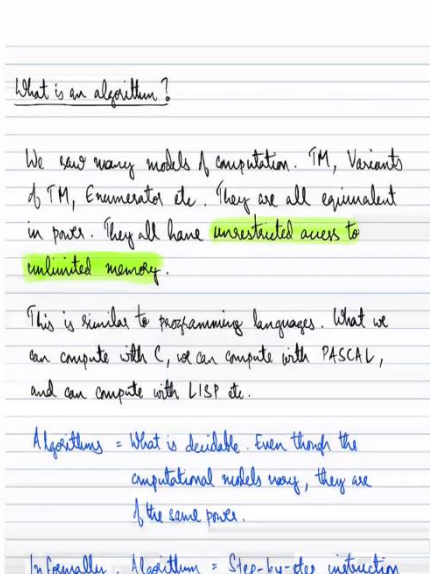
Mathematics (1970)



So, given a polynomial, can we tell whether it has an integer root or integer solution? And then we learnt that this is an undecidable problem meaning there is no way you can write an algorithm for it. There is no way you can build a decider for it. So, now in the upcoming lectures, we will ask questions like this. Like is A_{DFA} a decidable problem.

Given a DFA and a string can we tell whether this string is accepted by this DFA. Given a Turing machine and a string, does this Turing Machine accept the string. Is this A_{TM} , a decidable problem. So, we are now going to get into decidability undecidability theory. So, this was a precursor to that.

(Refer Slide Time: 28:34)





What is an algorithm?

We saw many models of computation. TM, Variants of TM, Enumerator etc. They are all equivalent in power. They all have unrestricted access to unlimited memory.

This is similar to programming languages. What we can compute with C, we can compute with PASCAL, and we can compute with LISP etc.

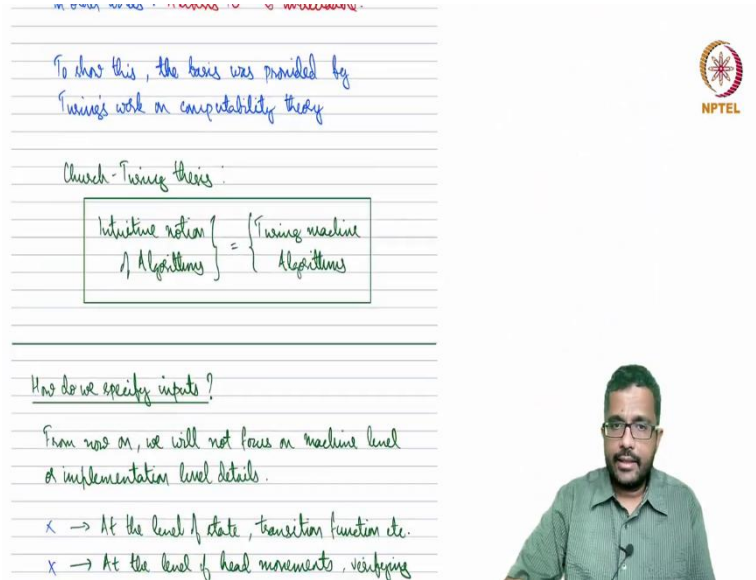
Algorithms = What is decidable. Even though the computational models vary, they are of the same power.

Informally, Algorithm = Step-by-step instruction



So, in order to understand that we first formalize the notion of algorithms, which we said is going to be deciders.

(Refer Slide Time: 28:40)



To show this, the basis was provided by Turing's work on computability theory.


Church-Turing thesis:


$$\left. \begin{array}{l} \text{Intuitive notion} \\ \text{of Algorithms} \end{array} \right\} = \left\{ \begin{array}{l} \text{Turing machine} \\ \text{Algorithms} \end{array} \right.$$

How do we specify inputs?

From now on, we will not focus on machine level or implementation level details.

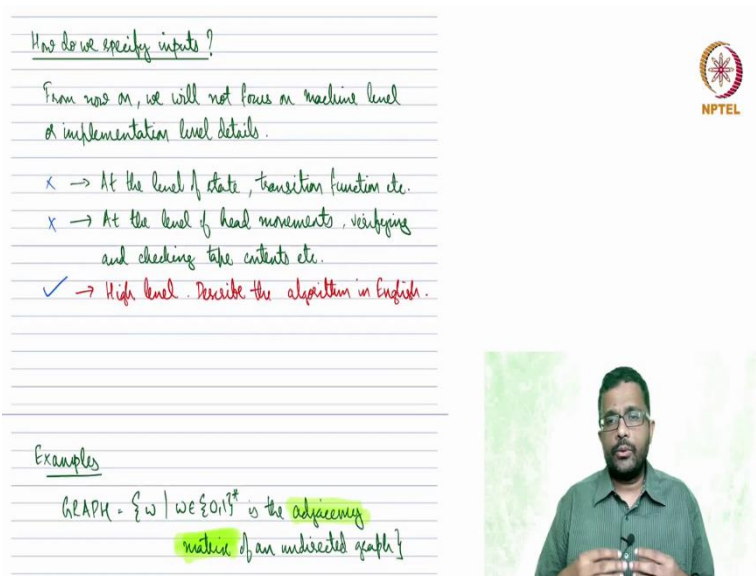
- x → At the level of state, transition function etc.
- x → At the level of head movements, verifying





And we stated the Church-Turing thesis. Now, we are explaining how to and how almost anything can be specified as an input to a Turing machine.

(Refer Slide Time: 28:52)




How do we specify inputs?


From now on, we will not focus on machine level or implementation level details.

- x → At the level of state, transition function etc.
- x → At the level of head movements, verifying and checking tape contents etc.
- ✓ → High level. Describe the algorithm in English.

Examples

GRAPH = $\{ \omega \mid \omega \in \{0,1\}^* \}$ is the adjacency matrix of an undirected graph.





→ Given TM M : Does M accept any palindromes?
 → Given TM M and w : Does M accept w ?
 ATM ↗
 → $A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$
 → $A_{NFA} = \{ \langle B, w \rangle \mid B \text{ is a NFA that accepts } w \}$
 → $E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA, } L(A) = \emptyset \}$
 → $EQ_{DFA} = \{ \langle A, B \rangle \mid A, B \text{ are DFA's, and } L(A) = L(B) \}$
 → $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that accepts } w \}$



And in the next upcoming lectures, we are going to focus on these languages like these. We will see whether they are and will try to understand whether they are decidable, whether they are recognizable, etcetera. So, we will see that there are languages that are not decidable, there are languages not even recognizable. So, surprising, as it may seem there are such languages too. But with this, we conclude week 6. This also concludes Chapter 3 in the textbook.

So, in week 6, we saw variants of Turing machines. We saw the Church-Turing thesis, which is in this lecture, and we set the stage for the decidability theory that is going to come in the remaining weeks. So, that is all I have for you in Lecture number 32 and in week 6. So, see you in week 7, thanks.