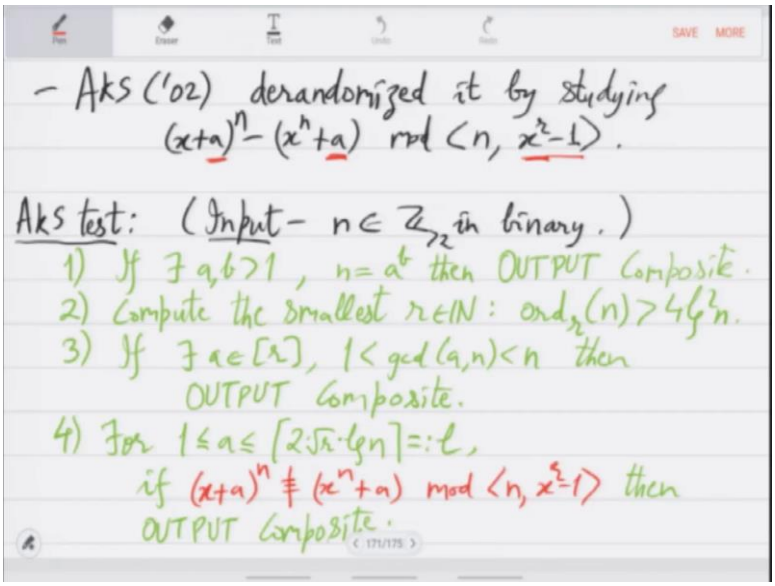
Computational Number Theory and Algebra
Prof. Nitin Saxena
Department of Aerospace Engineering
Indian Institute of Technology - Kanpur

Lecture – 26
Integer Factoring Smooth Numbers and Pollard's Rho Method

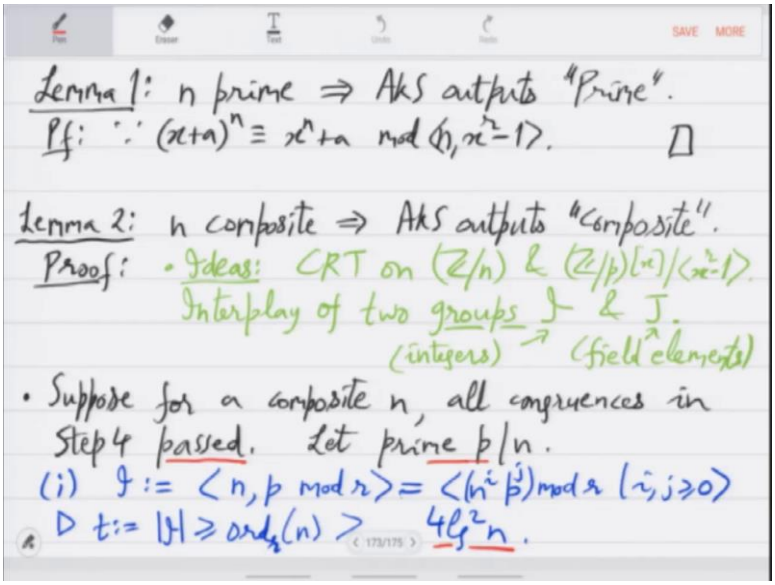
(Refer Slide Time: 00:20)



So last time I wrote this algorithm this AKS Primality test in the input you are given a number n in binary. So first is just this preprocessing you check whether n is a perfect power or whether n is even and then if it is you output composite of course if not then the the algorithm the main part of the algorithm starts. So it finds a number r such that order of n mod r is large which is which should be at least 4 log square n + 1 or more.

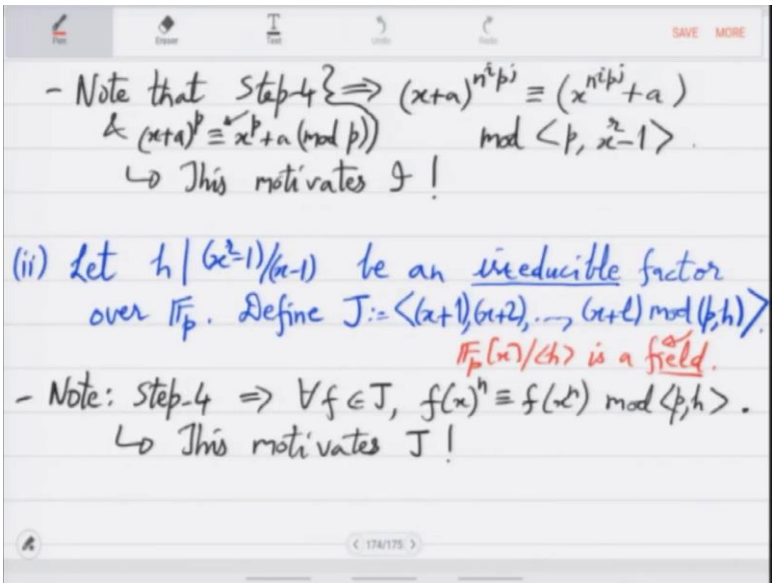
And then this red check is done for several a's you check whether x + a to the n is x to the n + a modulo x to the r - 1, n. If any of these tests fail then you are sure that n is composite because it should have passed if n was prime. If all the tests pass for a 1 to 2 square root r log n then you give up and you say that n is prime. Now we have to show that when we say prime it is n is indeed prime there is no mistake.

(Refer Slide Time: 01:35)



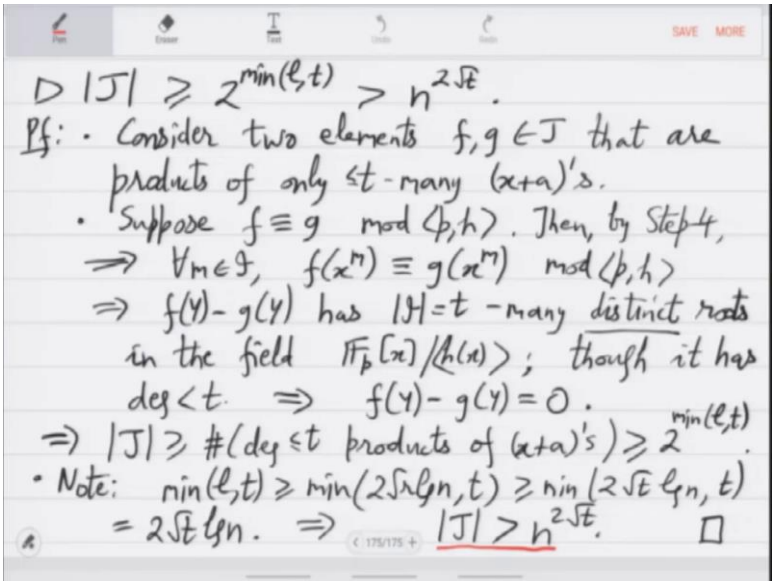
So for that we identified two groups the first group is group of numbers which is i so we identified this subgroup i which is the numbers generated by n. So, prime p divides n so look at this now sub group generated by n and p mod r. So basically these are product of n power p power and this group has size at least 4 log square n by assumption.

(Refer Slide Time: 02:11)



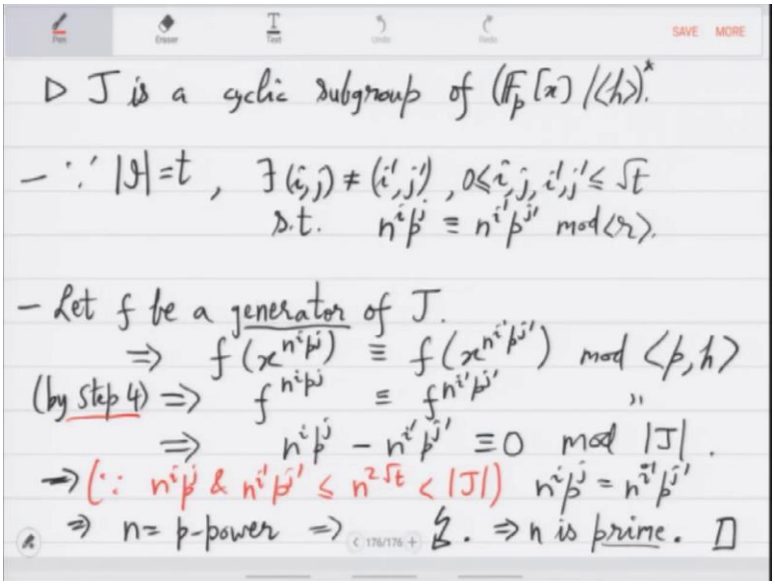
The second subgroup is that of field elements we are calling it J so this is the subgroup generated by these polynomials for which the test was done x + 1 dot dot x + l but viewed as field element so mod p, h where h is an irreducible factor of x to the r - 1 mod p. This also is a large subgroup.

(Refer Slide Time: 02:47)



So we showed that J is at least n raised to square root t two square root t. In fact so now let us use these two lower bounds these two big subgroups to get what we want the correctness proof of the algorithm.

(Refer Slide Time: 03:08)



So note that J is a cyclic subgroup of the field why is that? Well because this multiplicative subgroup of the finite field is cyclic so every subgroup is cyclic J is J in particular is also cyclic. So the way you can use it is since the size of i the subgroup i is t there exist two different pairs between all smaller than square root t at most square root t such that n raised to i p raised to j is n raise to i prime p raise to j prime mod r.

Right this is true because the number of n raised to i p raise to j is more than square root t times square root t which is more than t but the size of the subgroup i is only t which means

that two elements will be the same. So we can find i j and i prime j prime says that they are the same mode r but i j and i prime j prime are different pairs. Now for these two pairs what can you deduce.

So let f be a generator of j j is a cyclic subgroup pick a generator we call it f then you know that f of x to the n to the i p raise to j is the same as f of x to the n to the i prime p to the j prime in the finite field right simply because this n to the i p raised to j and r is to i prime p raised to j prime they are the same mod r. So these x powers are the same so f evaluations are the same. But then by step four of the algorithm what is step four of the algorithm?

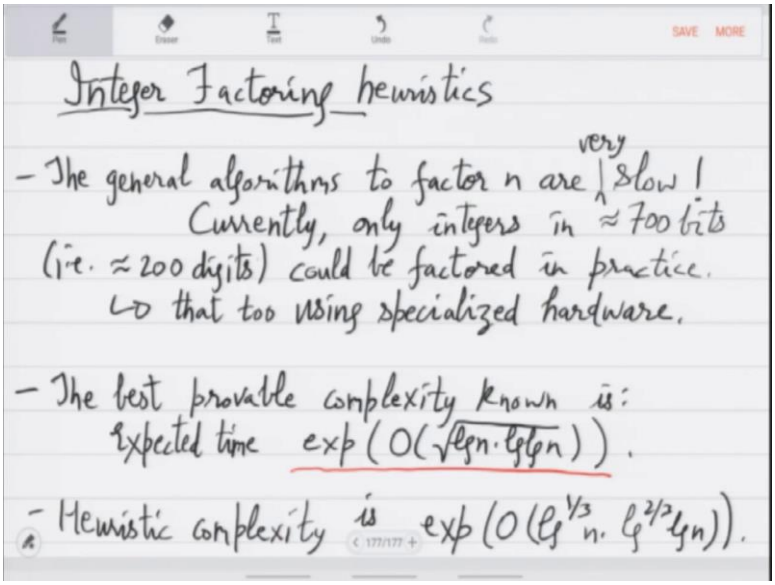
Step four is this red test. The red test is passing. So if x + a to the n is x to the n + a then since f is a product of such things you get that LHS will be equal to f raised to n raised to i p raised to j and RHS will be equal to f raised to n raised to i prime p raised to j prime mod p, h. But f is an element of high order right so this means that in fact size of j should divide n raise to i p raise to j - p raise to j prime right.

So this is a very important step this is the key step which will finish the proof because we are deducing from the fact that these two products are same mod are we are deducing that in fact they are the same modulo a much bigger number which is size of J. So what does that mean so since n raised to i p raised to j and n raised to i prime p raised to j prime both of them in value is smaller than n raised to 2 root t which is smaller than the size of j.

What you deduce is that they are absolutely equal not just mod absolute equality which means that n is a power of p but n cannot be a perfect power that we have ruled out in preprocessing step one. So that gives you the contradiction. So this means that if step 4 passes then n has to be prime that finishes the proof. So these algorithm steps one to four this algorithm takes remember log to the 10.5 n time this is the time complexity.

And if n is composite it outputs composite if n is prime it outputs prime right. So that is the full proof of a case Primality test. So you have seen some very practical Primality test you have seen this deterministic polynomial time Primality test. Next question is once you have verified that n is composite how do you find factors integer factoring.

(Refer Slide Time: 10:31)



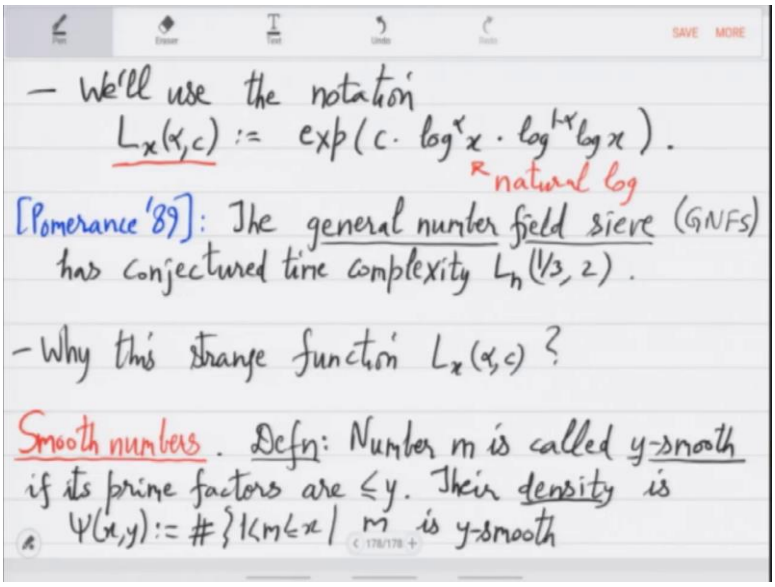
Let us start with that so this area of integer factoring consists of only heuristics. The formal algorithms and formal analysis is currently missing even at the level of heuristics these algorithms are not fast but whatever there is will study. So that it gives you an idea of how possibly you can factor integers that are not too large. So the general algorithms to factor n are slow in fact very slow.

So, currently only integers in around 700 bits which in terms of decimal digits is 200 digits. So, around integers of around 200 digits up to that could be factored that too using specialized hardware, so 200 digit long integers are still quite large they are like 10 raised to 200 in value it is amazing that they can be factored in practice but anything bigger than that it is very hard to get to an answer.

Even for specialized hardware this is a very tough problem it takes forever to find a prime factor or any factor. Theoretically the situation is quite bad so the best provable complexity that is known is expected time. So all these are randomized algorithms heuristics expected time is 2 raise to big O of log n times log log n square root. So compare this with brute force brute force is exponential in log n and here you have square root of that in the exponent.

So it is certainly faster but it is still not polynomial time and it is not very practical if log n is large. And heuristic complexity is better is one third here and two third here. So heuristic is a kind of one third of log n, log n to the one third. Provable is log n to the half and this is happening in the exponent. So this is a weird complexity function right we have not seen this before.

(Refer Slide Time: 15:26)



Let us talk about it a bit. We will use the notation L x alpha, c to denote this exponential in c times log to the alpha x log to the 1 - alpha log x and to be sure this log is natural log that is important because we want to really talk about the constant multiples as well here because that may change the practice of it how fast it will be in practice. So, we want c to be small and also we also want this log to the alpha x to be small log x to the alpha to be small.

So the best algorithm known is due to an analysis by pomerance it is called the general number field sieve in short GNFS number field sieve is GNFS generalized number field c with GNFS. So using a GNFS you can factor integers in the fastest possible way currently it has conjectured time complexity L n one-third two so exponential in 2 times log x to the log n to the one-third times log log x log log n to the two thirds the dominating part is log to the one third that is the conjectured complexity of GNFS.

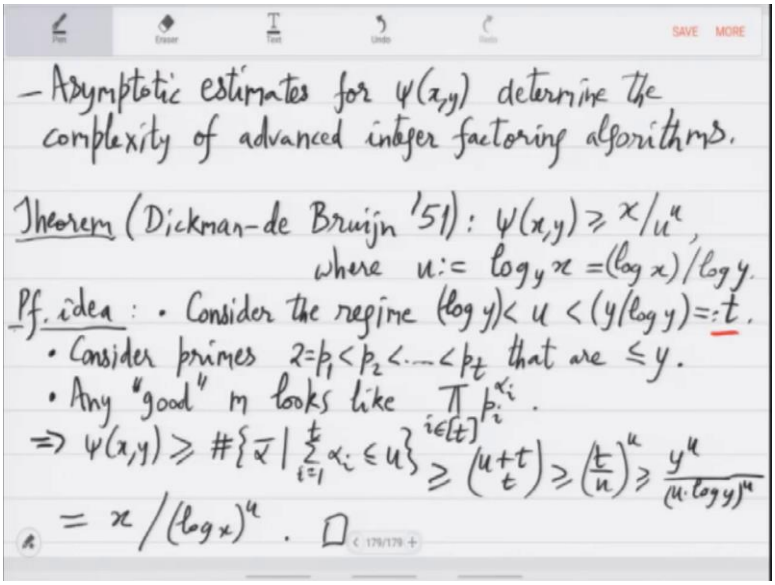
So in these two lectures last remaining lectures you will get a good idea of how this is done you will not be able to see all the details but you will get the basic idea. So let us start with the with the reason why this function appears? And the reason for this is actually quite interesting and a very important topic it is the topic of smooth numbers. Smooth numbers are basically numbers that have small prime factors.

So number n is called y smooth if all its prime factors are less than equal to y then we call let me call it m so number m is called y smooth if all the prime factors of m every prime factor is smaller than less than equal to y. Their density will be important how many numbers below x

are y smooth. So their density is denoted by psi x, y and this is the number of m such that m is y smooth.

So psi x, y divided by x you can say is the density. So we are interested in this as a function of x and y. So it is actually this density that decides the complexity of integer factoring algorithms.

(Refer Slide Time: 21:31)



So asymptotic estimates for psi x, y determine the complexity of advanced integer factoring algorithms. So let us first see a see a proof sketch of a good estimate of psi. So this was shown by Dickman de Bruijn it is an old result. So it says that psi x, y is at least x divided by u to the u where u is log of x base y which is also the same as log x by log y. So in terms of this ratio of log x log y the number of so the density of y smooth numbers up to x is u raise to – u.

So let us see a proof sketch it will not give the full result but it will give a decent estimate and you will see y u to the u and then why this L function appears. Consider the regime so let us take u to be at least log y. So u is not very small and we also do not want u to be very large in the proof. So we want we our proof will work when u is not very small and it is not very large. So for example this in particular means that y has to be large as well y cannot be too small because u less than y over log y means that y has to be more than log x.

So if y is smaller than log x then the proof will not work and similarly y should not be too large because this first inequality means that log y should be less than or log y whole square

should be less than log x so y should not be too large. So under this in this regime the proof works what is the proof actually now the proof is very simple the proof idea is very simple. It is that you consider the primes that are less than equal to y.

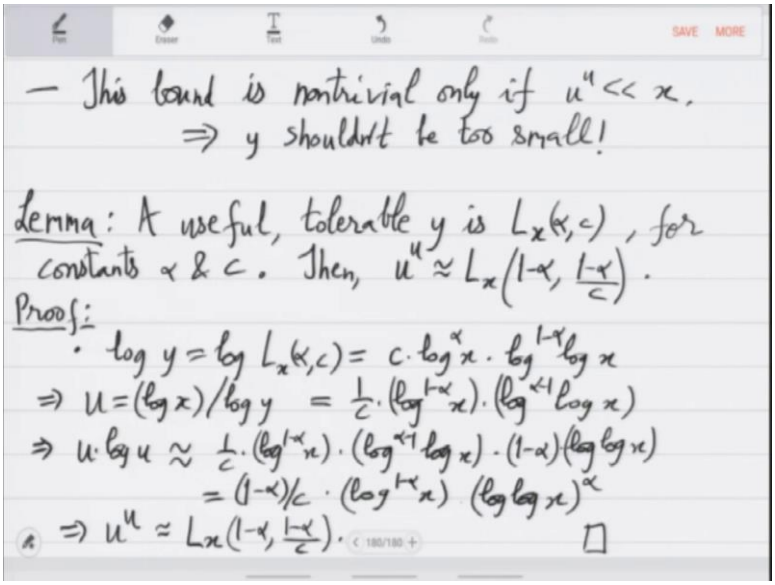
So consider all the primes that are y or less we call them p 1 to p t. Notice that the prime numbers are they are asymptotically they are like y over log y so they are around t many. So let us enumerate them p 1 to p t. And now these are the this is kind of the base on which you will build smooth numbers m right so you will multiply these primes with repetition. So any y smooth number m will look like product of pi to the alpha e.

So this means that the number of numbers such numbers you will get below x if you take alpha i's appropriate alpha i's then this product will be below x right. So in particular you can use these alpha i's such that the sum of alpha i's less than equal to u because u is log x by log of x to base y p i's are all smaller than y right. So if the sum of alpha i is less than equal to u then this product of p i alpha is smaller than x.

That was the reason we defined u actually like that and this then is at least you use the binomial estimate. So u + t choose t then you use this simple estimate of by u to the u t is in our regime t is bigger than u so u + t choose t will be t by u raised to u and then you substitute for t so you get y raised to u divided by u times log y to the u which is what is y to the u? y to the u is x what is u times log y?

That is log x so that is that is some decent estimate it is still not like u to the u because u is actually smaller than log x but we can stop here because x by log x to the u is giving us some idea of how many y smooth numbers there are below x. But this can be made more refined and then you can get this Dickman the Bruijns estimate. So let us assume that it will work.

(Refer Slide Time: 30:27)



So then using that theorem or estimate first thing you can observe is this bound is non-trivial only if u raise to u is sufficiently smaller than x otherwise x over u raised to u will become one or less that would not make much sense. But if u raise to u is much smaller than x over u raised to u is a decent quantity so that is a good lower bound. And for that to happen y should not be too small.

If y is too small like a constant then your u is log x and then u raised to u will be much more than x. So why should it be too small it should not be too large right that that was our regime anyways in the proof. So let us clearly write down u as a function of x and y which will work and then you will see the L function so a useful tolerable y is l alpha sorry L x alpha, c for constants alpha c.

So x is the variable if you take alpha c to be absolute constants then for this y, y equal to L x alpha, c this the brown Dickmann turboroid estimate will be good so in that case u raised to u comes out to be actually L x 1 - alpha 1 - alpha by c. So u raised to u 1 - alpha appears there in the second exponent if you started with alpha. Let us look at the calculation because this is this is an important way to understand the reason why this strange function appears.

So what is log y? Log y is log of this L x alpha, c which by definition if you look at the definition it will be what we ignore the constants and we come to the main function main function which is log x to the alpha times log 1 - alpha log x this is what log y is. So what is u? So this will be you just want to divide by the above quantity right log x divided by that so

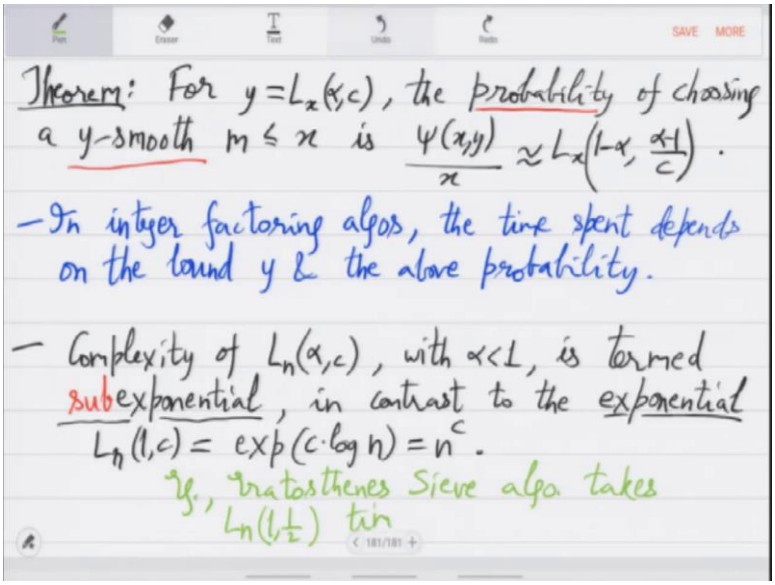
you will get let me make this exact because I need that this actually is just c times its base it was exponent exp function was b c.

So this is just c times log alpha times log 1 - alpha log so then i can write it exactly so this is equal to 1 over c times log of 1 – alpha and log of alpha - 1 log x right that is what you get. What is u raised to u? Let us calculate u raised to u so which is instead to keep calculation manageable we work with u times log u. So u times log u will be the above so 1 by c times log of this times log of alpha - 1 log x and log of u will be;

So here I make a make some approximation so I forget about log of 1 by c because it is only a constant contribution log of log of log to the 1 - alpha gives me 1 - alpha log log x and I get some again additive lower order term log of log of log of x which I ignore. So let us just make it approximate instead of doing the exact calculation and then you will notice that this is coming out to be 1 - alpha by c times 1 - alpha in the exponent.

And then you have log of log of x to the alpha so which means that u raised to u will be e of this e raised to this right which is same as L x 1 - alpha, 1 - alpha by c that is the full calculation. So you get that u raised to u is again L of 1 - alpha if u was L of alpha. So that is a nice relationship. It will keep appearing in integer factoring analysis.

(Refer Slide Time: 37:55)



So let us collect this make it a theorem. So if you take y to be l x alpha c then the probability of choosing a y smooth m less than equal to x. This probability is psi x, y the number of y smooth numbers divided by x and it is 1 over the previous bound that we got so L x 1 - alpha

so alpha - 1 by c this is the this is really the probability of y smooth numbers when y you take as an L function L of constants.

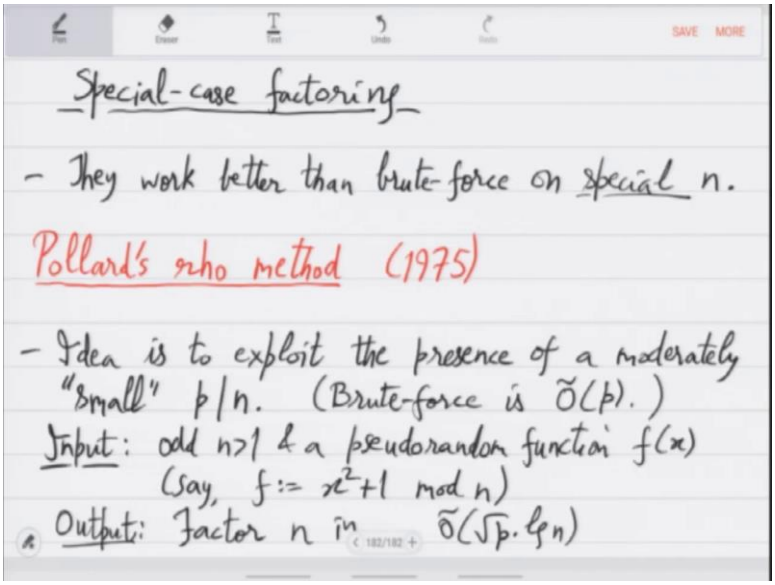
So for L of constants thus the smooth such smooth numbers is actually decent it is again L of constant. So that is the reason why L function appears. So in integer factoring algorithms the time spent it depends on the bound y and the probability because one over probability is expectation. So you have to try these many numbers to get a smooth number. So actually the time complexity of integer factoring uses this L function.

Another point is time complexity of this form L n alpha, c where alpha is of course less than one this is termed in the literature of integer factoring this is termed sub exponential. Why is the term sub exponential? Sub exponential time complexity to contrast with L n 1, c what is L n 1, c? L n 1, c is exponential in c times log x right which is x to the n. So n to the c for any constant c no matter how small the constant c is this is a very slow algorithm because n will be huge in value that is clearly exponential time.

And if you can reduce alpha below one then in the literature it is called sub exponential. So for example Eratosthenes Sieve so you have to receive is just you divide by all numbers from 2 to square root n to factor n right. So the complexity will be around square root n which is a half c equal to half so that takes L n 1, half time. So we are up against that L n 1, half is kind of the trivial algorithm to factor n that we want to reduce to L n alpha, c where alpha is less than 1.

So, much for the L function and smooth numbers, now let us look at some concrete factoring algorithms we will start with an easy one.

(Refer Slide Time: 43:36)



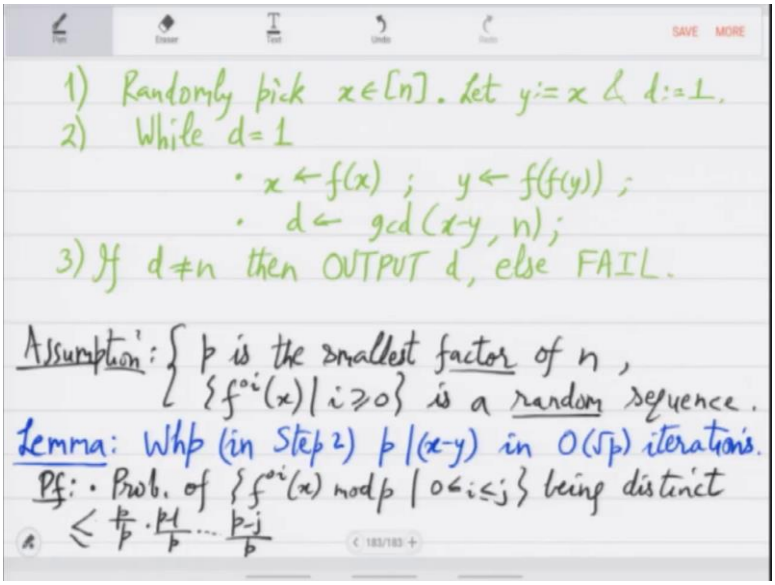
But they are still better than Eratosthenes c better than the brute force. So let us start with some special case factoring algorithms. So they will they work better than brute force on special numbers n they make some assumption on n then they work better. So first is it is called Pollard's Rho method what does it do? Suppose one of the prime factors p of n is moderately small you want to use that fact.

Obviously if you enumerate numbers from 1 to p then it will take step p but can you do better than p that is what Pollard's Rho method does. So idea is to exploit the presence of a moderately small p dividing say you want to solve it in square root p time instead of trivial p. So, brute force would have been O tilde p that you want to do better than that. So this is a clever algorithm it will achieve square root p how it will do?

It will take a random looking function f and it will start with a value with the point x and then it will apply f on x again and again application of f on x again and again. So let us first write down the algorithm input is odd n and a pseudo random function if actually you can even take f to be x square + 1 mod n. Just introduce some non-linearity and then usually the function usually these non-linear functions they behave in a pseudo-random way.

So, factor n in O tilde square root p times log n time that is the goal factoring in square root p time.

(Refer Slide Time: 47:57)



Let us write down the algorithm the algorithm as I said randomly picks a starting point and let us take another variable y set it equal to x and let us set a new variable third variable d to be 1. Then what you do there is a kind of an infinite while loop what it does is compute f of apply f on x and apply f on f y. So on x you apply f once on y you apply f twice and then you compute the gcd of x - y with n.

Basically checking whether x is y mod n you are taking the difference or think in terms of prime p dividing n. So you are trying to check whether x and y are the same mod p. If it is; if they are then d will become p heuristically. So if that happens so if d is not equal to n then output d so this loop will come out only when d is not 1 right. So d is between 1 and n if it is not 1 it may also be n but if it is no if it is neither one nor n then it is a prime factor then it is a factor of n.

So that is a success case otherwise you will say fail and then probably you have to repeat the experiment that is the algorithm. So we will make two assumptions and then analyze this. So assumption is p is the smallest prime factor of n and second is that if you keep applying f on x then you get a random sequence. So let us say p is the is the smallest factor of n. So it is prime and if you keep applying f on x then you get a random sequence with these two reasonable assumptions in fact the assumption is only the second one.

With that assumption we will show that it takes square root p time. So with these two assumptions what we will show is that with high probability in step 2 p will divide x - y within square root p iterations that is what we claim. So this while loop in step 2 will not

really be infinite very soon which is square root p iterations it is highly non-trivial because we are just applying f once and f twice again and again on x and y respectively.

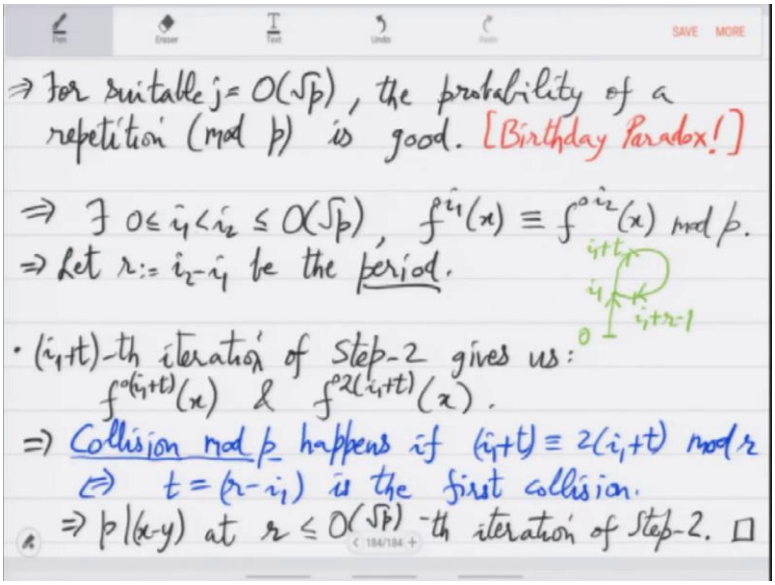
But within square root p iterations it is expected that this difference will be divisible by p and so will come out and output a factor of n obviously this is a heuristic using the assumption it's a probabilistic claim. So how do you show this well since you are assuming that this f applied i times on x is a random sequence. So the probability of f 0 i x viewed mod p for let us say i 0 to j – 1.

So these being distinct there is an upper bound on this probability right because the numbers the residues you can get is 0 1 2 dot dot p - 1. So in the first case for i equal to 0 this can be any value so that probability is basically 1. So I write it as p over p possibilities in the second case since you do not want to overlap with the first one this is p - 1 by p in the end you get you get p - j + 1 by p which I write as you go up to you maybe you go up to j.

So at i equal to j you have this p - g excluded g excluded so it is p - j by p. So you see this as 1 - 1 over p dot dot 1 - j over p. So this is e raised to - 1 over p - 2 over p dot dot - j over p which is around e raised to - j square by p that is the probability estimate. So if you take j to be less than square root p by 2 or square root p by 10 then you see that this probability is sufficiently small.

So these elements in the sequence random sequence being different mod p is actually low probability so with high probability there is an overlap and overlap means that the difference will be divisible by p.

(Refer Slide Time: 56:10)



So this implies that for g smaller than square root p the probability for a suitable fixing of j let us say square root p by 10 the probability of a repetition obviously mod p is good. This calculation should remind you of something called the bird a paradox that is what is going on here. So if you collect square root p random things then numbers then two of them will be the same mod p that is what it is saying.

So this means that there exists i 1 i 2 at most square root p such that f applied i 1 times is the same as f applied i 2 times mod p. So this seems so what how does this relate to step 2 of the algorithm right there we were just applying x we were applying f once and y we are applying it twice. So how does this relate to what we have deduced in terms of i 1, i 2. We are almost there we just have to connect it properly.

So after the what is happening is that you started with something and then in the sequence f i f composed i times x mod p that starts with some value and then it starts repeating. So this is the 0th iteration this is the i1th iteration and then let us say this period is r. So let us so this will be the i 1 + t iteration and this will be i 1 + r - 1th iteration and in the next iteration which is i 1 + r the thing repeats.

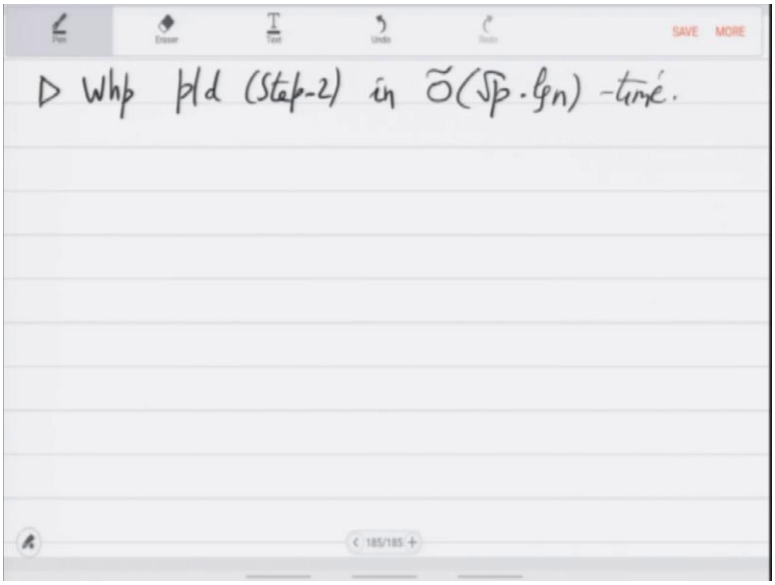
So this is how it is moving forward. So this is also the reason why it is called the Rho method because you can look at this picture how the iterations are going which is how the sequence f i x mod p is changing. So it may start at some value but after few iterations that particular value after i at the ith iteration that particular value will repeat with period r. So let r defined to be i 2 - i 1 be the period that is what we have learnt.

This happens with high probability this rho picture. So in step 2 what happens i 1 + tth iteration of step 2 gives us i 1 + t times composed x with f and twice that on x right these are the 2 values you are getting because x you are going one step at a time why you are going 2 steps. So in i 1 + t iteration you are getting i 1 + t and 2 times i 1 + t. So a collision occurs that is the key thing.

So collision mod p happens if these 2 values overlap mod r the period that is the period of the circle in rho which means that the first time you will see the collision right that will be when t is equal to r - i1 this is the first collision. So i 1 + t you take it to be r and that thing will at that iteration you will see the collision and hence we are done. So this means that p will divide x - y at r which is at most square root p iteration of step 2.

So this finishes the proof of this lemma statement in blue with high probability p will divide x - y in step two in only square root p iterations.

(Refer Slide Time: 1:04:11)



So the time complexity of this is so with high probability p divides d in step 2 in O tilde square root p iterations. In; each iteration you are doing everything around log n time. So that is the algorithm Pollards Rho algorithm okay.