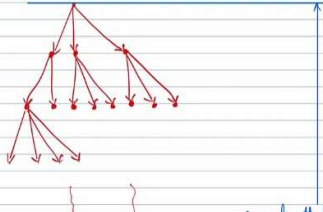


Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Non-Deterministic Polynomial Time _ Part 1

(Refer Slide Time: 0:16)

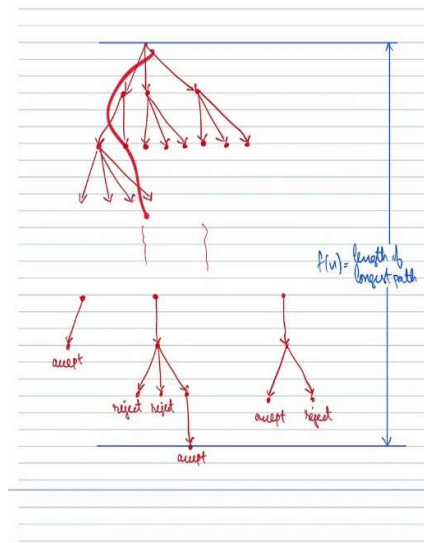
Non-deterministic Polynomial Time

Def 7.9: Let N be a non-deterministic 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 .



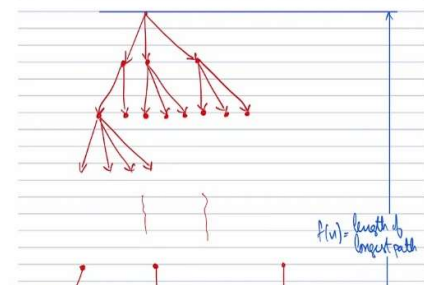
Hello and welcome to lecture 46 of the course Theory of Computation. In the previous lecture, we saw time complexity and we saw the complexity class P which corresponds to the class of problems or languages that can be decided in deterministic polynomial time. So, now in this lecture we will start seeing non deterministic polynomial time. So, before getting into non deterministic polynomial time, we first have to understand what is the time complexity or running time of a non-deterministic Turing machine.

(Refer Slide Time: 0:54)



Non-deterministic Polynomial Time

Def 7.9: Let N be a non-deterministic 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 .



In the case of a deterministic Turing machine the time is fairly clear because it is a deterministic Turing machine, it takes a specific number of steps, the running time or the time complexity was given as a function f based on the length of the input. So, the $f(n)$ the running time is defined as the maximum time it takes or the maximum number of steps it takes on any input of the same length n .

In the case of non-deterministic Turing machines this is the same, the running time is the maximum number of steps, $f(n)$ is the maximum number of steps it takes on any input of length n . However, there is one additional thing in the case of a deterministic Turing machine an input takes a fixed number of steps.

Here there are several possible computation branches, there are several possible computation branches and all of which may be of different length there could be different branches of different length. So, for instance, you could have branches that end very quickly, there could have branches that go a significant take a significant number of steps. So, but we define the running time to be the running time of a specific input.

So, this is all for a specific input, to be the maximum number of steps taken by the longest path. So, if this is a specific input and this is the configuration tree this particular, the one that is at the bottom this particular accepting configuration, the accepting path that comes some way like this, this seems to be the one that takes the longest time.

So, $f(n)$ is should be for this particular input should be this the length of this path, but then we take maximum over the, for every input the $f(n)$ is the length of the longest path and then we further take the maximum amongst all the inputs of a certain length. So, $f(n)$ is a maximum number of steps that the non-deterministic Turing Machine capital N uses on any branch of its computation on any input of the of the length n.

So, you have to maximize on all the inputs of a certain length and for each input all the branches of computation you want to see the maximum number of steps that is how we measure the or we assess the time complexity of a non-deterministic Turing machine.

(Refer Slide Time: 3:48)

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

Def 7.22: $NP = \bigcup_{k=1}^{\infty} NTIME(n^k)$

We will see that this is equivalent to the Guess & Verify model.

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



So, given this the rest of the definitions are fairly straightforward once we understand what it takes for, what is how is it time measured for a non-deterministic Turing machine? So, similar

to $DTIME(t(n))$, we have $NTIME(t(n))$, so $DTIME(t(n))$ was the set of all languages that can be decided in $t(n)$ time by a deterministic Turing machine.

So, $NTIME(t(n))$ is a set of all languages that can be decided in order $t(n)$ time by a non-deterministic Turing machine. So, this is exactly the same the only differences is this N in the definition $NTIME(t(n))$ and the fact that here we have an NTM instead of a DTM. So, just like we have $DTIME(t(n))$, which is a class of all languages that can be decided in order $t(n)$ time by a deterministic Turing machine we have $NTIME(t(n))$, which is a class of all languages that can be decided by an NTM, a non-deterministic Turing machine in $O(t(n))$ time.

And just like we had P, the class of all languages that can be decided by a deterministic Turing machine in polynomial time. We have NP, which is a class of all languages that can be decided in polynomial time by a non-deterministic Turing machine. So, it is exactly the same, but instead of P we have NP and instead of $DTIME(t(n))$ we have $NTIME(t(n))$.

So, the definition for P was a union of all $k = 1$ to ∞ , $DTIME(n^k)$, here we $NTIME(n^k)$ the class of all languages, that can be decided by a non-deterministic Turing machine in polynomial time. So, later, we will see that NP has a characterization in the Guess and verify model also. So, but for now, I am just mentioning this so, guess and verify model is when we make the non-deterministic choices, right at the beginning of the computation. Anyway, we will first see some examples.

(Refer Slide Time: 5:56)

Def 7.22: $NP = \bigcup_{k=1}^{\infty} NTIME(n^k)$

We will see that this is equivalent to the Guess & Verify model.

Examples:

1) SUBSET-SUM = $\{ \langle S, t \rangle \mid S = \{x_1, x_2, \dots, x_k\},$
 $\exists T \subseteq \{1, 2, \dots, k\}, \text{ s.t. } \sum_{i \in T} x_i = t \}$

On input $\langle S, t \rangle$:

1. Non deterministically select/reject each of x_1, x_2, \dots, x_k .
2. Add all the selected x_i and verify if they add up to t .
3. If $\text{sum} = t$, then accept. Else, reject.



So, the first example is something that I described briefly in the previous lecture. It is a language called subset sum. So, we are given 2 things, a set S of integers and target sum t . The question is, so, S is a set consisting of x_1, x_2 up to x_k , the question is, is there a subset t . So, here the set subset is written as a subset of the indices such that, the elements of the subset sum up to the target sum t .

So, here the subset is written indicated by the indices. So, if the subset is 1, 3, 5 then the corresponding subset is x_1, x_3 and x_5 , but the question is the same is there a subset of S that sums up to t , is there a subset that sums to the given sum. So, let me describe so, let us first describe a very direct brute force approach to this problem. So, if we have to do it with a deterministic Turing machine, one possibility is we try out, so this is a naive approach you try out all possible subsets.

So, given that there are k elements of the set, the number of possible subsets is 2^k we try out all possible subsets and for each subset you check whether the sum is equal to t and if you find a subset whose sum is equal to t you say yes, this is a yes instance. Otherwise, you go through all possible subsets and you do not find any subset which sum to t at which point you say no there is no subset that sum to t .

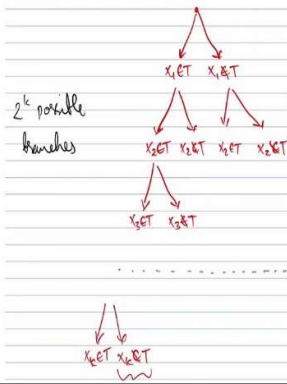
But then that requires you to check 2^k subsets so, or $2^k - 1$ subset or whatever. The S has k elements there are 2^k possible subsets. So, this is a deterministic algorithm that I just described but you have to check 2^k subsets and for each 2^k subset you have to verify so, that requires some addition et cetera.

So, that may take some polynomial time et cetera, which I am not really measuring, but it takes an exponential time in the number of elements of S . However, now, let me describe a non-deterministic Turing Machine, non-deterministic approach for this. So, one approach is the non-deterministically select or reject each element of the set S . So, we come to x_1 , you either select x_1 or reject x_1 .

(Refer Slide Time: 8:38)

2. Add all the selected x_i and verify if they add up to t . $\rightarrow O(k)$

3. If $\text{sum} = t$, then accept. Else, reject.



Examples

1) SUBSET-SUM = $\{ \langle S, t \rangle \mid S = \{x_1, x_2, \dots, x_k\}, t \in \{1, 2, \dots, k\}, \text{ s.t. } \sum_{i \in S} x_i = t \}$

On input $\langle S, t \rangle$:

1. Non deterministically select/reject each of x_1, x_2, \dots, x_k .
2. Add all the selected x_i and verify if they add up to t .

3. If $\text{sum} = t$, then accept. Else, reject.

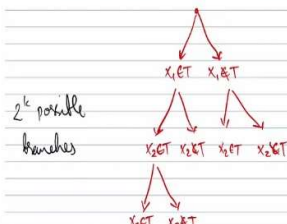


$$T \subseteq \{1, 2, \dots, k\}, \text{ s.t. } \sum_{i \in T} x_i = t$$

On input $\langle S, t \rangle$:

1. Non deterministically select/reject each of x_1, x_2, \dots, x_k . $\rightarrow O(k)$ time
2. Add all the selected x_i and verify if they add up to t . $\rightarrow O(k)$

3. If $\text{sum} = t$, then accept. Else, reject.



So, I have described it in a tree here. So, first we decide to include x_1 or exclude x_1 , left means including, right means excluding. And then, whichever way we go we decide to include x_2 or to exclude x_2 and even if we exclude x_1 we have the same 2 options and then whichever way we go we decide to include x_3 or exclude x_3 and so on. So, this now we keep building the string.

And now, at the end we will be asking the question should we include x_k or exclude x_k and we will have altogether we will have 2^k such paths. Starting from the left most such path which where everything is included and the right most path which is basically an empty set no, no element is included and everything in between where some things are included and some things are not. And for each of these.

So, now we have these non-deterministic choices so, we walk through x_1, x_2 et cetera. And at each point we non-deterministically decide to accept or include x_i or not include x_i . So, this results in 2^k possible branches of computation. And then at the end, you just add up. So, let us say over here, you just add up, check if sum is equal to t , and then accept or reject accordingly.

So, which is what I have written here, we non-deterministically decide to select or reject, sorry, select or reject each of x_1, x_2 up to x_k , and then add the selected x_i and verify if they add up to t . If they add up to t we accept otherwise we reject so. that is that is pretty much it. That is a very simple algorithm. And it is not very difficult to see that this is correct.

If there is a subset, so, we are going through all possible so there are 2^k possible subsets checked here. So, 2^k possible branches and if there is a subset that sums to t that will correspond to some computation in branch here and that will get accepted which means a non-deterministic Turing machine accepts this instance.

If this instance was a no instance meaning there is no subset that sum to t then all possible computation branches, all possible branches will reject because none of them will. So, by assumption, no subset should sum to t , so everything will reject. If is a yes instance there is some subset and the branch corresponding to that subset will accept. So, clearly this is a correct non deterministic approach.

But let us see the time taken. So, first this step non-deterministically selecting or rejecting each step, each one of them this takes order k time. So, this takes, this takes order k time or, in fact that actually not even $O(k)$, it is just directly k time and then you add up the selected x_i and

verify if they add up to t . So, this also may be something like order k time, because there are k numbers maximum. So, it takes order k time.

So, this whole thing runs in order k time. But I am talking about non deterministic running time, not the deterministic running time. In non-deterministic running time, if there is a correct approach, so, we do not have to add up the lengths of all the branches. If there is a correct approach, we somehow the machine will find it if this is the correct accepting path, the machine will find it.

So, the running time is order k for the non-deterministic Turing machine, it is a fairly straightforward algorithm. So, the approach is clear, we non-deterministically select or reject each x_i and then so, we are left with some subset we added the elements of the subset and verify they add up to the desired sum. If it is adding up to the desired sum we accept, if it does not add up to the desired sum we reject.

So, if there is a subset which adds up to the desired sum, there is some computation branch which will lead to an acceptance. If there is no subset that adds up to the desired sum then all the parts will reject which is exactly what we want in a non-deterministic Turing machine and also this running time is order k which is polynomial in the description of the input and subset is in NP. Hence, subset sum is in NP, takes polynomial time by a non-deterministic Turing machine.


However, I cannot say that subset sum is in P because the deterministic algorithm that I described 2^k time which is not polynomial in the description of the input. So, there was not a deterministic algorithm, but it took more than polynomial time.

(Refer Slide Time: 14:09)

SUBJECT SUPPLEMENT

2) 3-COLORABLE = $\{ \langle G \rangle \mid G \text{ is 3-colorable} \}$

→ If G has n vertices, brute force takes 3^n time
deterministic



On input $\langle G \rangle$:

- Go through the vertices $1, 2, \dots, n$ and non-deterministically assign colors R, G, B . $O(n)$
- Go through each edge of G and check if it is properly colored. $O(m)$
- Accept if coloring is valid. Else reject.

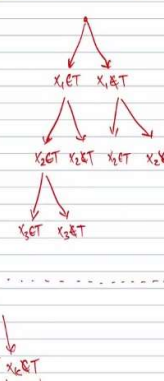
3-COLORABLE \in NP.

Note that a DTM can be considered as an NTM.

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



- Add all the selected x_i and verify if they add up to t . $\rightarrow O(k)$
- If $\text{sum} = t$, then accept. Else, reject.



2^k possible branches



So, the next language is 3 colorable. So, what is 3 colorable? 3 colorable is given a graph we are asking whether it can be colored using 3 colors. So, maybe one example is this. So, this is a yes instance, this graph can indeed be colored using 3 colors, it is a red, green, red, green, blue, this can be colored using 3 colors. So, if you look at any edge, the endpoints do not share the same colors that is the requirement.

So, given a graph how do we check this? Again, I explained the brute force approach which is deterministic. So, for each vertex how many assignments are possible? Let us say red, green and blue are the 3 colors. So, if the graph has n vertices, so suppose the graph has n vertices maybe I will just move this down a bit so as to make some space. If G has n vertices brute force takes 3^n time.

Because each vertex has 3 possible choices of colors red, green, and blue. And then it takes 3^n time and there are 3^n possible assignments. And then, once the assignment is completely assigned, we need to check, we need to go through all the edges. So, that takes let us say the m edges $3^n \times m$ time.

So, maybe I will just say that $3^n \times m$ time, which is not polynomial, brute force by that I mean deterministic approach takes $3^n \times m$ time. However, let me just describe so maybe I should write it slightly above. Brute force takes $3^n \times m$ time, this is deterministic, but the non-deterministic approach is the following. So, there are similarities to the approach that we followed here.

So, what we do is we walked let us say the vertices are 1, 2, 3 up to n . So, we go to the first vertex, we non-deterministically assign a color red, green or blue. Then we go to second vertex we non-deterministically assign a color red, green or blue. So, there are 3 options for each vertex and we non-deterministically assigned one of these options, and at the end of that computation path, we have made some assignment and then this assignment we deterministically check whether this assignment constitutes a proper coloring.

So, meaning we go through all the edges of the graph and see whether there is any clash between the colors of the endpoints. So, this takes actually, the first step takes order n time, this takes order n time, because we have to walk through n vertices. And this step takes order m time, where m is the number of edges.

And if the graph is indeed 3 colorable, meaning if there is a correct 3 coloring, one of these multiple computation paths will lead to accept. If the graph is not 3 colorable, whatever you try, there will be some edge with a clash. So, all paths will lead to reject. So, what I am saying is that the running time of this non deterministic algorithm is $O(n) + O(m)$ which is $O(m)$, let us say, assuming the number of edges are more than the number of vertices.

So, the running time is polynomial hence, 3 colorable is also in NP, maybe I will just use a different color, because we can do it in this much time. Whereas, the brute force takes exponential time, which is yet another instance of a problem where the non-deterministically is really seem to help. So, we sought two examples one is subset sum and another one is 3 color ability.

Both we followed a similar approach we non-deterministically made some choices and then we verified it. In this case we non-deterministically assigned the coloring and then we check

whether this coloring is proper. In the earlier case we non-deterministically chose a subset and then check whether the subset added up to the required sum. So, there was a similarity in both of these things. Anyway, we will come to that a bit later.

(Refer Slide Time: 20:33)

on input x .

- Go through the vertices $1, 2, \dots, n$ and non-deterministically assign colors R, G, B . $\left. \begin{array}{l} \text{1. Go through the vertices } 1, 2, \dots, n \text{ and} \\ \text{non-deterministically assign colors } R, G, B. \end{array} \right\} O(n)$
- Go through each edge of G and check if it is properly colored. $\left. \begin{array}{l} \text{2. Go through each edge of } G \text{ and check if} \\ \text{it is properly colored.} \end{array} \right\} O(m)$
- Accept if coloring is valid. Else reject.

3-COLORABLE \in NP.

Note that a DTM can be considered as an NTM.

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

$$DTIME(n^k) \subseteq NTIME(n^k)$$

So $P \subseteq NP$

P vs NP question: Is the above containment proper? Is $P=NP$ or $P \subset NP$?

on input x .

- Go through the vertices $1, 2, \dots, n$ and non-deterministically assign colors R, G, B . $\left. \begin{array}{l} \text{1. Go through the vertices } 1, 2, \dots, n \text{ and} \\ \text{non-deterministically assign colors } R, G, B. \end{array} \right\} O(n)$
- Go through each edge of G and check if it is properly colored. $\left. \begin{array}{l} \text{2. Go through each edge of } G \text{ and check if} \\ \text{it is properly colored.} \end{array} \right\} O(m)$
- Accept if coloring is valid. Else reject.

3-COLORABLE \in NP.

Note that a DTM can be considered as an NTM.

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

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

So $P \subseteq NP$

P vs NP question: Is the above containment proper? Is $P=NP$ or $P \subset NP$?



One point that I want to mention is, a comparison between the classes P and NP. So, first of all any deterministic Turing machine can also be considered as a non-deterministic Turing machine. Because non-deterministic Turing machine has the capability to make multiple choices for a given configuration. So, each configuration there could be multiple next moves possible. So, delta can be a function that maps to a set. So, it can map to more than one possible option.

So, a configuration can have multiple possible successors, valid successors, but multiple possible one it could also be one successor it could also be zero successor. So, a deterministic Turing machine can also be seen as a non-deterministic Turing machine just that I am not using multiple successes at any instance. So, anything that can be done by a deterministic Turing machine we can do it with a non-deterministic Turing machine also, because, I can just have the same machine and view it as non-deterministic Turing machine.

So, anything that can be done in deterministic time $t(n)$ can be done by non-deterministic time $t(n)$ as well because, the same machine works. So, anything that can be done by deterministic time n^k and we consider in non-deterministic time n^k . So, deterministic time $t(n)$ is a subset of non-deterministic time $t(n)$ for any function $t(n)$. Specifically for the function n^k deterministic time n^k is a subset of non-deterministic time n^k .

Now, if I take union of k equal to 1 to infinity for both these terms we get P in the left hand side at NP in the right hand side. So, union over all k deterministic time n^k is P. Union over all k non deterministic time n^k is NP.

So, this gives us that P is a subset of NP, it is fairly simple anything that can be done by a deterministic Turing machine in polynomial time can also be done with non-deterministic Turing machine polynomial time. So, P is a subset of NP. It is a fairly simple thing to say and an easy thing to see.

(Refer Slide Time: 22:59)

D TIME (n^k) \subseteq N TIME (n^k)

So $P \subseteq NP$

P vs NP question: Is the above containment proper? Is $P = NP$ or $P \subset NP$?

$P = NP$ NP

This is a big open question. One of the biggest open questions as of today.

Formally asked by Stephen Cook (1971) and independently by Leonid Levin (1973)

Now, let me come to the question which is very, very famous, called the P versus NP question. So, it is one of the most famous questions in computer science, certainly in theoretical computer science, and if you have at least heard about computer science in popular science or literature or something, you might have seen come across this question P versus NP. So, now we have formally defined what is P and what is NP?

Both of them are complexity classes, P is the class of all languages that can be decided in polynomial time by a deterministic Turing machine. NP is a class of all languages that can be decided in polynomial time by a non-deterministic Turing machine. So, that is P and NP. P is languages that can be decided in polynomial time by a deterministic Turing machine, NP is languages that can be decided by a non-deterministic Turing machine in polynomial time.

So, sometimes some people who are not really students of computer science end up saying that sometimes they think that NP stands for not polynomial. No, NP stands for non-deterministic polynomial time not for non-polynomial. So, as somebody who has listened to me delivering lectures in this course and somebody who have taken this course, at least I expect the students of this course to know what is P?

P stands for polynomial time or deterministic polynomial time and NP stands for non-deterministic polynomial time, not non polynomial time. So, let me come back to the P versus NP question. So, as I said, P is contained in NP, P is a subset of NP. Now, the question is, is this subset, is this containment proper or is this containment strict?

So, one possibility is that P and NP are the same. Anything that can be decided by non-deterministic Turing machine in polynomial time can also be decided by a deterministic Turing machine in polynomial time. So, P and NP are the same. So, this is what I have depicted here, P and NP are the same set. All the other possibility is that P is a subset, a strict subset of NP, a proper subset of NP.

By that I mean there are languages in NP, which are not in P. So, these are the 2 possibilities is P equal to NP, which is one possibility, or P contained in NP, but not equal to NP. These are the 2 possibilities. So, the question is which of these cases is it? So, we know that it is a subset? But is it a proper subset? Are there languages over here? Sorry? Are there languages over here? Or P and NP are equal?

One of these has to be the case, we do not know. The answer is that we do not know which one of these cases is the correct one. Even now, we do not know. So, this question was first kind of asked around 50 years back, even now, the answer is unknown. So, it is one of the biggest open questions in computer science, and certainly in theoretical computer science.

And it has got a lot of attention by a lot of computer scientists, and also a lot of people who are not computer scientists. It was one of the so called millennial problems. So, there was an American institute called Clay Math Institute, who offered a prize of 1 million dollars if you solve any of, like the there was a list of seven questions, seven open questions.

And if you solve one of them, they promise any one of them, each one of them, they promise 1 million dollars to the person who solved it. One of those questions is the P versus NP question. So, first of all, we do not even know which one is the case, even if we know there is no proof. So, many people believe that, this second is the correct situation.

Many people think that P is not equal to NP, but it is still unknown, nobody has an approach. Nobody has actually been able to show that. So, to show that P is not equal to NP, one has to figure out a language which is in NP and show that it is not in P. To show that P is equal to NP, we will see some techniques or we will see some more theories soon, which will help us develop a theory for this.

So, this is a big open question. And, and it is said that many, many things like the cryptographic systems as of today will collapse if it is shown that P is equal to NP. So, it has another significance as well, not just as a curiosity of an open question. This was first asked by Stephen Cook in 1971. And later, almost in parallel by Leonid Levin by Cook and Levin, 2 people who

posed this question independently. So, in those days, the thing was that we did not have like internet, email was not there.

So, it is very possible that somebody in one part of the globe works on some problem. And somebody who is in some other part of the globe also works on the same problem. They even publish it in their, let us say, their own journals or whatever. But then by the time person, the first person reads that the second person has also worked on it by then both of them have built the theory and published it. It was asked around 50 years back. And it is still kind of unknown. The answer is unknown. We do not know which one is a correct case.

(Refer Slide Time: 28:54)

This is a big open question. One of the biggest open questions as of today.

Formally asked by Stephen Cook (1971) and independently by Leonid Levin (1973)

- Also asked by others: 1956 letter by Kurt Godel to John von Neumann.

SAT: Short for Satisfiability. There is a Boolean formula and we want to know if it is satisfiable



QUESTION 4

SUBSET-SUM ∈ P

2) 3-COLORABLE = {G | G is 3-colorable} ∈ P

→ If G has n vertices, brute force takes 3^n time. deterministic

On input G:

- Go through the vertices $1, 2, \dots, n$ and non-deterministically assign colors R, G, B. $O(n)$
- Go through each edge of G and check if it is properly colored. $O(m)$
- Accept if coloring is valid. Else reject.

3-COLORABLE ∈ NP.

Note that a DTM can be considered as an NTM.

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



3-1000-0000-0000

Note that a DTM can be considered as an NTM.

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

$$DTIME(n^k) \subseteq NTIME(n^k)$$

So $P \subseteq NP$

P vs NP question: Is the above containment proper? Is $P=NP$ or $P \subsetneq NP$?

This is a big open question. One of the biggest open questions as of today.



In fact, as I said earlier in the previous lecture, that the paper by Jack Edmonds also talked about difference between polynomial time and exponential time. Many of these ideas, probably were kind of in the air, so to speak. But people thought about these ideas, but maybe not formalized. In fact, it was later discovered that in a letter, in 1956, Kurt Godel had asked John von Neumann the questions which is similar in nature, not so formal. It was just a letter, not a formal theory or formal academic paper.

But he had asked about P versus NP, or a question which would imply, or which would relate to P versus NP to John von Neumann. So, there is a Wikipedia page for P versus NP and you should certainly have a look at it. So, the thing is this, just to give you a slightly more philosophical overview, if you look at 3 colorability. A non-deterministic Turing machine is able to guess some coloring and verify it.

Whereas a deterministic Turing machine has to kind of check determine the 3 coloring in a brute force manner. So, one is about just verifying, non deterministic machine is able to automatically guessed the correct coloring and it just has to verify. Whereas in deterministic Turing machine it has to go through and identify the 3 coloring as well.

So, one is about actually finding out the 3 colorability, the deterministic Turing Machine. Non deterministic Turing machine, it is about verifying the 3 colorability. So, P versus NP can be thought of as actually computing something versus verifying something. Or another way to think about it is like, if you are like a music composer or something, being able to compose the music versus being able to appreciate the music.

So, both of them, you feel that it should be easier to verify than to come up with answers. So, that is one reason why people feel that P is not equal to NP meaning the latter is the case. This is the case. So, it is like asking is deciding something the same as verifying something? So, that is the kind of very, very high-level overview about the P versus NP question.

It is one of the most important problems, please have a look at the Wikipedia page or other resources. But now, as somebody who has taken this course, I want you to know what is P versus NP? So, P is polynomial time or deterministic polynomial time, NP is non deterministic polynomial time and not non polynomial time. And the question is, are they the same? So, P is contained in NP. The question is, are they the same? Or is P a strict containment in NP? And the answer is we still do not know.