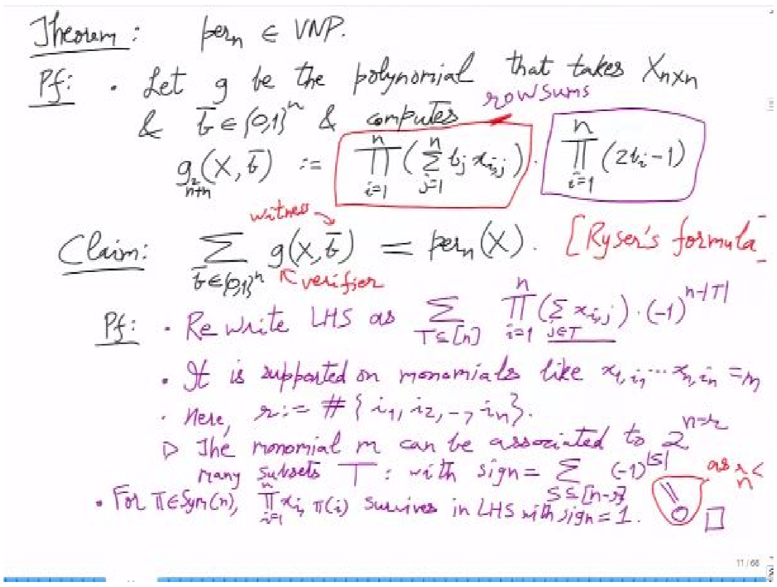
Arithmetic Circuit Complexity
Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 3

Last time we were showing permanent in VNP, any questions?

(Refer Slide Time: 00:18)



Theorem: $\text{per}_n \in VNP$.

Pf: • Let g be the polynomial that takes $X_{n\times n}$ & $\bar{t} \in \{0,1\}^n$ & computes

$$g_{n^2+n}(X,\bar{t}) := \boxed{\prod_{i=1}^{n}\left(\sum_{j=1}^{n} t_j x_{i,j}\right)} \cdot \boxed{\prod_{i=1}^{n}(2t_i-1)}$$

row sums

Claim: $\sum_{\bar{t}\in\{0,1\}^n} g(X,\bar{t}) = \text{per}_n(X)$. [Ryser's formula]

witness

verifier

Pf: • Rewrite LHS as $\sum_{T\subseteq[n]} \prod_{i=1}^{n}\left(\sum_{j\in T} x_{i,j}\right)\cdot(-1)^{n-|T|}$

• It is supported on monomials like $x_{1,i_1}\cdots x_{n,i_n} =: m$

• Here, $r := \#\{i_1, i_2, \ldots, i_n\}$.

▷ The monomial m can be associated to 2^{n-r} many subsets T: with sign $= \sum_{S\subseteq[n-r]} (-1)^{|S|} \overset{\text{as } r<n}{=} 0$

• For $\pi \in \text{Sym}(n)$, $\prod_{i=1}^{n} x_{i,\pi(i)}$ survives in LHS with sign $=1$. □

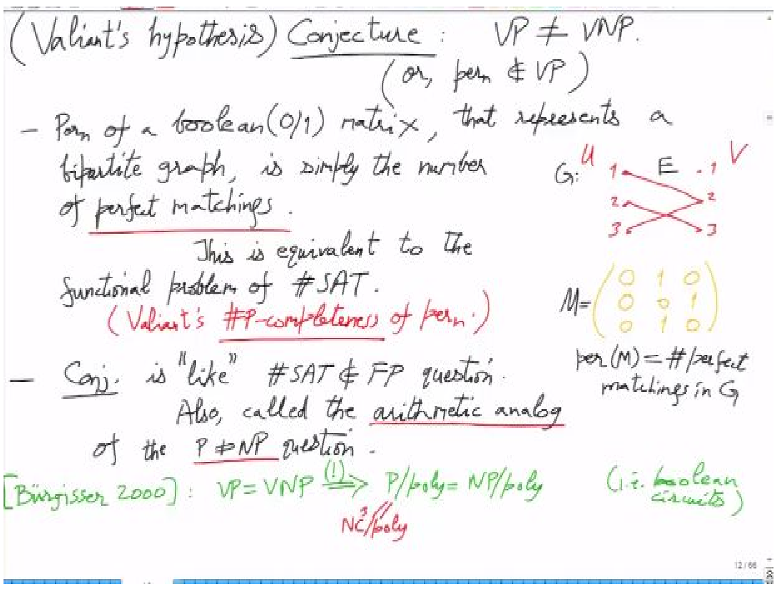
There were some questions after the class and there is a small mistake. So recall this definition of g, right which is basically a product of the row sums. These are the row sums. You are multiplying n of these row sums. And so in the claim when we are showing that some of these products is permanent, $\Sigma\prod$ is permanent.

What we should correct here is suppose the monomial we are looking at is $x_{1,i_1}x_{2,i_2}\ldots\ldots x_{n,i_n}$ say r is the number of distinct subscripts. So let $r := \#\{i_1, i_2\ldots\ldots\ldots i_n\}$ okay and then after this the calculation here will not be 2^{r-1} ,but in terms of $n-r$. we will just change it to this 2^{n-r} and this is over $n-r$.

And then as long as the $r < n$ this sign will be 0. This will come out to be 0 as $r < n$. And if $r = n$ then it means that $i_1\ldots\ldots i_n$ are distinct. In the distinct case you will get the monomials corresponding to permutations and they will come with sign 1.

So that is exactly then computing permanent, With this correction let us proceed. Now let us write down a conjecture which was made by Valiant. So it is called Valiant’s hypothesis.

(Refer Slide Time: 02:43)



(Valiant's hypothesis) Conjecture : $VP \neq VNP$.
($\text{or}, \ per_n \notin VP$)

- Per$_n$ of a boolean(0/1) matrix, that represents a bipartite graph, is simply the number of perfect matchings.
  This is equivalent to the functional problem of #SAT.
  (Valiant's #P-completeness of per$_n$.)

G: U, V, E; 1, 2, 3; 1, 2, 3

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

per(M) = #perfect matchings in G

- Conj. is "like" #SAT $\notin$ FP question.
  Also, called the arithmetic analog of the $P \neq NP$ question.

[Bürgisser 2000] : $VP = VNP \overset{(!)}{\Longrightarrow} P/poly = NP/poly$ (i.e. boolean circuits)
$NC^3/poly$

12 / 66

It is his hypothesis which means, which is still unproved. So it is a conjecture, Valiant’s conjecture. This says that VP≠VNP. Again VP is the set of those polynomials that can be computed by small arithmetic circuits and VNP is the set of polynomials that can be written as a big sum of small arithmetic circuit and Valiant’s hypothesis claims that these two are different .

So obviously VP $\subseteq$ VNP. So the conjecture is saying that there is a polynomial which you can write as a big sum of small circuits, which cannot be written as a small circuit, overall. And we will not be doing that prove, but permanent is essentially VNP complete. This alternately you can see $per_n \notin VP$

“Professor - student conversation starts” Here in completeness how do you define the reduction? **Professor:** The reduction will just be is there a polynomial in which if you set some variables, you get your favorite polynomial projections. By projections and yeah the bigger polynomial should have a small arithmetic circuit. So is there a small arithmetic circuit which on fixing some variables gives you your polynomial.

“Professor - student conversation ends”.

Permanent is not in VP. This is an alternative formulation of Valiant’s hypothesis. These questions are the central questions in algebraic complexity theory and even in complexity theory because of the following connection. If you look at permanent of a Boolean matrix,(0/1)matrix suppose this (0/1) matrix is the it is the adjacency matrix of some bipartite graph.

That represents a bipartite graph. let us draw a small example. Say you have three vertices on each side. So 123 here and 123. This is left side is U right side is V. How do you represent it as a Boolean matrix? It will be a 3 by 3 matrix where 1, 2 so 1, 1 will be 0. 1, 2 will be 1. 1, 3 will be 0. 2, 1 is 0. 2, 2 is 0. 2, 3 is 1 and so on. So this is the matrix that represents this bipartite graph.

U V and the edges are in the middle E. So this matrix M , if you look at the permanent of this matrix, what does it represent? You can show that permanent of M is in this graph G, the number of matchings in G, perfect matchings. So this is easy to see because when you look at the permanent of the matrix you will by definition it is the sum of monomials, .

Every monomial basically corresponds to picking ones in those places, if they are present otherwise this monomial this product will not contribute. You can pick these ones only when there are these disjoint edges to pick . So that every monomial product corresponds to a perfect matching. And then you are taking the sum without sign. This gives you the exact count.

Permanent of a Boolean matrix actually gives you the count of perfect matchings. This is simply the number of perfect matchings. In practice given a graph, can you compute the number of matchings? Can you check whether there is a perfect matching? So checking whether there is a matching this has a practical algorithm, which is a very famous algorithm given in the 60s or 70s.

But to count them is considered to be very hard, it is harder than NP hard problems. Okay. So permanent on even this (0/1) matrix is actually hard to compute in practice. This is actually equivalent to, this is equivalent to the functional problem of # SAT. , #SAT is the problem of it is the counting version of SAT, formula satisfiability, Boolean formula satisfiability.

Given a Boolean formula, how many satisfying assignments are there? For n variables it can be a number between 0 to 2^n, . Even testing whether it is positive this is the NP complete problem and here you are asking to actually count it. Clearly this is this you would expect to be harder much harder than just SAT. So there is no hope of solving this in practice.

So this also is due to Valiant. So this is called the Valiant's # P completeness of permanent which is Boolean permanent. So Valiant showed that permanent, the Boolean permanent or computing the permanent of a Boolean matrix which corresponds to a graph. This is equivalent to a #SAT. And #SAT defines as a class which is called #P.

I would not go into the definition or the proof of this because this we cover usually in competition complexity course. This is to do with the Boolean world. Here in the arithmetic world, now we want to study permanent as a polynomial okay. So can you compute permanent as a polynomial and obviously, this should remain a hard question, because intuitively if you can write the permanent polynomial as a small arithmetic circuit, then there is in all likelihood, you will also be able to solve permanent in practice.

But that would mean that you are solving not only SAT but #SAT, . You are solving all these hard optimization problems if you solve this. So that is not generally believed. This motivation then suggests that the conjecture, Valiant's conjecture this is like the conjecture of #SAT not in polynomial time, functional polynomial time, .

It is not formally correct, but when Valiant conjectures that VNP $\neq$VP which is $per_n \notin$ VP it is like saying that if you wanted to count the number of satisfying assignments that you cannot do in polynomial time . Arithmetic circuit size corresponds to Boolean time or time in practice and so you could think of the Valiant, think of Valiant hypothesis as this and both these things we would expect to be true, okay. But there is no proof. These are open questions.

This we call the arithmetic analog of the P $\neq$ NP question. So P $\neq$ NP is the classical question in the Boolean world or in the real world. And the algebra or the arithmetic analog of it is the VP $\neq$VNP. Is there a formal connection between them? Formally there is no direct connection known, but there is a result due to Burgisser which I will state.

It is not hard. So you can even do this as an exercise. It says that if VP =VNP then what happens? S P and NP are the same /poly. So , P / poly = NP /poly.

I will not go into the details of this implication. There is some little assumption based on what field you are working in. But if you are working in a finite field, then this implication is exactly true.

Burgisser showed that VP =VNP would imply that P /poly = NP / poly, which is much stronger than saying P=NP. Well, I mean technically it is incomparable but, I mean from this you cannot really deduce whether P is equal to NP or P is different for NP. Because, when you talk about the P and NP classes, you are talking about Turing machine.

You are not talking about Boolean circuits. So, P / poly is actually the problems that you can solve using Boolean circuits and NP/ poly is also defined in a similar way using Boolean circuits. The meaning of efficiency in Boolean circuits and the meaning of non-determinism in Boolean circuits, these two things are the same. This is what the equality is saying.

Equality is saying that non-determinism and efficiency are the same. Intuitively we do not believe this. Intuitively we believe that non-determinism is far stronger than efficiency. But, this is talking about the Boolean circuits and original P $\overset{?}{=}$NP question is for Turing machines. On the left hand side you have an arithmetic circuit statement and on the right hand side you have RHS you have a statement about Boolean circuits.

This is pretty satisfactory although not completely satisfactory. But it suggests that if you want to show P different from NP then you should first show VP different from VNP, okay. You cannot just, most probably you cannot skip this step. In fact, it shows something far stronger. So you may or may not know the definition of this class, but Burgisser also shows his proof also shows that this is an NC^3 /poly.

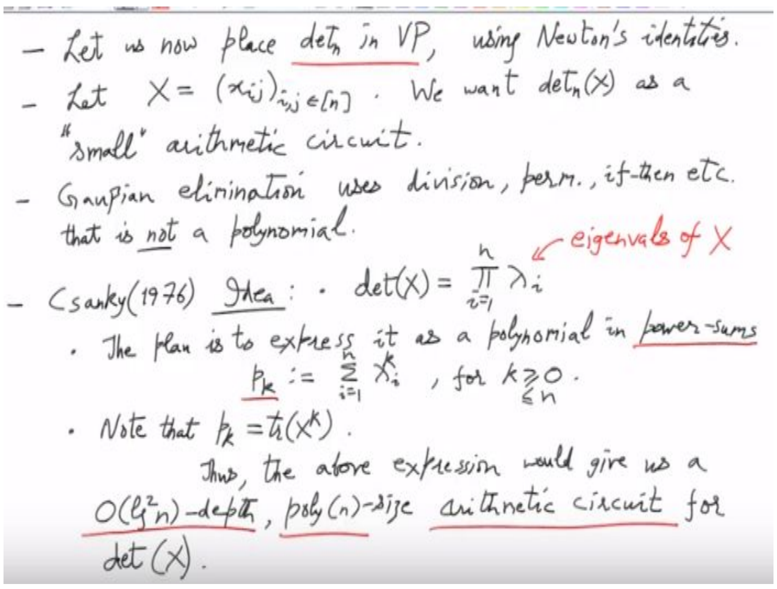
Do you know what is NC^3 or what is NC? These are Boolean circuits where the depth is only log^3 okay. So this is again, this exactly models very fast parallel algorithms. On an input size of n the parallel time complexity is only log^3. It is exponentially faster than what you would expect.

So if you show VP = VNP then Burgisser's proof actually tells you that NP / poly is not just equal to P / poly, it is even equal to NC^3. So non-determinism is the same as efficient parallel algorithms. You can solve in a way it is saying that you can solve SAT in parallel, fast parallel time okay. So everything on the RHS is actually unbelievable and hence LHS is also unbelievable.

That is the import of this implication. When you are working over characteristic zero then Burgisser's proof requires the assumption of GRH, generalized human hypothesis. This is why I put an exclamation mark but that is not very important. I think everybody in this class believes GRH, so you do not have to worry about those specifics. Right, so let us now move to easier problems than VNP.

The easier problem we will move to is determinant. So where is that? It is in VP. What is the proof of that?

(Refer Slide Time: 18:29)



- Let us now place $\det_n$ in VP, using Newton's identities.
- Let $X = (x_{ij})_{i,j \in [n]}$. We want $\det_n(X)$ as a "small" arithmetic circuit.
- Gaussian elimination uses division, perm., if-then etc. that is not a polynomial.
- Csanky(1976) Idea: • $\det(X) = \prod_{i=1}^{n} \lambda_i$ $\swarrow$ eigenvals of X
  - The plan is to express it as a polynomial in power-sums $p_k := \sum_{i=1}^{n} \lambda_i^k$, for $k \geq 0$, $\leq n$.
  - Note that $p_k = tr(X^k)$.

    Thus, the above expression would give us a $O(\lg^2 n)$-depth, $poly(n)$-size arithmetic circuit for $\det(X)$.

Let us now place determinant in VP, okay. This we will do using some classical identities from 1800s which are called Newton's identities. This is an old proof. Once we have done this we will also give you a we will also see a modern proof which will be far more efficient and cleaner. But the first proof of determinant in VNP was this. That I will now present using Newton's identities.

What do you want to compute? You have a fully symbolic matrix entries, $X = (x_{i,j})_{i,j \in [n]}$. So we want determinant as a small arithmetic circuit. And the only thing currently, the only thing shown in this class is the definition of determinant which requires $n!$ monomials. And now we want $poly(n)$ sized arithmetic circuits, right. So what is the plan?

The plan is to write determinant in some other way, which is not too big and then implemented as okay and so this arithmetic circuit will be computing determinant as a whole. It will not be evaluating values of the determinant. It will be actually computing determinant polynomial which is very different to what you do when somebody asks you to compute the determinant of a matrix say of entries 1, 2, 3, 4 .

There you do Gaussian elimination and you get a number, but that is not the same as saying that you have computed determinant polynomial as a whole. We will now achieve that through identities. Let me, anyways, write that down. Gaussian elimination uses, the problem is that it uses division and many other things. Uses division, permutation, if-then-else etc.

That we cannot naively write as a polynomial. It is using non arithmetic operations, okay. So that does not apply to our current problem. So here I will present Sankey's idea. Sankey's algorithm it is an algorithm. The idea here is what is the relationship of determinant with the eigenvalues. $det(X) = \prod_{i=1}^{n} \lambda_i$, eigenvalues of X.

Now we would not worry too much about where these eigenvalues live because they live in a complicated place. Your matrix X has just formal variables as entry. So if you try to compute eigenvalues they will not be complex numbers they will be actually complicated functions in these variables and technically they are in the algebraic closure of the function field.

We should not worry about its representation And that will be an amazing thing that without worrying about the specifics of λ_i still we will get they will help you in getting an identity. That will be very explicit okay. So let us just start with this, this is the first identity. And what is the plan after this? So the plan is to express it as a polynomial in the power sums.

Power sums means, well let us say $\Sigma\lambda_i$, or $\sum\lambda_i^2$ or $\sum\lambda_i^3$ and so on. These are the power sums. So as a function of the power sums you want to write the product. You must have seen this.

"Professor - student conversation starts" Sorry, what do you mean, characteristic polynomial. Right. Symmetric polynomial you can write it as a power sum. **Professor:** Yeah, so it has nothing to do with the characteristic polynomial, it is just any, product of anything you can express as, as a function of the power sums. **"Professor - student conversation ends".**

And in general as Abhibhav is saying that any symmetric function in lambda is, well lambda is really arbitrary. We will not use their eigenvalue property and is elementary symmetric function or actually any symmetric function you can write as a function of, in fact as a polynomial of power sums.

So this follows from Newton identities. This you must have seen or even computed in school. We want to prove that in full generality, Power sums are the kth power sum is basically the sum of kth powers. That is

$$p_k := \sum_{i=1}^{n} \lambda_i^k \text{ , for } k \geq 0$$

Why can you write $\prod \lambda_i$ as a polynomial n, p_k 's where k goes up to n. So let me also add that.

You will not need to go beyond n. So just from 1 to n, this p_1 to p_n will be enough to produce, . This is very mysterious. It is even hard to conjecture unless you have some experience. But anyways, this is the plan. We will implement this plan and let us answer the question, the next question why is it helpful? If you prove such a thing, can you write p_k as something, some small circuit.

What is the relationship of p_k with X? In fact, what is the relationship of p_1 with X? $p_1 = \Sigma\lambda_i$ right. It is the trace of X right exactly. That is the beauty. Note that $p_k = tr(X^k)$. So now this has a small circuit because X raised to so say case 2. So X^2 you can compute just by multiplying and adding the entries.

There is a trivial arithmetic circuit that can compute X^2 all the entries of X^2, in particular the diagonal entries of X^2 and then there is the addition gate that adds up those diagonal entries and you get trace of X^k, okay. You just have to compute the diagonal entries of X^k to compute trace and any entry of X^k can be computed via a circuit.

Now as k grows to n what you can do is something like repeated squaring you can just compute X^2, then you can compute X^2. If you have completed X^2 so these are you have n^2 outputs defining the matrix X^2 or describing the matrix X^2. you can then square it and the same. You can actually do this in just login steps, this process.

Even and it will ultimately turn out that the even the depth of the circuit will not be too much. , That is the point. It will be a really nice circuit not only the size is small, even the depth is small. That is the advantage of working with p_k. To begin with, it was not really clear, how will λ_i or λ_1 will be useful at all because λ_1 is something very complicated in terms of X.

But the thing is that it is not just λ_i you are actually looking at the product, and the product gets related to traces. And the traces are simple for arithmetic circuits to compute. Thus the above expression would give us O($log^2 n$)-depth, $poly(n)$-size arithmetic circuit for the determinant. The important not only is the size there which is actually all we wanted. So everything else is bonus.

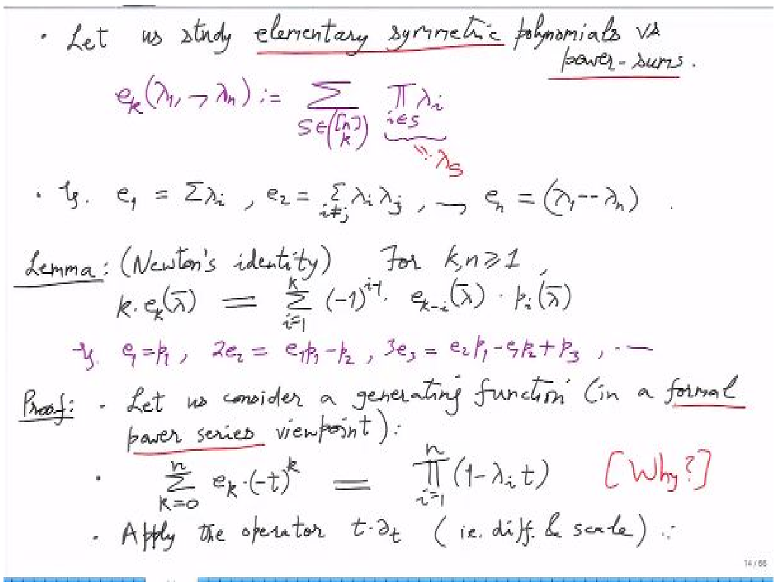
We just wanted poly(n) size. That it surely will be. This you can immediately see because of the X^k computation. The depth you will have to analyze carefully. I leave it as an exercise. But you will be able to show that the depth is also very small. It is only $log^2 n$. In fact, this was the reason why people studied this in the 70s because people wanted a very fast parallel algorithm for determinant okay.

The motivation was actually practical. Determinant is a very practical problem. In fact, all these amir companies, the only computation they do is either they multiply matrices or they solve a linear system. If you are solving a linear system, you have to invert, invert a matrix. You have to compute determinant. If you can do it in parallel, you can buy many computers and you implement determinant in parallel.

This is saying that for a huge n if you make that kind of, if you buy many processors then after this one time investment you can actually solve determinant on various instances in only $log^2 n$ - time once you have set up the parallel architecture. which can be very important if you work if you want real time solutions.

This depth is actually the motivation why people studied it, but we currently only care about the size. This is enough to show the determinant is in VP. I leave the details in as an exercise. Let us do the main thing which is connection with power sums. How is that done? How is that proved? We will prove the general thing.

(Refer Slide Time: 31:59)



- Let us study elementary symmetric polynomials vs power-sums.

$$e_k(\lambda_1, \to \lambda_n) := \sum_{S\in\binom{[n]}{k}} \underbrace{\prod_{i\in S} \lambda_i}_{=: \lambda_S}$$

- Eg. $e_1 = \sum \lambda_i$, $e_2 = \sum_{i\neq j} \lambda_i \lambda_j$, $\to$ $e_n = (\lambda_1 \cdots \lambda_n)$.

Lemma: (Newton's identity) For $k, n \geq 1$,

$$k \cdot e_k(\bar{\lambda}) = \sum_{i=1}^{k} (-1)^{i-1} \cdot e_{k-i}(\bar{\lambda}) \cdot p_i(\bar{\lambda})$$

Eg. $e_1 = p_1$, $2e_2 = e_1 p_1 - p_2$, $3e_3 = e_2 p_1 - e_1 p_2 + p_3$, $\cdots$

Proof: - Let us consider a generating function (in a formal power series viewpoint):

- $$\sum_{k=0}^{n} e_k \cdot (-t)^k = \prod_{i=1}^{n} (1 - \lambda_i t) \quad [\text{Why?}]$$
- Apply the operator $t \cdot \partial_t$ (ie. diff. & scale) :

14 / 66

Let us study elementary symmetric polynomials versus power sums. What is this identity? Whose existence is not clear? I mean, this is just some claim that I am making, but if you think about it, it is not clear why this connection should even exist. The elementary symmetric polynomial e_k $(\lambda_1, .., \lambda_k)$ is defined as so this would be what? .

So k times, you pick k things, multiply them and then do this in all possible ways. It is about subsets, k subsets of [n]. You pick an S, and then you multiply these λ_i. That is it. So e_1 for example, is just the sum and then e_2 is product of λ_i, λ_j, for distinct i, j.

This thing we may sometimes write as λ_S. Okay, so it is just $\prod_{i \varepsilon S} \lambda_i$.

Okay, so example is $e_1 = \sum\lambda_i$, $e_2 = \sum_{i \neq j}\lambda_i \cdot \lambda_j$, .., $e_n = \lambda_1 \cdots \lambda_n$ and e_n has only one monomial which is the full product. So these are the n elementary symmetric polynomials. Now the problem does not happen in the ends, the problem happens in the middle because there you are looking at $n/2$ subsets and they are exponentially many. So that is a large sum right.

Computing those things is complicated, also in practice. How do you make it efficient. This is what we call in the course Newton's identity, this is a recurrence to simplify our life. So for k, n $\geq$ 1 it writes

$$k \cdot e_k(\overline{\lambda}) = \sum_{i=1}^{k}(-1)^{i-1}.e_{k-1}(\overline{\lambda}).p_i(\overline{\lambda})$$

It is just this convolution type of product of e's and p's okay. This is a this is a recurrence which means that if you know the expression for e_k up to $.e_{k-1}$ if you know the relationship with power sums then this immediately gives you the relationship of e k with the power sums and you only need power sums up to p_k. For e_k you only need power sums up to p_k . You do not need anything else okay.

This is exemplified as so $e_1 = p_1$ unsurprisingly. So $2e_2 = e_1p_1 - p_2$ which is then $p_1^2 - p_2$. And $3e_3 = e_2p_1 - e_1p_2 + p_3$ which you can again right using the previous things. **"Professor - student conversation starts"** Apriori it is not even clear that the power sum representation of p_n is like it will be a polynomial in the p_n . **"Professor - student conversation ends".**

That is a good point, but remember that you have to write down in arithmetic circuit. So assuming that even to e_{k-1} you know the arithmetic circuits. On top of this you can produce e_k. Yes, that is important. Yeah and maybe you will appreciate the fact that you actually this actually produces a circuit not a formula as far as small size is concerned, it because in formula you will be not you will not be able to reuse the output.

And if you have to implement this recurrence you have to keep reusing it. Otherwise, things will start blowing up. So you really use the power of a circuit and so more than a formula and then you can write e_k as a function of ultimately the circuit in terms of λ_1 to λ_n. Okay, so let us do the proof details, finally. How do you think you will prove this? It is impossible to guess this.

But once I have given you this, how will you prove it? But even induction does not look easy. Because what is induction telling you. It is saying that e_{k-1} is a function of lower p's. But then who will dare substitute that expression. So we will actually give a cute proof using generating functions. Let us consider right a generating function in a formal power series where as a formal power series, okay.

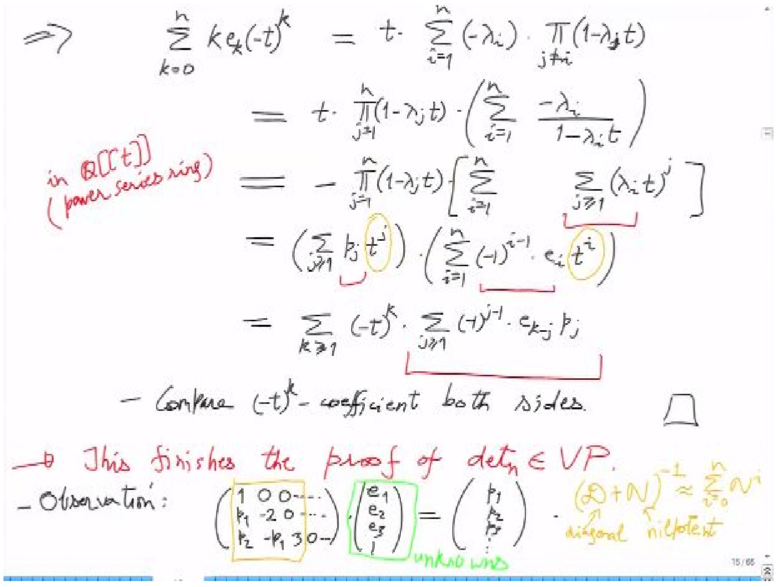
The mathematical tool being used is actually formal power series. So it is an infinite sum that we will look at. So what is the generating function for e_k? So you should look at $\sum_{k=0}^{n} e_k(-t)^k$. So this is the generating function. There is no reason to stop at n. But, if you look at e_{n+1} it is already zero.

This is why we are truncating at n but this you can think of as an infinite sum goes beyond t^n and so this is this is the generating function of the e's. There is another way to write this. You can factorize this to get what? Exactly. So you know that the roots of this polynomial by fixing t or λ right or 1 over λ kind of. you can actually factorize it as $(1-\lambda_i t)$.

So just you can check this. This just follows from polynomial multiplication. You just look at the RHS and the way you multiply polynomials you will actually get elementary symmetric polynomials, all these e_k's and yeah so now this is a good identity to start with to get a recurrence for e_k. So do you remember the high school method of getting recurrences? Yeah you have to differentiate.

You differentiate and then you will be done. Basically apply the operator again for notational convenience we will apply $t \cdot \partial_t$, okay. So we differentiate and scale up differentiate and scale. Now what is the result?

(Refer Slide Time: 42:37)



$$\Rightarrow \sum_{k=0}^{n} k e_k (-t)^k = t \cdot \sum_{i=1}^{n} (-\lambda_i) \cdot \prod_{j \neq i} (1-\lambda_j t)$$

$$= t \cdot \prod_{j=1}^{n} (1-\lambda_j t) \cdot \left(\sum_{i=1}^{n} \frac{-\lambda_i}{1-\lambda_i t} \right)$$

in $\mathbb{Q}[[t]]$ (power series ring)

$$= - \prod_{j=1}^{n} (1-\lambda_j t) \left[\sum_{i=1}^{n} \underbrace{\sum_{j \geq 1} (\lambda_i t)^j}_{} \right]$$

$$= \left(\sum_{j \geq 1} \underbrace{p_j}_{} t^j \right) \cdot \left(\sum_{i=1}^{n} \underbrace{(-1)^{i-1} \cdot e_i}_{} t^i \right)$$

$$= \sum_{k \geq 1} (-t)^k \cdot \underbrace{\sum_{j \geq 1} (-1)^{j-1} \cdot e_{k-j} p_j}_{}$$

- Compare $(-t)^k$ - coefficient both sides. $\square$

$\longrightarrow$ This finishes the proof of $\det_n \in VP$.

- Observation: $\boxed{\begin{pmatrix} 1 & 0 & 0 & \cdots \\ p_1 & -2 & 0 & \cdots \\ p_2 & -p_1 & 3 & 0 \cdots \end{pmatrix}} \cdot \boxed{\begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \end{pmatrix}}$ unknowns $= \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \end{pmatrix}$. $(\underset{\text{diagonal}}{D} + \underset{\text{nilpotent}}{N})^{-1} \approx \sum_{i=0}^{n} N^i$

This will on the LHS this will give you $\sum_{k=0}^{n} k e_k (-t)^k$. And on the RHS you will get, so on the RHS you had a product. When you differentiate it using you can use the Leibniz identity which is you differentiate one factor at a time, take the sum. That will give you $t \sum_{i=1}^{n} (-\lambda_i) \prod_{j \neq i} (1-\lambda_i t)$ we are only differentiating the ith factor . That is the sum, .

Any questions about this? We have differentiated and then multiplied by t, just two write a better expression. And once you do this it will be better to actually re-express the RHS as a sum of fractions. Let us bring out the full product and be left with $-\lambda_i/(1-\lambda_i t)$.

(That is what we are left with and there is this full product. Now what?

Now you observe that $(1-\lambda_i t)$, its inverse exists in the power series, in the power series ring, so you expand that out. So $(1-\lambda_i t)$ will expand as, let me delete that. Let me absorb the t first. It is just this double sum, right. And there was a hanging, there will be a hanging minus sign basically you get, in the inverse you get

$$(1 + \lambda_i t) + (\lambda_i t)^2 \ldots\ldots..$$

And there is a $\lambda_i t$ already sitting outside. So take it inside. So your series starts with the $\lambda_i t$ and then its square cube and you are summing it up. And then for various i's you take the sum. So it becomes a double sum. This technically is actually happening in the power series ring. So this power series ring is you can take over rationals. This is this ring.

If you want to be technical, about where does this computation happen, it is happening in the power series ring. In this power series ring basically the elements of this ring are an infinite sum of t monomials but taken in this increasing order. The bigger the power of t is, the smaller that term is considered. This is the algebraic version of analytic power series. This is called a formal power series, okay.

In this power series we have these two expressions LHS and RHS and then we should compare t monomials, . So let us we will do that but let us first bring it in a form that we want. So this part right this $\sum_{j\geq 1}(\lambda_i t)^j$ which we have to swap the Σ. Let us first swap that.

Let us have this thing as, okay let me just write the expression before giving the explanation. That is the expression.This double Σ, you can swap the sigma and then you are looking at $\sum_{j\geq 1}(\lambda_i t)^j$ for all these i's. That is p_j. So the coefficient of t^j is just p_j. That gives you this term. And next you look at the product of $(1 - \lambda_j t)$ which before just this.

Look at the coefficient of t^i there and that is just the elementary symmetric sum with the appropriate sign. So you have now these two polynomials, well first one is infinite but will only go up to t^n because there is no point going beyond. you have now these two polynomials and their product is equal to LHS. So now you should just simply compare the t^k coefficients.

And when you compare you will get to this. Let us just write it in terms of t^k. So minus if you look at the coefficient of $(-t)^k$ what you will see is this, This part is the coefficient $(-t)^k$ above. To get t^k you have to look at $j+i=k.$ Because you have this term t^j and t^i here.

You need to look at those j and i's such that $j+i=k$. Let us so that will be j and $k-j$. So j will give you p_j and the (k-j) will give you e_{k-j} with the sign, that is the sum. Now when you compare the coefficients of $(-t)^k$ both sides, you get Newton identity as claimed. Now finally, compare $(-t)^k$ coefficient, both sides. That is the proof.

Any questions? That should already convince you that determinant is in VP. Because we have related the determinant which is product of λ_i s to a polynomial in the power sums. And this is a very simple expression as the recurrence is very simple. It is easily implementable as a circuit, poly size. And the final point is that the power sums are trees of matrix powers and matrix powers fully can be computed again as small circuits, $poly(n)$ size circuits.

So with poly and size part should be clear. We have shown the determinant is in VP. This finishes the proof of determinant in VP. That must we have already done. Any questions about this statement? But this does not stop here. This is actually giving you a lot more. This analysis actually gives you a practical implementation for determinant that is extremely fast in the parallel sense.

A parallel algorithm is obtained. I will not go into all the details. Maybe I will put it in the assignment. But, one observation is that the recurrence that you have got right it is you can think of it as, you are interested in elementary symmetric polynomials.

You are interested in e_1, e_2, e_3 finding them and the recurrence is actually giving you a linear system to find them where the entries are, so e_1 is the same as p_1.

You have p 1 here and let us say you just put p_1, p_2, p_3 here and so on. So e_1 is p_1 which is that gives you the first row from the recurrence, I mean using the recurrence. Then the next recurrence for k = 2 tells you that p_2 is, so e_1 p_1 - $2e_2$ and then zero. Then the third recurrence for k = 3 gives you p_3 is equal to e_1 p_2 - $e_2 p_1$ + 3 e_3 .

If you look at this just a 3 by 3 part what do you see? This is the matrix. Why is it special? Well is a lower triangular matrix, right. You have a linear system where the e's are the unknowns. This is what you want to find. These are the unknowns. And to solve this linear system you have to invert the triangular matrix, right. A tree boils down to the inversion of the triangular matrix and you want to make this implementation faster.

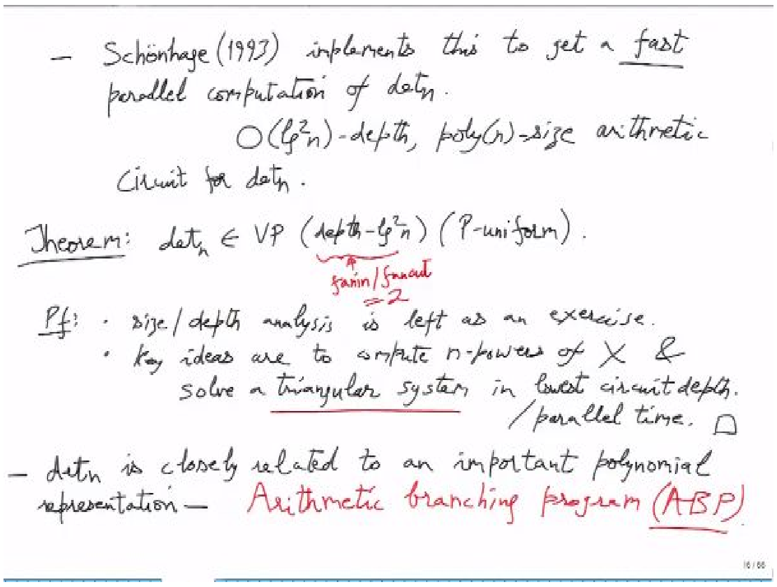
There is a very nice way to invert triangular matrices. You write your triangular matrix as a diagonal matrix + nilpotent matrix and you want to compute this. D is diagonal. And N is nilpotent. Nilpotent because if you raise it to, let us say the dimension of the matrix, it vanishes. how do you compute $(D+N)^{-1}$ in a fast way?

And usually inversion of a matrix is pretty complicated because you have to at least compute the determinant which will put us in a vicious cycle because our goal was to compute the determinant and now so inverse also will, so somehow you have to break the cycle. So you have to find a different way to compute this inverse which will be yeah. The amazing thing is that this also can be done using power series.
The same power series. So think of this as $(1-n)^{-1}$ and that is equal to $1 + n + n^2 + n^3 + \ldots$ and after a point it vanishes. So this actually is a, this is essentially sum of powers of n, something like this. So you just have to compute the powers and then sum them, okay. This is the idea. This is why solving the linear system actually does not use, does not need determinant.

It can be done just by powering an addition. And when you formalize this, when you do this implementation now actually it is after this point it is quite straightforward. Once you do this you will see that even the depth of the circuit is very limited . And that gives you a very fast parallel algorithm to compute determinant. So we, so as you saw here we started with this abstract looking model and we prove the statement that determinant is in VP.

But the proof of it is actually giving you even a new way to compute determinant in practice, which is extremely fast and which you cannot guess. You cannot guess it at all from looking at the definition of determinant, okay. So this implementation details I leave for the assignment.

(Refer Slide Time: 58:46)



– Schönhage (1993) implements this to get a <u>fast</u> parallel computation of $\det_n$.
$O(\lg^2 n)$-depth, poly(n)-size arithmetic circuit for $\det_n$.

<u>Theorem:</u> $\det_n \in VP\ (\underbrace{\text{depth-}\lg^2 n}_{\substack{\text{fanin/fanout}\\ =2}})\ (P\text{-uniform})$.

<u>Pf:</u> • size/depth analysis is left as an exercise.
• key ideas are to compute n-powers of X & solve a <u>triangular system</u> in lowest circuit depth /parallel time. $\square$

– $\det_n$ is closely related to an important polynomial representation — Arithmetic branching program (<u>ABP</u>)

16 / 66

Schönhage idea implements this to get a fast parallel computation of determinant and no big constants are used. Although VP allows the use of arbitrarily large constants. But in this proof you are using very small constants. The theorem that you will get is that the depth is only $\log^2 n$, the size is only poly for determinant n. Let us just state it as a theorem that $\det_n$ ε VP that is actually depth $\log^2$ n and another qualification is that as you vary n can you describe the circuits?

Because in the definition of VP that is left totally arbitrary. There may not be any connection between the circuit for n and the circuit for n +1, that is the built in non-uniformity in VP. But that is not what this proof is giving you. So here the proof is actually giving you an explicit fast way to compute the circuits, describe the circuits as well. It is P-uniform .

And another point here is that this depth is actually with the fanin and fanout 2. So the fanin and fanout is also very small, okay. So it is constant. Depth is just $\log^2$. Every circuit for every for a given n, the circuit can be computed by a Turing machine in polynomial time. I mean this gives you everything except maybe some you might expect some more improvements, but other than that asymptotically it is giving you or functionally it is giving you the best possible results for determinant.

You may ask whether $\log^2$ can be reduced to log. You can ask whether the size is if it is n^5 can it be reduced to n^2. So those questions you can still ask but yeah even this is quite a strong thing for a polynomial like determinant, .The proof is as we have already sketched. So we leave the size and depth analysis as an exercise. And key ideas are to compute n powers.

And the second idea is to solve a triangular matrix equation, triangular linear system in lowest depth, which is also in parallel time. The parallel time complexity should be very fast. Fine. So this is yeah so we showed determinant in VP and more.

"Professor - student conversation starts" Sir, what should be P non-uniform if we know for n and we cannot explain. **Professor:** We, there is no fast algorithm as in the sense of Turing machines that can produce the circuit in poly n time. So it may need exponential in n time. **"Professor - student conversation ends".**

I mean within exponential in n time I think you that is a lot of time. That kind of uniformity I think you always have but the thing is whether there is a polynomial time algorithm which when given n or to be formal 1 raised to n when given n in unary then it outputs the nth circuit.

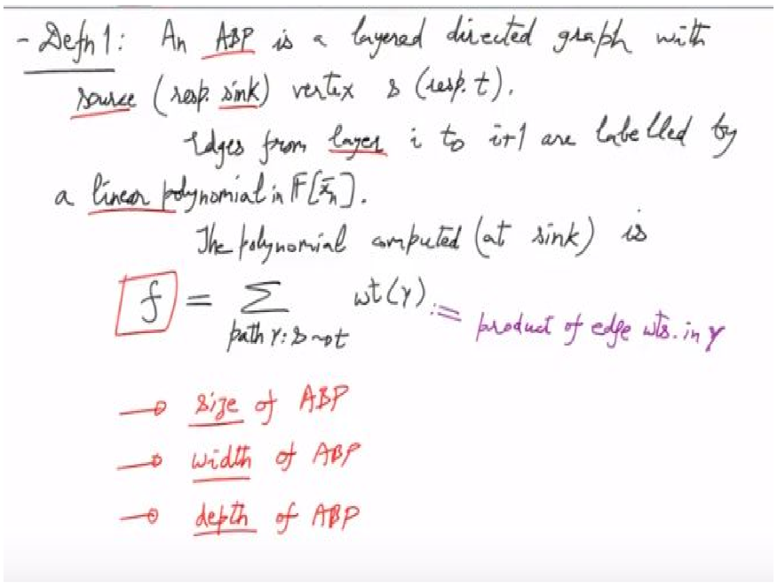
The circuit building should not be left to the user. Because that could be again that is extremely hard. How do you come up with a circuit for a given n. If n is 1 million how do you come up with the circuit that is running on 1 million inputs. So that also somebody has to provide it. So is there a Turing machine that can do it in a fast way? But in the for the definition of VP that was left free, that was left unfixed.

In the case of determinant, we are actually getting this additional property, which is why all these things are good in practice. But now we will actually go through a different route and prove something even better for determinants,. we will actually define a complexity class based on determinant. Any polynomial that you compute in that class, you can reduce it to determinant.

And you can also compute determinant in that class. We will actually make this really optimal. We will come up with a class that is determinant hard and complete. Do you know the name of this class? This is called arithmetic branching programs. Determinant is closely related to an important polynomial representation which will be different from circuits, formulas and so on.

It is a completely different representation defined in a different way. It is called arithmetic branching programs, ABP. You might have heard the term branching program. There are also Boolean branching programs. This will be obviously inspired from that. It will be an arithmetic version of the Boolean branching programs, but it will have a lot more structure.

(Refer Slide Time: 1:07:40)



- Defn 1: An ABP is a layered directed graph with source (resp. sink) vertex s (resp. t).

Edges from layer i to $i+1$ are labelled by a linear polynomial in $\mathbb{F}[\bar{x}_n]$.

The polynomial computed (at sink) is

$$\boxed{f} = \sum_{\text{path } \gamma : s \rightsquigarrow t} wt(\gamma) := \text{product of edge wts. in } \gamma$$

- size of ABP
- width of ABP
- depth of ABP

This is basically a layered graph. You can think of it from, going from left to right, so the computation moves from left to right. In every step, there are layers. It is a layered, directed graph. So an ABP is a layered directed graph with source and sink. Source vertex is called s and sink you can call t. Now edges from layer i to layer $i+1$ are labeled by a linear polynomial.

The linear polynomial you can assume is in n variables field F, . You have these edges going from the source to the sink and when you are at layer i to layer $i+1$, you will draw edges. The edges will have labels or weights which is a polynomial say $x_1 + x_2$ could be one weight or just x_1 or maybe 0 or 1, . Can also be field constant.

Once you have this picture, what do you think is the polynomial that this computes? How should you define the final polynomial that you get at the sink? In a path from source to sink whatever you whatever linear polynomials there are, you take the product and then for every path from source to sink you add these products. That is the polynomial.

The polynomial computed at sink is:

$$f = \sum_{path\ \gamma :\ s \rightsquigarrow t} wt(\gamma) := product\ of\ edge\ weights\ in\ \gamma$$

So that is it. ABP is source, sink. You have layers. You have linear polynomials as labels and finally the polynomial computed is just this f. Yes.

I do not want to see it as a circuit yet, but this expression for f is the sum of product of linear polynomial. It is sigma pi sigma. These are the three operators you are using in this order. And then what are the parameters or resource parameters for this. That when will you say that the ABP is small? Well, obviously when the graph is small, only then should you say that the ABP is small.

If the graph is large, then it is a large representation. So that is one thing. Size of the graph is one parameter. Size of ABP that is one thing. The second thing you can look at is in every in the ith layer, how many vertices do you see? So maximize over i. So that is called the width, width of ABP. So max number of vertices in a layer. And finally depth. how long is the ABP?

So depth is the longest path from source to sink, right. So first you want to optimize the size, or optimize as in, say for n variate polynomial you want an ABP of size $poly(n)$. That would be the ideal goal. Once you have that, then you can talk about other things like can I make the width small? Can I make, I mean width small as in can I make width, let us say three? Obviously it is $poly(n)$, but can I make it even smaller?

Can I make it 3? Can I make it 10? And then depth here. But that, so depth is usually correlated with the size. We do not do much with the depth. But we talk about size and width most of the time. So we will see an example and do it, I mean study this next time. Any questions? Okay.