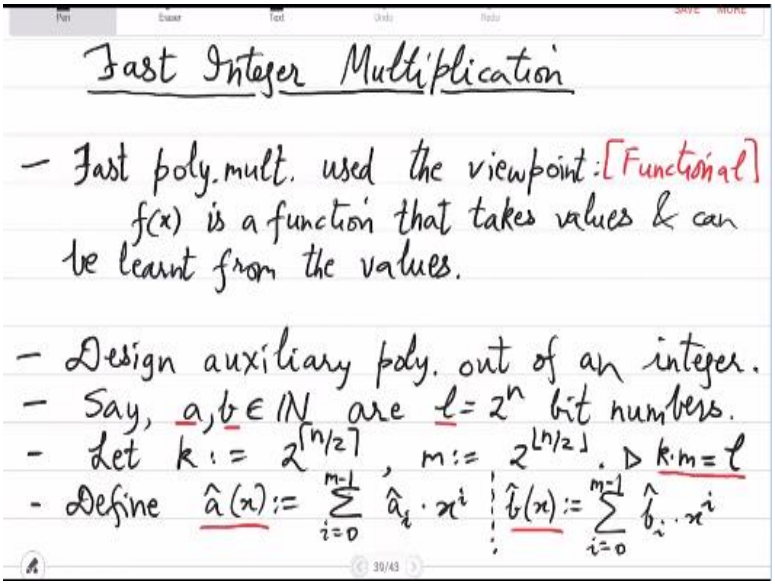
Computational Number Theory and Algebra
Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 06
Fast Integer Multiplication and Division

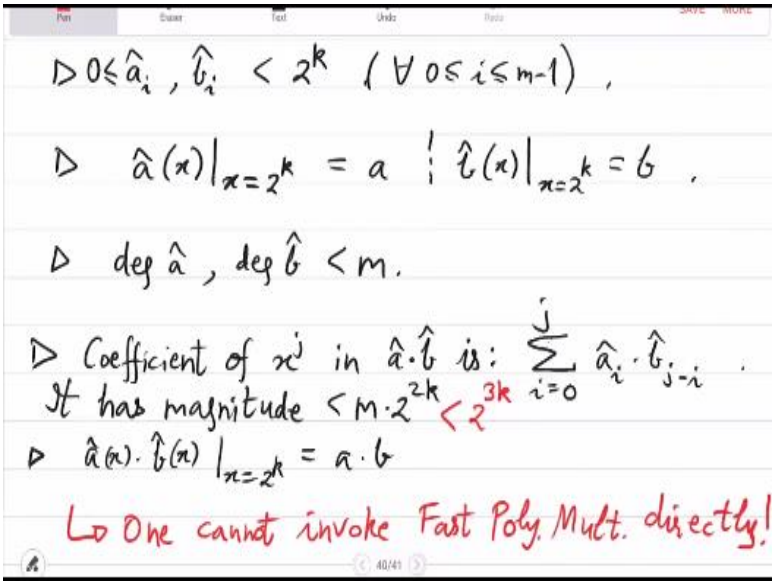
(Refer Slide Time: 00:16)



Okay, so last time we started doing fast integer multiplication by converting these given integers a and b. So input is a and b. These are l bit numbers, positive integers. So we can again, without much loss of generality assume that l is power of 2. And as expected, we convert a into a polynomial. So we break it into essentially square root l chunks where each chunk has square root l bits.

So these square root l bits they give you coefficients. So you have a polynomial called a hat x and similarly b hat x. So formally, a gives you a hat of degree m - 1 and the coefficients a i hat they are k bits. So they are numbers between 0 to 2 raised to k. And we will assume that km is l okay.

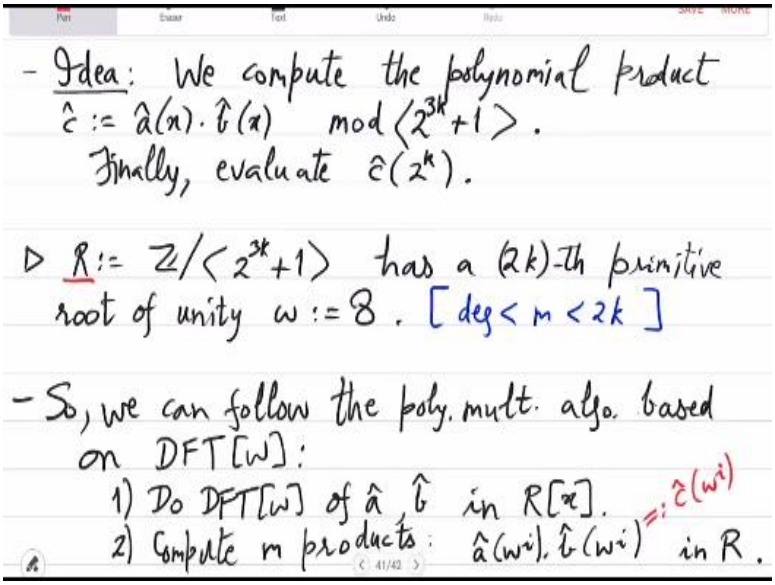
(Refer Slide Time: 01:26)



So you can of course, multiply a hat and b hat using the polynomial multiplication algorithm, but the serious limitation is that it will only tell you time, it will tell you l log l is the time complexity but in terms of Z- operations. And Z- operations is everything. I mean that your original question was about a single Z- operation, right. So that is really no progress.

So what this needs is going deep inside that polynomial multiplication algorithm and make all the claims in terms of bit operations. That is actual time. So this is why we cannot just stop here. You cannot just invoke fast polynomial multiplication. So we have to now analyze. And remember that once you have computed a hat times b hat you just have to substitute x to be 2 raised to k to get back numbers. Any questions? Okay.

(Refer Slide Time: 02:36)



So yeah so to go into bit operations, what we will do is we will first I mean since the coefficients are only 2 raised to k in magnitude. So maximum you will get is 2 raised to 2k. When you multiply in slightly mode it was m times 2 raised to 2. Okay that calculation also we did. So we got this, m times 2 raised to 2k, which is definitely smaller than 2 raised to 3k.

So we will do arithmetic modulo 2 raised to 3k + 1. This is important because the coefficients cannot be just stated as integers. We also have to take into account how many bits there are and how many multiplications and additions. How will it correspond to bit operations? So we have to fix the integer precision. So we are fixing it to 2 raised to 3k, okay. So all the arithmetic will happen modulo 2 raised to 3k + 1.

And then the c hat you will get you can just substitute 2 raised to k and everything is the same as before. So our ring is this R integers modulo 8 raised to k + 1. And which makes 8 a 2k-th primitive root of unity, okay. So this ring has already has a primitive root of unity with order much bigger than required. What was required is only degree of the polynomial a hat which is less than m that is less than 2k, okay.

So at this point we can now do the DFT steps. So let us start that. So we can follow the polynomial multiplication algorithm based on DFT with respect to omega. So these are basically three steps. You do DFT, then you multiply, then you do DFT inverse. So you get c hat. And then in c hat you substitute 2 raised to k right. So now four steps.

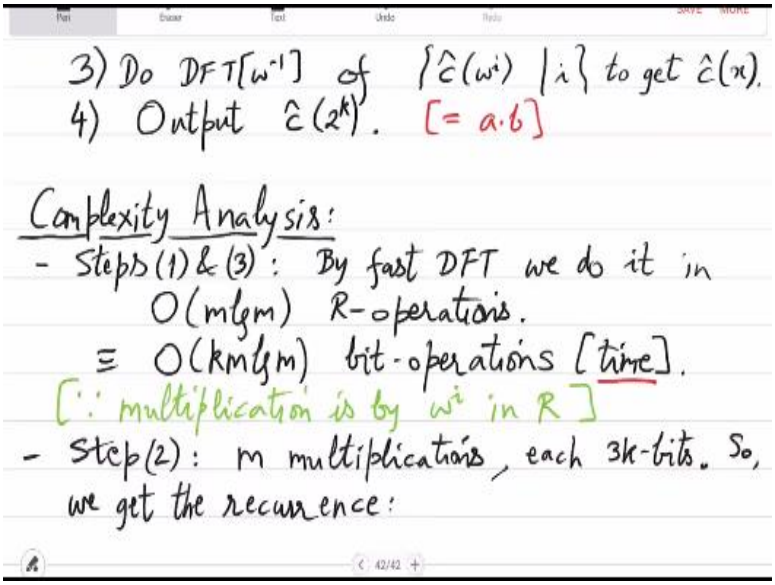
So first step is do DFT omega of a hat b hat in the ring R x which is nothing but computing the values of a hat for all these powers of omega and same for b hat, right. So just do that evaluation. To make it fast you have to do it by Gaussian trick, so by recursion. Divide a hat into two halves and then recurse in the way of merge sort and so on. Merge. Then you do find m products.

So these m products are product of these respective values. So you have a hat omega i b hat omega i. These are all values in R and i goes from 0 to this order of omega, so 2k – 1. So it is basically 1 to 2k. So all these 2k actually you can also stop at m because the degree is m -1. So just i can go from 0 to m - 1 also. That is also fine. So here now we have to analyze the time complexity carefully.

We cannot say that this is these are just m products so it is m time, right? So R is actually this integer ring and how many bit operations it will actually take. So what is the bit size of a hat omega i value? How big is that as an integer? 3k bits. You are finding a remainder mod 2 raised to 3k + 1. So these numbers are at most 2 raised to 3k. So the bits are at most 3k.

So you are actually multiplying two 3k bit numbers and then you have to take remainder and in the end, you will be left with the 3. So all the elements in R are 3k bits or less. So it will not just be m multiplications. But also each multiplication will cost something in terms of k, function of k right? So that we have to evaluate, estimate.

(Refer Slide Time: 08:04)



Third step is inverse. So DFT omega inverse of the c hat to get c hat polynomial c at x; c hat is just this. So from these values you can recover c hat as a polynomial. And then you compute or output c hat's value at 2 raised to k. So this is a times b, right? So it is a very roundabout complicated way of multiplying two numbers, right. So you convert the numbers to multiply the polynomials.

Do the arithmetic modulo a number, and then you do DFT inverse, get the polynomial and then evaluate the polynomial. Any questions? So these are the four clear steps. All that remains is the time complexity analysis. So let us do that. How fast is this? It will not be l times log l, because now we will also have to count bit operations. So it may be slightly more than that.

So how much more? So steps 1 and 3, which is the DFT part. So by fast DFT. So in first you talk about R- operations. So how many R- operations will this DFT take? So how long are these vectors that DFT is being applied on? m, so this will take m log m. So m log m R- operations. But then this R- operations you have to count in bit operations. So how much will that be?

Right so you are working with integers in R, which are k bits, around 3k bits. So you can call k bits. So you multiply this with k. But is that really enough? So for additions it is order k. But what about if you are multiplying two numbers? Yeah, so you have to see, you have to look into the details of what you actually do in DFT. Fast, this

FFT. So you have to see that there if you multiply with a number, then that is only omega to the i.

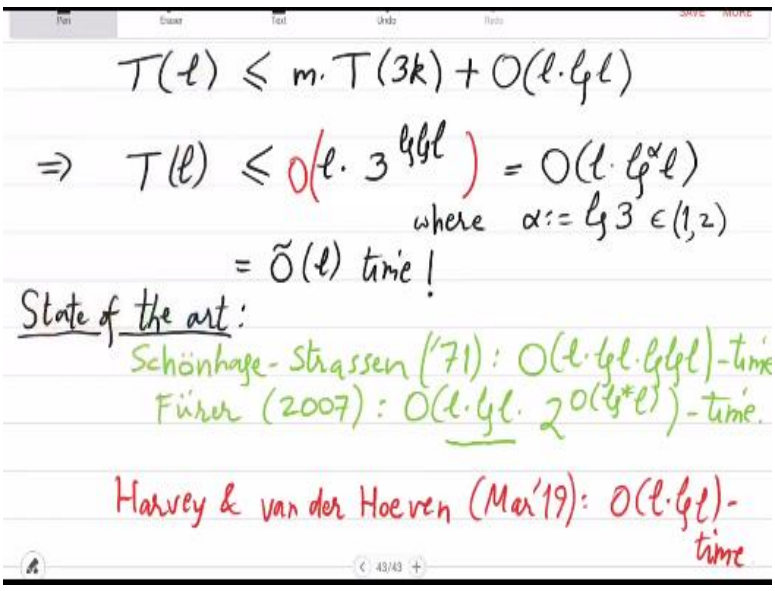
And then you take a simple linear combination, which is sum. So everything is simple sum except multiplication by omega to the i. But that also is a very simple arithmetic in this ring, right. So that is just a game in the exponent. So that also reduces to addition. So this is actually in our case it is km log m actual time. So km is? Is l. So this is actually l log l, right.

So steps 1 and 3 are actually quite cheap. So reason here is since multiplication is by omega to the i in R. So that can be done very efficiently. The simple shifting, simply a bit shift. So like before, step 2 will be the expensive one, because you are doing many multiplications and each of them is order k bit. So that is step 2. So this is m multiplications each 3k bit.

So what do you do here? You do not know how to multiply numbers, right. So this here you will have to recurse on integer multiplication itself, right. So these are clearly smaller integers than l bits. 3k is around square root l. So this is much smaller than the original instance. So it is fair to say that I will do each of these multiplications recursively by using fast integer multiplication itself.

So the recurrence will be? So let us do it on the next slide.

(Refer Slide Time: 14:31)



So recurrence is you wanted to multiply l bit numbers and overhead is l log l. So that is not too much. How are you recursing down? So m instances, right? And this is 3k. So this is T l equal to square root l times t of 3 square root l plus order l log l, right. So what? Who will solve this? So what is this as a function of l? Without the three it was as we solved before l log l log log l. But there is a 3 multiplying now.

So this gives you an extra factor. What is the factor? So I would say this is l times 3 by 2 log log l. Will this work? So the only way to check this is you substitute this back. Does it work? Yeah, so all that will get absorbed in this because this is 3 by 2 is more than 1. So this already contains log l. I mean 2 raised to log log l yeah I do not see 3 by 2. I mean 3 is clear. So every time you get a multiple of 3 and this will happen log log l time.

So you get 3 raised to log log l. That I think is clear. And this will give you how much? Okay let me see I think we there was some intricacy in definition of km. Now if n is even, then k and m both are square root l. So that is actually correct. So anyways, it is easy to see that l times 3 raised to log log l is will satisfy the recurrence, but even 3 by 2 works, and this will then give you l times log l to some constant.

So the factor is log l raised to a constant. The constant is just this, alpha. Where alpha is yeah log 3 by 2 to base 2. No if I work with 3 by 2, then I get this. Log 3 by 2 base 2. It is positive, but less than 1. So this alpha is between 0 and 1. So this you can check that you get. Okay, that will be a problem. That is too good to be true. In fact, that is an open question. So let me keep it like that okay.

So that is what you get, you get slightly more than 1 in the exponent. So it is l log l times not log log l, but in fact a small exponent over log l. So it is slightly worse than fast polynomial multiplication. But remember fast polynomial multiplication was in terms of ring operations. This is in terms of bit operations. So this is actual time. So you have shown that this is a soft O of l time right.

So you can multiply integers l bits in truly O tilde l time, that is the result. Okay, any questions? So once you have learned how to multiply integers in actual O tilde l time,

now you should ask about integer division. And once you have learned integer division then about GCD, Chinese Remaindering and all that.

So all these basic arithmetic which you always knew in quadratic time could that be now made linear time or O tilde l time right. So to begin with integer division itself is not clear whether it can be done fast because this method is very specific to multiplication. So why is it very specific to multiplication? Why cannot you do the same thing for division, integer division?

The problem is that you are converting to polynomials. So when you convert to polynomials then polynomial multiplication is automatically defined, but division is not. So a tilde, b tilde does may not divide a, a tilde. So b divides a but then b tilde may not divide a tilde. So in that case your method gets stuck. So we have to see that. So that result is also there.

You can do actually everything in O tilde l. So we will do it in a different way and then invoke this result at some point. Before that, let us talk about the state of the art of this. So this is not the best result known. This has been superseded multiple times. In fact, this class or next class we will improve this to Schonhage-Strassen's result. So Schonhage-Strassen results form 71 is l log l log log l time for fast integer multiplication l bits.

Okay, that was there were some results before this also. But they were not so interesting. So there is a famous result by Karatsuba, which gives you something like l raised to an exponent between 1 and 2 log 3, yeah l raised to log 3. So that result is there, but that is not really linear in l in the O tilde's n. So this is the first O tilde l algorithm. It is far more advanced than current Karatsuba's algorithm.

Then it was improved in a big way, after several decades by Furer. So Martin Furer gave l times log l times 2 raised to log star l. So who knows the definition of log star l? Yes. So you are interested in composition of the log operator. So you are given in the input l argument is l and you start taking log of log of log of that and the number of times you have to take it let us say i times before you go below 1.

That is the that is called log star l. So that i is being put in the exponent. So it is essentially it is l log l with a multiplier that is so small that you will have difficulty even making this log star l 5 in reality. Because making it 5 means that you are looking at l bits where l is a tower of 2 of length 5, it is huge. So for all practical purposes, this is l log l. But theoretically it is not.

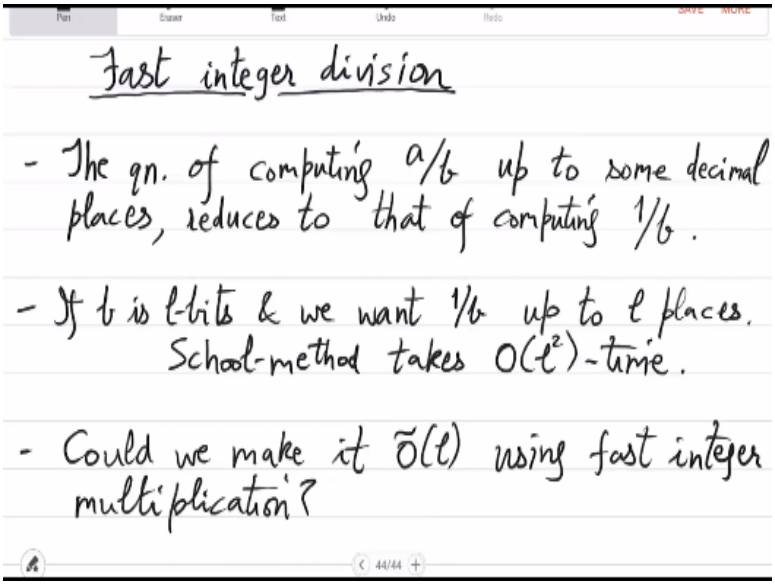
So that problem has been resolved just nine months ago. Okay, so last year Harvey and Van der Heuven, so they made it all l log l time, okay. So that solves the old question of multiplying numbers in l log l, okay. So Schonhage-Strassen you will very soon see once we have done division etc., we will come back to this.

What Furer and then Harvey and Van der Hoeven have done is instead of defining a hat a univariate polynomial, they define a multivariate polynomial okay. So given an integer a, you define a polynomial that has d variables and from that polynomial you can come back to the number by simple evaluation and then these two d variates polynomials are multiplied.

Now it is shocking how can that be easy. Multiplying two d variates polynomials should be even more expensive than univariate multiplication or integer multiplication. But the thing is that you have to design the ring R in such a way that multiplications are easy, okay. So the ring R is so special that it is also a d variate ring or d – 1 variate ring. But it is so special that somehow these d variate polynomial multiplication, it is very simple and it is fast.

So that multivariate fast Fourier transform is computed. If anybody is interested in this last year's paper, we can talk about in an extra talk. I would not be able to cover this. Any questions? Okay, so let us see applications of this to other operations equally basic.

(Refer Slide Time: 27:06)



So first is fast integer division, how do you divide numbers. So the question of computing a by b. So b may divide a in which case you will get an integer or you want to do long division. You may also want the quotient and the remainder because that will be helpful in GCD. So the question of doing this division let us say up to some decimal places, instead of remainder we talk about decimal places.

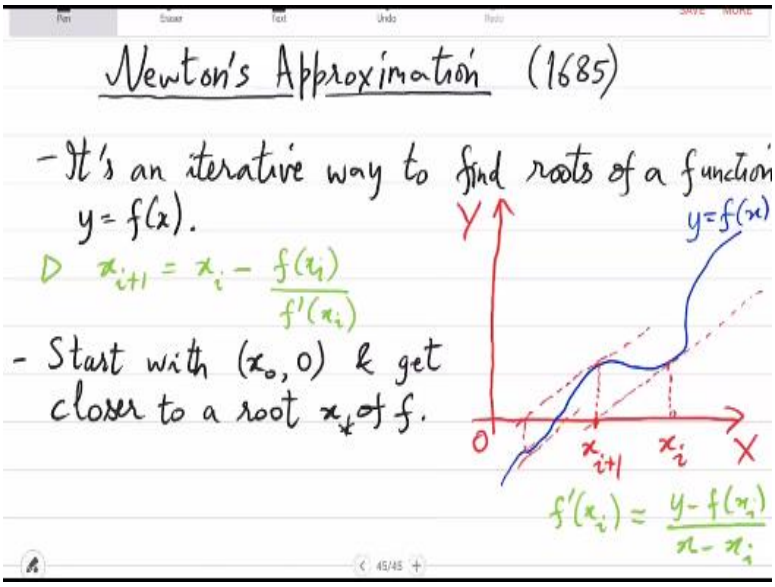
So this reduces to simply an inversion problem, which is inverting b. So you just want to compute 1 by b up to some precision in decimal representation. So basically you compute 1 by b to enough decimal places and then you do integer, fast integer multiplication with a right so that will give you a by b also with the remainder. The remainder version also you will be able to simulate by this procedure.

So we will only focus on how to compute 1 by b to let us say, d decimal places, okay how much time will that take? So that is the thing we want to solve. So let us say b is l bits, l bit integer and we want 1 by b up to l places as well after decimal point. So the school method that you know how much time will that take? Computing each decimal place takes how much time?

At least l and for l places it will take l square. So you cannot think of a simple way to improve on l square. Okay, l square is all you know at this point. So this is what we want to make O tilde l. So how do you do that? How do you use integer multiplication to make it O tilde l? On the face of it, it is a purely division question. It is 1 by b. There is no scope for any multiplication.

So how will you do this? How will you use integer multiplication here? So hint is that you already know this, most of you being engineers, you must have done courses where this is hinted. Even the math students know this, because these are math courses. So have you heard of Newton's iteration?

(Refer Slide Time: 31:34)



It is also called Leibniz Newton iteration. So what is that? Yes. So yeah if you have a moderately well behaved function univariate f of x, then to find the root of it. It is a let us say you are looking at polynomial with real coefficients. So there is this real metric and you want to find a real root up to some approximations some decimal places.

So you start with a at an arbitrary place on the x axis right and then you look at the value of f there, draw a tangent and wherever the tangent cuts you go there and repeat the process, right. And God willing, it will lead you to a root. Otherwise it will lead you haywire. So there is no way to predict where it will lead. But in experience, it works well if you choose a decent starting point.

Also the function has to be well behaved. So if the function is well behaved and you have a good starting point then this will very rapidly converge to a root. So on the x axis wherever the curve intercepts this process will lead you to that intersection point. So the tangent is yeah, helping you converge very fast to that place. So it is a very old idea.

It is from 1600s. So it is an iterative way to find roots of a function y equal to fx. So it draws a curve on the xy plane. So you have let us say curve is. So if you start with this point, well not start say in some intermediate step of the process you are here. This is x i. So you look at the value of y at this point on the curve. And from there you draw a tangent.

So this tangent you see meets the x axis here. So this is x i + 1. So now at x i + 1, you look at the value of y and draw a tangent there. Then here draw a tangent here, right. So I have gamed this curve so that the tangents are going closer and closer to the point where curve meets the x axis. But usually this works, okay. So for good functions, this is a good process. And when it works, it works extremely fast, okay.

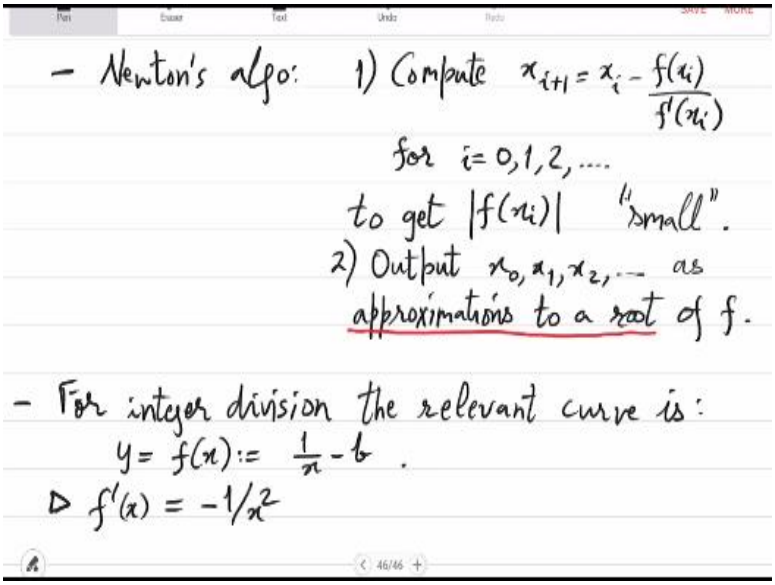
So we will actually analyze how fast it will work in our case. It is a exponentially fast, converging process to the root. So this basically is giving you a recurrence between x + 1 and x i. What is that? Yeah, use the slope equation. So the slope equation says that f prime x i, or the slope of the curve at x i is given by this equation. That is the slope equation.

And in the slope equation, since you x i + 1 will be the place where y is 0. It is on the x axis. So this you set in this equation y = 0 and set x to be x i + 1 and you get the recurrence. So the recurrence is x i + 1 = x i – f x i over f prime x i, okay. So if you start with a good x i then you get the next x i + 1. That is the Leibniz or it is called I think Newton-Raphson iteration.

This is the Newton-Raphson recurrence. Obviously, if f prime x i vanishes, then this is a bad formula. So those things you have to avoid in general. We would not have any of that trouble as you will very soon see. So this process starts with x 0 on the x axis and get closer to a root of f. Let us say root x star. So x 0, x 1, x 2 may meander a bit, but as you go further and further, ultimately it will be converging to a root under some moderate conditions.

So yeah I can give the basic algorithm which, in general you use.

(Refer Slide Time: 38:41)



So compute x i + 1 = x i minus for i = 0, 1, 2 till f x i is small enough or maybe slightly different. So you keep computing these x i's till you make f (x i) as small as you want. Ideally it should become zero, but it is a real process so it may never become zero it may just converge to zero so at some point you will stop and at that point your output x 0, x 1, x 2 has approximations to a root of x, to a root of f.

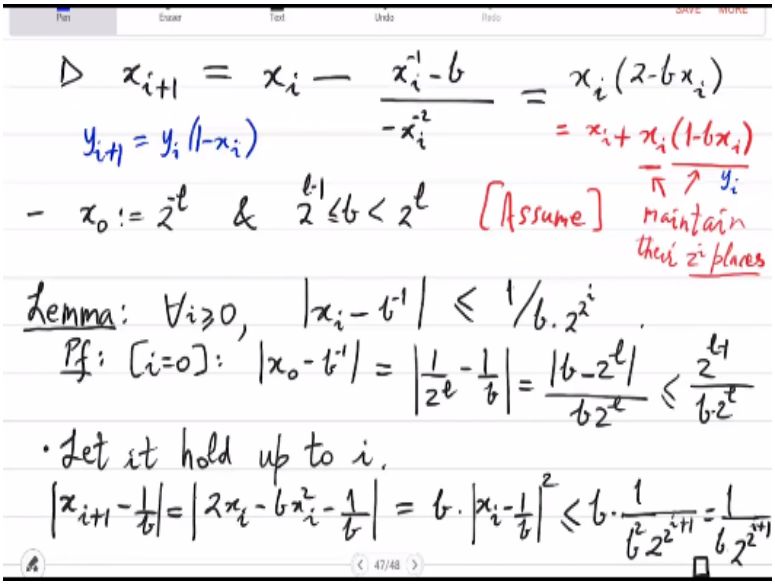
So these are the approximations, okay. So what is the function of our interest on which we want to run this pseudocode or this paradigm? Yes. So you want to find x which is equal to 1 over b right. So this x you can first guess the first decimal place that is only one digit so that you can call x 0. It is some progress and then you want to find x 1 which will be up to two decimal places.

And then three decimal places and four and so on, okay. So as you run this Newton iteration, you will discover more and more decimal places of 1 by b. That is the simple idea. f dash yeah we will do the calculation. Do not worry. Do not use the b x - 1. Let us use 1 over x – b, right? Yeah, I do not know why it works, but your method did not work because of that problem.

So you use a different function with the same root. So for integer division, the most relevant curve is y = fx equal to defined as 1 over x – b. Use this. So for this f prime x is - 1 over x square. So the point is that when you divide by f prime, you do not want to get stuck in division, you want to reduce it to multiplication. So this 1 over x square will go on the top.

So you would this one step is actually not division but multiplication and you know how to multiply, fast multiply. So here you will invoke Schonhage-Strassen or whatever. This is – 1 over x square. And let us see Newton iteration step 1.

(Refer Slide Time: 43:34)



So f of x i is x i inverse – b and f prime x i is – x i – 2. So which is okay? So 2 x i and then – b x i square. This is what you get. So you have this is just multiplication, okay. Division has been reduced to a sequence of multiplications. So that is big progress. What will you take x 0 as? x 0 is 0, but then you will not make any progress everything you can find it yeah, so actually here it seems that 2 raised to - l will work.

It is in a way approximation to 0 because b is l bits. So we are taking x 0 to be very small. So it is kind of close to 0. So we will start with this and then we will see how this proceeds. So x 0 you can start with this and b you have you know is smaller than 2 raised to l and at most, at least 2 raised to – l, l - 1. Yeah, I want that. There is really no loss of generality here because you started with an integer.

So we are just saying that it is between two successive powers of 2. And for this setting, you take x 0 as 2 raised to – l, okay. So we have to now measure our progress. What is the progress or how good is the approximation in the i-th step of Newton iteration. So x i - b inverse value we want to show that it is very small. In fact, as i is increasing, this difference is only getting smaller and smaller.

So that ultimately x i is 1 by b, right. So what do you think is the upper bound that you can show? 2 to the – i? Yeah, that is true. But something even more stunning is true. We will show this. That it is not just 2 to the - i. It is actually 2 raised to – 2 raised to i, okay? This is what I meant when I said that Newton iteration actually converges exponentially faster than you want, okay.

So it is a highly convergent process when it wants to converge. For bad functions it may want to diverge, but this function is so good that it will converge in its best way which is exponential convergence. So roughly what this lemma once you have shown this lemma and the proof is very easy. Once you fix this upper bound, you can simply prove it by induction.

The proof will be simple. So the consequence will be that if you wanted l digits after the decimal place, so how many iterations will you need in the Newton process? Only log l okay. So to get l digits, you only need log l iterations of this, right. So this is why overall everything will be very fast here. Okay proof we can do as a formality. So what is base case i equal to 0? So x 0 – b inverse, x 0 is 2 raised to – l, which is b – 2 raised to l.

2 raised to l – b you know is positive. You use the fact here that yeah that the b – 2 raised to l value cannot exceed 2 raised to l – 1. It is at most this. And b is at least 2 raised to l – 1 or maybe wait. Yeah, let us rewrite it. Yes. So you get 1 over 2b. That is i = 0 case. Now let us assume it to be true. So induction hypothesis is that it holds still i, some value i. And let us look at i + 1.

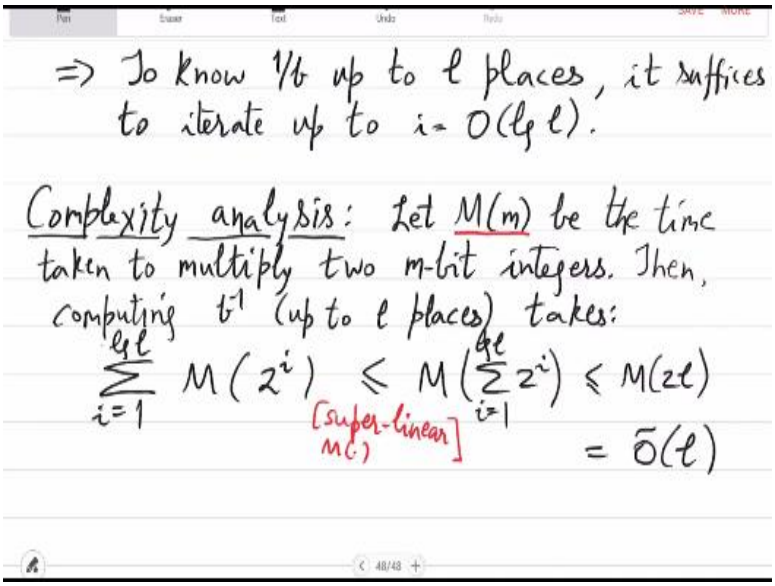
So here you use the directly the formula, put that here. So that is 2x i – bx i square – 1 over b. And what is this? So magically this formula simplifies a lot. So this actually becomes 1 over b times it is a square. So you get x i – 1 over b square. And x i – 1 over b you have an upper bound and since it is positive, upper bound can be put here and squared. So you will get less than equal to 1 over b times by b. Yes.

So the bound you know by induction hypothesis is 1 over. So x i – 1 over b you assumed is at most that bound in the lemma. So just square it and you get what you wanted. So for i + 1 you get 2 raised to 1 + 1 in the exponent, okay. So you can see

why this is happening. The previous error is being squared, okay. So the new error is always is related to the square of the previous error.

So that gives you the exponential convergence, okay. So with this stunning convergence, we can now analyze the time complexity. We know how many iterations there are. And we know that in every round or every iteration, there is just integer multiplication happening. So we just have to sum up the complexity. Let us do that.

(Refer Slide Time: 52:26)



So for l places it suffices to iterate up to i equal to log l, order log l many iterations. So let M of small m be the time taken to multiply two m bit numbers. So we are not fixing the fast integer multiplication algorithm, let it be just m of the number of bits complexity algorithm and use this in Newton iteration. So then computing b inverse up to l places takes how much time?

So it is the sum from i 1 to log l each iteration i. In each iteration, how much integer multiplication is happening? M of, so how many decimal places of x i will you care? So you will only care about those places which you know are correct. After that point it is garbage. So you will not use that part right. So how much is that part which we actually know is matches with 1 by b?

2 to the i, right, that is by the lemma. So you just keep that many digits. So that is M of 2 to the i. So you invest only this much time in the i-th iteration, not more. In the

iteration you have x i times 2 – b x i. I think I can multiply this by l also l times 2 raised to i. No, I do not want to do that. Sorry? Just add it. Yeah, I can say m l here but then you will get m l times log l. I was planning to get something better than that.

Okay, so some simplification is needed here. So if you see this as two things, there is x i and then there is x i times 1 – b x i. So how many digits of 1 – b x i are important? That is 2 raised to i. So the actual multiplication which is happening is actually only in 2 raised to i digits. A b is sitting there but you are actually using 1 – b x i and 1 - b x i is only getting smaller.

But yeah after the decimal place, the digits are increasing. The l will not appear in this calculation. No, we cannot ignore it. But if you look at 1 – b x i then it is only 2 raised to i digits which are important. **"Professor - student conversation starts"** How to evaluate the value of bx i? **"Professor - student conversation ends".** Yeah that has to be done iteratively.

So I think I have to write down another recurrence. You can keep track of these two things separately x i and 1 – bx i. You keep these things in two registers and from these two registers, so x i and 1 – bx i in both the registers there are only 2 raised to i digits which are important. And you do that multiplication and from that you get x i + 1. So you maintain them in two different registers.

So let me just finish this. The point here is just that once you accept this M of 2 raised to i in the implementation, this is actually less than equal to M of sigma 2 raised to i. Do you agree? This is because M is a super linear function. It is at least linear, even worse. So the sum of its values will always be upper bounded by M of the sum of the arguments, right. That is just super linearity.

So by that we get M of sum. M of sum is just M (2l) which is in all the algorithms it is O tilde l in particular. But if you look at M (2l), this is just saying that essentially the amount of time it takes to multiply integers in the same amount you can also divide, okay. The time is not so much different. That is all. That is the simple analysis based on Newton iteration. Any questions? Okay.

“Professor - student conversation starts” Can you explain that registers? **“Professor - student conversation ends”.** Oh, the register thing. That is an implementation detail. From the base case itself, you maintain x i and 1 – bx i separately and when x i gets updated then you have to ask the question, how much time will it take to update 1 – bx i.

“Professor - student conversation starts” The idea is for updating is like say 1 – bx i is y i then y i + 1 is y i – 1 – x i. **“Professor - student conversation ends”.** Yeah, it is free of b. It is free of b. So maybe you state the recurrence again. Sure. So this is this is y i. Then y i is, y i + 1 is y i times 1 – x i. Yeah, that is the important thing. So keep this independent of b. So this you can quickly compute. Okay, thanks.