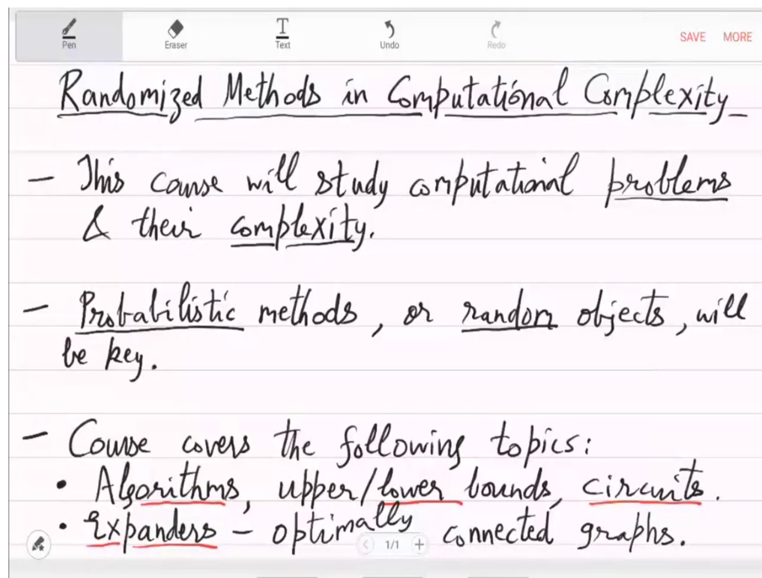
Randomized Methods in Complexity
Prof. Nitin Sexena
Department of Computer and Engineering
Indian Institute of Science – Kanpur

Lecture 01
Course Outline

(Refer Slide Time: 00:19)



Welcome to this course on randomized methods in computational complexity theory. As the title suggests, this will be a course slightly more advanced than complexity theory. But I do not expect you to have done this course on complexity theory. I just expect you to know little bit of algorithms, Turing machines, probability and algebra. So based on those things you will be able to understand this course.

This course will study computational problems and their complexity. The two things are problems and their complexity. These problems will be computational problems which are slightly different from mathematical problems or other types of problems you face in everyday life. Here computational means that there is a precise input and there is a precise output. Input and Output are strings.

And that is what you want to achieve as fast as possible; so that is what is complexity which will formalize. And randomization in the title refers to probabilistic methods or random objects. Random objects, pseudo random objects, probabilistic methods, probability; these things will be key in this course. It is necessary to have some intuition about probability.

Today I will not go into the details of anything I will just give you a list of topics and objects and tools that this course will expand upon.

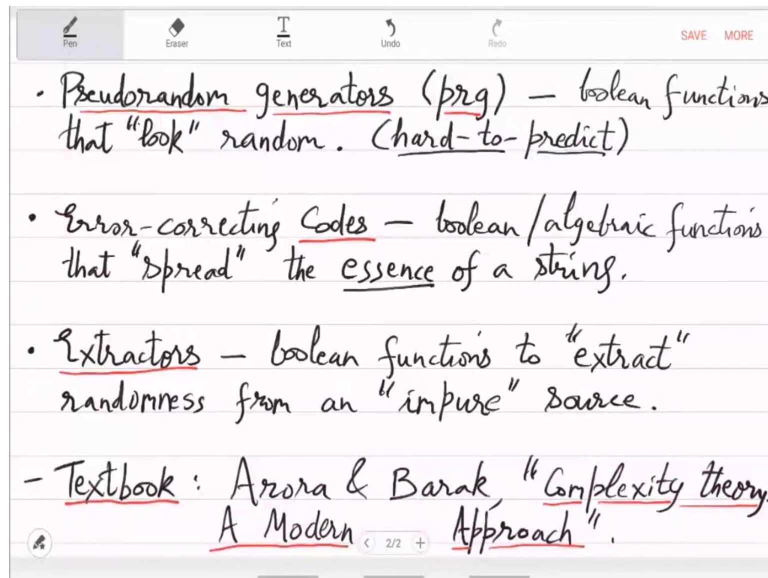
This lecture gives you the overview. The course will cover the following topics or at least the initial ones and then depending on time we can do more things. First is algorithms; but this is not an algorithms course, so will not really do everything in algorithms but just the notion of algorithms. We ask what is the problem? What is an algorithm? What is the solution? What are the upper and lower bounds?

There is a different way to formalize computation which is called Circuit. We will use this extensively: Circuits. The circuit should remind you of electronic circuits, something which is used to make a computer. We will formalize that and use it mathematically. Next is expanders. So what is an expander?

At this point it will be hard to define but think of expanders as a graph with the high connectivity or to be precise high reachability. It is a graph where if you take a few random steps then you can reach almost anywhere with equal probability. If there are n vertices you want to design a graph where there are n vertices and the edges. You also want to minimize the number of edges but they should be distributed such that in $\log n$ random steps you should be able to reach every vertex with equal probability or almost equal probability.

These are optimally connected graphs. Essentially, expander is a graph construction. We will look at such constructions and we will study it very rigorously. This is a beautiful and a very useful theory that we will develop. Something else that we will develop and is even more important is what is called a pseudo random generator.

(Refer Slide Time: 06:30)



We will shorten it to prg. These are boolean functions, the output of which looks random. This also means that for a low computational device it will be very hard to predict the values of these functions. For that computational device it will be a hard function because if it was not hard then a small computational device can actually predict the values. This, in particular, also means that this will be a hard function and its output in particular will look random to low computational devices.

This is a very useful object. Something on which a lot of what you do in security, cryptography is based on. These are boolean functions that look random. Which means it is hard to predict. We have to formalize and define prg and then we have to see whether these things exist and how fast can you compute them. Next is error correcting codes. This is hugely important. In fact it is so important that if error correcting codes did not exist then none of your electronic machinery will exist.

The intelligent electronic machinery - computer, smartphone, communication. Everything depends on correction of errors. Physical devices have a lot of errors. For example, a hard disk has a lot of errors. You still have to store correct information, you have to correct the errors. Obviously up to a limit. If the errors are for example more than 50% or 50% then it is impossible to correct.

But suppose the errors are less than 50% like 45% on your hard disk then can you store information in such a way that it can be recovered safely and correctly? We will see a lot of such constructions. These will be, again functions based on mostly actually the best

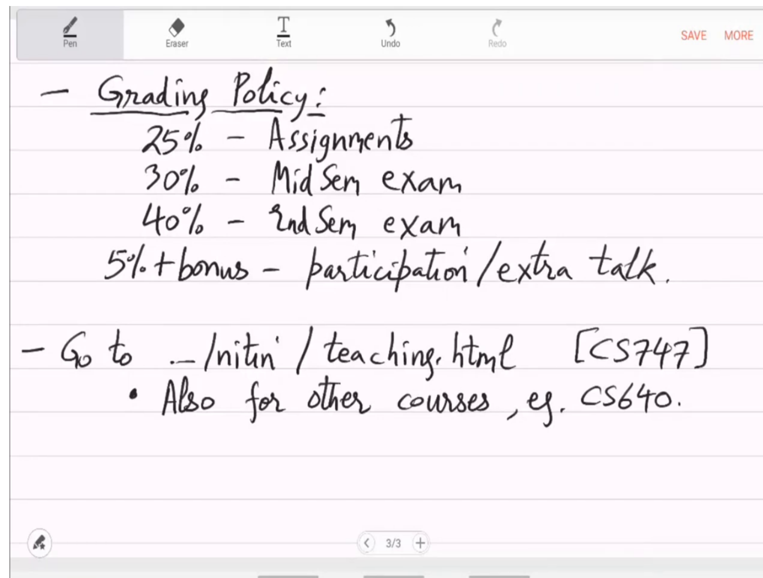
constructions are algebraic. So these are boolean or algebraic functions that spread the essence of a string.

Whatever information that string contains, you basically want to spread that duplicate, introduce some redundancy and then store it on a hard disk. So that even if a part of the hard disk gets corrupted still you can recover that essence of the string. It is about the diffusion or the spreading of the information. So those kind of boolean functions. Do they exist? if they exist, can we construct them? If we can construct them, how much error can they correct or can they handle?

We will see all of that and finally extractors. So what are extractors? Extractors are boolean functions that can extract randomness from an impure source. It basically takes in a pseudo random function which may have lots of bits but the randomness is only in a few places so what this extractor function does is it extracts those few places. In the ambient space it compresses and then gives you a more pure form of randomness.

So this is very vague but we will formalize and then we will see whether these things exist and how to construct them. If we have time then we can do more things. But usually I have seen that these topics are already a lot. If you are interested in reading a book together with this course then a recommended textbook is Complexity Theory: A Modern Approach by Arora and Barak.

If you have not done a computational complexity theory course, this book is highly recommended. The initial seven chapters of this will give you the basics of complexity theory and this course builds on that formalism. We will not need all the theorems and results but we will need the formalism and the definitions. These advanced objects and lower bound results we will discuss in this course. That is the part that comes in the book after chapter eight.
(Refer Slide Time: 13:52)



When this course is taught in IIT the grading policy is that

- 25% - Assignments.
- 30% - Mid Sem Exam.
- 40% - End Sem Exam.
- 5% + bonus - participation and extra talk.

The exams will all be take-home, since it's an advanced course. Questions will take longer than three hours. So I usually give it as a take-home exam. The remaining 5% and maybe bonus marks that are for if you are in the class if you are attending this by a physical class.

Then participation or if you give an extra talk or if you prepared an extra report. That is the usual grading policy in a usual class. In NPTEL it will be slightly different, which you can see from NPTEL rules. The lecture notes are also available; they are already available on the home page. So you can check at my CSE homepage - cse.iitk.ac.in/users/nitin/teaching.html

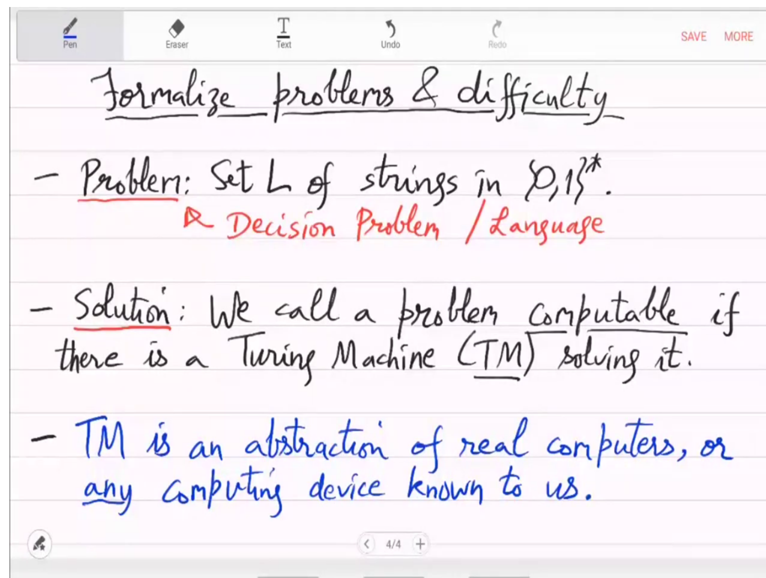
You can get lecture notes for CS747. I have taught it many times and every time there are some changes. So you can look at the most recent one. Those lecture notes are in PDF, so you can easily download it and read it if you want something with the video then it is a good option. Also if you need to read about other courses in particular complexity theory that is also available and you can have a look.

So it is also good for other courses. CS640 is the course that comes before this particular course. There is the administrative part and the overview. Let us now do some formalization which will be necessary for this course. It will remind you if you have done a course on

complexity theory. And if you have not done any such course then it will introduce you very quickly into the basics on which complexity theory builds up.

So everyone here has already done theory of computation, so it is basically some part of theory of computation that I will quickly remind you.

(Refer Slide Time: 17:10)



Let us formalize Problems and Difficulty. What is a Problem? In computer science, a problem is just a set of boolean strings. Because computers essentially only understands on or off, which is 1 or 0, true or false. Everything that is happening in computer science is actually ultimately implemented as just a string. Inputs are boolean strings and output is also a boolean string.

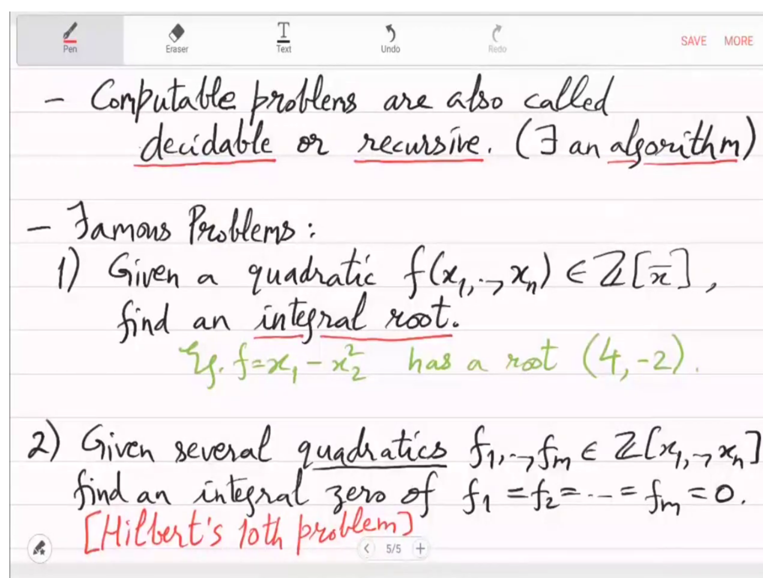
Input problem is basically a set of strings and this is what we call a decision problem or it's called a language. When you model a decision problem as a language or as a set L the strings which are in the set, they are called the strings and the ones which are not in L are called the no strings. You are dividing the space into yes or no and obviously the problem then is to distinguish -Given a string, how do you find out whether it is yes or no?

So that is a decision problem. Now what is the meaning of solving a problem? The solution to a problem is in this course it will mean, which is also true for computer science in general, we call a problem computable if there is a Truing Machine (TM) solving it.

That is when we say that a problem is solvable. That deciding whether a string is in the language or not, that can be done by a Turing machine decider. I will not define the Turing machine, since this you have already seen in great detail and also you have seen models weaker than Turing machines like finite automata, non-deterministic finite automata, push down stack and so on. So I will not go into all that. That really needs a course in itself.

I will just say that the Turing machine is an abstraction of real computers and in fact any computing device known to us. Any computing device and especially real computers, are all abstracted by a Turing machine. This is how you should remember it. In this course, whenever I say algorithm, solution, problem, I am talking about Turing machines solving it.

(Refer Slide Time: 21:46)



Computable problems are also called decidable or recursive. Decidable is a standard term. Computable and decidable are interchangeable. Recursive is an older term, which comes from logic. But they all mean the same for us, which is also the same as their being an algorithm. So these terms will really mean the same thing. When you write a C program or a java program that is an algorithm and that it is also a Turing machine.

And the problem you are solving by the C program, that problem is decidable, recursive, computable. That is how we will talk about it. Now let us look at some famous concrete problems in computer science. Suppose you are given a quadratic polynomial. The polynomial has n variables and it is an integral polynomial, that is, its coefficients are integers. You are given an integral polynomial and the question is whether there is an integral root.

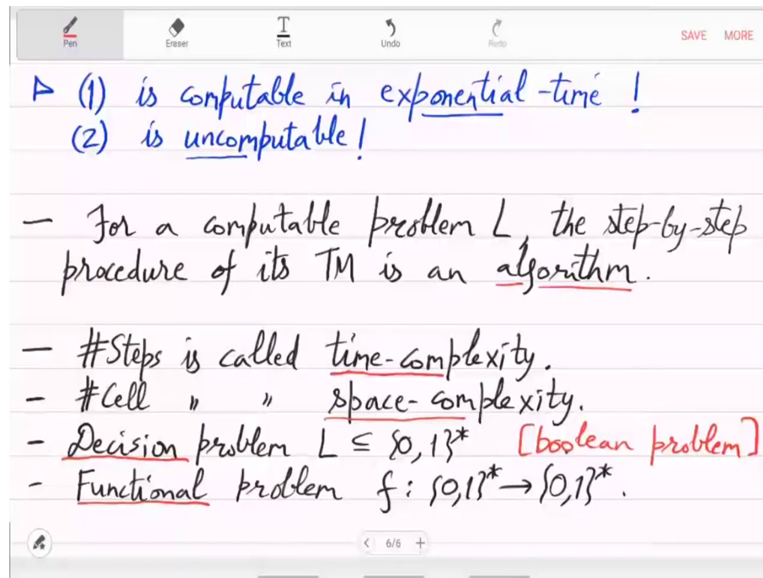
And a simple example for this is $f = x_1 - (x_2)^2$ has a root. Example, take $x_1 = 4,\ x_2 = -2$. You are given say some complicated polynomial f like this or much more complicated than this and the question is whether there is an integral root. This problem is also called the diophantine problem. Think about this, how will you solve this?

And you can make it even more complicated by considering a polynomial system. So not just one f, but many polynomials are given, and you want a common integral root. This $\bar{x}$ or variables $x_1, \cdots, x_n$ will be used interchangeably. So you are given m quadratic polynomials in n variables, find an integral zero of $f_1 = f_2 = \cdots = f_m = 0$. In this polynomial system whether there is an integral zero? And if there is then find it.

So it is a very famous problem with a history of around 120 years. It is called Hilbert's tenth problem. Okay. The question that Hilbert asked is whether there is an algorithm for this, hoping that there indeed is an algorithm. You just have to find it. Now note that there are infinitely many integers. So the space of integral 0s which satisfy $f_1 = f_2 = \cdots = f_m = 0$ is infinite. And how do you search in an infinite space?

Even a brute force algorithm is not clear. It was shown after 50 years of this problem being posed that this cannot be solved. It is an uncomputable problem.

(Refer Slide Time: 27:19)



So problem number 1 is computable in exponential time. This is not easy to see but somehow it can be solved in exponential time. While problem number 2 is uncomputable. With one polynomial there is still hope of doing it but with many polynomials this just becomes impossible; there is no algorithm. So the way you prove this is you pick some uncomputable problem like halting problem and reduce it to this.

Again this I would not be able to cover the details but you can have a look at the literature if you are interested. Whenever we say something is computable then we have to show an algorithm. For a computable problem L the step by step procedure of its turing machine give you an algorithm. That is the algorithm.

This can be written in C, Java or Python or whatever. Then a computer can execute it. Let us now move to the complexity part. So like I said, problem number one is exponential time. This basically means that the turing machine or the algorithm the number of steps there grows extremely fast in terms of the input size something like 2^n or worse.

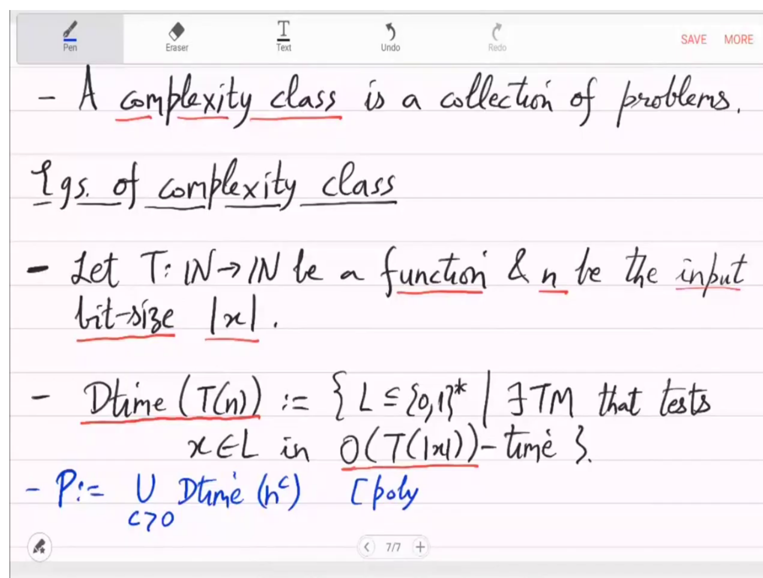
As soon as you have a Turing machine definition you can specify time, space and other resources. The number of steps is called time complexity and the number of cells on the turing machine is called space complexity. Both time and space have been mathematically formalized. Let me give you an example of a decision problem versus a functional problem.

Decision problem L is a subset of strings while functional problem functional problem will be a function from strings to strings. We will also sometimes use this qualification whether a problem is decision or functional. Decision would mean that the output is yes or no, 0 or 1 also called boolean problem. Functional problems will output a string.

For example in problem one when we are looking for integral roots, if you just ask for whether their integral root exists. Then the answer will be yes or no. That is a boolean problem. The input string is a description of f. The coefficients and the output is yes or no whether there is an integral root or not. Functional problems will be or functional versions of this problem will be that give me an integral root. Then your output will also be a string. So that is the distinction.

Once I have defined let us say decision problems, I can look at a collection of problems that is called complexity class.

(Refer Slide Time: 33:19)



First I started with a bit, a collection of bits I called a string, a collection of strings I called a language or a problem and now a collection of problems I call complexity class. You must have seen a few one or two complexity classes like polynomial time and exponential time in theory of computation. Let me just remind you examples of complexity classes.

It makes sense to define complexity classes with respect to some resource. The most common resource is time or space. You can say that all the problems that are solvable in time n or n square, I collect them. These will be like easy problems. In general let T be a function and n

be the input bit size which we also denote like $|x|$. In terms of the input bit size, how many bits there are in terms of that you can define this complexity class Dtime T(n).

Think of T(n) as n^2. Then we are looking at Dtime n^2 and so on. Collect all the problems which are solvable in this much time. All the languages or problems L such that there is a turing machine that test whether x ∈ L or not, in T(|x|) time. That is how we collect problems. Problems that have similar complexity time complexity. All these problems are solvable in T of size of x time but there is a big O.

Recall the asymptotic notation big O means that the number of steps of the turing machine will be some constant times T. Similarly there is big Ω. Which means at least constant times T and there is a big Θ. When you have both upper and lower bounds. Recall your asymptotes and that defines Dtime. Once you have Dtime you can create a lot of new classes so polynomial time is union of Dtime n^c, for all c.

We are allowing the time to grow as a polynomial power of n, n^2, n^3. Go over every positive integer, so that is polynomial time.

(Refer Slide Time: 37:59)

And you can give more time so E is union of Dtime 2^{cn} where c is again positive real. This is called simply exponential time. Further there is EXP, which is more time, it is Dtime 2^{n^c}, so this is exponential in n to the c, for some constant c, where we go over all constants. These

are a lot of problems, some very hard problems sit here. This is exponential time. Then there is a SUBEXP which is the intersection of these.

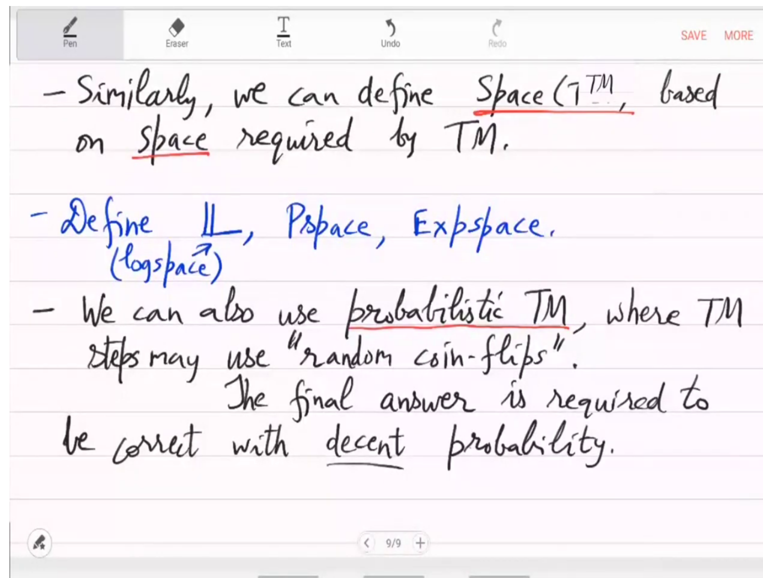
Sub exponential time, as the name tells time complexity of these problems is 2 raised to something where the something is smaller than every n^c. An example of this is Dtime $2^{\log^2 n}$ since $\log^2 n$ is smaller than every n^c. A problem with this much time complexity is in SUBEXP.

You can go on as there are many ways of defining complexity classes. Another way is by using non-deterministic turing machines where the turing machine transition function can make a choice, then it is called non-deterministic. Those classes are called Ntime. Non-deterministic turing machine based algorithms. These use non-deterministic turing machines (NTM) instead of DTM.

Just like you must have seen a non-deterministic finite automaton. That kind of non-determinism you give to turing machine and then you define classes. The key thing there is, as I just said they can guess bits to solve a problem. If the steps are not fixed; the machine can actually make choices. As we did before, you will get the complexity classes - for P you will get NP, E will become NE, EXP will become NEXP, Subexp will become SUBNEXP.

This is by the resource of time but you can also do this by the resource of space and finally you can also do it by the resource of randomization - random bits. Computers can also flip a coin and then make decisions. That will give you a different set of complexity classes. Let us quickly go through that because you will be seeing those.

(Refer Slide Time: 42:41)



Similarly we can define Space T(n) based on the space by the best turing machine. Then you get this class which is L for a class of log space solvable problems. For polynomial space you will get Pspace and for exponential space you will get Expspace. These are the space classes that we will be talking about. We can also use probabilistic turing machines, where the fair turing machine where a step may use random coins. This is similar to a non-deterministic turing machine but also very different.

In a non-deterministic turing machine a choice is made but there was no random coin flipping. In a probabilistic turing machine there is an actual random coin flipping and then the path or the step is taken. Out of 2 steps one step is taken by looking at the coin whether it is head or tails and then the answer which is obtained should have a guarantee of correctness. There is a bigger demand here. The final answer is required to be correct with high probability.