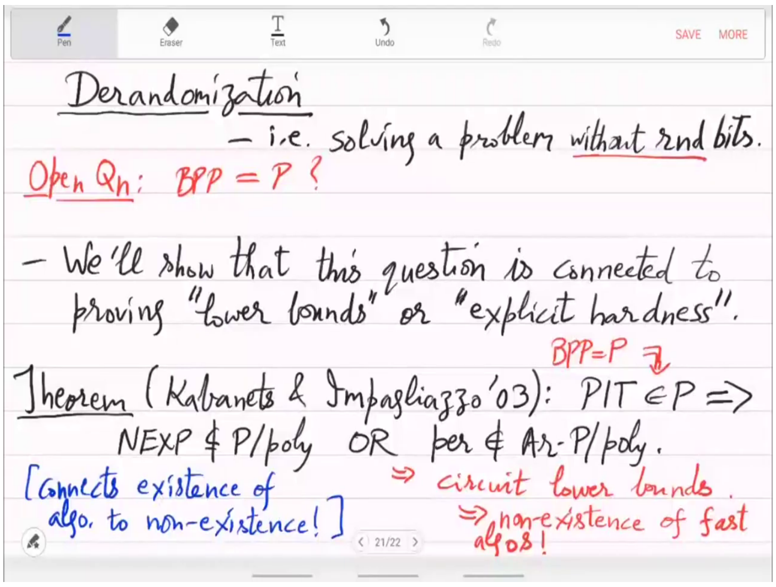
Randomized Methods in Complexity

Prof. Nitin Sexena
Department of Computer and Engineering
Indian Institute of Science – Kanpur

Lecture 03
Derandomization and Lower Bounds

(Refer Slide Time: 00:16)



Derandomization

— i.e. solving a problem without rnd bits.

Open Qn: BPP = P ?

— We'll show that this question is connected to proving "lower bounds" or "explicit hardness".

Theorem (Kabanets & Impagliazzo'03): $\overset{BPP=P \Downarrow}{PIT \in P} \Rightarrow$ $NEXP \notin P/poly$ OR $per \notin Ar\text{-}P/poly$.

[Connects existence of algo. to non-existence!]

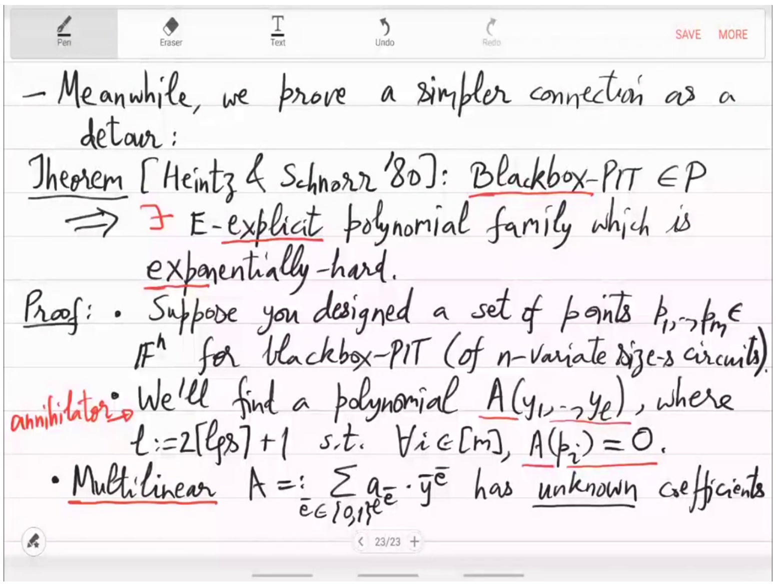
$\Rightarrow$ circuit lower bounds.

$\Rightarrow$ non-existence of fast algos!

Last time we started this new concept of Derandomization. Which is basically the question of BPP = P. Where BPP is the class of problems that are solvable in randomized polynomial time and P is the class of problem solvable in deterministic polynomial time. Whether these two sets are the same. In other words can randomness be eliminated in an efficient way. So this theorem on the slide proved by Kabanets and Impagliazzo is what we are interested in, which shows that if BPP is equal to P then there is hardness in nature.

It is an either or statement, one of these at least has to be true. Either NEXP does not have small boolean circuits or permanent does not have small arithmetic circuits. Before embarking on the proof of this which will be quite complicated and long, let me give you some hint about this connection by a different theorem.

(Refer Slide Time: 01:24)



– Meanwhile, we prove a simpler connection as a detour:

Theorem [Heintz & Schnorr '80]: Blackbox-PIT $\in P$

$\Rightarrow \exists$ E-explicit polynomial family which is exponentially-hard.

Proof: • Suppose you designed a set of points $p_1,\dots,p_m \in \mathbb{F}^n$ for blackbox-PIT (of n-variate size-s circuits).

annihilator $\to$ • We'll find a polynomial $A(y_1,\dots,y_\ell)$, where $\ell := 2\lceil \log s\rceil + 1$ s.t. $\forall i \in [m],\ A(p_i) = 0$.

• Multilinear $A =: \sum_{\bar{e}\in\{0,1\}^\ell} a_{\bar{e}} \cdot \bar{y}^{\bar{e}}$ has unknown coefficients

Meanwhile we will prove a simpler connection as a detour. So this is an old result due to Heintz and Schnorr from the 80's. This says that if black box PIT is in P. A Black box PIT means this question of polynomial identity testing that we defined before checking whether a given circuit is 0. It is the same problem you want to solve but now you cannot see the the details of the circuit.

The circuit is given as a black box so all you can do is just evaluate the circuit at some points which have to be prefixed. You have computed these points in advance and then given any black box you compute the value of the black box on these points, pre-computed points and check whether the black box computes non-zero. If it does then the black box is non-zero otherwise the black box is zero.

That problem has a very simple randomized practical algorithm: you just pick your points randomly and evaluate your black box. But is there a deterministic polynomial time algorithm? So Heintz and Schnorr showed that if you find a deterministic polynomial time algorithm for black box PIT then there exists an E explicit polynomial family which is exponentially hard.

The two important things here are first is obviously we want a hard polynomial family but at the same time we also want it to be explicit. What this theorem is saying is that if you can solve black box PIT in a derandomized way then you will have an explicit polynomial family which is hard and very hard. Let us prove this. This has a surprisingly simple proof. This is again the connection between derandomization.

Or this you can think of as the first connection between derandomization and hardness or proving lower bounds. How do you do this? Suppose you have designed a set of points $p_1, \cdots, p_m \in F^n$ for black box PIT of n variate, size s. These points that you have precomputed are $p_1, \cdots, p_m$ they have n coordinates. So the property of these points is, solving black box PIT in P. They are efficiently produced and a non-zero circuit of sizes will evaluate to non-zero on one of these points, at least one of these points say at p_i will be non-zero.

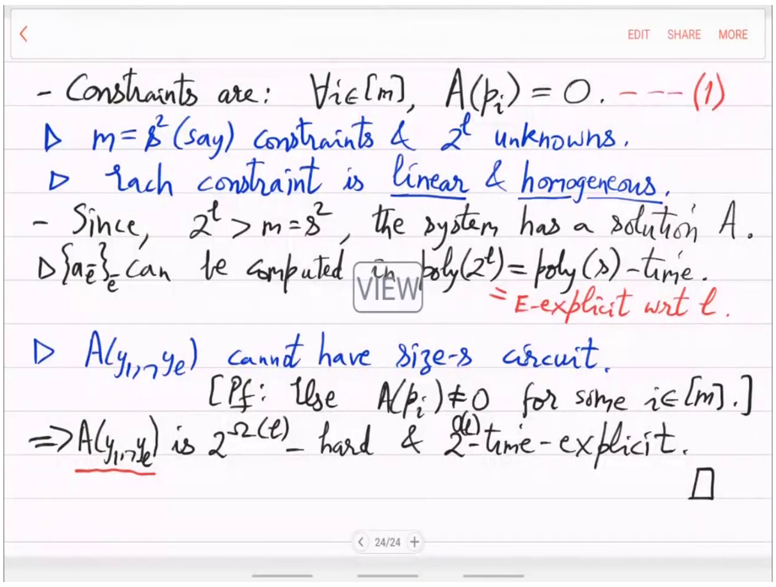
Let us define a polynomial which vanishes at all these points. What will you know about that polynomial? You will know that the polynomial which vanishes on all these points cannot have size s, it has to have size more than s. That is the potentially hard polynomial. Let us do it a bit more systematically, so we will construct or we will find polynomial A in l variables where l is let us say $2 \log s + 1$.

So we will find a polynomial A whose number of variables is around $\log s$ such that for all these points 1 to m, A vanishes. This polynomial we will call an annihilator, that is the goal. Let us try to find a polynomial A which vanishes on all these m points, an annihilator polynomial. If it exists and it is explicit then this is a good candidate for hardness. Because this circuit cannot have have size s circuits.

Remember if it had a size s circuit then it cannot vanish on all the points $p_1, \cdots, p_m$. Let us now look at the properties of this polynomial, what are the parameters, how explicit is it, how hard is it and whether it exists? And actually we will also ensure that this is multilinear. Let us do that here. Multilinear polynomial A will have coefficients $a_{\bar{e}} \cdot \bar{y}^{\bar{e}}$ for all $\bar{e} \in \{0, 1\}^l$.

We actually think of A as a multi-linear polynomial which means that y_1^2, y_2^2 or y_3^2, they do not appear; it is just individual degree one monomials that appear. So y_1 appears, y_2 appears or $y_1 \cdot y_2 \cdot y_3$ appears and so on. This $a_{\bar{e}}$, these are the unknown coefficients. These are the coefficients we want to find such that $A(p_i) = 0$.

(Refer Slide Time: 09:57)


- Constraints are: $\forall i \in [m]$, $A(p_i) = 0$. --- (1)
▷ $m = s^2$ (say) constraints & 2^l unknowns.
▷ Each constraint is linear & homogeneous.
- Since, $2^l > m = s^2$, the system has a solution A.
▷ $\{a_{\bar{e}}\}_{\bar{e}}$ can be computed in $poly(2^l) = poly(s)$-time.
= E-explicit wrt l.
▷ $A(y_1, \ldots, y_l)$ cannot have size-s circuit.
[Pf: Use $A(p_i) \neq 0$ for some $i \in [m]$.]
$\Rightarrow$ $A(y_1, \ldots, y_l)$ is $2^{\Omega(l)}$-hard & $2^{O(l)}$-time-explicit.
$\square$

The constraints are that for all i ∈ 1 to m, $A(p_i)$ vanishes. This means that there are m constraints. Here m in general will be polynomial in s but you can think of it as s^2. There are my constraints above and how many unknowns? So there are 2^l many $\bar{e}$ there. Each constraint is just giving you a linear relationship between $a_{\bar{e}}$.

Here you can see that since 2^l is more than m, you have more unknowns than constraints and these constraints are homogeneous. It is a linear combination of $a_{\bar{e}}$ equal to 0, there is no constant term. And there are fewer constraints than there are unknowns. So there is a solution; the system has a solution and that gives you the A by solving the linear system which is this.

By solving the linear system given to you by equation 1, you will be able to find A. How fast can you find it? So e can be found out in polynomial in so this is $2^l \times 2^l$ matrix. So polynomial in 2^l where l was log s. This is polynomial in s time.

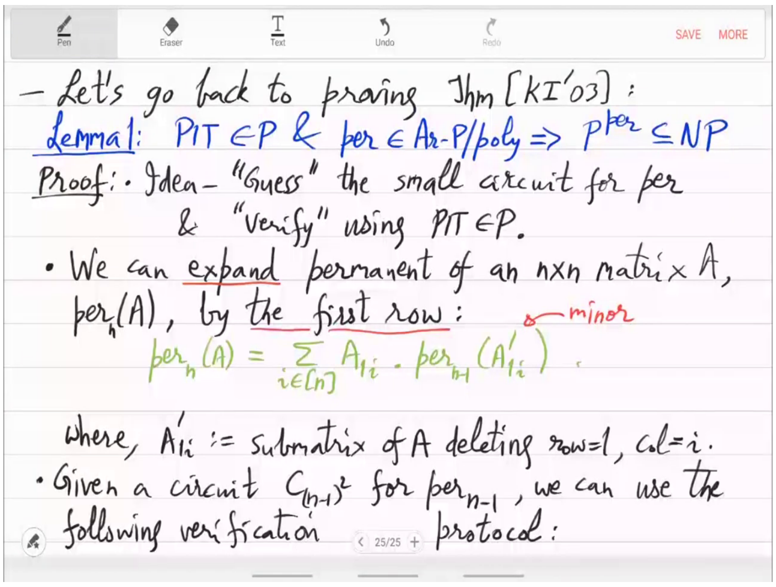
So these $\bar{e}$ can be computed in poly s time. In terms of l this is actually E-explicit. With respect to the l parameter which is the number of variables that polynomial A has, this is an E-explicit polynomial and this you can compute immediately using the points p_1 to p_m. And how hard is this? So $A(y_1,...,y_l)$ cannot have size s circuit. The proof is very simple, this is just because else $A(p_i)$ would not vanish.

Since we have picked a polynomial that vanishes at all p_i's by the property of pi's A should have size more than s which means that this polynomial A is $2^{\Omega(l)}$-hard and $2^{O(l)}$ -time-explicit. And how many variables does it have? l variables. So this finishes the theorem of Heintz and Schnorr that if you can solve black box PIT optimally then you have this l variate polynomial which is 2^l time explicit and it is 2^l size hard.

II hope that this theorem with a small proof convinces you that there is a strong connection between polynomial identity testing and existence of hard polynomials that are explicit. The proof of this is relatively easy because our assumption was that we can solve black box PIT. We are actually solving PIT without even looking at the circuit so that was a strong assumption.

Now we will go back to our theorem of Kabanets and Impagliazzo which will only assume PIT in P. This particular algorithm may even look inside the circuit and use the circuit gates to make a decision. This is a weaker assumption, it is a weaker algorithm and consequences will also be much weaker. You will get either NEXP or permanent hardness. But the proof will not be so easy.

Let us now start the proof. It will actually require many of these interesting techniques that you usually learn in the first course on computational complexity.
(Refer Slide Time: 17:22)



– Let's go back to proving Thm [KI'03] :

Lemma: PIT $\in$ P & per $\in$ Ar-P/poly $\Rightarrow$ $P^{per} \subseteq NP$

Proof: • Idea – "Guess" the small circuit for per & "verify" using PIT $\in$ P.

• We can expand permanent of an $n \times n$ matrix A, $per_n(A)$, by the first row:

minor

$$per_n(A) = \sum_{i\in[n]} A_{1i} \cdot per_{n-1}(A'_{1i}) .$$

where, A'_{1i} := submatrix of A deleting row=1, col=i.

• Given a circuit $C_{(n-1)^2}$ for per_{n-1}, we can use the following verification protocol:

Let us go back to proving the theorem by Kabanets and Impagliazzo (KI) from 2003. First we will prove a lemma which will use the PIT algorithm on permanent. If PIT is in P and permanent has small size arithmetic circuits. So let us look at the statement. Suppose PIT is in P and the second conclusion (in KI) is false which means that permanent has arithmetic circuits then what happens?

Then we will show that P^{per} is in NP. This means that any polynomial time algorithm that uses permanent as a subroutine will have those problems which can be solved this way will be contained in NP. This is a very strange conclusion. We do not believe that this is true. Because remember, Permanent is as hard or even harder than or it is as hard as computing the number of satisfying assignments.

And what this conclusion is saying is that even that can be done in NP. So conjecturally we do not believe this to be true. But if you assume permanent has small circuits and PIT is in P then you actually get this, this is what we will show. The idea is, guess the small arithmetic circuit for permanent and verify using PIT in P. So the idea of this proof is first guess the small circuit. Why can you do that well because you are assuming that permanent has a small circuit.

You can guess in NP and then you verify whether your guess is correct using the PIT algorithm and thus you have a chance of showing that P^{per} is in NP. Remember NP is the collection of problems which can be solved by guessing and verifying. The goal is to show that permanent is of a similar type. All we have to show is this verification protocol. How do you verify?

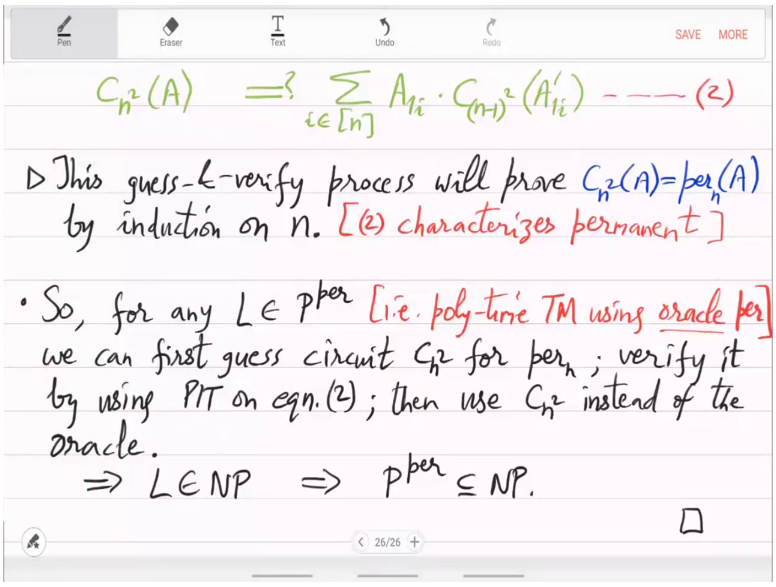
So what you have to observe is that we can expand permanent of an $n \times n$ matrix. Let us call it $Per_n(A)$. You can expand it by the first row. Expanding by the first row you could recall this from the determinant. In determinants also you can expand it by the first row or the first column. What that means is that you look at the entries which appear in the first row and then for each entry look at the corresponding minor which will be an n-1×n-1 matrix.

And look at the permanent of that so that expansion will give you the following identity. Permanent of A is equal to expand by the first row. So A $_{1i}$ is the ith entry in the first row, i will go from 1 to n and then put the minor. So, Per$_{n-1}$. That is the minor, so permanent expands like this, this you can do as an exercise. Recall the same or very similar expression that you get also for determinant.

Now think of this expression as a recursive procedure. Given an expression for permanent n-1 you can derive an expression for Per$_n$. This can be used to get a verification protocol. You basically look at the circuit that is given at n-1 ×n-1 matrix input and then check whether the left hand side and hand side are the same. Where $A'_{1i}$is the sub matrix of A deleting row 1 and column i. Basically minor just means that you delete the first row and delete the ith column.

What remains is permanent of n-1 ×n-1 square matrix. Now given a circuit $C_{(n-1)^2}$ where (n-1)^2 is the number of variables in the minor to compute the Per$_{n-1}$. Once you have this circuit we can use the following verification protocol.

(Refer Slide Time: 25:47)



$$C_{n^2}(A) \overset{?}{=} \sum_{i\in[n]} A_{1i}\cdot C_{(n-1)^2}(A'_{1i}) \text{ --- } (2)$$

▷ This guess-&-verify process will prove $C_{n^2}(A) = per_n(A)$ by induction on n. [(2) characterizes permanent]

• So, for any $L \in P^{per}$ [i.e. poly-time TM using oracle per] we can first guess circuit C_{n^2} for per_n; verify it by using PIT on eqn. (2); then use C_{n^2} instead of the oracle.

$\Rightarrow L \in NP \Rightarrow P^{per} \subseteq NP$. □

You check whether C_{n^2}, the circuit computing permanent for n by n matrix, that you have guessed whether this is equal to $A_{li} \cdot C_{(n-1)^2}$ on the sub matrix. That is your recursive verification protocol. This guess and verify process will prove that $C_{n^2}(A)$ is indeed permanent of A, by induction on n. You can go through this inductive proof. The idea is quite simple. The idea is just that when you apply this recursive definition on 2×2 or recursive kind of equation on 2×2 matrix.

Then you will get the 2×2 permanent. This C_{2^2} will be written as C_{1^2}. And C_{1^2} is just an entry. You can see that actually what you get on the right hand side in this green equation is the 2 by 2 permanent. Then you repeat this for three by three permanent. You will see that this C_{2^2} will be 2 by 2 permanent, C_{3^2} will be 3 by 3 permanent and so on.

By induction actually it is always true that if C_{n^2} satisfies this recursive equation then it is permanent. This is actually a characterizing equation. You just have to verify this on the circuit. What you have done till now is for any problem in P^{per} which means that is a polynomial time algorithm or polynomial time turing machine using permanent as an oracle.

Any such problem which can be solved using permanent as an oracle, we can first guess the circuit $C_{n^2}$for permanent then verify whether it is indeed permanent or not. Verified by using PIT. Equation 2 is exactly an instance of PIT. You have a circuit on the left and circuit on the right and you want to test whether they are equal.

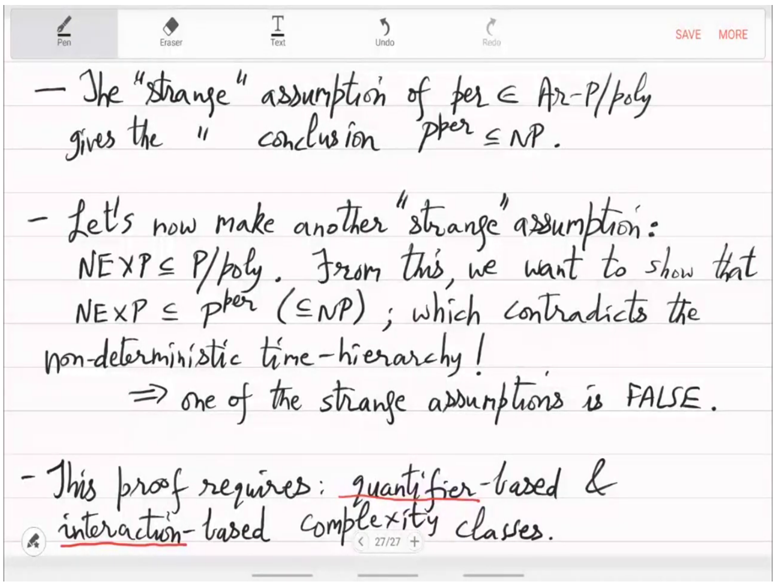
You guess a circuit, verified by PIT algorithm which you are assuming exists and then useC_{n^2} instead of the oracle. Oracle now can be eliminated from your polynomial time algorithm for l. You can in fact use this arithmetic circuit. This means that once you have guessed the circuit you have a deterministic polynomial time algorithm. And because of the guess it is actually a non-deterministic turing machine.

This means that L is actually in NP which means that anything in P^{per} is in NP and any problem in NP can obviously be solved using permanent. This actually means that these two are equal. In lemma 1 we have achieved that if we assume that PIT can be derandomized and permanent has small circuits then permanent is as good as being in NP.

This is a very strong conclusion. We do not believe this to be the case actually. Which suggests that we are making some wrong assumption and our conjecture is that permanent does not have small arithmetic circuits. We do not believe this assumption. That is the first thing which was relatively easy to prove. Now permanent being in NP from this point, we want to reach NEXP not in P/poly; that was the hardness.

Remember that this conclusion is not a hardness result we want to get to a hardness result. We wanted to connect PIT to some sort of hardness either arithmetic or Boolean. So how do we get there from here?

(Refer Slide Time: 33:16)



– The "strange" assumption of $per \in Ar\text{-}P/poly$ gives the " conclusion $P^{per} \subseteq NP$.

– Let's now make another "strange" assumption: $NEXP \subseteq P/poly$. From this, we want to show that $NEXP \subseteq P^{per}$ $(\subseteq NP)$; which contradicts the non-deterministic time-hierarchy!

$\Rightarrow$ one of the strange assumptions is FALSE.

– This proof requires: quantifier-based & interaction-based complexity classes.

So what we have achieved is the strange assumption of permanent in arithmetic P slash poly gives the strange conclusion P raised to permanent is in NP. So let us now make another strange assumption which is NEXP is in P slash poly. So we will assume this, that NEXP is in P slash poly and we will try to get a contradiction which will definitely mean that one of these strange assumptions is false. So either permanent is not in arithmetic we slash poly or next is not in P slash poly.

So what will happen with this change assumption that we want to show from this; From this we want to show that if NEXP is in P slash poly then NEXP is in P raised to permanent which we have shown to be in NP. You want to actually put NEXP in P raise to permanent which you have shown to be in NP before, in lemma 1 which is a contradiction. Which contradicts the non deterministic time hierarchy theorem.

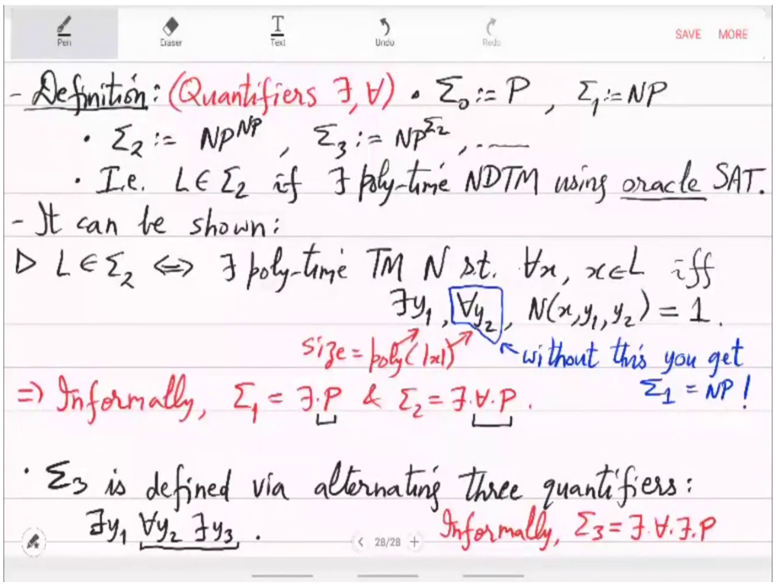
So this will be the end. We would have contradicted nondeterministic time hierarchy. We know that NEXP is actually bigger than NP. It is not equal to NP and that will be a contradiction. This contradiction would tell you that one of the strange assumptions is false. This would mean that one of the assumptions is false. Okay that is the plan of the proof. This proof requires a lot of tools, quantifier based and interaction based complexity classes.

Basic example of quantifier based complexity classes is NP, because in NP the key quantifier is ∃ (there exist) . There exists a correct guess which can be verified by the verification or by the verifier. On top of there exists you can throw in ∀(for all) and then based on there exists for all you can actually have more complexity classes which will potentially be stronger than NP. They will continually solve harder problems.

You can use more quantifiers and build more complexity classes. A third quantifier is "for most" strings. You may ask whether for most strings something is true. That gives you BPP but then when you mix these three for all, there exist, and for most, then you will get a lot more complexity classes and there we have to then discuss what is the meaning of interaction. Let me give you a crash course on this.

This is usually covered in detail in the first course on complexity but we will not be needing everything here. Let us just go into the definition and hopefully you will get some insight or some intuition from the definitions.

(Refer Slide Time: 39:19)



- Definition: (Quantifiers $\exists, \forall$) • $\Sigma_0 := P$, $\Sigma_1 := NP$
  - $\Sigma_2 := NP^{NP}$, $\Sigma_3 := NP^{\Sigma_2}$, ...
  - I.e. $L \in \Sigma_2$ if $\exists$ poly-time NDTM using oracle SAT.
- It can be shown:

▷ $L \in \Sigma_2 \Leftrightarrow \exists$ poly-time TM N st. $\forall x$, $x \in L$ iff $\exists y_1, \boxed{\forall y_2}, N(x, y_1, y_2) = 1$.

size = poly(|x|) without this you get $\Sigma_1 = NP$!

$\Rightarrow$ Informally, $\Sigma_1 = \underbrace{\exists \cdot P}$ & $\Sigma_2 = \underbrace{\exists \cdot \forall \cdot P}$.

- Σ_3 is defined via alternating three quantifiers: $\underline{\exists y_1 \forall y_2 \exists y_3}$. Informally, $\Sigma_3 = \exists \cdot \forall \cdot \exists \cdot P$

First quantifier based classes - there exist and for all. So you can define Σ_0 to be the complexity class P and Σ_1 the complexity class NP. Then you can define Σ_2 to be the complexity class NP with NP as an oracle - NP^{NP}. This is like a non-deterministic turing machine which is using SAT as an oracle and then you can repeat this process to infinity.

Further Σ_3 will be NP to the NP as an oracle so you get $NP^{NP^{NP}}$ and so on. That is a language L is in Σ_2 if there exists a polynomial time non-deterministic turing machine (NDTM) using SAT as an oracle. Think of somebody giving you a subroutine that solves SAT. Using that subroutine, what can an NDTM do? Those problems are these Σ_2 problems. Now why did I call this quantifier based?

An alternate classification of language L is or characterization of this is that there exists a poly time turing machine N such that for all input x, x is in L if and only if there exists a y_1 string for all y_2 string, $N(x, y_1, y_2)$ is equal to 1, where y_1 and y_2 are not too large. The size is polynomial in the input. Let us read this again carefully. So what this alternate characterization is saying is the problems you are solving by Σ_2 these problems have a characterization based on quantifiers where you will use both the quantifiers $\exists$ and $\forall$.

Some string x $\in$L or it is a yes string if and only if there is some y_1 such that for every y_2, N accepts. This is the important part. If this for all y_2 was not there then it would have been just NP; there exists y_1 such that N accepts. But with this $\forall$ this is something else it is a class stronger than Σ_1. So without this you get Σ_1. The reason why this is happening is I mean one intuitive reason is that when you look at NP to the NP.

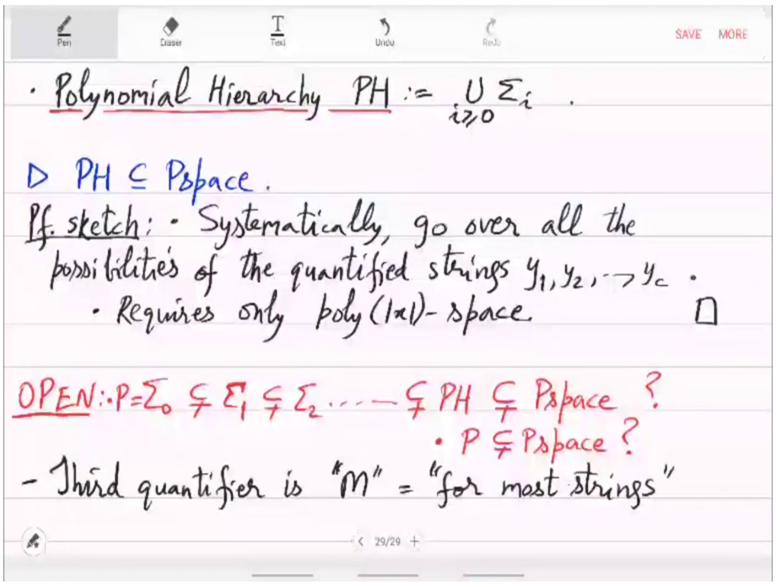
NP is using SAT but then you can also think of it as NP using complement of SAT. Complement of SAT are satisfying assignments which are always true; tautologies. Instead of $\exists$ you can think of the dual which is $\forall$. So Σ_2 actually is using both $\exists$and $\forall$. From NP^{NP} you can actually use this intuition to give a formal proof of this $\exists$, $\forall$ characterization.

First start with Σ_1. So sigma-1 is $\exists$P and Σ_2 is $\exists$ $\forall$ P. This is a simple way to understand and remember what we just did. You saw the definition of Σ_1, Σ_2. Σ_1 is $\exists$ quantifier so you guess and you verify. Σ_2 is $\exists$ which means you guess but then the verification is more complicated.

Here the verification is for every string y_2 the thing has to be true. So the verification is more complicated than Σ_1 and we can obviously make it more general. We can say Σ_3 is defined via alternating three quantifiers. So $\exists y_1 \forall y_2$ like Σ_2 and then there will be $\exists y_3$ also. The verification is happening in a more complicated way which is this part. You have to first guess a string y_1 and then you have to verify for every y_2 whether there is some y_3.

Again informally you can think of this as $\exists$ $\forall$ $\exists$ P. We have defined these infinitely many complexity classes in two ways. We defined Σ_i as a tower of NP, NP raise to NP raise to NP and we are claiming that there is an alternate equivalent characterization which is by a tower of alternating quantifiers which is $\exists$ $\forall$ $\exists$ $\forall$ and so on.

Remember these two definitions of Σ_i and once you have these Σ you can look at the union of this which is called Polynomial Hierarchy(PH).
(Refer Slide Time: 48:14)



· Polynomial Hierarchy PH $:= \bigcup_{i \geq 0} \Sigma_i$.

▷ PH $\subseteq$ Pspace.

Pf. sketch: · Systematically, go over all the possibilities of the quantified strings $y_1, y_2, \ldots \to y_c$.

· Requires only poly($|x|$)-space. $\square$

OPEN: · $P = \Sigma_0 \subsetneq \Sigma_1 \subsetneq \Sigma_2 \cdots - \subsetneq$ PH $\subsetneq$ Pspace ?

· P $\subsetneq$ Pspace ?

- Third quantifier is "M" = "for most strings"

This is the union of everything, all the Σ_i's. You can think of this as either a constant number of NP towers of constantly many NP's or constantly many alternating quantifiers ∃ ∀. Any problem that you can solve this way is said to be in PH. Now it is easy to show that the polynomial hierarchy is completely contained in Pspace.

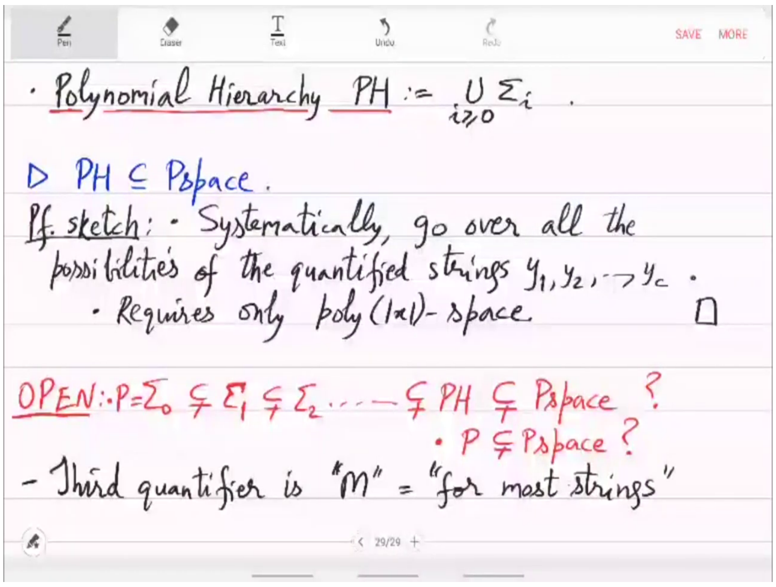
Well Σ_1 is NP, which is contained in Pspace because you can go over all the string y_1's. Now the first question is why is Σ_2 in Pspace? $\exists y_1 \forall y_2$, why is this thing in Pspace? Again you can actually enumerate over all possible y_1's. And once you fix y_1 then you can go over all y_2. That is the proof. Systematically go over all the possibilities of the quantified strings y_1 ,y_2 and only constant many times to y_c.

Going over all the possibilities of y_1 to y_c systematically requires only polynomial space, polynomial in the size of x. So in Pspace you can solve anything that can be solved by PH. Which is essentially saying that any problem in Σ_i can also be solved in Pspace. This is a large number of complexity classes that we have created as generalization of NP and obviously all these questions are open questions.

Whether Σ_0 is smaller than Σ_1 is smaller than Σ_2 is smaller than the PH, and is it smaller than Pspace? These infinite towers are collapsing or are things equal. These questions are open questions and Σ_1= P even that question is open. Whether P is different from Pspace. Although Pspace seems to have very difficult problems which we do not expect to have polynomial time algorithms, practical algorithms.

But still we do not have a proof of the strength of Pspace over P. This is an outstanding open question. That finishes the discussion about quantifier based complexity classes. Next time we will start another or analogous tower based on some other quantifiers. The starting point would be for most quantifiers. So, the third quantifier that we can use is M, which is for most strings.

(Refer Slide Time: 54:20)



- Polynomial Hierarchy PH $:= \bigcup_{i \geq 0} \Sigma_i$.

▷ PH $\subseteq$ Pspace.

Pf. sketch: • Systematically, go over all the possibilities of the quantified strings $y_1, y_2, \ldots, y_c$.
• Requires only poly($|x|$)-space. □

OPEN: • $P = \Sigma_0 \subsetneq \Sigma_1 \subsetneq \Sigma_2 \cdots \subsetneq PH \subsetneq$ Pspace ?
• $P \subsetneq$ Pspace ?

- Third quantifier is "M" = "for most strings"

Let us define it. This statement M quantifies a string y. So M_y let us say in the space $\{0, 1\}^n$. We say that N(y) is equal to 1. Think of N as a turing machine. We say that N(y) evaluates to 1 for most strings y. We say this is true if the probability that N(y) is 1 in the space $\{0, 1\}^n$ is high, let us say 3/4. In simpler words if turing machine N is true for at least three fourths of the strings in the space then we say that N is true for most of the strings.

And for that we use the quantifier M_y. This is what we have defined as a third quantifier. Using this quantifier with ∃, ∀ we can actually define more complexity classes. We will call them interaction based complexity classes. We will see next time why this can be seen as an interaction between two parties.