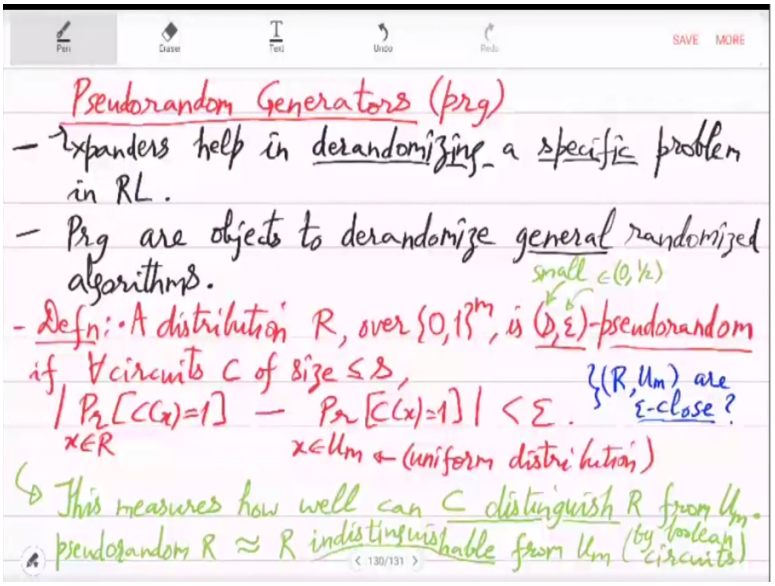
Randomized Methods in Complexity
Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology - Kanpur

Lecture – 17
Pseudorandom Generators

(Refer Slide Time: 00:22)



Last time we finished the construction of explicit expanders and we started this new topic called Pseudorandom Generators. We will shorten it to prg and this will be the most important object in this course on randomized methods because this you can think of as the way to generate random numbers efficiently. So, in practice, if you want to generate if you want to simulate a coin toss you want it to be efficient, so you can think of prg's in that context.

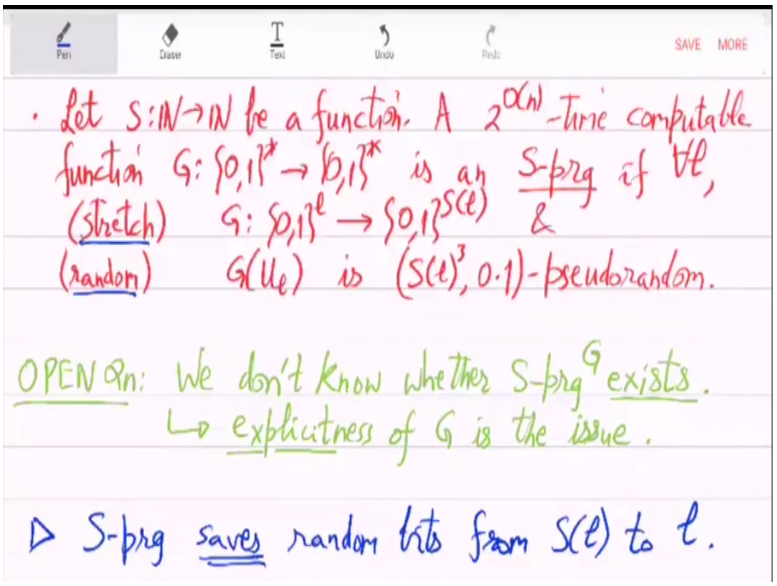
So, the first thing is how do we mathematically, how do we formally define this object? What do we want from it? So, what we want is that this s, epsilon, there will be 2 parameters, s will be the resource which your prg can fool. So, in this case, it will be small-sized circuits of size s. Epsilon will be an error, how much is it able to fool, ideally, you would want epsilon to be 0 because no circuit of small size s should be able to distinguish between this distribution R and the uniform distribution U_m.

So, ideally, epsilon should be 0, but then we have to tolerate some error because of practical considerations because of efficiency. We will allow epsilon to be something smaller than half

a constant. You can think of epsilon as 0.1. First, we define the distribution. A distribution R is called s, epsilon pseudorandom if any circuit of size s is unable to distinguish R from U_m.

The probability of C being 1 on either of these distributions is very close, it is close by epsilon it is epsilon close, so we can also write that. You can think of this notion of closeness, but remember that it is with respect to a limited resource and that is the size s circuits. And so what is a prg now? Once you have defined pseudorandom distribution, a prg is efficient you can think of it as a circuit or an algorithm, it is an algorithm that can produce this distribution R.

(Refer Slide Time: 03:12)



Let S be a function at 2^n time computable function, G it is multivalued. It is not a Boolean function, it is actually a function is an S-prg. If for all l, G stretches so G stretches l bits to S(l). So think of S as the stretch function for this prg G. It will stretch l bits to S(l) bits and if you look at $G(U_l)$, this is pseudorandom. A prg is basically some function that is easy to compute, very easy, it is very relaxed, it is 2^n.

This is actually smaller than exponential, the class E and not the class EXP. But with that you have enough time to compute this. So, G is easy to compute and it will stretch a seed. So, l seed will stretch to S(l) and then the output of G if you look at the whole space it will look more or less random two circuits of size s^3. Remember that the output you are getting is S(l) bits and essentially in that match of size circuits will be fooled.

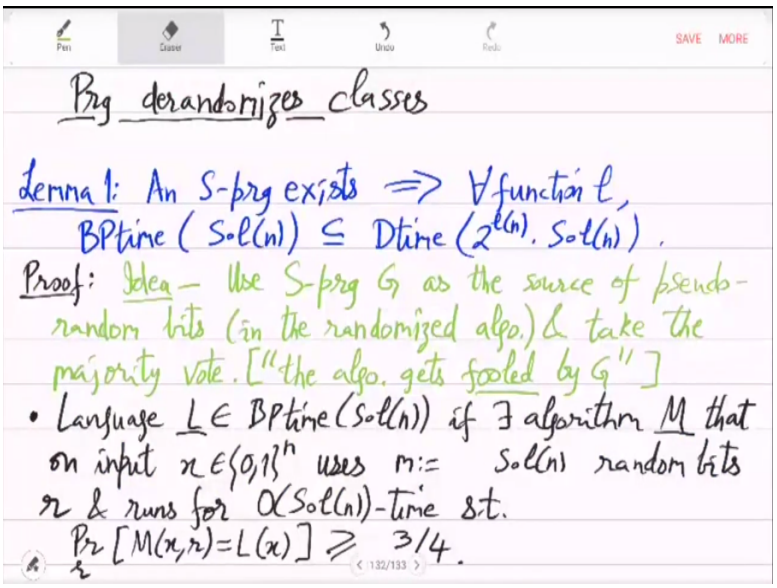
So, circuits of size S(l) cube will be fooled and then the error we are allowing is 0.1. It is not 0 exactly, but it is 0.1. These constants 3 and 0.1 are kind of arbitrary. We have picked them so that all the subsequent theorems work, but you can actually change them also, it is not something very specific. What is the consequence of this definition and then this object? First question you should ask is whether this object actually exists? That is an open question currently.

We do not know whether these things exist. And the problem is this time, 2^n time computability, this kind of makes it explicit. Explicitness is the issue. If this condition of 2^n time computability was not there, then we could show that actually random functions G will work, they will be able to fool, stretch and fool circuits but then they will most likely be uncomputable. I mean at least uncomputable in 2^n time.

This 2^n time is the important condition. And that is why we do not know whether they actually exist. If they exist, then we can actually derandomize complexity classes. What can you derandomize? Remember that stretch is l to S(l). Basically, any algorithm that requires that many bits you can provide that randomized algorithm with this S-prg. So, that will reduce the random bits from S(l) demand S(l) to seed l.

If l is very small then your algorithm will save the random bits. So, S-prg saves random bits from S(l) to l that is the idea and let us now show this as a theorem. How much is the saving or which complexity classes are being derandomized?

(Refer Slide Time: 08:57)



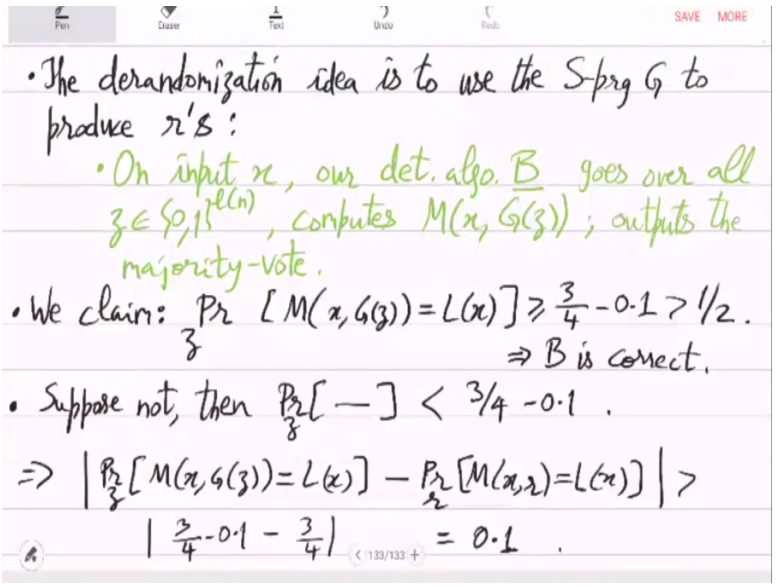
We will show so an S-prg exists. This implies that for every function l, BP-time S(l) n can be derandomized and so brute force would have been $2^{S \circ l(n)}$ because you have to essentially go over the whole space of random strings but this prg will make it much smaller. It will make it $2^{l(n)}$. So, basically focus on this $2^{l(n)}$. What you are getting from S-prg is $2^{S \circ l(n)}$ is being reduced to $2^{l(n)}$.

And this is happening because the random bits S(l) is being reduced to random bits l by the stretch that the prg gives. But of course, we have to see why exactly, what is this deterministic algorithm first of all and why will it work? That we will do essentially by taking a majority vote. A trick that you have seen before also many times in fact. The idea is to use S-prg G as the source of pseudorandom bits in the randomized algorithm and take a majority vote.

The idea is that your randomized algorithm will be fooled, gets fooled by G that is how you can put it. The randomized algorithm would not know whether it is getting random bits or pseudorandom bits, but of course this we have to now formulize how exactly is this fooling happening. Language L is in BP time if there exists an algorithm M, so Turing machine that on input x n bit input uses $m := O(S \circ l(n))$ like these many random bits r and runs for that much time.

This algorithm M requires m random bits and also it runs for a similar amount of time and then it outputs an answer which will be correct with probability three-fourth. So, the output of M on x on the random bits r will match whether x is in l or not, so that I am rooting by l of x with probability at least three-fourth. That is the meaning of L being in BP time. We will now change this algorithm M by using G.

(Refer Slide Time: 14:57)



So, the derandomization idea is to use the S-prg the one that exists from the hypothesis of the theorem to produce r's. So, basically the algorithm is just this on input x our deterministic algorithm B goes over all z $\{0,1\}^l$ computes M(x, G(z)). Remember G(z) will be stretching l bits to M bits, M we have defined as S(l) and let me make it precise, let me just make it M is just this.

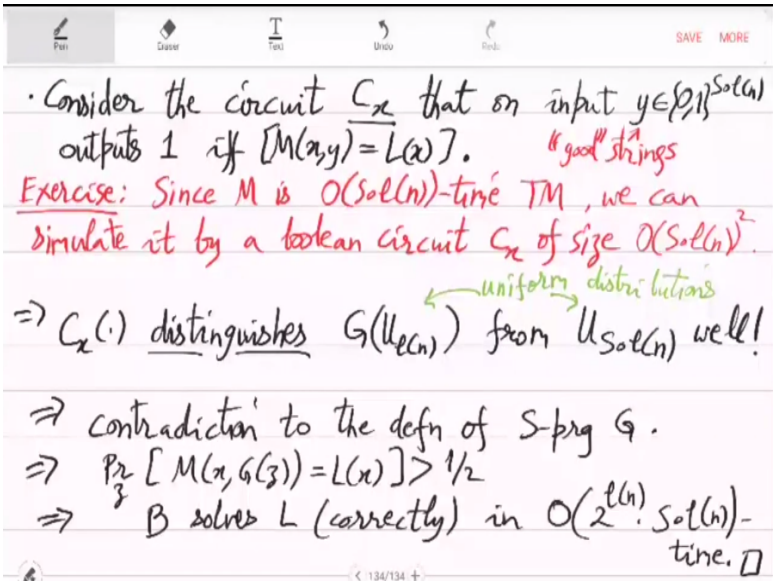
So, it will stretch G will stretch l bits to M bits which is S(l) n and then this randomized algorithm M will use this output of G and then it outputs the majority vote. This overall z takes the majority vote. If 0 appeared more output is 0, if 1 appeared more output is 1. So, the claim is that this deterministic algorithm B correctly solves l, let us see that. So, we claim that the probability over z that M(x, G(z)) matches L(x).

This is at least 3/4 - 0.1, which is more than half. We want to show that over the z the probability is more than half which means that the majority vote is correct. So, the answer of B is correct implying that B works. So, suppose this probability estimate is wrong. Suppose not, then the probability is less than 3/4 - 0.1 which means that the probability over z of M(x, G(z)) being equal to L(x) minus the probability over r of M(x, r) = L(x).

This probability difference that M runs using G and M runs without using G. This difference is large because you are saying on one hand that probability over z is less than 3/4– 0.1. On the other hand you are saying probability overall is more than 3/4. So, this is larger than 3/4 – 0.1 – 3/4 which is equal to 0.1. So, this means that M is able to distinguish between z and r which should be a warning sign.

The way we define G nothing should be able to distinguish its output from random, but here it seems that there is a distinguisher. So, to make this formal you have to talk about a circuit because S-prg was defined by a circuit, M is not a circuit it is an algorithm. So, let us make it into a circuit.

(Refer Slide Time: 20:21)



So, consider the circuit C_x, x is something fixed, in this argument we are looking at x as fixed that on input y, outputs 1 if and only if M(x, y) = L(x), so y are kind of the good random strings. This is what C_x is identifying and as an exercise show that since M is $S \circ l(n)$ time Turing machine, we can simulate it by a circuit C_x of size $O(S \circ l(n))^2$.

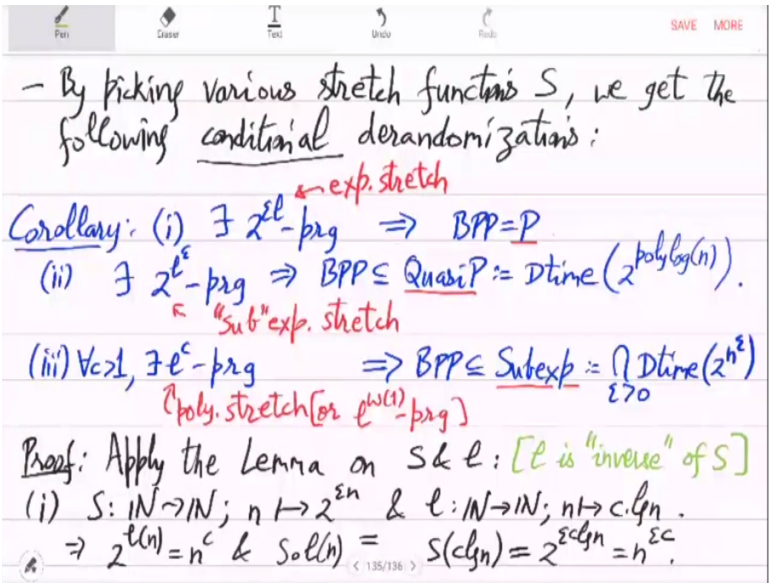
So, whatever is the time complexity of the Turing machine basically every transition step of the Turing machine you can simulate by these and or not gates and the overall size you can easily show is at most quadratic, in fact you can show something better also, but for us quadratic is fine. Basically there is a circuit C_x which on y is correct, its value is L(x) for all these y's in the domain.

So, what this means is that C_x can distinguish $G(U_{l(n)})$ from $U_{S \circ l(n)}$ the uniform distributions. So, U denotes the uniform distribution of appropriate size and $G(U_{l(n)})$ you can distinguish it from $U_{S \circ l(n)}$ using C_x because we have assumed that the probability difference is larger than 0.1. So, this is what we are using and that is contradicting the definition of S-prg G.

So, this contradiction to the definition of S-prg G which means that actually the probability difference is small, so it is as shown before more than half as claimed before. This implies that the probability over z of M(x, G(z)) set being L(x) is greater than half which means that majority vote works which means that B is correct. B solves L correctly, in how much time? So, in the definition of B remember in this green definition it is going over all the z side.

So that is $2^{l(n)}$ and time complexity is that of M, so that $S \circ l(n)$. So, this is how you use an S-prg. You can derandomize and equivalently you can reduce the number of random bits. So, now to see the impact of this observation let us look at various parameters or parameter settings for the stretches.

(Refer Slide Time: 26:28)



So, by picking various stretch functions S, we get the following conditional derandomizations, conditional because I mean you have to assume that an S-prg exists. Those questions we do not have an answer for currently, but nevertheless this even conditionally randomizations are quite instructive in the sense they will tell you what S-prg do you need. So to get BPP = P, so if there is an exponential stretch prg, then BPP = P.

So that is an exponential stretch which means that l you are stretching 2^{l}. This is the maximum you can hope, you cannot do more than this. So, if you can really construct optimal stretch prg, then you can solve any randomized algorithm you can make a deterministic polynomial time. If there exists $2^{l^{\varepsilon}}$ prg then BPP you will put in QuasiP.

So, this is slightly more than polynomial time, this is $2^{poly(\log n)}$ and the stretch that this is demanding is not exponential, it is we call $2^{\varepsilon l}$ this here now you can use for example $2^{\sqrt{l}}$ or $2^{\sqrt[3]{l}}$ stretch. So, you can call it sub exponential and the weakest stretch you can hope for is like quadratic stretch, cubic stretch, polynomial stretch.

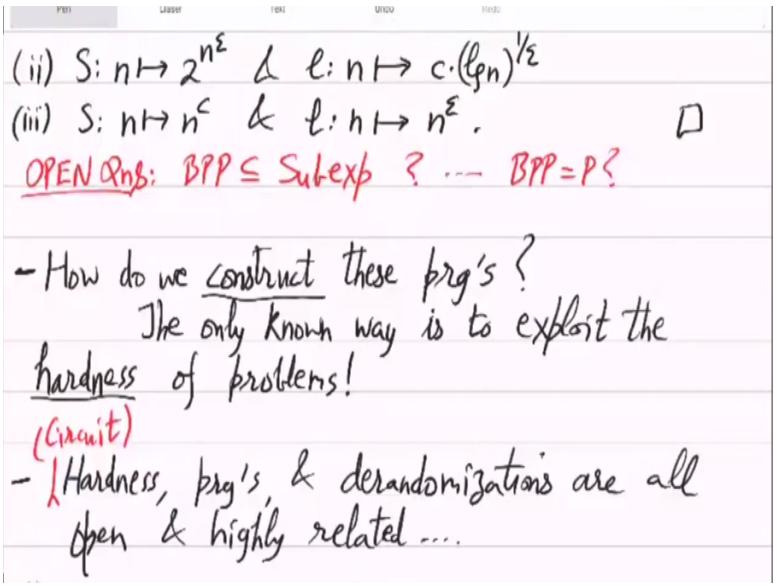
So, if there exist l^c for c greater than 1, then what happens? Even then something happens. So BPP then would be in Subexp like we defined before. So, you go to P or you go to Quasi P or you go to Subexp. This is polynomial stretch. One important thing I forgot is I have to add here for every c, for every c if you have an l^c prg only then. So we will prove this quite simply using invoking the previous theorem.

It will also imply that for every c there is an l^c prg and in that case you can put BPP in Subexp. So, what is the proof of all this? The proofs will be similar. You just have to save what basically in this Lemma that we had what is S, what is l and then look at the RHS, what is 2^l that is all, that is all you have to check. So, we will say apply the Lemma on S and l. So, first you apply it S is in this case the function that sends n to $2^{\varepsilon n}$ that is exponential and l is the function that sends n to c log n.

Since you have in condition in this item one you are assuming that there is an exponential stretch $2^{\varepsilon l}$ prg so that is why the stretch n to 2^{ε} is possible. And then when you apply this S on l you will see that you will get the following. So, $2^{l(n)}$ is equal to n^c and $S \circ l(n)$ is equal to $S(c\ log\ n)$ which is equal to $2^{\varepsilon(c\ log\ n)}$ which is $n^{\varepsilon c}$.

So, both $2^{l(n)}$ and $S \circ l(n)$ they are both polynomial in n. So, in the Lemma statement LHS S of BP time, BP time $n^{\varepsilon c}$ this is being put in Dtime n^c times the same thing $n^{\varepsilon c}$. And the assumption was that there is an S-prg. So that stretch function immediately gives you the BP time Dtime connection and the same thing you do for other items 2 and 3.

(Refer Slide Time: 34:35)



So, here you will pick S to be 2^{n^ε} because that is what prg you are promised and l will be; so l here, remember that in this proof l is always kind of the inverse of S. So, here you can pick l to be $c(\log n)^{1/\varepsilon}$. So, when you compose S with l, you will see that you will get polynomial in n that is the idea and in the third case S will be polynomial stretch stretching n to n^c and l will be kind of the inverse.

So, this will be n^ε. So, l is always kind of the inverse of S, l is inverse of S. Based on that principle, we can show the collapse of BPP. BPP is BP time poly n and depending on what kind of a stretch you have you can put it either in P or Quasi P or Subexp, even Subexp is very nontrivial. All these are open questions on their own. So, as you know BPP in Subexp is an open question we do not know the proof of this and in fact we believe that BPP is in P.

What this previous connection is showing this color in blue is showing is all these stretch prg's these are also open questions, we do not know whether they exist. So, how do we construct these prg's? So, the only known way, we obviously will not be able to construct them in this course, unconditionally we will not be able to do that, but we will relate this problem to the problem of hardness.

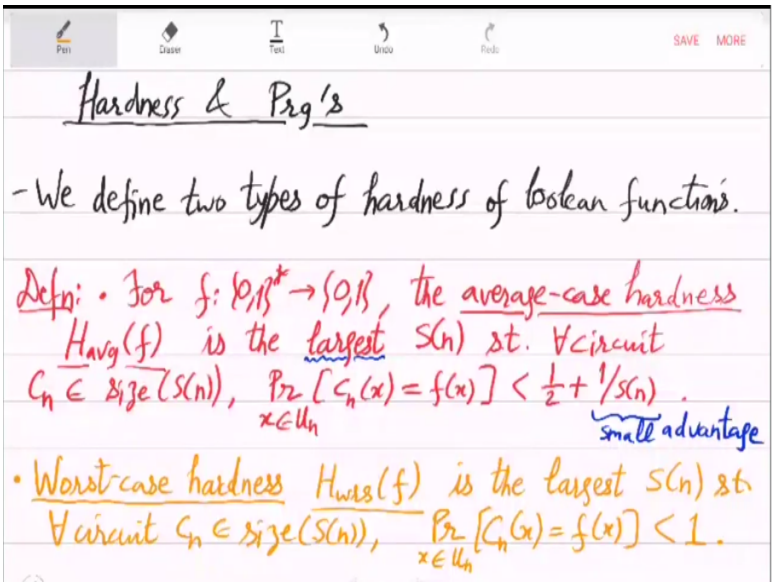
So, that in itself will be an amazing connection, we will show that if there are hard problems, then these prg's will exist and then all these derandomizations would happen. So, the only known way is to exploit the hardness of problems. So, what kind of hardness will suffice and

then what is the proof of this connection? So, let us move to that. So hardness, prg's, and derandomizations are all open and highly related.

This is what we will now see and we will spend the remaining course on these connections. You have already seen this easy connection from prg to derandomization. Now, we will move on to the harder connection which is between hardness and prg's and this hardness also to be useful it has to be for explicit functions. So, it should be explicit hardness that is why it is challenging.

So, actually we know that there are hard problems like halting problem is hard, SAT is I mean we do not know that SAT is hard, but at least you know that halting problem is hard and problems which are in EXP you know there is a problem which cannot be solved in polynomial time. So that kind of hardness you already have, but the thing is we want more explicit hardness. So, let us now get deeper into this.

(Refer Slide Time: 40:34)



So, hardness and prg's: We define two types of hardness of Boolean functions. I should say here actually the focus will be not so much on explicit but circuit hardness, I should say that. So, you have to show hardness in terms of Boolean circuits. It is not this P versus EXP or P versus E. This is in a way stronger. So, the first definition is average-case hardness and the second will be worst-case hardness.

So, for a Boolean function the average-case hardness will denoted by $H_{avg}(f)$, this is the largest S, now I am using the same symbol S like stretch and later on we will see they are

actually this is for a reason, but again a function S will be called the average-case hardness of Boolean function f such that for every circuit C_n of size S(n), C_n will not be able to solve f very well.

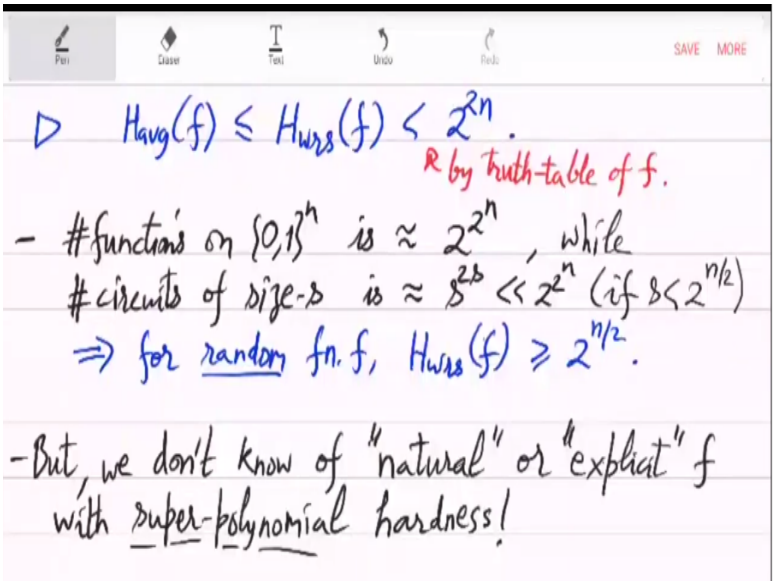
So; in other words the probability of $C_n(x)$ being equal to f(x) on x in U_n so $\{0,1\}^n$, uniform distribution. If you just did a random choice or give a big zero or one; yes or no in a random way, then the probability would have been around half or at least half and this circuit C_n is not able to do much better that is what we want to say. We will say 1/2+ 1/S(n). So, this you can see as the advantage.

So, we say that f is average-case hard or f has average-case hardness S if circuits of size S cannot give you an advantage more than 1/S or they have an advantage less than 1/S. So half is for free just by tossing a coin and giving the answer but nothing better these circuits are able to do or not much better. So, that largest S is what you will pick. Notice that S will increase the size S, S will increase the size of the circuit C_n.

The C_n will start approximating f better and better on more and more inputs, so the advantage will actually increase. So, which is why saying largest is fine because this largest will exist, it is not unbounded. And the second notion will be worst-case hardness. We will denote it by $H_{wrs}(f)$. This is again the largest S such that for every circuit of size S(n), now you want in worst cases actually want to be correct on every x.

So, we will say that this circuit C_n are not able to or they make some mistake, at least one mistake, so probability less than 1 that is all. So, here even a single mistake is considered a failure. So, what is the largest S circuits that will be a failure for computing f, so that will of course be is expected to be bigger than the average case hardness?

(Refer Slide Time: 47:44)



You can write the relationship quite easily amongst these two notions. So, the average case hardness is at most worst-case hardness which is at most, so average case hardness note that this random choice thing already gives you half rate. So, if you take the size to be more than 2^n, then you can just use the truth table of the function and you can exactly compute f. So, this is less than 2^n, and cannot exceed 2^n by truth table of f.

And the size when you convert truth table to circuit that will be slightly more than 2^n, so maybe we make it a bit more. Let us make it 2n in that much. So, worst-case hardness obviously it cannot exceed much more than exponential in and average-case hardness will be smaller. So, I mean obviously, the open questions here will be then can you bring the worst-case hardness close to this upper bound?

Can you bring it close to 2^{2n}? So, the number of functions on $\{0,1\}^n$ this will be how much? 2^{2^n}, while number of circuits of size n is around n^n. So, you have n gates and each has a fan in n and fan out also at most n, so you can see that the way you can arrange the gates is something like n to the let us say 2n which is much smaller than 2^{2n}.

So, this means that usually for random f the worst-case hardness is, let me actually I have to replace this by some size let me replace it by size s, so this is something like s^{2s} which you can make much smaller than 2^{2n} if you take s to be sufficiently smaller than 2^n. Let us say I take $2^{n/2}$. If I take s to be less than $2^{n/2}$, then you can see that this inequality holds, s^{2s} is less.

So, the number of circuits is actually much smaller than the number of functions of this much size. From that you can see that for random functions worst-case hardness will be this s which is $2^{n/2}$. So, generally, functions are supposed to be pretty hard in terms of Boolean circuits, but we do not know of natural functions or explicit f with super-polynomial hardness.

Actually we will be interested in these explicit functions with super-polynomial hardness. In the ideal case exponential hardness and then will relate it to the construction of prg's and then from prg's you will get the derandomizations.