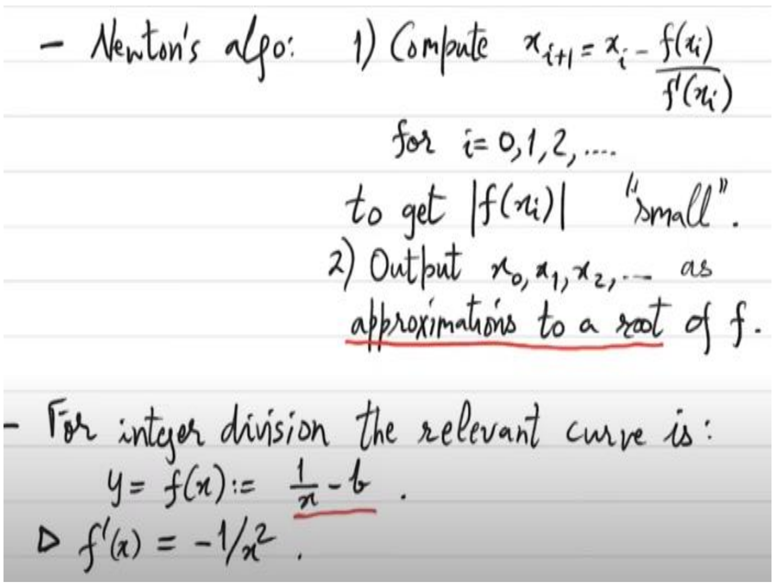
Computational Number Theory and Algebra

Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture – 07
Fast Integer Arithmetic and Matrix Multiplication

Okay, so last time we were looking at fast integer division. So we started Newton-Raphson iteration basically. So you want to find essentially a root of this, 1 over x - b.

(Refer Slide Time: 00:28)



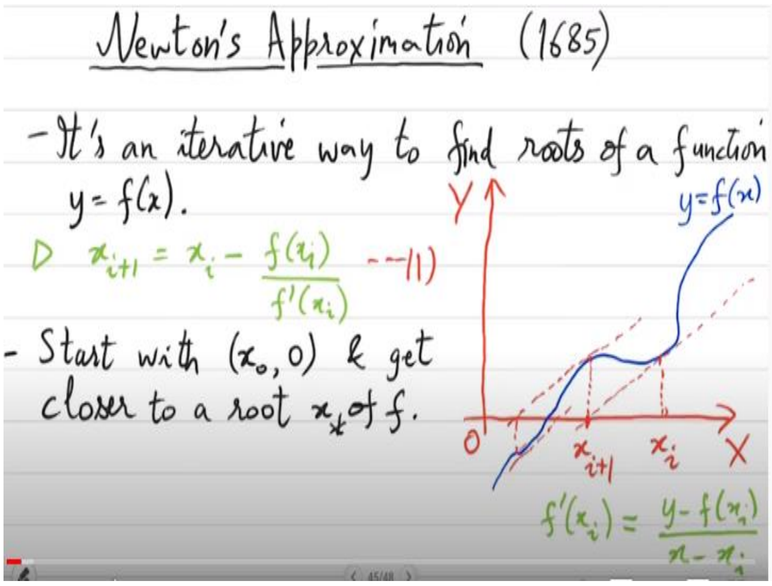
- Newton's algo: 1) Compute $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ for $i = 0, 1, 2, \ldots$ to get $|f(x_i)|$ "small".
  2) Output $x_0, x_1, x_2, \ldots$ as approximations to a root of f.

- For integer division the relevant curve is: $y = f(x) := \frac{1}{x} - b$.

$\triangleright$ $f'(x) = -1/x^2$.

So this is a nice function where b is given to you an integer and you want to find x up to some decimal places. So finding root of this function we will use Newton iteration.

(Refer Slide Time: 00:46)



Newton's Approximation (1685)

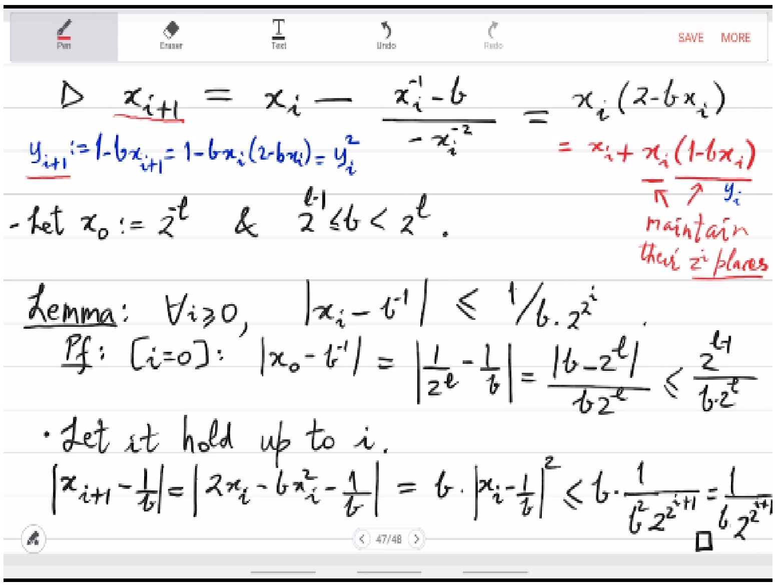
- It's an iterative way to find roots of a function $y = f(x)$.

$\triangleright$ $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ ---(1)

- Start with $(x_0, 0)$ & get closer to a root x_* of f.

So the formula for that is x i + 1 = x i. So this formula. This equation 1 is x i + 1 = x i – f over f prime. So using this if you want to find b up to l decimal places or l bits then you will have to do this iteration only log l times. There were some questions after the class.

(Refer Slide Time: 01:18)



$$\triangleright \quad x_{i+1} = x_i - \frac{x_i^{-1} - b}{-x_i^{-2}} = x_i(2 - bx_i)$$

$$y_{i+1} := 1 - bx_{i+1} = 1 - bx_i(2 - bx_i) = y_i^2 \qquad = x_i + x_i(1 - bx_i)$$

y_i

maintain their 2^i places

- Let $x_0 := 2^{-l}$ & $2^{l-1} \le b < 2^l$.

Lemma: $\forall i \geq 0$, $|x_i - b^{-1}| \leqslant 1/b \cdot 2^{2^i}$.

Pf: [i=0]: $|x_0 - b^{-1}| = \left|\frac{1}{2^l} - \frac{1}{b}\right| = \frac{|b - 2^l|}{b 2^l} \leqslant \frac{2^{l-1}}{b \cdot 2^l}$

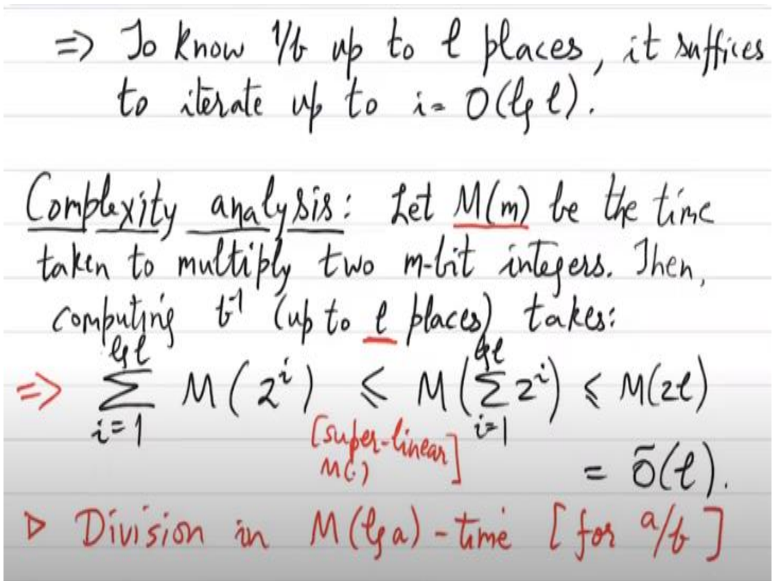
- Let it hold up to i.

$$\left|x_{i+1} - \frac{1}{b}\right| = \left|2x_i - bx_i^2 - \frac{1}{b}\right| = b \cdot \left|x_i - \frac{1}{b}\right|^2 \leqslant b \cdot \frac{1}{b^2 2^{2^{i+1}}} = \frac{1}{b\, 2^{2^{i+1}}} \quad \square$$

So one correction is this formula we have x i + 1 = x i times 2 – b x i. And we do not want to work with b. So instead what we will do is we will maintain x i and y i. So this expression is actually x i + x i times 1 – b x i. 1 – b x i you can call y i. So y i + 1 is 1 – b x i + 1 which you can see is actually y i square, okay. So we will maintain y i in two different registers and to compute y i + 1 you only have to square y i.

And once you have y i + 1 then using that once you have y i and x i, using that you can compute x i + 1, okay. So we will eliminate the need of b. Because b had a lot of bits, so we did not want to use all the bits at once. We wanted to do this iteratively in a slow, the number of bits should grow slowly.

(Refer Slide Time: 02:29)


$\Rightarrow$ To know $1/b$ up to ℓ places, it suffices to iterate up to $i = O(\lg \ell)$.

Complexity analysis: Let $M(m)$ be the time taken to multiply two m-bit integers. Then, computing b^{-1} (up to ℓ places) takes:

$$\Rightarrow \sum_{i=1}^{\lg \ell} M(2^i) \leq M\left(\sum_{i=1}^{\lg \ell} 2^i\right) \leq M(2\ell) = \tilde{O}(\ell).$$

[super-linear $M(\cdot)$]

$\triangleright$ Division in $M(\lg a)$-time [for a/b]

So that was useful in this final complexity analysis where big M is the complexity of multiplying small m bit many two integers each of small m bit size. In terms of that you get this expression for integer division, okay. So this expression is sum over big M of 2 to the i. And you want to compute 1 over b up to l places, okay. So sigma M of this is at most M of sigma, which is at most M of 2l and which you can use now any integer multiplication algorithm.

All of them will give you O tilde l, right. So you have a soft O linear time algorithm to divide. First you compute 1 over b then you multiply by a. So you can compute the quotient a by b. And once you have the quotient you can also compute the remainder, right. It is another integer multiplication. So hence you can do division in this much time.

This will take, so if you want to compute a by b, suppose a is longer, so M of log a time for a by b computation. Okay, so this long division you can now do. Essentially in the same time as multiplication of a times b. Any questions till now? Okay. **“Professor - student conversation starts”** Why we are doing it up till order of log l? Up to order of log l, the number of iterations? **“Professor - student conversation ends”.**

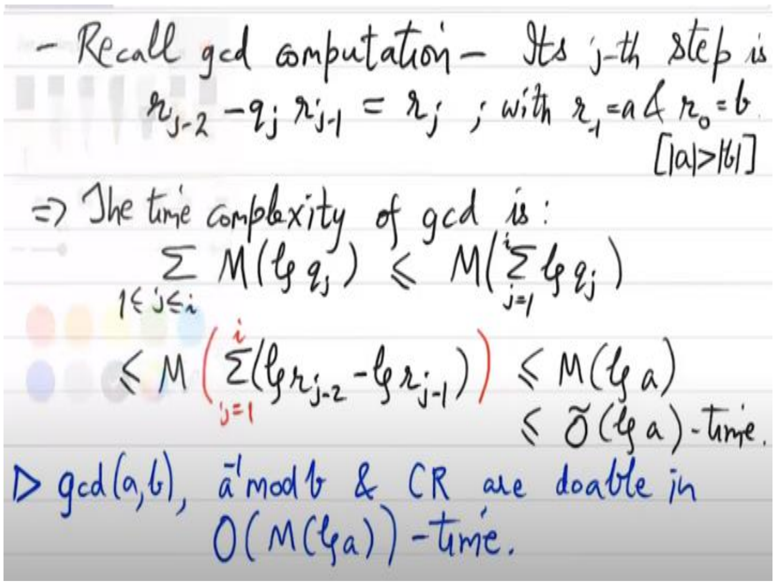
That was the previous calculation. You have shown that this difference X i – 1 over b. So as you do the next iteration go to the ith iteration then the difference between x i and 1 over b is smaller than 2 raised to - 2 raised to i. And when you want to compute

1 over b up to l decimal places or l binary places, it basically means that you want the error to be something like 1 over 2 raised to l + 1.

So you set l + 1 equal to 2 raised to i. So i is only log l. So the convergence here is actually it is called quadratic convergence. Quadratic because every time you are essentially doubling the number of binary places. Those many binary places are correct, guaranteed to be correct, okay. So once we have this, once you know multiplication and division, you can do all other basic operations.

So starting with the gcd, right. You can also analyze gcd now. It will have a similar analysis.

(Refer Slide Time: 05:52)



- Recall gcd computation – Its j-th step is

$r_{j-2} - q_j r_{j-1} = r_j$; with $r_{-1}=a$ & $r_0=b$

$[|a|>|b|]$

$\Rightarrow$ The time complexity of gcd is:

$$\sum_{1\le j\le i} M(\lg q_j) \le M\Big(\sum_{j=1}^{i} \lg q_j\Big)$$

$$\le M\Big(\sum_{j=1}^{i}(\lg r_{j-2} - \lg r_{j-1})\Big) \le M(\lg a)$$

$$\le \tilde{O}(\lg a)\text{-time.}$$

$\triangleright$ gcd(a,b), $a^{-1} \bmod b$ & CR are doable in $O(M(\lg a))$-time.

So recall gcd computation. So what happens in the jth step? So r j – 2. So this is the formula we have in the jth iteration of Euclid gcd algorithm. So basically you are dividing r j – 2 by r j- 1 and you will get the next question q j and the next remainder r j right and you keep repeating this. You have started with a and b. So initialization is r - 1 is a and r 0 is b. We have the usual assumptions that a is greater than b.

Well, we can also take mod. We can also take this modulus magnitude. So a is the bigger one. So hence it makes sense to divide by b and get the remainder. So how many iterations were there in this in Euclid gcd algorithm? Yes, so there were log b, around log b many iterations, right. So here the number of iterations is not so small. It is not in contrast to Newton iteration. Here the number of iterations is more.

But still we can do the same analysis. So what you will say is the time complexity of gcd is, it is again the sum of basically multiplication. So what are you multiplying? You will first do division which will have complexity same as multiplication. And you will get the quotient q j then you will multiply q j with r j – 1 to compute the remainder. So it is log q j, okay.

So the complexity of one step long division is basically M of big M of log of q j for all j, right. And again, same trick. So since the matrix the integer multiplication complexity is at least super linear this sum of M is at most M of sum. So let us take the sum inside. What can you say about the sum now? Log sum log q j? So this we had done also in Euclid gcd right? What did you do with sum log q j there?

You telescope you express this in terms of the two r's, the two previous r’s; r j – 2 and r j – 1. And then you will get a telescopic sum. So that was the efficient analysis. Do the same thing here. So you will get log of r j – 2 – log of r j – 1. And j goes from 1 to i. Okay, so this then becomes M of log a okay. So it does not matter actually how many iterations there are. That was more critical in Newton approximation analysis.

Here since you have a telescopic sum it does not matter. So here you just get, anyways you will get the sum log a. So you have learned that doing long division, doing a computing gcd by doing multiple long divisions, it takes essentially the same time as multiplying a with b okay. So gcd is also the same time complexity as multiplication. So that is a surprising fact. But the proof is quite simple.

There might be these inequalities, there might be some constant factor that I have ignored. When you approximate log q j as this difference you might have to add some constant there. M log q j? This is the this I use what we just learned before division. So you have to find q j, q j is unknown. You know the previous two remainders, r j – 2 and r j – 1. You have to compute the ratio approximately.

So that approximate division you use Newton iteration. So that will give you q j. Once you have q j then you will get the exact r j. So those things I have combined in this big M of log q j. So some constant multiples may have to be introduced, but ultimately it will come out to be this. And definitely it is, in all cases it is soft O of log a. And this is actual time. So you can compute gcd also in almost linear time, right not quadratic.

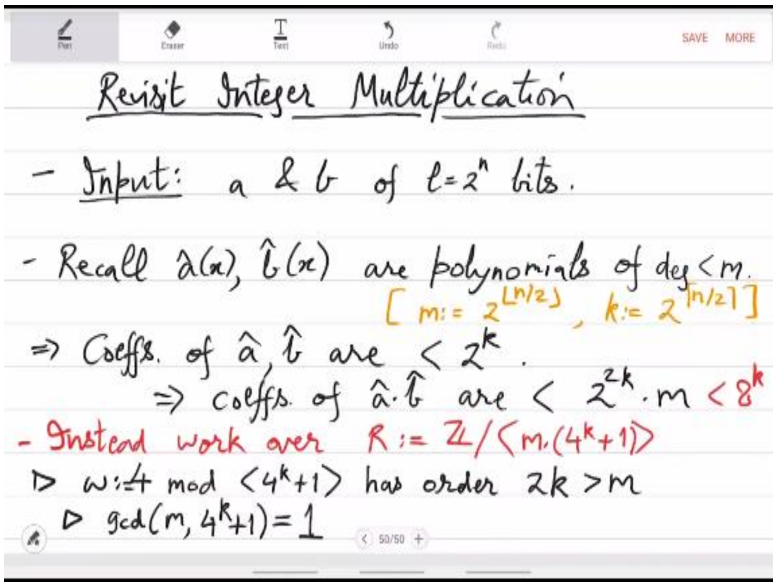
So this is a big improvement over the school algorithm. Okay, once you know how to, any questions? Do you see this gcd algorithm? So once you know gcd computation you also know how to compute this Bezout identity in the same time. So u a + v b = 1. You can find now u and v, which means that you can find inverse, right. So inverse is also same complexity, same algorithm.

So gcd of a, b or a inverse mod b and once you can compute a inverse you can also compute Chinese remaindering. I mean the constructive version of CRT takes the same time are doable in the same time as multiplying log a bits, okay. So use any multiplication algorithm and you get same complexity for gcd, division gcd inverse Chinese remaindering.

So from now on, we will not, I mean we will we will just implicitly assume this theorem. All our algorithms for basic arithmetic will be fast. Any questions? Yeah, so let me revisit integer multiplication. The algorithm I gave was, it was definitely slower than the algorithm we saw for polynomial multiplication, right. So the complexity we saw was n times log n to the alpha.

It was slightly more than, the multiplier was slightly more than log n. We want to make that multiplier log n times log log n for now. So we will see a simple modification of integer multiplication algorithm that you have seen, that gives us log n times log log n.

(Refer Slide Time: 15:25)



Revisit Integer Multiplication

- Input: a & b of $\ell = 2^n$ bits.
- Recall $\hat{a}(x), \hat{b}(x)$ are polynomials of deg $< m$.
  $[\, m := 2^{\lfloor n/2 \rfloor},\ k := 2^{\lceil n/2 \rceil} \,]$

$\Rightarrow$ Coeffs. of $\hat{a}, \hat{b}$ are $< 2^k$.

$\Rightarrow$ coeffs of $\hat{a} \cdot \hat{b}$ are $< 2^{2k} \cdot m < 8^k$

- Instead work over $R := \mathbb{Z}/\langle m(4^k+1) \rangle$

▷ $\omega := 4 \bmod \langle 4^k+1 \rangle$ has order $2k > m$

▷ $\gcd(m, 4^k+1) = 1$

So let us revisit that algorithm. Even this revisited algorithm will not be as fast as state of the art. So the current fastest is completely solves the problem. It removes the log log n factor as well right. But that we will not discuss. That is more complex. So you want, so in the input you are given a and b of how many bits? I think you are calling it l bits and l we are assuming to be power of 2.

So this is the input and to get the product a times b we converted this into these auxiliary univariate polynomials a hat and b hat. a hat x, b hat x they are polynomials of degree less than m. So basically m will be around square root l. So we will divide this l sized binary string into square root l blocks. Each block is of size square root l.

And each block will give us a coefficient and we will get a polynomial of degree square root l. So I think what exactly what was the definition of M? 2 raised to n by 2 I guess. 2 raised to ceiling or floor? 2 raised to floor? Yeah, let us recall that. And there will be a k, which is 2 raised to n by 2 ceiling. So basically k times m is l. This is what we have done.

And the k and m are around square root l, okay. The coefficients of a hat b hat are at most 2 raised to k. In fact, they are smaller than 2 raised to k. They are only k bits. So what can you say about the coefficients of the product a hat times b hat? So we said that you will multiply a coefficient of a hat with that of b hat. So it is at most 2 raised to 2k. But then there is also this convolution when you multiply polynomials.

So you have to sum up m such products because degree is m. So this times m, right. Definitely the coefficients of a hat b hat cannot even equal 4 raised to k times m. That is the bound. In the previous integer multiplication algorithm, we said that this is smaller than 2 raised to 3k, 8 raised to k. So you just go modulo 8 raised to k + 1. And that recurrence was not good.

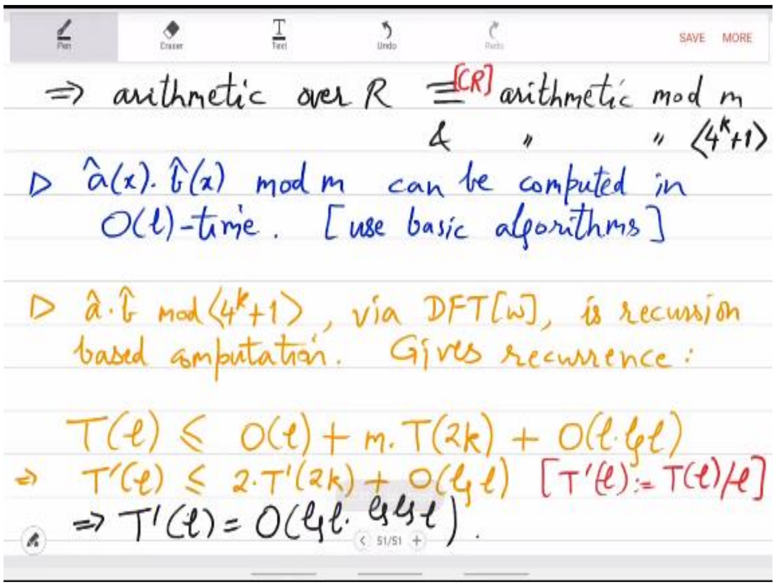
So that time complexity recurrence gave you a log l to the alpha factor. So we will now improve that. So we will say that you actually work modulo this number this bound plus 1. Or in fact not plus 1 you work exactly this over this. So that is the change we are introducing now. So this we arbitrarily bounded by 8 raised to k. We will not do that. So instead work over R.

Next time we have to solve this problem of flickering. So m times 4 raised to k + 1. This is slightly bigger than that bound. So work modulo this and so if you take omega to be 4, what is the, so the point is that m and 4 raised to k + 1 these two are co-prime numbers. So by you can do Chinese remaindering. So this ring by Chinese remaindering arithmetic over r can be broken into two parallel arithmetic one over mod m and one mod 4 raised to k + 1.

Now notice that m is it is just 2 raised to n by 2. So actually these are only n by 2 bits. So that is a very small computation, okay. The original computation was for n bits sorry 2 raised to n bits. Mod m computation is only for n bits. So this is an exponentially low scale arithmetic. So this part actually you can assume to be nearly free, free of cost. The expensive part is mod 4 raised to k + 1.

So in that ring what is the order of 4? It is 2k right. So we have a 2k th root of unity. So which is 4. So we will work with this. So omega in this arithmetic has order 2k, which is more than we need. It is more than m. Okay. Also as I said the gcd of m and 4 raised to k + 1 is 1. Why is that? Yeah m is a power of 2 and the other thing is a power of 2 + 1. So there is no chance of gcd. So this is this justifies the choice of the ring r.

(Refer Slide Time: 23:38)



$\Rightarrow$ arithmetic over $R \overset{[CR]}{\equiv}$ arithmetic mod m
& " " $\langle 4^k+1 \rangle$

▷ $\hat{a}(x) \cdot \hat{b}(x)$ mod m can be computed in $O(\ell)$-time. [use basic algorithms]

▷ $\hat{a} \cdot \hat{b}$ mod $\langle 4^k+1 \rangle$, via DFT[ω], is recursion based computation. Gives recurrence:

$$T(\ell) \leq O(\ell) + m \cdot T(2k) + O(\ell \cdot \lg \ell)$$
$$\Rightarrow T'(\ell) \leq 2 \cdot T'(2k) + O(\lg \ell) \quad [T'(\ell) := T(\ell)/\ell]$$
$$\Rightarrow T'(\ell) = O(\lg \ell \cdot \lg\lg \ell).$$

So arithmetic over R is the same as arithmetic mod m and that mod 4 raised to k + 1. This is by Chinese remaindering and the efficient version of Chinese remaindering. So if you can multiply those polynomials and get the coefficients mod m and mod 4 raised to k + 1 then very efficiently you can convert it back to actual product. So this is the idea. And now we all we have to do is give the new recurrence for time complexity for all this.

So okay the first observation is a hat times b hat and remember these are polynomials mod m can be computed in how much time. So m is just well m is n by 2. Yeah maybe I misspoke before. So anyways m is clearly n by 2 bits. No, yeah okay. So and so the coefficients of a hat and b hat you have to multiply and divide by m to get the remainder. So even if you use brute force methods all that will still give you quadratic in n, right.

So which is much smaller than order l time. Remember l is 2 raised to m. So for this computation you have actually exponential time available. So even if you use brute force everything can be done in l time without any optimizations. Just use basic algorithms okay. So modern computation is easy. Is this clear that you can do it in l order l time? So this is far less than what we are willing to afford.

We are willing to afford l times log l times log log l. So in that sense this is really free. So let us look at the other part a hat b hat mod 4 raised to k + 1. So what do you do here? So here you have to go through this DFT based idea, right. So you have omega available which is 4, which is more than the degree of the product a hat b hat. So you can compute the evaluations of omega powers.

Evaluations of a hat at omega to the i for the amount you need. So compute those values of a hat and similar values of b hat. Then multiply these two coordinate wise and then apply DFT inverse, right. So that will give you a recurrence. So what is the recurrence? So overall if the time complexity to multiply a and b is T (l). So T (l) is at most there is this overhead that we spent above which is order l plus you are computing.

So we have assumed that the product of that degree of a hat is m. So you will be computing a hat at m values at m arguments omega to the i, right. So you have essentially m integer multiplications to do. So that is m times what is the bit size? This is 2k, correct. And another overhead of, yeah which will be how much? Is it m (k)? I think it is more. Oh, so I think we are forgetting DFT computation.

So DFT is also a recursive algorithm. That will give you l log l. Yeah, so what is the difference between this recurrence and the one we did in the first integer multiplication algorithm? Yes, so the only difference is this right? This 3k has been reduced to 2k. I am not sure, is it just that? I think there was a 2m also outside. Okay. So we can write it, we can divide both sides by l.

So that will give us let us call T l by l T prime. So T prime l is at most, l is km. So this will become 2 times T prime 2k plus order log l. So let me define T prime. Okay. So what is T prime l now? From this recurrence what do you get? T prime of 2k think of it like 2 T t prime of square root l. And if you take T prime to be T prime l to the log l times log log l T prime of square root l will actually give you a factor half which will cancel with the factor 2, okay.

So then your inductive proof for the solution of T prime will work. And if you work it out, it will give you that T prime l is at most log l times log log l, okay. So this improves on our previous analysis. So you get that T l is l times log l times log log l. Is that okay? So you can check this. Just fit this T prime l in the recurrence and verify that it actually is a solution. You do not need log l to the alpha like in the previous proof.

“Professor - student conversation starts” Sir we used to do polynomial multiplication in the order of l log l log log l times. So if we can map individual multiplication to polynomial multiplication, why cannot we bring that problem to the similar problem we solved before, polynomial multiplication. **“Professor - student conversation ends”.**

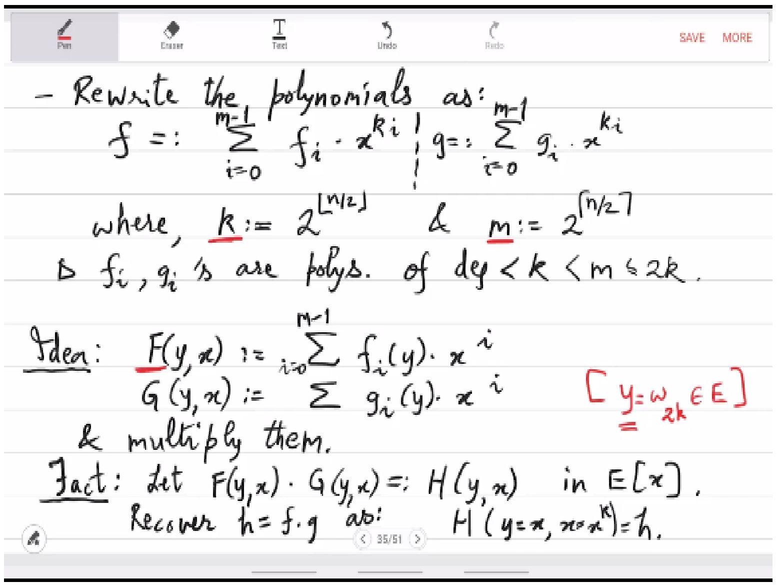
Yeah, so polynomial multiplication analysis was done with respect to ring operations. Ring operations here are they have to be counted because when you multiply integers you cannot say that the time complexity is small in terms of z operations. That would be a trivial statement. So you have to reduce z operations to bit operations. That is the point we have already discussed.

So this is why you cannot just directly invoke polynomial multiplication. So we are doing actually what we did in polynomial multiplication, but our estimate has to be in terms of bits, which means actual time. Not in terms of the ring, because ring is the ring of integers. There you cannot say that one multiplication is unit time because original question was actually that.

How much is unit time for multiplying two integers. Yes, this is a small modification which using Chinese remaindering actually reduces 3k to 2k and gives you this complexity which stood for very long. And then recently it has been brought down to just l times log l. This is also practically fast. Even doing a few iterations of this recursion if you implement it, it is much faster than the school algorithm.

Okay, any other questions? There was a question after class regarding polynomial multiplication. Some corrections I propose there. And then it will actually look very much similar to integer multiplication what we just saw. Basic things were fine. The change is when you define k and m. Even in the ring actually there is a change.

(Refer Slide Time: 35:52)


- Rewrite the polynomials as:

$f =: \sum_{i=0}^{m-1} f_i \cdot x^{ki}$; $g =: \sum_{i=0}^{m-1} g_i \cdot x^{ki}$

where, $\underline{k} := 2^{\lfloor n/2 \rfloor}$ & $\underline{m} := 2^{\lceil n/2 \rceil}$

▷ f_i, g_i's are polys. of $\deg < k < m \leq 2k$.

Idea: $F(y,x) := \sum_{i=0}^{m-1} f_i(y) \cdot x^i$

$G(y,x) := \sum g_i(y) \cdot x^i$ $[\, y = \omega_{2k} \in E \,]$

& multiply them.

Fact: Let $F(y,x) \cdot G(y,x) =: H(y,x)$ in $E[x]$.

Recover $h = f \cdot g$ as: $H(y = x, x = x^k) = h$.

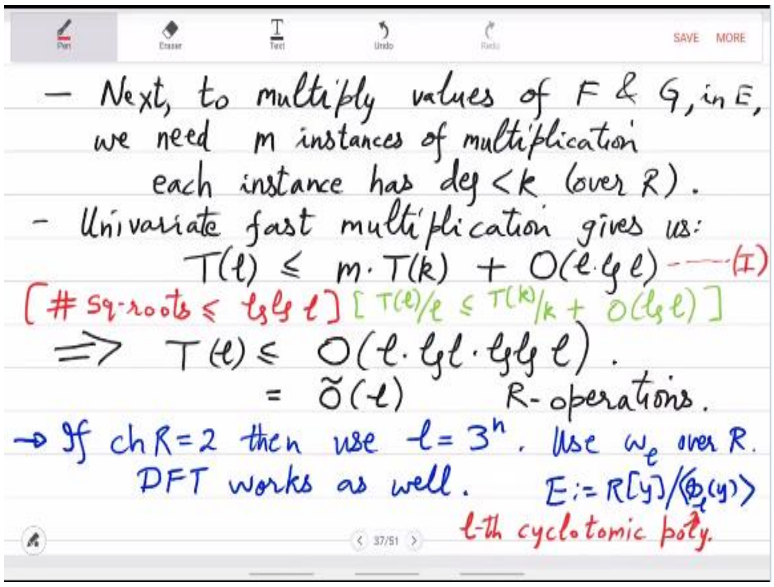
So the in your notes just make this change. So define k and m as we define right now. Take k to be square root of l, but exactly this 2 raised to floor of n by 2 and m is l over k and then you decompose your polynomial f in terms of these blocks, m blocks each of size k okay. So then your auxiliary polynomial big F change the definition slightly. So look at a f i as a univariate in a new variable y.

So basically from univariate we are going to bivariate. So f i in y and instead of x to the k i you actually look at x to the i. So big F is a bivariate polynomial where the individual degree of x is m – 1. And the individual degree of y is k – 1, right. So both the individual degrees are around square root l, but exactly this. Now you want to multiply the auxiliary polynomials big F with big G.

So in that multiplication you have to basically substitute for x powers of omega. And it will suffice to use since the individual degree of x is m – 1, you use this y. Use y to be 2 kth primitive root of unity. And this is what you should attach in E. Or E should be defined by this y to the k + 1. So r, y, mod, y to the k + 1. And then you use that y. So when you substitute those powers of y in big F it will become univariate in Y.

And the degree will be, what will be the degree in y after substituting powers of y in x? Well you are modding out by y to the k + 1. So the degree will be smaller than k right. So you have basically reduced the multiplication of the bivariates into many multiplication instances of univariates each of degree at most k. So you get the following recurrence.

(Refer Slide Time: 38:30)



- Next, to multiply values of F & G, in E, we need m instances of multiplication each instance has $\deg < k$ (over R).
- Univariate fast multiplication gives us:

$$T(\ell) \le m \cdot T(k) + O(\ell \cdot \lg \ell) \text{ --- (I)}$$

[# sq-roots $\le \lg\lg \ell$] [$T(\ell)/\ell \le T(k)/k + O(\lg \ell)$]

$$\Rightarrow T(\ell) \le O(\ell \cdot \lg \ell \cdot \lg\lg \ell).$$
$$= \tilde{O}(\ell) \quad R\text{-operations}.$$

→ If $\mathrm{ch}\, R = 2$ then use $\ell = 3^n$. Use ω_ℓ over R. DFT works as well. $E := R[y]/\langle \Phi_\ell(y) \rangle$

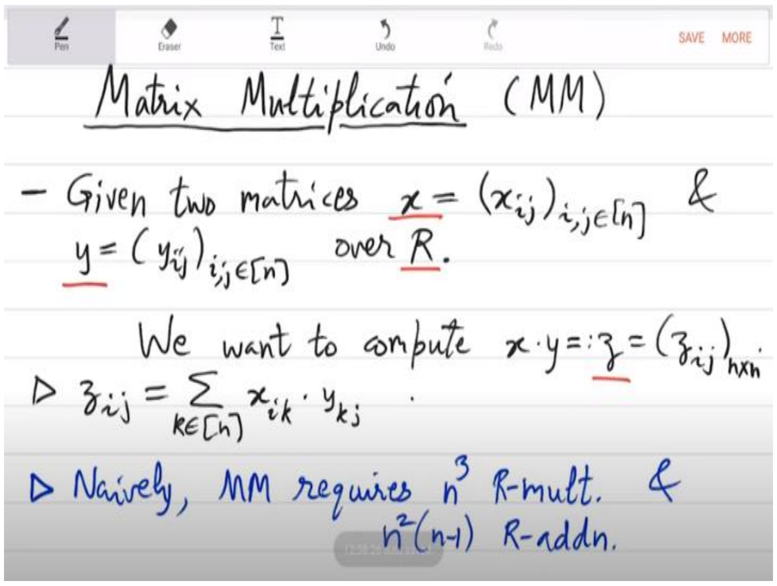
ℓ-th cyclotomic poly.

The recurrence you get is this. So this equation 1 is what you get that multiplying two l degree polynomials reduces to m instances of k degree polynomial multiplication with the DFT cost of a log l as overhead. And a solution of this is l log l log log l. But some correction was needed. It was pointed out by someone that, what is your name? Yes, so it was pointed out by Rishabh that previously the instances were not degree k.

They were slightly more and then the recurrence would get messed up. So instead you should use this. And this is exactly what we did also in integer multiplication. So actually the two proofs are identical, okay. Integer multiplication, the way we have done finally, is the same that you will do with polynomial multiplication. You will get the same time complexity as well.

Obviously, the units are different. So here multiplication in the base ring is considered free. In integer case you go at the level of bits. Okay, so you can make these changes. And if there is a question ask me next time. Okay. So with that we will move on to the next topic. Is there any question till now?

(Refer Slide Time: 40:29)



Matrix Multiplication (MM)

- Given two matrices $x = (x_{ij})_{i,j\in[n]}$ & $y = (y_{ij})_{i,j\in[n]}$ over R.

We want to compute $x \cdot y =: z = (z_{ij})_{n\times n}$.

▷ $z_{ij} = \sum_{k\in[n]} x_{ik} \cdot y_{kj}$.

▷ Naively, MM requires n^3 R-mult. & $n^2(n-1)$ R-addn.

So then let us start this fundamental objects matrices. Right, so since now you know how to multiply in a faster way polynomials, integers. So it is natural to ask how will it lift to matrices, right. So what is your algorithm to multiply two n by n matrices? How much time does it take? Yes, so the definition of matrix multiplication already takes n cube assuming the base ring operations to be free.

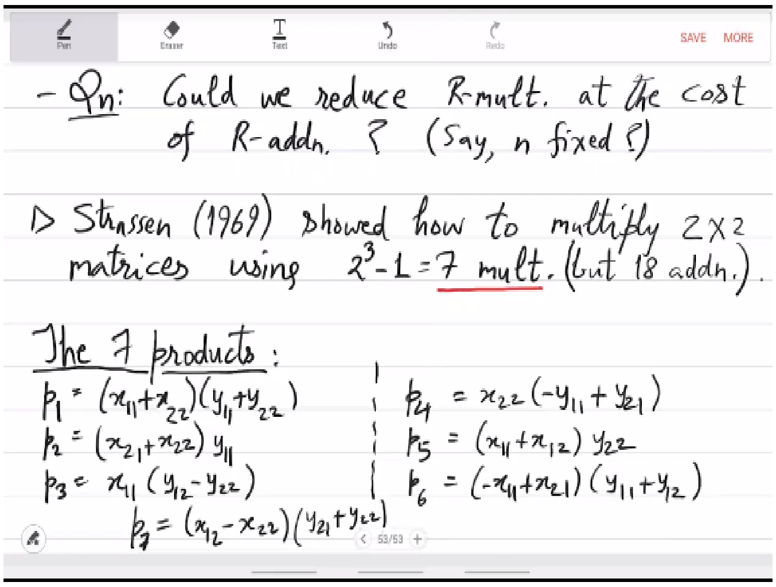
So it is actually order n cube ring operations, R operations. Now the question is can it be improved? Right, if the definition is already taking n cube why should it be possible to improve it below n cube? So that is what we will do now. And this also is an evergreen area. This is still not solved. So questions here are still open. Every few years there are surprises. So given two matrices, we will use small x, it is a matrix x ij.

So this is an n cross n matrix, x ij's are just ring elements over a ring R. Yes all you need to define multiplication of x with y is this base object R to be a ring, right. Then you have the classical definition. So you want to compute x times y which is say z. This will also be n cross n of course. So in the input you are given x and y and R. In the output you have to output z and what is the definition of z ij?

Yes, it is the inner product of row with vector. So ith row inner product with jth column for all i, j. Right, this is what you have to compute and there are three subscripts i, j and k, right. So to compute all these products, you will obviously need n cube multiplications. Each subscript takes values 1 to n. So that is the naive algorithm. Matrix multiplication requires n cube multiplications which is R.

n cube R multiplications and how many additions? Each i, j requires n - 1. So n square times n – 1 R additions. So the question is can we reduce, I mean for all rings, for almost all rings R multiplications is the expensive part. So we want to reduce this part, not worrying too much about addition. The addition over rings are is already cheap, is always cheap. So can we reduce multiplications at the cost of additions? Can we reduce it below n cube?

(Refer Slide Time: 45:53)



- Qn: Could we reduce R-mult. at the cost of R-addn.? (Say, n fixed?)

▷ Strassen (1969) showed how to multiply 2X2 matrices using $2^3 - 1 = 7$ mult. (but 18 addn.).

The 7 products:

$p_1 = (x_{11} + x_{22})(y_{11} + y_{22})$

$p_2 = (x_{21} + x_{22})\, y_{11}$

$p_3 = x_{11}(y_{12} - y_{22})$

$p_4 = x_{22}(-y_{11} + y_{21})$

$p_5 = (x_{11} + x_{12})\, y_{22}$

$p_6 = (-x_{11} + x_{21})(y_{11} + y_{12})$

$p_7 = (x_{12} - x_{22})(y_{21} + y_{22})$

And you could actually even assume for a fixed n. So you do not even need to do this asymptotically. Say n is equal to 10 instead of thousand multiplications make it 999 multiplications, right. So even just numerically can you reduce multiplications? This is the first question? And believe it or not if you can solve this.

So for a fixed n if you just reduce multiplications from n cube to something smaller then actually it has a cascading effect and it reduces the complexity of asymptotic matrix multiplication as well. So if for any constant n, you have a faster way to multiply then you have a faster way for general n by n matrices which is faster than n cube. It is something like n to the 3 minus epsilon.

That you can show by recursion. You can divide your general n by n matrix into let us say n zero blocks assuming that n zero cross n zero matrices you can multiply faster. And then recursively you will get a faster than n cube. So that is what Strassen actually showed in a concrete way. So Strassen showed that and it was a big surprise. So how to multiply two by two matrices using 1 less than 8.

Okay, so 2 cube is by definition he can do it in 7. Additions will be slightly more. So additions in the base algorithm was 4. In his case it is 18. But that is not important. So what is important is how did he get 7? So this algorithm you must have seen in prior courses right? What is the idea to get from 8 to 7 and what effect will it have for n cross n matrix multiplication?

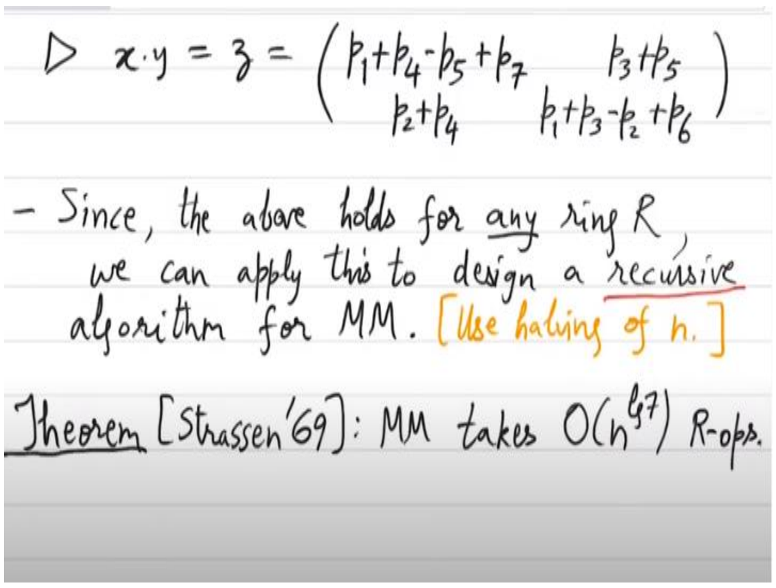
Yeah, so if you go with the divide your n by n matrix into blocks of 8, blocks of 2 and that 2 by 2 matrix multiplication you can do faster so for n cross n you will get actually, for n cross n you will get n to the log 7, log is to base 2, which is slightly less than 3. I think it is 2.8 or something. So you get n to the 2.8. And what are the 7 products that you have to compute?

So this so there is this magical 7 products right, which nobody can remember. Because they are totally unmotivated. So there are these 7 products. So the idea is that Strassen defined these 7 products p 1 to p 7 such that the 4 entries of the z matrix, they are just addition. So sum or difference of these p i’s in fact linear combinations of the p i's using only plus minus one coefficients.

So multiplications are only 7, additions are more. So p 1 is, p 2 is, so which is p 1 is kind of the diagonal, the trace of x; p 2 is using the second row of x. p 3 is which is like the second column of y. p 4 is which is the first column of y. So there are some symmetries, but it is very hard to guess. p 6 and finally p 7 is x 12 – x 22. So these are the seven products that you have to compute.

So the idea is that instead of just computing x i k times y kj, which was just a monomial. Instead of computing a monomial, you compute these products which actually use linear combinations of x part and respectively y part and multiply those combinations. Okay, so you can see that in all these products x and y's are separated. They do not mix and yeah, then there are equally magical combinations which will give you z.

(Refer Slide Time: 54:33)



$$\triangleright\ x\cdot y = z = \begin{pmatrix} p_1+p_4-p_5+p_7 & p_3+p_5 \\ p_2+p_4 & p_1+p_3-p_2+p_6 \end{pmatrix}$$

- Since, the above holds for any ring R, we can apply this to design a recursive algorithm for MM. [Use halving of n.]

Theorem [Strassen'69]: MM takes $O(n^{\lg 7})$ R-ops.

So final identity is that z is equal to x times y which is z is equal to this. So the diagonal entries are combinations of 4, but with plus minus 1 coefficients, the other entries are even simpler, right. So proof of this you can easily check unless I make some mistake in the definition. But, clearly there are only 7 multiplications to be done and additions you can count. It will come out to be 18.

Also note an important fact that no division was done. So the ring could be very arbitrary. It does not, it only needs multiplication and addition of ring elements. So this is a very general formula. It solves matrix multiplication in all cases. There is no assumption which is why it is a fundamental identity. So the above holds for any ring. So oh, so there is a important point.

Since this identity holds for any ring in particular, it also holds for the matrix algebra. So these entries of x, x 11 for example, this can be a matrix in itself, right. So this is what allows you to do recursion. So 4 by 4 matrix you can divide into 2 by 2 blocks. And then you can again use the same formula because it actually also holds for matrix algebra. So that gives you the recursive approach.

So we can apply this to design a recursive algorithm for M. Okay, so yeah, the recursive thing is quite simple. It is just the natural thing you would do. You assume this n cross n matrix, we assume n to be a power of 2. And then you just each time you halve n. So if you halve n then n cross n matrix looks like a 2 cross 2 matrix. And you can use the recursive algorithm.

The recursion will smoothly proceed and the base case will be 2 cross 2. So you will get the following theorem by that. So use halving of n. So if you do the recursive analysis the time complexity recurrence, you will get the following theorem that matrix multiplication takes n to the log 7 R operations. So the recurrence will be, how many multiplications are needed for n cross n matrix multiplication?

So T (n) will be equal to 7 times T n by 2 plus overhead which is order n. But this 7 times T n by 2 if you solve this you will get 7 to the log n, which is the same as n to the log 7, okay. So this immediately improves on the basic algorithm the n cube time algorithm. And there have been there is an ever growing literature which brings log 7 to various weird numbers. So currently it stands around more than 2.4.

Okay, even 2.4 has not been achieved, slightly more than 2.4. Conjecture is that you can make it arbitrarily close to 2. That is a major unsolved problem.