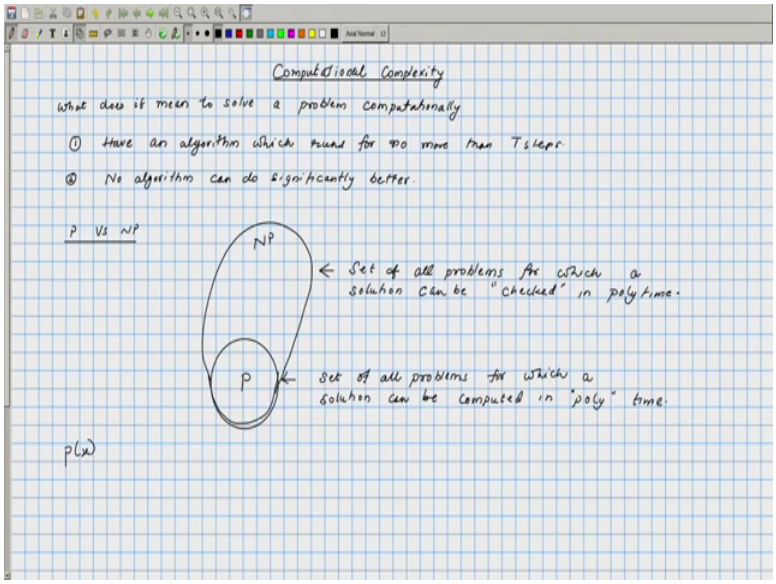
**Randomized Algorithms**
**Prof. Benny George Kenkireth**
**Department of Computer Science & Engineering**
**Indian Institute of Technology, Guwahati**

**Lecture – 22**
**Introduction to Computational Complexity**

In today's lecture, we will learn about some notions in Computational Complexity.

(Refer Slide Time: 00:36)




So, the basic question that we ask ourselves is what does it mean to solve a problem computationally? So, when we say that we have solved a problem from a computational point of view, what do we mean?. The first requirement is we should have a program which runs in certain number of steps. So, have an algorithm which takes no more than let us say some T steps which runs for no more than T steps ok, when we have one such algorithm natural question is are the algorithms which do even better than this.

So, we would like to say some things give some guarantee that, no algorithm can do significantly better significantly better, cannot take much smaller than T steps ok. So, here we are addressing the time taken for solving the problem one could also look at other resources like space or the amount of randomness used or the amount of communication used and so on.

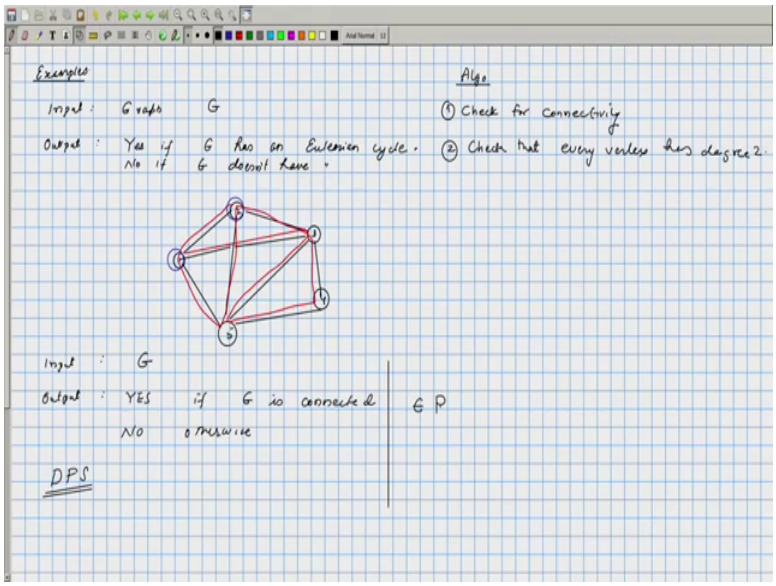
So, it is natural to look at various problems and look at which problem is computationally more difficult. If we were looking at time complexity classes it would mean which problems takes more time compared to other ok. So, first of all you need to have an algorithm and then you want to say that algorithm is optimal and when we are comparing two problems we will say that one problem is harder if the optimal algorithm for one takes significantly more time than the optimal algorithm for the other problem ok.

So, computational complexity essentially studies or classifies problem based on their computational difficulty ok. So, the famous question P versus NP basically arises when we are trying to classify problems on the basis of their computational difficulty. So, informally we can just look at the collection of problems called P and that is nothing, but this is the set of all problems for which solution can be computed or found in polynomial time ok. So, we have not formally defined what a problem is, we will see some examples and that will make things clearer later on ok.

So, the collection of all problems for which we have an algorithm which works in polynomial time. So, if the input is let us say of size x the running time the algorithm should be bounded by P x, where P is some particular polynomial and NP is a superset of this and this consist of all problems for which a solution can be checked in polynomial time can be checked or verified ok.

So, what does it mean to check a solution when we are thinking about these problems? If somebody claims that a particular problem has a particular solution, finding the solution may be difficult, but if one can verify that a purported solution as a correct solution in polynomial time then we will say that it belongs to NP.

(Refer Slide Time: 05:29)




So, let us see some examples and that will make things clear. Take an example of a couple of problems. So, let us look at one problem where the input is a graph ok. The input graph we call it as G and what we want is the following output. The output should be yes, if G has an Eulerian cycle and the answer should be no, if G does not have an Eulerian cycle.

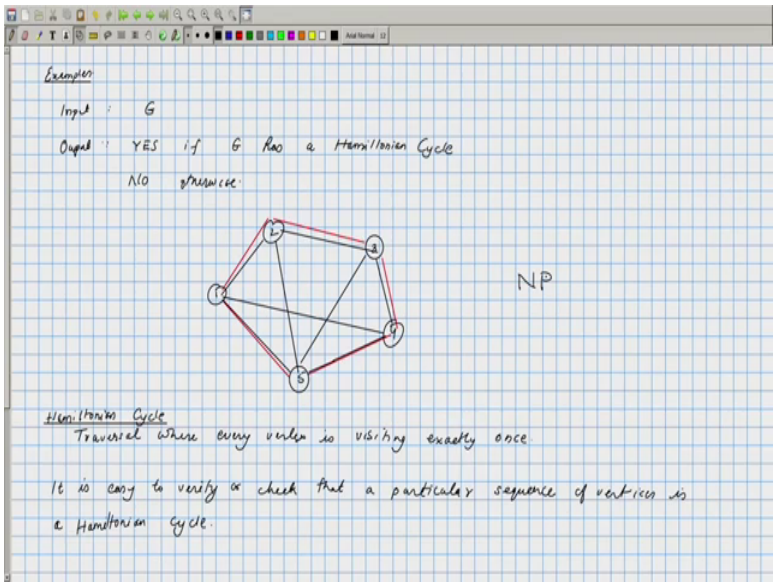
So, let us take an example. This is a graph on five vertices. What we want to know is whether we can start at a particular vertex and traverse every edge of this graph without visiting any edge twice. For example, we could start of at vertex 1 and traverse along this path and then come back here ok. So, we could traverse at this way and we have visited every vertex, but the starting vertex was 1 and the ending vertex was 2. Can we do a traversal says at the starting vertex and the ending vertex are the same? In this graph one can show that it is not possible to do this ok.

So, Eulerian cycle is a traversal of the edges such that no edge of the graph is traverse more than once or in other words every edge has to be visited exactly once ok. So, that is an Eulerian cycle. So, we want a program or an algorithm which takes graphs as input and tells whether the graph has an Eulerian cycle or not and the computations should work in poly time and there are algorithms to do it. All the one has to check is every vertex is of even degree.

So, this should be the algorithm basically checks for connectivity and then check for connectivity; if you take two disconnected graphs and if both of them have Eulerian cycle that will not be considered as an Eulerian cycle because we do not really have a cycle it is part of two different components. So, we check for connectivity and check that every vertex has degree 2 ok. So, this is a polynomial time algorithm ok. So, one can see that the solution can be computed based on this algorithm one can in fact, find an Eulerian cycle and if somebody gives an Eulerian cycle we can even check it. So, both finding and checking are easy.

Let us take another example the input is a graph again and the output should be YES if G is connected and NO otherwise. The familiar DFS algorithm can be modified to check of the graph is connected. You start at any particular vertex and do the DFS and keep track of the number of vertices visited. If it is equal to the total number of vertices in the graph then it is a connected graph otherwise it is not a connected graph. So, this problem also belongs to P. There is a polynomial time algorithm which solves this problem.

(Refer Slide Time: 09:57)



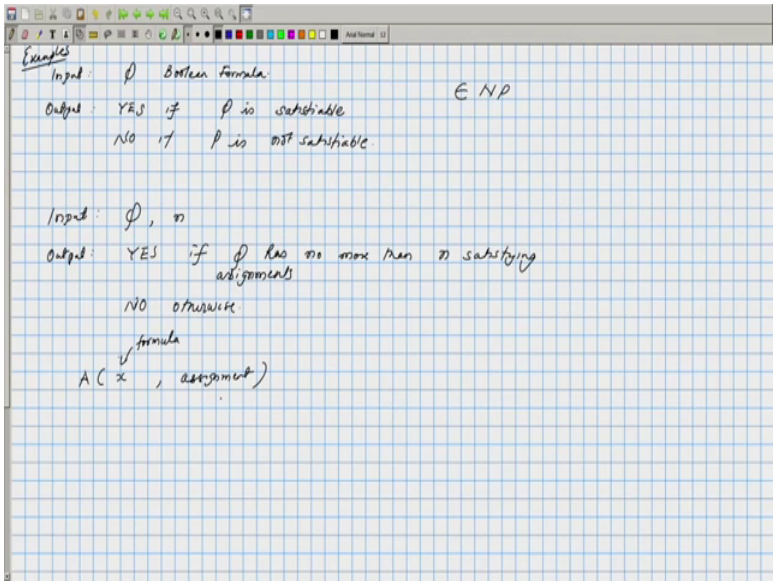

Let us look at another problem. So, here the input is again a graph G and the output is YES if G has a Hamiltonian cycle and it is NO otherwise ok. So, let us take an example. So, we want to know whether we want a traversal where every vertex is visited exactly once, and no vertex should be visited twice. So, Hamiltonian cycle this is a traversal where every vertex is visited exactly once ok. So, you can ignore the fact that if you start

at the vertex you come back to the vertex. So, when you come back to the same vertex that is just counted as once ok. So, you start at the vertex keep on traversing and come back to the same vertex.

So, in this graph it is possible for example, you could not just do this particular traversal this visits every vertex once you can imagine that there is a cycle on which no vertex appear more than once and every vertex appear at least once ok. So, given a graph we want to check if it contains a Hamiltonian cycle. There no known good algorithms to solve this problem, but we can see that if somebody has given a particular sequence of vertex one can easily check that the answer is correct. So, it is easy to verify or check that a particular sequence of vertices is a Hamiltonian cycle. It is not clear whether there are good algorithms to find one such. So, this is a problem where verification is easy ok.

So, we will say that the verification can in fact, be done in polynomial time because we just need to check whether all the vertices which are next to each other in the path we need to verify whether they are connected in the graph or not. Let us look at couple of other problems as well.

(Refer Slide Time: 13:27)

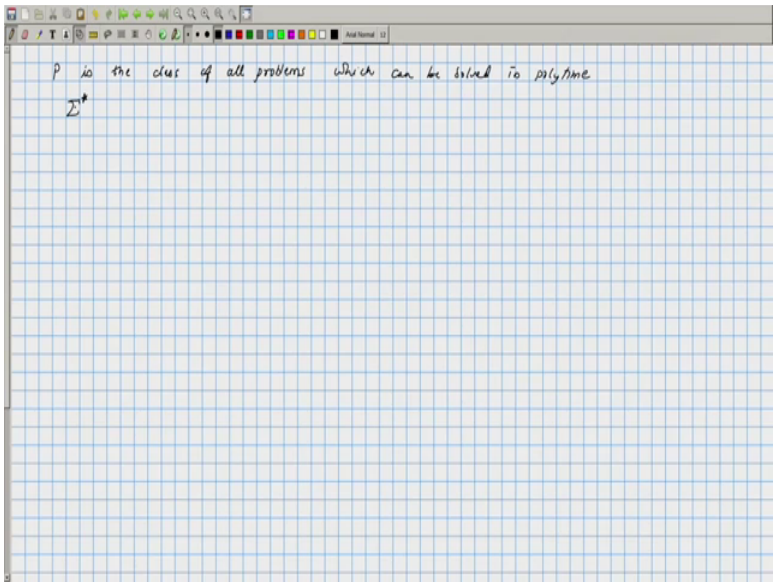



Now, the input is going to be a Boolean formula and the output should be YES if phi is satisfiable and it should be No if phi is not satisfiable. So, given any Boolean formula we can and a particular assignment one can easily check if it is satisfiable whereas, finding

that assignment is not so straightforward. Again it is not known whether there are good algorithms to do that ok. So, this is again a problem belonging to NP.

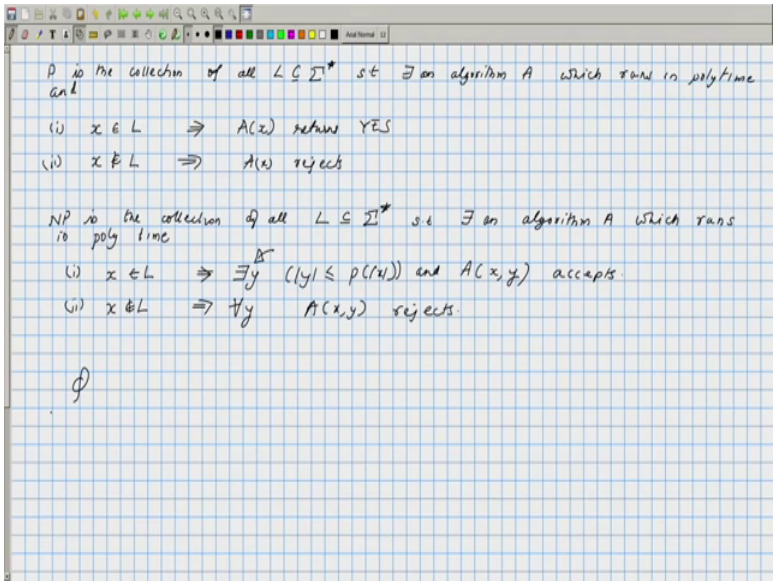
Let us look at another example. The input now will be Boolean formula and a number n and the output should be YES if phi has no more than n satisfying assignments and NO otherwise. In this case it is not even clear that we can verify this part easily. Suppose, somebody gives you n satisfying assignments we need to figure out that these n satisfying assignments are let us say the only satisfying assignments for this particular formula. Now, it is not clear if there is a polynomial time algorithm for solving this problem and we do not know whether this belongs to NP as well.

(Refer Slide Time: 15:48)




So, now since we have seen many different problems we can formally define what is P? So, P is the class of all problems or you can say P is the class of all problems which can be solved in poly time. If we take sigma star as our universe what we require is the following.

(Refer Slide Time: 16:19)




P is the collection of all L subset of sigma star such that there exists an algorithm A which runs in poly time and we require two conditions; x belongs to L implies A x returns YES or it accepts and x does not belong to L implies A x rejects or says NO saying returns YES means accept, rejects means return NO ok.

So, all the languages for which you can do this is the collection called P, when we think of graphs with containing Eulerian cycles language L there would consist of all strings which are valid encodings of graphs which have an Eulerian cycle in them and NP is the collection of all L belonging to sigma star such that again there should exist an algorithm which runs in polynomial time.

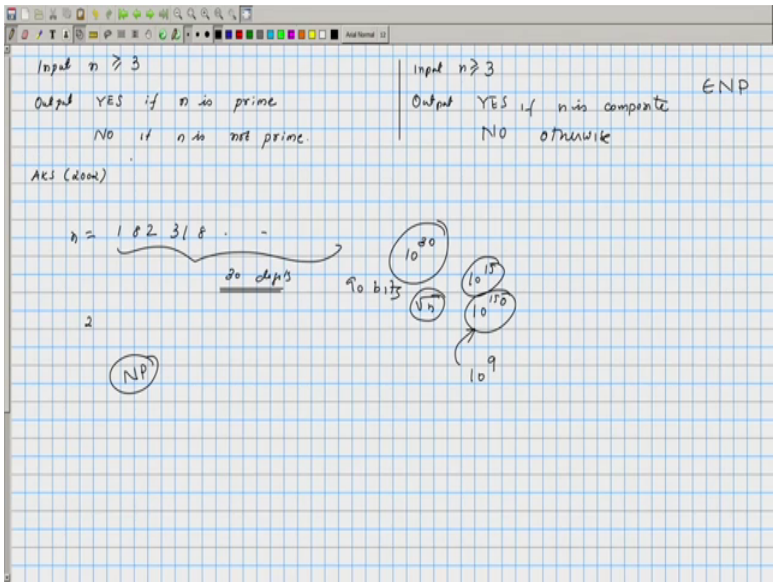
And then there are two requirements; if x belongs to L and that would imply there exists a y and the length of this y is bounded by a polynomial in the size of x. So, y should be less than P of x. And the algorithm now the algorithm has two inputs into A x, y accepts when x does not belong to L this would imply that for all y A x, y rejects ok. So, this y you can think of it as a certificate or the proof that x belongs to L.

So, when we look at our formula phi and we wanted to look at this problem is phi satisfiable we can think of the algorithm A as an algorithm which takes two inputs; the first input as a formula and the second part is the assignment ok. So, it takes these two input and then evaluates the formula at that particular assignment that evaluation can be done in polynomial time.

And note that for any satisfiable formula there will exist an assignment and the length of the assignment is at most equal to the number of variables in the formula which is bounded by a polynomial in the length of the input and A x, y is going to accept because that was a satisfying assignment. Whereas, for an unsatisfiable formula whatever assignment you give the algorithm is going to find that that particular assignment no longer does not satisfy the formula and therefore, A x, y rejects ok. So, that is a problem which belongs to NP and this is the formal definition.

Now, we will see a classic problem and see that it belongs to NP. The problem that we look as the following.

(Refer Slide Time: 20:55)




The input is a number n and the output should be YES, if n is prime and it should be NO, if n is not prime or if it is composite we want to say no and we will assume that the input is greater than or equal to 3.

Now, then currently there are very good algorithms to do this. There are polynomial time algorithms the so, this is a problem which belongs to P, but that was a recent result came out. In the AKS algorithm is a polynomial time algorithm which solves this problem. So, we will look at what was known before 2002. So, in order to understand the difficulty here, let us look at this problem a bit more carefully.

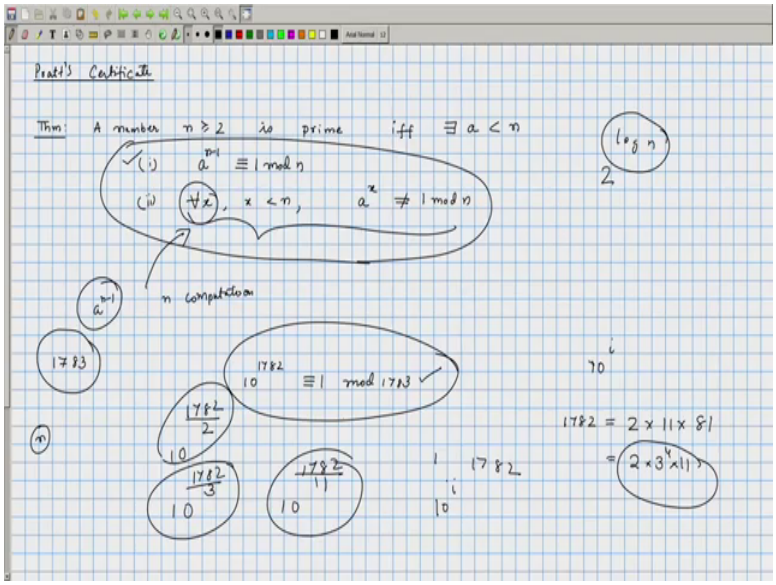
Let us say I give you a 30 digit number ok; some number which is 30 digit. A number is about 10 to the power 30 in size. We could run a naive algorithm which starts at the number 2 and then repeatedly checks whether the number divides the input n. If you find the number which divides n then we know that this is not a prime number this is a composite number ok. Now, this naive algorithm is not computationally efficient in the sense we will never be able to run this even on modern day computers ok.

So, we are basically checking till let us say if you check till n by 2 or even square root n there is a factor which is no larger than root n ok. So, we are going to check till root n. So, root n root 10 to the power 30 is about 10 to the power 15 ok. If we had a 300 digit number then this is something like 10 to the power 150 ok. This is the number of divisions that you have to do if we do 10 to the power 9 computations per second even then we will not be able to reach anywhere near 10 to the power 150 ok. So, brute force algorithm clearly does not work and that is precisely the problem here.

When we measure the input the input is measured in terms of the size of the number of bits required to represent the number. So, 30 digits would essentially mean 90 900 mean 90 bits something like that. So, what is the way out? Is there some way by which somebody can convince me that this number is a prime, without having to check all the numbers up to square root n.. So, maybe to show that this belongs to P is difficult, but we can be try and show that this belongs to NP; is there small certificate that is what we will be seen. If you look at the complement problem where the input is again the same and the output is going to be YES if n is not prime or composite and NO otherwise ok.

We look at this complement problem, then its clear that this belongs to NP because for any number if it is composite it has a factor and the algorithm just takes that factor and takes that number and checks that if that it is really the factor. So, compositeness or the set of all composite numbers they do belong to NP. Can we do a similar thing for primes, that is something known as Pratt's certificate.

(Refer Slide Time: 25:24)




This is based on the following theorem. A number say n greater than or equal to 2 is prime if and only if there exists a number a which is less than n such that a to the n minus 1 should be congruent to 1 mod n; that means, compute a to the power n minus 1 and divided by n the remainder should be 1, that is the first requirement. And then the second requirement is for all x, x less than n, a to the power x should not be 1 mod n. So, if these conditions are true then the number is prime ok.

How do we come up with a non-deterministic algorithm or an algorithm which certifies that this number is prime. So, this theorem asserts that there is surely for every prime number which satisfies these conditions. So, if we can take the day and do these computations that would suffice, but again here there is a lot of computations. First of all we have to compute a to the power n minus 1 and then we have to compute.

So, this probably by repeated multiplications we can compute, but here we have a lot of computations; we have n computations in the second step ok. The input size is something like log n because you are representing it in let us say binary. So, there are log n digits and when you say n computations, that is, something like 2 to the log n ok. So, that is not really a good way to check this.

Pratt's certificate basically overcomes having to check for all the x's here it says that if you check for a subset of x's that is as good as checking for all x ok. So, we will do this with an example. So, suppose given the number 1783 as input this is actually a prime

number. Now, how do I prove to somebody that this is a prime without him having to check for all the factors till under root of 1783? The first thing that I will ask somebody to check that 10 to the power 1783 is congruent to 1 mod n.

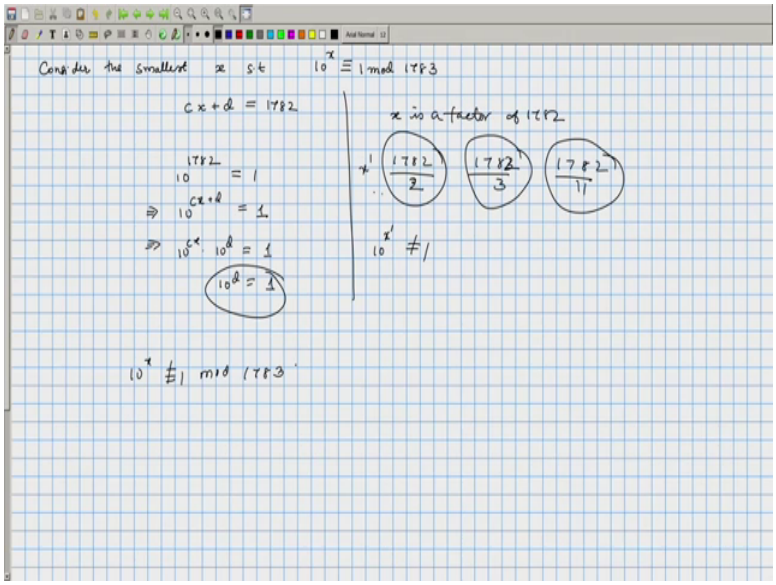
This calculation is not very difficult because we can do this by repeated multiplication. So, this can be checked ok. So, that basically checks the first thing, but we are essentially asserting is that when a is equal to 10 that will certify that 1783 is prime. So, 10 to the 1782 somebody can compute ok, anybody can just do this computation by a calculator the value of n here is 1783 ok.

But, now how do I know that 10 to the power i, where i is varying from 1 to 1782, none of them will give me 1. So, instead of checking for all of them I will check for only few numbers and these numbers are nothing, but they are dependent on 1782. So, 1782 one can factorize it ok. So, somebody tells me that this is it is factorization 2 into 11 into 81 that is equal to 2 into 3 raise to 4 into 11 you need not believe me on this you can just multiply yourselves and check that 1782 indeed factorizes.

So, any number n if we want to check for want to verify that it is prime the proven must provide you with a factorization for n minus 1 ok. So, 1782 can be in fact, written as 2 into 3 power 4 into 11 and we can compute 10 to the 1782 by 2; this is again a computation of this kind ok, repeated multiplication or we can do fast exponentiation and we will also check 10 raise to 1782 by 3 and 10 raise to 1782 by 11. So, these are three computations which involve further multiplications ok.

But, basically we did one such computation for each prime factor of 1782. What we will show is if none of these are 1, then there cannot be any other number between 1 and 1782 such that 10 to the i is going to be 1. So, we need not check for all these 1782 numbers we need to just check for three numbers the three numbers being 1782 by P i, where P i is the prime factor of 1782 ok. Why would this suffice?

(Refer Slide Time: 31:51)




So, consider the smallest x such that 10 to the power x is congruent to 1 mod 1783. Now, you can solve this equation cx plus d is equal to 1782 ok. So, just take 1782 divided by x and you will get your quotient and remainder ok. So, now let us look at 10 to the 1782. We have verified that 10 to the 1782 is 1. So, this is equal to 1 mod 1783 we will not write it each time. So, all these calculations are done modulo 1783.

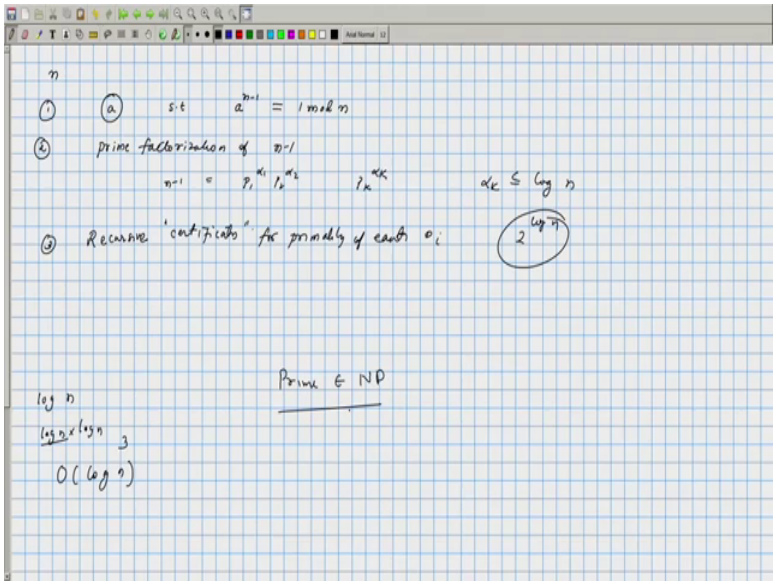
So, this would imply that 10 to the cx plus d this is equal to 1 mod 1782; that would imply 10 to the cx times 10 to the d is equal to 1 mod 1782 ok. Now, 10 to the x is 1, so, 10 to the cx is also 1. So, this would imply 10 to the d is equal to 1. Now, d is a reminder that you get when you divide 1782 by x. So, d has to be less than x, but 10 to the d is 1. So, the smallest and we had assumed that x is smallest, so, d must essentially be 0 ok. If d is 0, that would mean that x is a factor of 1782.

So, look at all the factors of 1782, three of those factors were 1782 by 2, 1782 by 3 and 1782 by 11. For each of these factors we had verified that. So, let us call these factors these as x primes. For each of these x primes we have verified that 10 to the power x prime is not equal to 1. Now, if 10 to the power x is any other factor check that e those x's will surely divide it one of these factors because these are the largest factors of 1782, other than 1782 ok. So, if none of them gives 1 when 10 is raised to those powers, we can guarantee that none of the smaller factors would be giving 1 ok. So, this would imply

that and if you verify for these, then none of the smaller factors are going to matter ok; so, that is basically the proof.

So, this means that in order to check that 10 to the power x is not equal to 1 mod 1783, we need to check only for three numbers namely 1782 by 2, 1782 by 3 and 1782 by 11 ok. Now, you can extend this to a general scheme.

(Refer Slide Time: 35:26)




So, if your number is n and if somebody wants to show that this is a prime number, the verification step would first what needs to be provided as certificate for the primality of n would be an a such that a to the power n minus 1 is equal to 1 mod n and then the second component would be prime factorization of n minus 1 ok. So, that means, n minus 1 should be presented as P 1 to the alpha 1, P 2 to the alpha 2, P k to the alpha k.

Note that each of these P 1, P 2, P k are going to be prime factors. So, the maximum number of alpha case is going to be. So, alpha k is going to be less than log n; if n was a number then it cannot have more than log n distinct prime factor because each prime factor is at least 2; so, 2 into 2. So, product is going to be at least 2 raise to log n which is n, ok. So, number of distinct prime factors is less than log n. So, these P 1, P 2, P n are going to be given.

So, the certificate would basically have a component a which is at most log n digits long because its a single number and then log n numbers which are factors; if somebody gives

you P 1, P 2, P k you can multiply and see if that is indeed a factorization, ok. So, that is going to be something of the length log n times let us say log n. Each of the factors is at most log n bits long and there can be at most log n factors. So, these are the two parts and then we need to have a box which will basically be a recursive called for which certifies the primality of each P 1.

So, recursive certificates for primality of each P i ok. Together all these things we can account for and we can argue that the total length of this is going to be bounded by log n to the power 3 and that is a polynomial time algorithm in terms of decides the input. So, this is known as the Pratt certificate and that would show that primes belongs to NP. We will stop here and we will continue on primality testing in the coming classes.