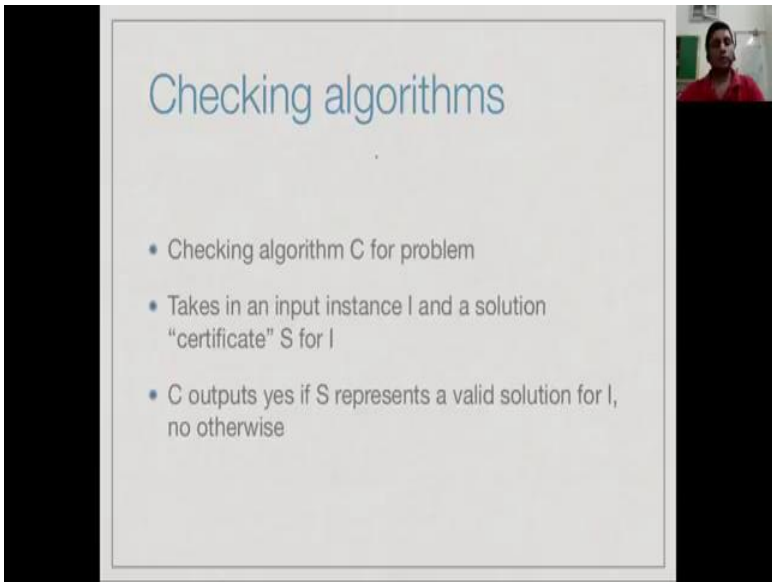
Design and Analysis of Algorithms

Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 08
Module - 07
Lecture - 56
Intractability: P and NP

In the last lecture of this course, we will look at this celebrated question about P and NP, which is often mentioned as one of the last and most important open problems in Computer Science.

(Refer Slide Time: 00:14)

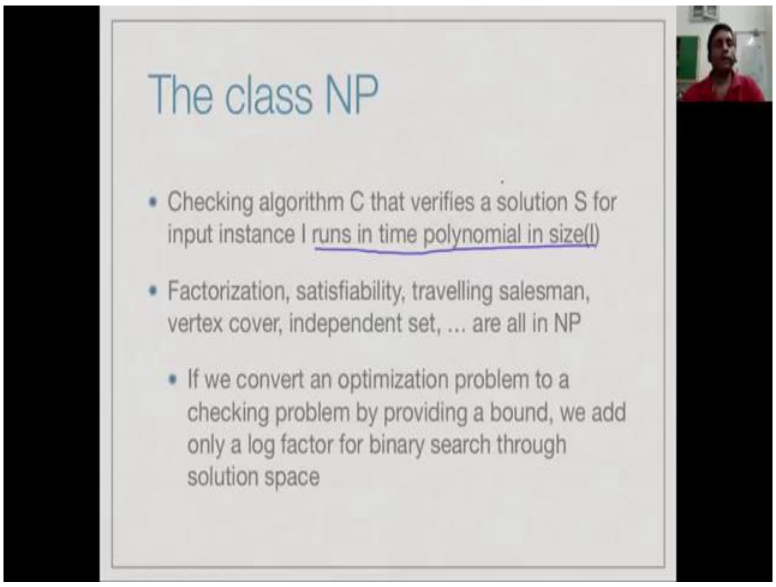


Checking algorithms

- Checking algorithm C for problem
- Takes in an input instance I and a solution "certificate" S for I
- C outputs yes if S represents a valid solution for I, no otherwise

So, we have seen that a checking algorithm is something which allows us to verify whether or not a given problems solution is valid. So, a checking algorithm does not necessarily solve the problem itself, but what it can do is take an input to the problem, a potential solution to that problem, and validate whether or not that solution is indeed one. So, without having solved the problem itself, it can just check if a given solution is a valid solution or not.

(Refer Slide Time: 00:44)



The class NP

- Checking algorithm C that verifies a solution S for input instance I runs in time polynomial in size(I)
- Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP
  - If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor for binary search through solution space

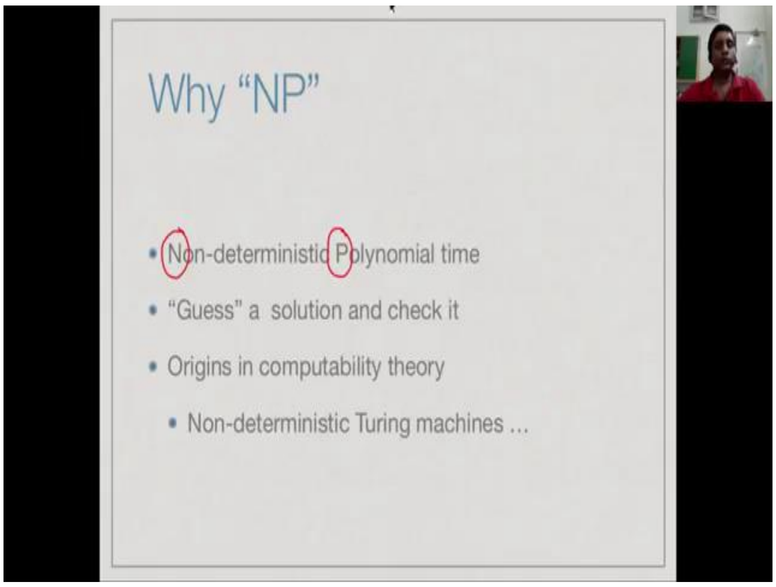
So, the class NP is a class of problems for which such checking algorithm is exist with the additional constraint that the check runs in polynomial time in the size of the input. So, the checking should be efficient, so in all the problems, we have seen in the last lecture, this is in fact true. Factorization is an NP, because given two factors, we can multiply them efficiently to get the answer, if we want and check.

Satisfiability, again, because we just have to take the assignment of true, false to the formula variables and verify, whether the formula evaluates to true after that. Again, traveling salesman with the bounded version, we can take a set of given path, check it is a simple cycle, add up the weights and verify the bound is satisfy, likewise vertex cover and independent set. So, these are all NP, because the solution can be validated efficiently with respect to the input size.

One thing to notice that in problems like traveling salesman, vertex cover and independent set, we took an optimization problem and converted it into a bounded checking problem. But we also say that the bound could then be exercised using binary search in order to find the actual answer, so this binary search has the logarithmic factor. So, overall in some sense there is no loss of generality, because you take a polynomial times step, and then you multiplied by logarithmic thing, so everything is still the same.

So, there is no difference journey between the pure checking problems like satisfiability and the modified checking problems which involve introducing a bound, a traveling salesman or vertex cover.

(Refer Slide Time: 02:24)



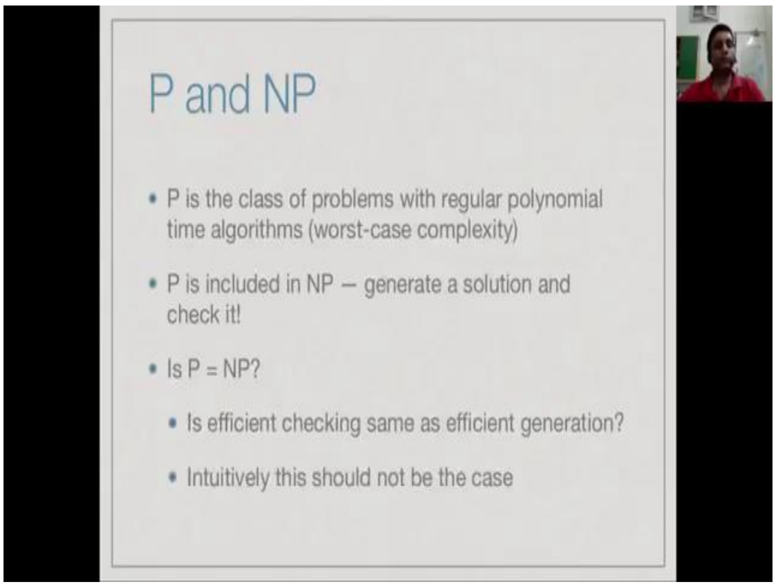
Why "NP"

- Non-deterministic Polynomial time
- "Guess" a solution and check it
- Origins in computability theory
    - Non-deterministic Turing machines ...

So, why is this class called NP, so NP comes from the word non deterministic polynomial time, so this is a historical reference. So, it refers to the fact that these correspond to problems where you can non deterministically guess the solution, you can magically produce the solution and then in polynomial time, you can verify whether the solution is correct or not.

So, the actual origin of this term comes from computability theory, from theory of non deterministic turing machines. So, if we have studied that, then you will know, why it is called NP, but if you are not studied that, it is not so important us to realize, what it implies for algorithms. So, NP comes from non deterministic polynomial time, which in terms comes from turing machines.

(Refer Slide Time: 03:13)



P and NP

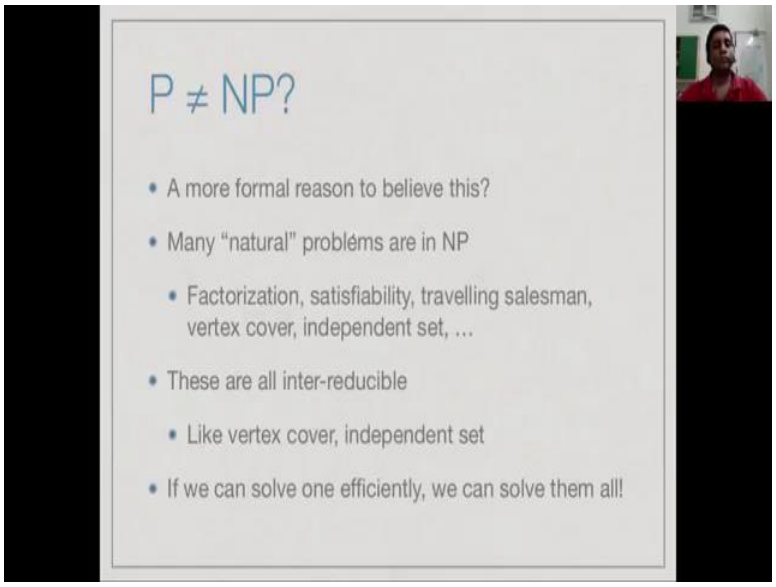
- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?
  - Is efficient checking same as efficient generation?
  - Intuitively this should not be the case

So, the two classes that people talk about a P and NP. So, we have just seen what NP is, so what is P? So, P is just the class of problems which we have been trying to explore in this course. These are problems for which you have a worst case algorithm which runs in polynomial time. So, recall that we said right at the beginning that we are looking at problems of the form N squared, N cubed then all that, we probably looking at even less in terms of practical things or things like N log N, but we certainly do not want to be a N and N factorial.

So, P is a class of all problems with regular polynomial time complexity in terms of worst case. Now, by all definition, everything in P is also in NP, because we do not need to get a solution to check it, because it is in P, we can actually generate the solution ourselves. So, checking algorithm ask somebody to give us an input and a propose solution and then validate it. A problem in P, we can actually generate the solutions, so we can validate again and therefore, every problem in P is actually in NP.

So, the question that people want to know is whether P is equal to NP. So, it is a converse group, is it possible that whenever I have an efficient checking algorithm, I also have an efficient algorithm to generate a solution. So, intuitively this is does not seen to be the case, as we observed, the teacher who was checking the factorization homework, need not even know how to factorize, teacher only needs to know, how to multiply, so it seems intuitively much simpler to check a solution, then to actually generate one. So, let us see whether this intimation can be made formal.

(Refer Slide Time: 04:51)



P ≠ NP?

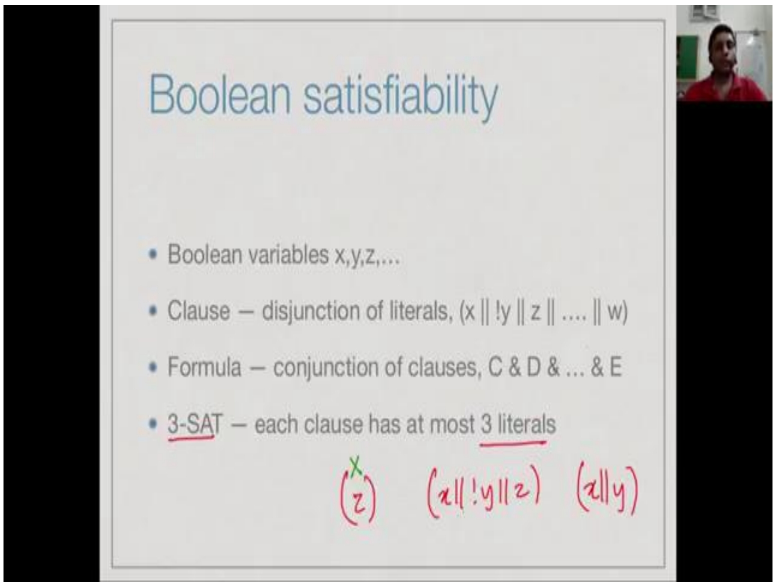
- A more formal reason to believe this?
- Many "natural" problems are in NP
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ...
- These are all inter-reducible
  - Like vertex cover, independent set
- If we can solve one efficiently, we can solve them all!

So, why do people believe that P is not equal to NP? Well, one of the reasons is just from experience or it is empirical. So, we have seen many natural problems, factorization, satisfiability, traveling salesman, vertex cover, independent set, these are all in NP. We have seen that vertex cover and independent set can be reduced to each other. Today, we will see that some other NP problems can also be reduced each other.

So, in fact, you can find that all these problem actually are inter reducing, this means that the effectively have the same level of hardness. Because, when things are reduced, one between one and other, either you can transport efficient solutions or you can claim inefficients. But effectively there all of the same type, you can solve one, you can solve the other in roughly the same amount of time.

So, all it takes is solve any one of these and all of these problems will then have an efficient solution. But people have been trying these problems from different angles from many years, now and nobodies have found one. Therefore, there is good experimental evidence that there is no good solution for these.

(Refer Slide Time: 05:56)



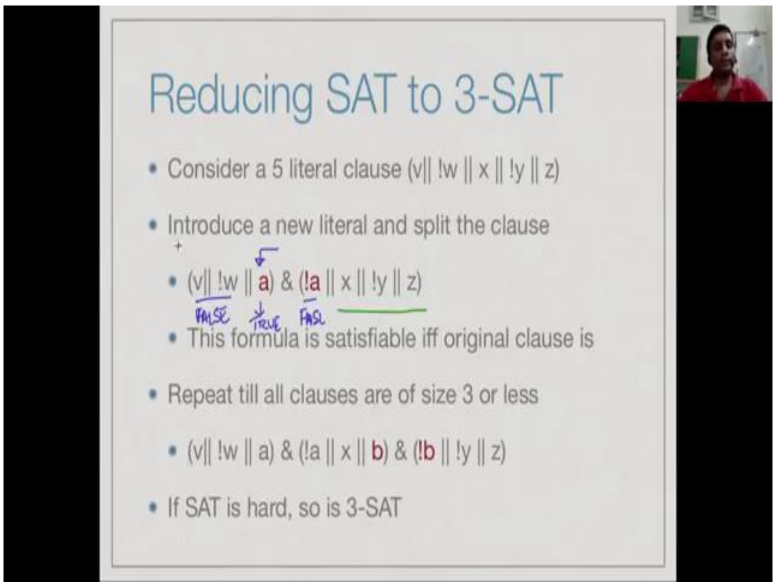
Boolean satisfiability

- Boolean variables x,y,z,...
- Clause – disjunction of literals, (x || !y || z || .... || w)
- Formula – conjunction of clauses, C & D & ... & E
- 3-SAT – each clause has at most 3 literals

So, let us look at these inter reducibility's a little more closely. So, we start with Boolean satisfiability, recall that in Boolean satisfiability, we have clauses, clauses are composed by disjunction of literals, so we take x or not y or z. So, we have these all stunt together to form a clause, put these clauses together with an act. So, in general, we argued that it is hard to think of how to find the satisfying assignment, but if we get an assignment of valuation, values for the variables, we can quickly check whether it makes true or not.

So, restricted form of satisfiability is called 3-SAT. In 3-SAT, we have only clauses like x or not y or z or we could have two also, x or y and so on. So, we could have even trivial things like z, but remember, if we have a clause it says that, the only way to make it, this forces mid to make z true, because there is no option. If I add x or y, I can make x true or y true, so our clause like this does not contribute much more problem, because it forces a valuation for a particular variable, and then I can reduce the problem to one which has 2 or 3. So, 3-SAT has typically two variables or three variables, but not have more than three, it does not have four or five variables.

(Refer Slide Time: 07:14)



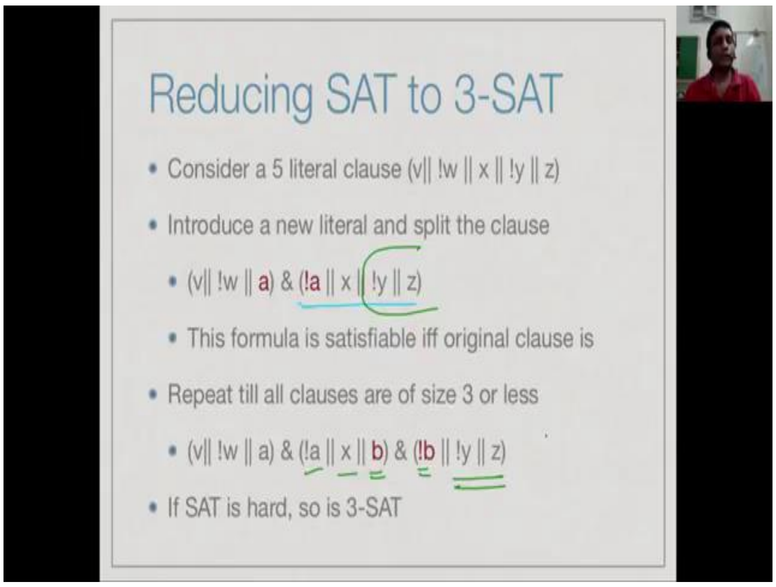
Reducing SAT to 3-SAT

- Consider a 5 literal clause (v|| !w || x || !y || z)
- Introduce a new literal and split the clause
  - (v|| !w || a) & (!a || x || !y || z)
  - This formula is satisfiable iff original clause is
- Repeat till all clauses are of size 3 or less
  - (v|| !w || a) & (!a || x || b) & (!b || !y || z)
- If SAT is hard, so is 3-SAT

So, the first claim is that 3-SAT is as general a SAT. So, we have show this just by a simple example. So, supposing I have a given clause, which has five variables like this. So, what I want to do is, I want to have only three variables, so what I will do, slice of this part. So, I will keep the first two and I slice of this part, so I will introduce a new variable a and I will say it is v or not w or a and I put a not with rest.

So, now what is this tell me, so this tells me that supposing in this case, that these are both false. Then, in order to make this true and must make a true, but a is true, then this is false on the other side, so that means that this clauses is true, one of these has true. So, in other words, it is saying both a v and w, not w are false, then one of the other three variables must be true, which is exactly what the original problem; one of these five must be true. On the other hand, if v or not w is true, then I can effort to set a false, if a is false not even become true, then I am no application of the remaining variables.

(Refer Slide Time: 08:35)



Reducing SAT to 3-SAT

- Consider a 5 literal clause (v|| !w || x || !y || z)
- Introduce a new literal and split the clause
  - (v|| !w || a) & (!a || x || !y || z)
  - This formula is satisfiable iff original clause is
- Repeat till all clauses are of size 3 or less
  - (v|| !w || a) & (!a || x || b) & (!b || !y || z)
- If SAT is hard, so is 3-SAT

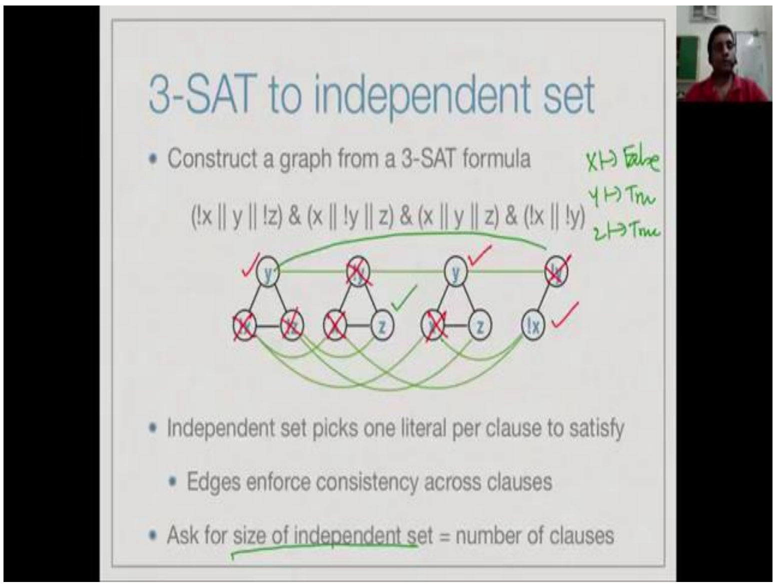
So, the claim is that by this expansion by removing by this one, splitting away the first two variables and adding this in intermediate thing linking the two clauses at produce two clauses, which have same satisfiability as the original clause. Now, again I have unfortunately too many variables here, so what I will do is, I will now again keep two and split the response.

So, I will keep not a x a little new variable b, and then I will take not b and take remaining b, at this point, fortunately I now reduce everything into three variables, I can stop it, if I have four variable still, I have do same thing a little c and not c and so on. So, in this way, I can take any clause which has more than three literals and systematically decompose it by adding new literals into sequence of three literal clauses.

So, I can convert any clause which is not in 3-SAT into a sequence of 3-SAT clauses and how big is sequence, well it is going to be portion to the number of literals in the clause. So, the blow up in my formula is going to be linear in the original formula, that is important, because we said that will be do reductions, we want the reduction part to be efficient to say that, we are not spending so much time of the reduction, other we cannot transfer efficient see claims at those.

So, what this tells us in that, if SAT is hard does we believe, then so is 3-SAT, because I reduce SAT to 3-SAT.

(Refer Slide Time: 10:01)



3-SAT to independent set

- Construct a graph from a 3-SAT formula

  (!x || y || !z) & (x || !y || z) & (x || y || z) & (!x || !y)

  x ↦ False
  y ↦ True
  z ↦ True

- Independent set picks one literal per clause to satisfy
  - Edges enforce consistency across clauses
- Ask for size of independent set = number of clauses

So, far we have seen reductions which are not very perhaps surprising, we have seen independent set and vertex cover, but these are similar problems. Similarly, we have seen SAT and 3-SAT, these are similar problems. Now, what we are going to do is, we are going to bridge the gap between these two different problems. We are going to look at 3-SAT and say that 3-SAT actually you can reduce to independent set, which is the little bit more surprising, because they seen come from variable different domains.

So, here is a typical 3-SAT formula. So, we have 1, 2, 3, 4 clauses and each of them as at most three literal, so this one has two literal, so other one that the… So, since we have 3-SAT, then we have a structure which is bounded by three. So, we will represent every clause by this triangle and in the triangle will put a label which indicates the literal. So, not x is replace by not x is in this thing, y and so this not x or y or z not z is this triangle. Similarly, this is not x or not y or z, so in this way every clause contributes the triangle to may graph.

So, what is this triangle signify, this triangle signifies that if may independent set picks one of these things, say y, then it cannot pick the other two. So, what this is saying is this is going to make exactly one of these things, it is going to identify for me which of the literals true in order to make this formula actually work out with. So, it is kind of giving me a witness from each clause.

So, not it well to be valid witness y is 2 here, then I cannot pick not wide to be the witness to the next hour, because y and not y contribute each other. So, in addition to the triangle, I have these green edges saying that variable and it is compliment will be connected by green edge. So, there should be some more green edges perhaps connecting this y to that not y is, so there should be clause, so this y to that not y.

So, every variable is connected to is compliment to indicate that if one is there, the other cannot be there. So, what we independent set is saying is this is time to pick out a solution, is trying to pick out a solution by looking at each clause. And saying that may be I pick y here, I pick y here, then I am go to rule out these things and I will maybe I can now I pick y there, then I to pick something here. So, maybe I pick x here, if I pick x here, I ruled out that not x, you have to ruled out this not x.

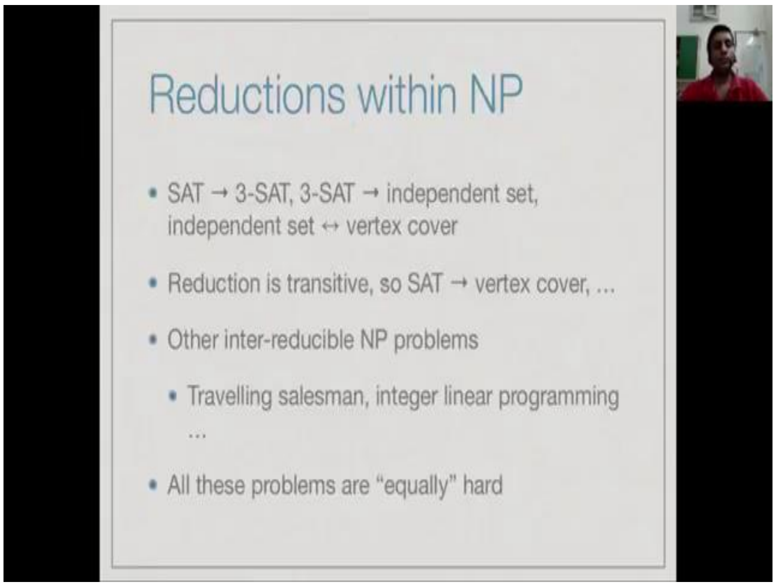
Now, I am trouble, because I do not have anything, so maybe I should could not a pick that x. So, therefore, I go back and say that, this should pick this not x, this not x will ruled out this x, ruled out this x, force meet to keep this z. Now, if I have an independent set of size 4, then I am done, so I can say that the solution now I want is that x goes to false, y goes to true and z goes to true.

So, the existence of independent set on this particular thing of size 4, which picks out 1 per cost. So, notice that within a clause, I can only pick 1. So, if I have four clauses and if I insist on the size are the independent set being the number of process, then I would definitely have got one witness per literal. And those witnesses at we mutually consist, because I connect z and not z x and not x everywhere by edges which rule out the same variable feb by founding reset true in one times, false in one times.

So, this independent set now picks out those literal which four, which make my formula true. So, if there is an independent set of size 4, then every clause can be made true and this set of four clauses. And therefore, I have a solution to 3-SAT and if there is no independent set of size 4, there is no solution. So, 3-SAT reduces independent set which is a bit more surprising, when our reduction as we set from SAT to 3-SAT, which are both similar looking problems.

Over from independent set to vertex cover and backward which are very directly structural related problem in the same type of graph. So, this is the surprising thing is Boolean satisfiability and vertex issues, I am independent vertices and graph are connect.

(Refer Slide Time: 14:26)



Reductions within NP

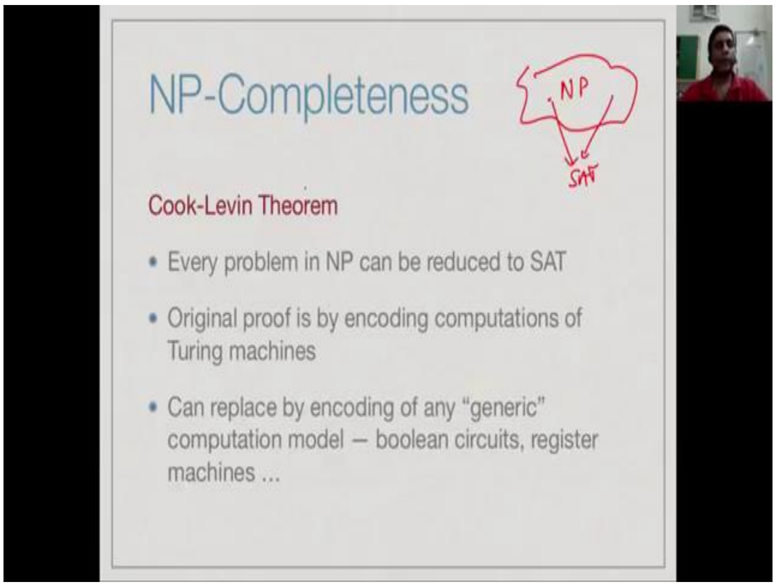
- SAT → 3-SAT, 3-SAT → independent set, independent set ↔ vertex cover
- Reduction is transitive, so SAT → vertex cover, ...
- Other inter-reducible NP problems
  - Travelling salesman, integer linear programming ...
- All these problems are "equally" hard

So, we are shown that 3-SAT reduces 3-SAT, 3-SAT reduces independent set, independent set and vertex cover are mutually reduce able. Therefore, for instance because I can now compose these reductions, SAT now reduces to vertex and you can find in the literature such reduction for various other problem that traveling salesman, integer, linear programming and so on.

Now, all these is important per remember that we are looking in this context to reductions, where the pre processing and post processing step at polynomial time. Because, we want to say there is no time loss in the reduction, therefore, all the complexity is in the solution, if I have a simple solution for b, I have a simple solution for a, when I reduce a to b.

If I have a complete solution for a, then I cannot say that the solution reduction is what may set complex, it is because reduction simple, it must be only the fact with two problems are similar that they are actually having the same complexity. So, given this when all these problems are inter reduce able, it means all of them are equally hard.

(Refer Slide Time: 15:26)



NP-Completeness

Cook-Levin Theorem

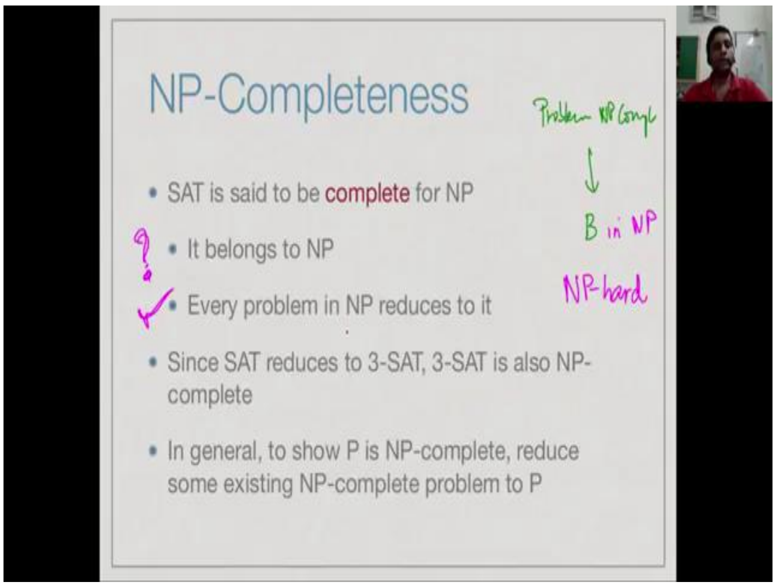
- Every problem in NP can be reduced to SAT
- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines ...

Now, the famous theorem by Cook and Levin says that every problem NP can be reduces to stack. Now, this is the rather amazing theorem, because this is that take everything an NP, anything there I can reduce the SAT. So, there is some reduction by taking an instant that problem, I can converted into next month of SAT, so all those insert of sat and go back it is.

So, we have definitely not going to try and prove this theorem here, it is enough to know that this proof is by encoding a suitably general module of computation. So, the original proofs of Cook use during machines, you can take Boolean circles are register machines, anything which is universal, anything which can compute every problem an NP. And then argue that the computation of that problem, so if something as NP, there is an NP computation, there is a checking problem, checking algorithm.

So, you can encode the behavior of that checking algorithm for that problem in SAT and therefore, it turns out that SAT is equivalent are everything reduces to SAT.

(Refer Slide Time: 16:34)



NP-Completeness

- SAT is said to be **complete** for NP
  - ? It belongs to NP
  - ✓ Every problem in NP reduces to it
- Since SAT reduces to 3-SAT, 3-SAT is also NP-complete
- In general, to show P is NP-complete, reduce some existing NP-complete problem to P

$$\begin{tikzcd}\text{Problem NP Compl} \arrow[d] \\ \text{B in NP}\end{tikzcd}$$

NP-hard

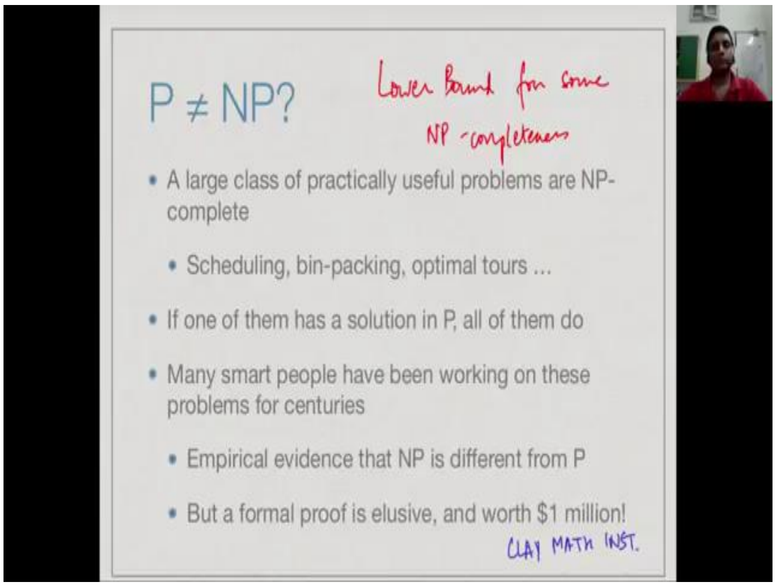
So, this gives us the notion of completeness, we say the SAT is complete, because it belongs to NP and every problem in NP reduces to it, this is the definition of NP completeness. A problem is NP complete, if first of all it has a checking algorithm belongs to NP, efficient checking algorithm. And in addition every problem in NP reduces to it by a polynomial time reduction. So, SAT by definition by the Cook Levin theorem is NP complete, because it belongs to NP, we know.

And the Cook Levin theorem establishes at every problem in NP reduces to it, but now we have seen that sat itself reduces to 3-SAT. So, every problem can be reduce to SAT and then in a further polynomial step reduce to 3-SAT. So, every problem now reduces to 3-SAT, the three sat is also NP, so 3-SAT is also NP complete. So, this is the generic technology now, so you have some problem which is NP complete.

And then I reduce it my problem here that makes b also NP complete, because provide b is an NP, if it is not an NP, then it is something weaker called NP hard, it is at least a hard every problem NP, but may not be NP itself. So, NP hard says that, it is every problem reduces to it, but you are not sure whether not belongs to NP, you do not have a checking algorithm for it.

So, there are few problems like this for which checking algorithm is not easy to describe, but you cannot certainly show reduction. But otherwise if can do both, you can find a checking algorithm and every problem, you have found some problem which in NP which NP complete problem which reduces to it, then it is also NP completed.

(Refer Slide Time: 18:20)



P ≠ NP?

Lower Bound for some NP-completeness

- A large class of practically useful problems are NP-complete
  - Scheduling, bin-packing, optimal tours ...
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries
  - Empirical evidence that NP is different from P
  - But a formal proof is elusive, and worth $1 million!

CLAY MATH INST.

So, this brings us back to the question of whether or not P is equal to NP. So, many useful problems which are extremely important day to day life scheduling, bin packing. So, bin packing is basically now if I want to put various objects into a container in an optimum way or traveling salesman tours, we want find optimal tours, all these are very important real life problems, which people would like the life solve efficiently.

The theory of NP completeness tells us that all of these problems are actually inter reduce able, and therefore equivalent to each other, I can solve one, I can solve them on. Now, we have this experimental evidence that since these are important problem are large number of smart people have been looking at this problems from even before the time that computer science was invented. People have been looking at scheduling problem and bin packing problems for very 100's of years.

Therefore, these problems, if they have not yet yield it and efficient solution, then it looks likely that there is no efficient solution. Therefore, from an empirical point of view from purely experimental evidence, it seems to be strongly suggest that N P is different from P. But this is not a proof unfortunately, mathematically one has to be argue that N P is different from P by giving a proof by saying that is the reason why problems some problems some N P complete problem can never be solve using a polynomial time. This is what is called as lower bound, so we need to establish a lower bound.

For some N P complete problem and this is very hard, this is very proved challenging, N P completeness was identified by Cook and Levin with early 1970's. So, it is now more than 40 years, since the theory of N P completeness itself was invented and a lot of people and looks at problem and nobody have yet found a direct way of doing this. So, formal proof is elusive and is worth a million dollars, because there is a prize given by the Clay Math Institute.

So, this is one of the celebrated open problems in Mathematics. So, with the Clay Institute has awarded a large prize. So, somebody does prove that this is indeed not the case, then there is a large prize at stake. So, with this it seems an appropriate place to end this course.

So, I thank you all of this time.