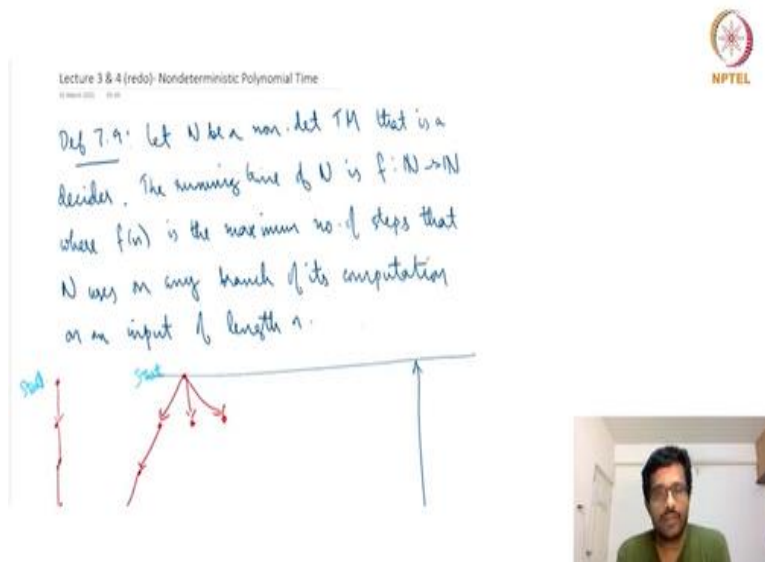


Computational Complexity
Prof. Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Lecture -03
The Class NP



(Refer Slide Time: 00:17)



Lecture 3 & 4 (redo) - Nondeterministic Polynomial Time

Def 7.9: Let N be a non-det TM that is a decider. The running time of N is $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum no. of steps that N uses on any branch of its computation on an input of length n .

The diagram illustrates the computation paths of a non-deterministic Turing machine. It shows a horizontal line representing the input string. From the left end, a vertical red arrow labeled 'Start' points down. From the point where this arrow meets the horizontal line, a red path branches out downwards and then upwards, ending at a point on the horizontal line. This path is labeled 'End'. To the right of this, another red path branches out downwards and then upwards, ending at a point on the horizontal line. This path is also labeled 'End'. A blue vertical arrow labeled 'End' points up from the right end of the horizontal line.



Hello, welcome to lecture 3 of the courses. Over the last two lectures, we saw an introduction to the course and introduction of Turing machines and then the class P which stands for polynomial time deterministic polynomial time. And today we are going to see what is non-determinism or non-deterministic polynomial time. So, let us see what non-deterministic polynomial time is?

So, I believe you would have seen the non-determinism in various places like be it in DFA's verses NFA's, deterministic finite automata non-deterministic finite automata or the pushdown automata that you would have studied most likely use non-determinism. And even by learning Turing machines in the computability theory you would have come across non-deterministic Turing machines.

But nevertheless, let me just quickly revise what are non deterministic Turing machines and how they how they function. In non-deterministic Turing machine, the key thing is that given a certain configuration. So, configuration is something that we defined in the first lecture or first second

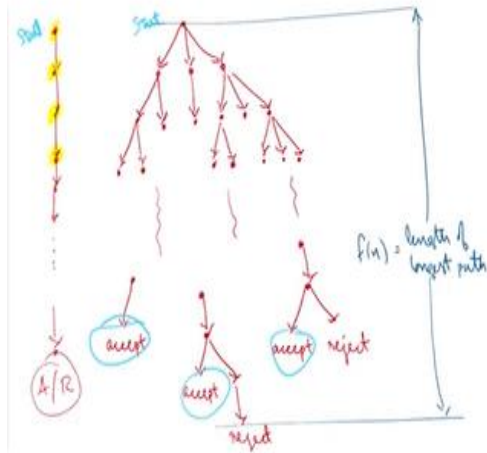
lecture where it is a snapshot of the current Turing machine. It consists of the state, the tape contents and the head position, head positions if there are more tapes.

So, if I tell you, I am stopping the computation like this, and this is the snapshot and then you could later resume the computation from that point by starting the Turing machine from such a position. So, given a certain configuration, a deterministic Turing machine has the next step is predetermined or pre decided by the transition function. The transition function for any configuration it tells you now, you are at this state with this tape content.

And the tapes are at here then it will say no go to the next state q_{10} then go to the right five on the tape and then go on the tape head let it move right something like this. Whereas, in non-deterministic Turing machine, there could be more than one choice, there could be multiple choices, there could be two, there could be three, there could be 10 choices there could also be zero choices. And it is also possible to in some cases there is exactly one choice available.

So, at a certain configuration in the case of non-deterministic Turing machines, there could be multiple successor configurations. Whereas in the case of deterministic Turing machine, the number of successor configurations is going to be exactly 1 whereas in the case of non-deterministic Turing machines, you may have 0, 1 or more than 1. So, let us so just to give you an idea.

(Refer Slide Time: 03:13)



See by this picture, I am depicting the configuration graph. So, each dot over here is a configuration. So, the starting configuration is unique. And suppose there are three successor configurations to the starting configuration. So, I have depicted like I have depicted here, and maybe the first successor configuration has maybe two other further successors, and maybe second one has one successor and the third one has three further successors.

So, what I am saying is that now, just after the starting configuration, there could be two possible, three possible successors. And for each of these successor configurations, there are different numbers of successor possible. Now again this could keep going on. So, now perhaps this one had two successors and this one had no successors let us say, that is also possible. This one had let us say three successors like that it can continue and so on.

So, this is the interesting part, this is what is interesting about a non-deterministic Turing machine. Each stage you could have one or zero or more than one successor to the current configuration. So, which means the Turing machine computation need not be a fixed path, like whereas in the deterministic case, it is like the picture on the left side here. You start with a certain configuration.

Let us say this, and then the next one once you start from a certain place the next one is predetermined, which is this and then this and so on till the machine finally decides to accept or reject. This is a case in a deterministic Turing machine. In the case of non-deterministic Turing

machine there are multiple such possibilities. And finally, what is interesting is that many of them the outcomes could differ some of them could go to accept some of them could go to reject.

And you will see examples, more details of why this or how this can happen. And when do we say that a non-deterministic Turing machines but this kind of a confusing outcome, but how do you make sense out of this. So, we say that a non-deterministic Turing machine on a certain input, let us say x accepts that input x . If there is at least one way to reach accept, so in this case again I have not drawn the entire configuration graph because the configuration graph could be too big.

It could have multiple branches of it, anyway it is a hypothetical thing. And the way I have drawn here there are some that say accept and then there are some that say reject and the remaining ones we are not drawn so, that could be anything. But the point is that there is at least one way to start from the starting configuration for the non-deterministic Turing machine on the input x and go to an accepting configuration like the rightmost one here or this one second one here and three ones that are depicted.

So, the non-deterministic Turing machine will accept the input x if the configuration graph looks like something like this over here in the right this big graph. Now what is the running time of the non-deterministic Turing machine? The running time is the maximum length of any of these paths. Length is the number of steps it takes to go from the starting configuration to a halting configuration, the halting configuration could be accept or reject. So, the length of the longest path is the running time. So, it is I am depicting it as this length.

So, where we can be length means the number of steps. And once again, we will for this course, we will assume that every machine that we have is a decider. So, everything will halt there will be no looping. So, in theory of computation when studying undecidability etcetera. you would have seen this looping, but we will for simplicity and for complexity theory, we will assume that all the machines halt.

For non-deterministic Turing machines, even all the different branches halt on all the steps. So, everything halts to know, which is the one that takes the longest to halt? That is the running time.

So, out of all these possibilities, f_n is the length of the longest path. And where they are the function f_n is the maximum number of steps that the machine uses. So, capital N is a non-deterministic Turing machine or any branch of its computation.

So, here I said the input is x , but f_n is an upper bound on all inputs of the length n . So, x could be just one input. So, we could have many inputs of the same length. So, 0 0 1 0, 1 1 0 0 they are all four-bit inputs. But then what I am saying is that running time is the maximum over all the computation paths over all the inputs of a certain length that length is n . So, this is the running time.

In the case of deterministic Turing machine, the running time was simply the maximum over all the inputs of a certain length. But here even within an input there could be multiple branches. So, we need to take those into account

(Refer Slide Time: 09:17)

A/R (Accept/Reject) diagram showing a state transition with 'Accept' and 'Reject' paths.

Def 7.21: $\text{NTIME}(t(n)) = \{L \mid L \text{ is decided by an NTM in } O(t(n)) \text{ time}\}$

Def 7.22: $\text{NP} = \bigcup_{L \in \text{NTIME}(n^k)}$
 This is equivalent to the guess and verify model.

SUBSET-SUM: $\{ \langle S, t \rangle \mid S = \{x_1, x_2, \dots, x_n\} \}$

NPTEL logo and a small video inset of a person speaking.

So, whereas in the case of deterministic machine, it is very simple, just one after the other. So, there is only one path, and it is very simple. So, the deterministic Turing machine is depicted in the left-hand side over here. This big thing is non-deterministic Turing machine. So, now just like we define $\text{DTIME}(t(n))$, we can define $\text{NTIME}(t(n))$. So, what is $\text{NTIME}(t(n))$? $\text{NTIME}(t(n))$ is a complexity class that consists of all the languages that can be decided by a non-deterministic Turing machine in order $t(n)$ time.

This is exactly that $DTIME(t(n))$, but just that now we allow the Turing machine to have non-determinism. And exactly like P we define NP to be the union of all the $NTIME(n^k)$, where k ranges from one to infinity. So, in the case of P it was the union of $DTIME(n^k)$. So, it is this class of all languages that have non-deterministic polynomial time algorithm. So, whereas P was a class of all languages that have a deterministic polynomial time algorithm.

So, we will come back to this sentence whatever said here, this is equivalent to the guess and verify model. So, we will go ahead for now.

(Refer Slide Time: 10:51)

The slide features a decision tree for the Subset Sum problem. The root node is labeled 't=47'. The tree branches into 'Yes' and 'No' paths. The 'Yes' path leads to a leaf node labeled 'Accept'. The 'No' path leads to a leaf node labeled 'Reject'. The tree is annotated with binary strings: '0000' for the 'Yes' path and '1111' for the 'No' path. The NPTEL logo is in the top right corner.

Handwritten notes on the slide:

- $t=47$
- $t=12$ $t=15$
- Example: SUBSET-SUM NP.
- On input $\langle S, t \rangle$:
- 1. Non-deterministically select/reject each of x_1, x_2, \dots, x_n .
- 2. Add the selected x_i 's and verify if they add up to t .
- 3. If $sum = t$ then accept. Else, reject.

A small video inset in the bottom right shows a man with glasses and a green shirt speaking.

What am I going to do? So, when again now, we may see you may think that there is some abstract sort of thing which is not very clear. But we will soon see that it makes it is not so difficult to imagine and it is something that you can understand. So, consider this problem called subsets sum. So, what is subsets, sum? So, you are given a set S and a number t, a target some t and you are asking whether there is a subset of S such that the subset sums to t.

So, maybe I will just use the subset sums to t. So, that is the question. So, for example, so, you could have the set to be 2, 6, 22, 42, 15 and 10. So, this could be a set and the target sum could be 47. Now, this is a yes instance. So, you would decide yes, if there is a way to if there is a subset of S that sums to 47. Is there such a subset? looks like yes. So, $22 + 15 + 10$ will add up to 47.

So, now maybe you want to take different subset maybe different targets some maybe you could I ask t equal to 12. So, for t equal to 47 in this set, the answer is yes. For t equal 12, what is the answer? Still the answer is yes, because 2 and 10 add up to 12 there is another subset. What about t equal to 13? There is no subset of the set that adds up to 13. So, that will be a no instance. So, now the problem is given S and t you have to decide if S has a subset that the subset t .

This is the claim is it, this is a NP problem, this problem is in NP. Let us see how that is an NP. So, what you can do is to you can come up with an algorithm. How can you come up with an algorithm? Basically, we use the non-determinism to choose a subset and then we verify that the subset sums to t . So, let us see so, what you can do is non-deterministically so, you should look at the picture over here in the right that I encircled it.

So, you start and then you non-deterministically choose to have x_1 in S choose to have x_1 in the subset. So, maybe let us see the set that you are building is called I do not know something A . So, non-deterministically decide whether you want to have x_1 in A or x_1 not in A . So, that is the first branch and then you decide whether x_1 should be in A or x_2 should not be in A . So, for each of these branches for the branch where x_1 is in A you have to decide whether x_2 is in A or x_2 is not in A .

And for the branch where x_1 is not in A again you have to decide whether x_2 is in A and x_2 is not in A and so on. In the next time you will decide whether x_3 is in A or x_3 is not in A and so on. So, basically this tree has suppose x_k element, this tree will have height k and it will have two power k leaves. Each leaf will correspond to a subset of S . So, if you look at this left most leaf, so left most this one it will be the subset A is equal to S and the rightmost leaf it will be the subset A equal to empty set.

So, these are trivial subsets. One of it is S itself one of it is empty set, but still you get so, here x_1 is not in A , x_2 is not in A , x_3 is not in A , everything is ruled out of A . And the left side everything A contains and in between you encounter all the possible subsets. So, what is the algorithm? So, non-deterministically you select or reject each of x_1, x_2 up to x_k . So, basically what are you doing,

you are non-deterministically selecting a subset. But instead of non-deterministically like how does that happen actually?

So, you cannot not just select a subset instead you are saying that I first non-deterministically select or reject x_1 , non-deterministically select or reject x_2 and so on. Another way to think of If you want to think of another way to see this whole thing is that you are non-deterministically picking out a k bit vector. So, the vector all zeros correspond to the empty set, all ones correspond to the set S itself.

And let us say, one followed by all zeros correspond to the set containing only x_1 and so on. So, there is a correspondence between the 2^k , k bit vectors and the 2^k subsets of S . So, here what is happening is that in the first step you are choosing the first bit of the k bit vector and then the second bit of the k bit vector and so on. So, this is what is happening. So, now once you non-deterministically select or reject each of x_i then what you do is?

You had the selected x_i 's and verify if they add up to t , the target sum and once you have a subset just verify whether they add up to t and then you check whether if sum is equal to t then you accept otherwise you reject. So, notice that if there is a subset here, we are actually checking all the possible subsets. If there is a subset the path corresponding to that subset we will go to accept.


So, suppose there is some subset the path corresponding to that subset will need to accept. Many others may or may not accept, but at least there will be one path. However, if there is no way to get the target sum t then all the paths will go to reject. So, hence this is not deterministic Turing machine. And you can see that it is in polynomial time because the selecting reject each step is just one for each element of is adding, performing and addition is also in polynomial time and finally its just an if condition.

So, we are using non-determinism and finally, we are deciding whether there is a subset which adds to the target sum or not. So, again the non-deterministic Turing machine does not actually evaluate all these options. If there is a correct option it leads to acceptance. Otherwise, if all the options are wrong it goes to reject. So, again one common mistake that people do is to think of

non-deterministic Turing machines as you follow one path then you follow the second path and then you follow the third path and so on.

But you do not have to do that. The non-deterministic machines if there is a path it will somehow find it. So, again it is an abstract notion which does not necessarily have a real-life parallel. That is the reason why it can be a bit counter intuitive. So, it would be helpful to think of this non-deterministic notion of non-determinism.


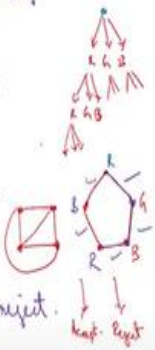
(Refer Slide Time: 19:51)



3- COLORABLE = $\{ \langle G \rangle \mid G \text{ is 3-colorable} \}$.

On input $\langle G \rangle$:

1. Go through the vertices $1, 2, \dots, n$ and non-det. assign each vertex colors $1, 2, 3$.
2. Go through each edge of G , and check if it is properly colored.
3. Accept if the coloring is valid. Else reject.



Another example is checking whether a graph is three colorable. So, how do you do that? So, it is like before. So, what you do is you order the vertices in some order, let us call them 1, 2, 3 up to n . Let us say the graph is n vertices. So, you start with the starting vertex and assign it what are the three colours let us call the colours red, green, and blue. And then you assign for each of these possibilities, you assign the second vertex red, green and blue and so on.

So, at this tree will have height n , where n is the number of vertices, and we will have how many leaves in here 3^n leaves because at each stage, we are assigning a colour to the i th vertex where i is the number corresponding to that stage. So, and there are 3^n possibilities because there are 3^n and there are n vertices, and each one of them can take one of the three colours.

So, what do you do? You go through the vertices 1, 2 up to n and non-deterministically assign each vertex, colours, red, green, or blue and we are just writing R, G or B. So, once that happens, the graph has an assignment. So, the graph has an assignment. So, let us say the graph is what we have here in the bottom right, it is just a five-cycle drawn in the shape of a pentagon. Now, what if we get this combination red, green, green, blue, blue, this is a possible three colouring.

But this is not a proper three colouring. So, then what do you do? Go through each edge of G and check if it is properly colored, accept if the colouring is valid else reject so, what you do is? There are three power n possibilities of colorings so maybe this is one instance. So, here you will check is this edge ok are red and green? Fine, this is fine. Is this edge, green and blue? Yes, that is fine. Is this edge blue and blue? No, it is not.

So, this way just this edge was okay well this edge was okay, while this edge was not okay because now it is not a proper coloring. So, this will be rejected this part will lead to reject. There are some other colouring let us say that this green and blue got swapped. So, if all the edges are properly coloured green, blue, red, green, green, blue, blue, green and red, blue, this will be accepted.

So, since this graph has proper three colouring valid three colorings it will get accepted because the non-determinism exhaustively is able to go through all the possible colouring options. Again, it does not do it like one after the other, it does not need to go cycle through one after the other, the cycles as possibilities one after the other. It has some way of magically picking out a correct option if it exists.

Maybe you can think of it has it having some magic parallelism. And so, in fact in this particular case, there are many possible three colouring also, if I change this green to red, it is also a valid three coloring. So, anyway the point is that there will be many paths that lead to accept. There will also be many paths to reject, but that does not matter. There is at least one path that needs to accept it cause to this is the essence and this is indeed as essence.

However, if we have a graph some other graphs that does not have a valid three colouring, let us say if a K_4 a complete graph on four vertices. So, there is no way to properly colour this graph

with it with three colours. So, now whatever you assign that the three power n options that you assign here, none of these options will work because basically all the pairs of vertices are adjacent.


So, however you assign three, three colours to all these four vertices, there will be two vertices that have the same colour and that will form a clash. Because this K_4 all the pairs of vertices are connected by edges. So, all the parts will need to reject and so it is a valid algorithm for testing three colourability. Again, the beginning stage takes n time because so just look at the height of the tree.

You do not again as I said again you do not need to cycle through each one of the options. The height of the trees making n assignments, that takes n time and then you need to go through each edge. So, if there are n vertices, there can be at most n squared edges. So, you go through each edge and check whether it is a valid colouring and that is it. So, it is also a polynomial time algorithm.

But it is a non-deterministic polynomial time algorithm. So, in both of these algorithms, you might have noticed or you can still notice that what we did was non-deterministically chose something and then decided to verify it. We non-deterministically chose something and then decided to verify it. So, this is what I mean by the guess and verify situation. So, you the non-deterministic choice of something can be thought of as making a guess followed by a verification.

So, you non-deterministic chosen colouring and then verify if it is a proper colouring or you non-deterministically chose a subset of S called it A and then then check whether the sum of elements of A adds up to t . So, in both of these cases, we have these guess first and then verify all the non-determinism happens at the beginning followed by the verification which is completely deterministic.

So, once you have a coloring there is no more non-determinism then the processes are completely deterministic. So, this we will come back to this later. But this is something that we will soon see.
(Refer Slide Time: 27:22)



3. Accept if the solution is valid. Else reject. \downarrow \downarrow
 keep reject

Notice that a DTM can also be considered as an NTM


So $DTIME(t(n)) \subseteq NTIME(t(n))$

$\cup_{k \geq 1} DTIME(n^k) \subseteq NTIME(n^k)$

$P \subseteq NP$

P vs. NP question: Is the above containment strict?

\therefore P vs. NP is $P \subseteq NP$?



One point that I can mention here is that any deterministic Turing machine can also be thought of as it is a non-deterministic Turing machine. This is because a non-deterministic Turing machine simply allows the machine to have more options it could have zero configuration successor, one successor or multiple successors. So, even one is a valid possibility. So, if at every stage you had exactly one successor configuration to all the possible configurations.

It will be a deterministic Turing machine. So, even the deterministic Turing machine can be viewed as a non-deterministic Turing machine. So, anything that can be done with using a deterministic Turing machine can be done as a non-deterministic Turing machine. Hence, the Class D time t_n is contained in N time t_n . And the class $DTIME(n^k)$ is true for any $t(n)$. So, if we replace it with n^k , it is also true.

Now I can take for this entire inequality or entire containment, I can take union over all k . So, k equals 1, 2, 3 and so on. So, the left-hand side is just gives us P and the right-hand side gives us NP . So, we saw what is non-deterministic Turing machine and then so, the definitions of n time t_n followed by the definition of NP followed by some examples.

(Refer Slide Time: 28:59)



P vs. NP question: Is the above containment strict?

Is $P = NP$? or Is $P \subset NP$?

This is a big open question.

Example ...
Consider Sudoku, a game where the player is given a partially filled-in grid of numbers and attempts to complete the grid following certain rules. (Learn an example. Sudoku grid of size $n \times n$. In each of them one cell is empty. Any proposed solution is easily verified, and the time to check a solution grows slowly (polynomially) as the grid gets bigger. However, all

Now, let me go to the P versus NP question. So, as I said P is contained in NP just over here, P is contained in NP. But is this containment a proper one meaning is a strict subset. So, there are two figures here, one I am circling in yellow, the right side one and one on the left side that I am underlining in yellow. So, the right side one piece contained within NP and there seems to be other elements that are in NP but not in P.

In the left side, one P and NP are both equal. So, which of these is the correct situation is? P equal to NP or is P a proper subset of NP? So, the left side is equal and the right side is a proper subset. So, it seems like it is very simple question to ask. But this has been a long-standing open question in computer science, and certainly in theoretical computer science. And arguably, this question has spurred the growth of this field of computation complexity theory.

Most of what we see in this course is a direct result of people trying to attack the P versus NP problem and coming up with various approaches for it. And it is still unknown whether P is equal to NP or is P a strict subset. However, it is still unknown but as a result of this question, and people being interested in this question. And again, not just people, but lots of people and lots of intelligent brilliant people who have who spend most of their careers pursuing it.

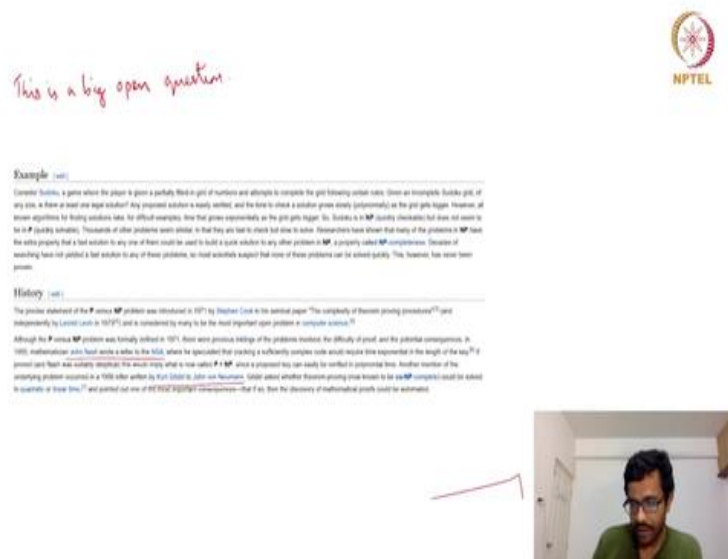
This field has really grown a lot and it has helped us understand the powers of computation or powers of the different models of computation a lot. So, on one hand, it is probably disheartening that it is still open. But it is on the other hand, it has led to so much work. This is a big open

problem. And you may be aware that there is a set of millennial open problems by the Clay math Institute which was set up the list of problems was set up at around the year 2000.

And this is one of the millennial problems. So, they set out seven open problems and said that if anybody solves any one of them, they will give you a million dollars. And out of the seven, I think only one has been resolved now. P versus NP is one of the remaining six and it still has no resolution. But it is a very interesting question. And let me spend a few minutes describing this question in a high-level sense.

So, we saw it in a very set theoretic or syntactic sense, where we define P, we define NP and then so what is it? But let me try to define it in a more high-level intuitive sense.

(Refer Slide Time: 32:13)



So, we saw the NP allows you to verify stuff. So, I like this example. So, this is a copy paste from Wikipedia, where they talk about Sudoku being one of the or use the example of Sudoku to explain people the difference between P and NP. So, you know the Sudoku problem puzzle. So, you have this 9 by 9 grid, where you each row has to have all distinct numbers from one to nine in each column, and each mini square.

So, again, if you if you do not know what it is, you can look it up. But it usually comes up in newspapers and people spend time solving it. So, the goal is you are given a partially filled set of

numbers or a grid of numbers and you have to complete it as per the rules. And usually takes a while sometimes it is very hard to analyze all the possibilities and rule out some things and fill it in.

So, but sometimes after spending one hour, half an hour depending on how good you are, depending on how much practice you have had, how much with these sorts of things it may take you half an hour maybe 5 minutes, maybe 10 minutes, maybe 2 hours, whatever. The point is that solving a Sudoku puzzle is not immediately easy. Whereas, if I give you a grid and I completely filled it up and ask you to verify whether this is a properly filled grid that is not so difficult to do.

Now you know what the process like you is check the first row, second row, first column, second column and so on and then the mini squares etcetera and then you are done. Whereas, if you are given a grid to solve, to solve actually and partially filled grid then you have to resort to various tries of attacking and finally, somehow, after a lot of time, you may get it done. So, clearly it seems that P is like solving it with a from a partially filled grid.

NP is like given a filled grid verifying it, so you may say, this is not a properly filled grid because this mini square contains two eights, or this row contains two sevens whatever. Or you may check everything and say okay this is this grid is good. I have checked all the possible things to check, and it is fine. So, certainly checking whether a filled grid is a valid grid, it is seems to be a more straight forward task than solving. So, P corresponds to the act of solving it.

So, you have to just given something you have to solve it. And NP corresponds to checking whether a given solution is correct or not. So, one is coming up with a solution rather is verifying its existing solution. So, the question are these two the same complexity. If P is equal to NP, then these two will be the same complexity. If P is not equal to NP, then verification that means that verification is strictly easier than decision.

So, again it is widely believed most computer scientists that you ask will say that they believe that P is not equal to NP because it seems that verification ought to be easier, then actually solving it. But even though it seems that one would intuitively think this should be the case, but it has evaded

the attempts of all the computer scientists. The problem P versus NP has evaded the attempts of all the computer scientists so far. So, that is the P versus NP situation.

So, one is like again, other another parallel is like coming up with composing a musical symphony. So, it requires a lot of creativity, versus listening and appreciating a musical piece. So, it is certainly much easier to appreciate a musical piece than compose one or that is what we think. But if P is equal to NP that will be saying in some high-level sense that being able to verify which is like appreciating music is the same as being able to come up with something so interesting. So, in some ways it is asking, is it possible to automate creativity?

P equal to NP then you are kind of saying that creativity could be automated. Again, this is a very high level or it may be a philosophical kind of way to look at the P versus NP question. But again, I am just doing this because it is the most important problem in theoretical computer science and perhaps all of computer science. And certainly, it is highly relevant to this course because almost all of the subject matter that we will see during this course came up as a result of people trying to address the P versus NP question. I Just want to briefly mention a bit of history.

So, this problem was initially stated in 70s 71, early 70s, by Steve Cook and independently by Levin, Leonard Levin. So, the problem is in the early days. In those days there was no email, there was no internet and research had to spread by hard copies, like somebody has to write a book or write a paper and it has to be published and then that journal has to travel across the globe for other people to know what there results are.

It is not like these days where I am delivering this lecture to you were internet. So, even though Steve Cook and Levin so, they might not have come up with the result on the same day. But it would have taken time for it to go from one of them to the other one, a Cook was in North America somewhere I think US and Levin was in Russia. So, and also do in those times after, because of the cold war that Russia would not have been talking to the Western Europe, US and Western Europe.

And this is when it was the problem was first formally stated. But however, there are a couple of instances at least a couple of recorded instances where scientists actually talked about this problem in an informal or high-level setting. So, one of them was John Nash, so if you know the word Nash equilibrium, he wrote a letter to NSA in the in the 50s, where he said that cracking a certain cryptography code, so a security code will require a long time unless P is equal to NP or assuming P not equal to NP .

And another instances the mathematicians slash computer scientists, Kurt Godel, he had written to John Von Neumann in 1950s, 1956. Again, he asked whether like stated a hard problem again I did not want to go into the details, can it be solved in polynomial time, quadratic time for instance. Again, if that was solved in polynomial time, it would imply that P is equal to NP . Again, that would mean that proving something could be automated.

So, again these are this is other evidence that brilliant minds already knew or already were aware that there is something the two classes P versus NP and that there are all these underlying issues going on, but the formalization took till early 70s. So, that is I think we will have crossed the 30-minute mark by now. So, I will wind up now, just quickly summarizing, we started by defining the non-deterministic Turing machine, the running time of it and n time t_n complexity class n time t_n , which is in parallel with D time t_n .

And then we define NP in parallel to P , non-deterministic polynomial time. So, two examples and then we stated that P is a subset of NP and we stated the P versus NP question which is whether P is a strict subset or whether P is equal to NP . I think that solved for this lecture and the rest will continue in the next lecture. Thank you.