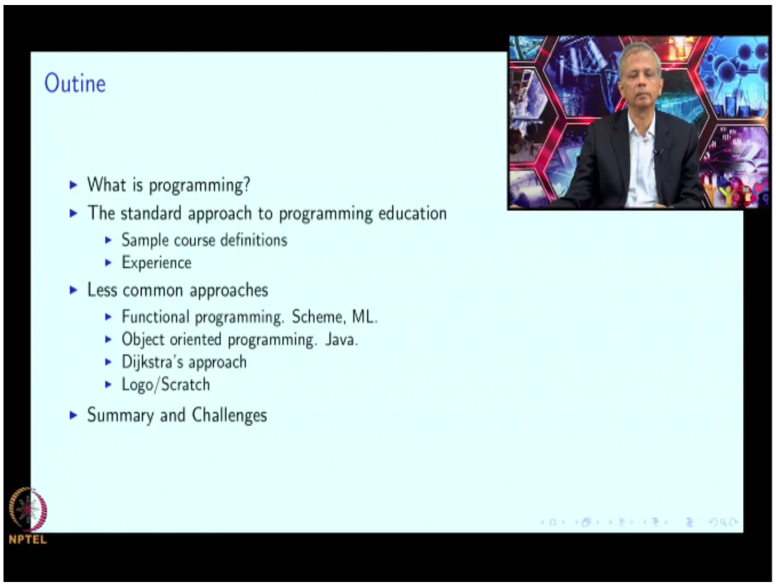
Design and Pedagogy of The Introductory Programming Course

Prof. Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology – Bombay

Lecture – 02
Introduction and Survey.0: The standard approach to introductory programming

Hello and welcome again. The outline for this part is as follows.

(Refer Slide Time: 00:30)



Outine

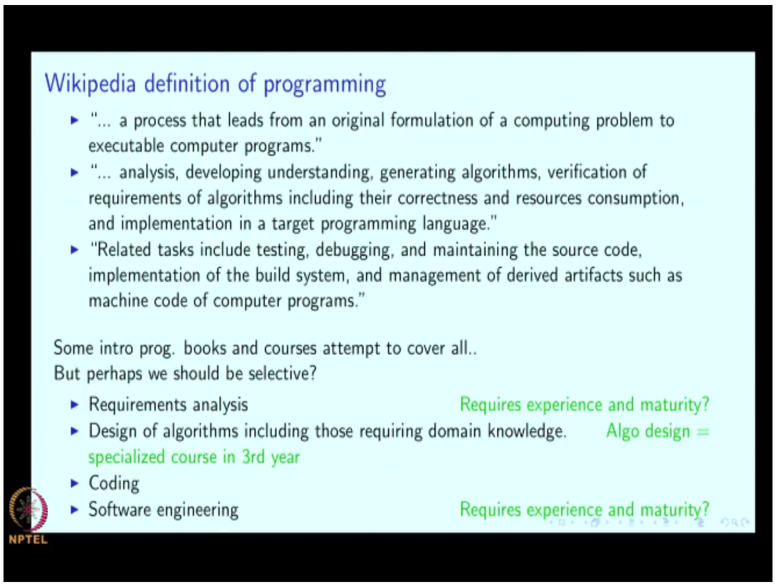
- What is programming?
- The standard approach to programming education
  - Sample course definitions
  - Experience
- Less common approaches
  - Functional programming. Scheme, ML.
  - Object oriented programming. Java.
  - Dijkstra's approach
  - Logo/Scratch
- Summary and Challenges

I am going to be starting by talking about programming in somewhat general terms. What exactly do we mean when we say programming or teaching programming and things like that? Then, I will talk about the standard approach to programming education. So this is the approach which is followed say in 90% of the universities and so we will talk about it at some length. We will have some course definitions that have been used okay.

And we will talk about the experience, so what kinds of measurements people have done, what kinds of surveys people have done with these courses. Then, I will also talk about some less common approaches. So these will be based on say something like functional programming or object-oriented programming, Dijkstra's approach. The idea will not necessarily be that we should use these less common approaches.

But I still want to survey them in order to figure out what is interesting about them and can we perhaps use those ideas when we teach and in fact one of the interesting approaches is going to involve two languages which are commonly used for teaching programming to children Logo and Scratch and I will conclude this part of the course after that by summarizing and indicating what the challenges are okay.

(Refer Slide Time: 02:00)



Wikipedia definition of programming

- "... a process that leads from an original formulation of a computing problem to executable computer programs."
- "... analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resources consumption, and implementation in a target programming language."
- "Related tasks include testing, debugging, and maintaining the source code, implementation of the build system, and management of derived artifacts such as machine code of computer programs."

Some intro prog. books and courses attempt to cover all..
But perhaps we should be selective?

- Requirements analysis — Requires experience and maturity?
- Design of algorithms including those requiring domain knowledge. Algo design = specialized course in 3rd year
- Coding
- Software engineering — Requires experience and maturity?

So let us begin with a definition of programming. So here is a definition from Wikipedia. It says that programming is a process that leads from an original formulation of a computing problem to executable computer programs. A more detailed definition; analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resource consumption and implementing in a target programming language.

Related tasks include testing, debugging and maintaining the source code, implementation of the build system and management of derived artifacts such as machine code or computer programs. Some introductory programming books and perhaps some courses as well somehow try to cover everything maybe something superficially maybe some things in detail but I feel that we should be more selective.

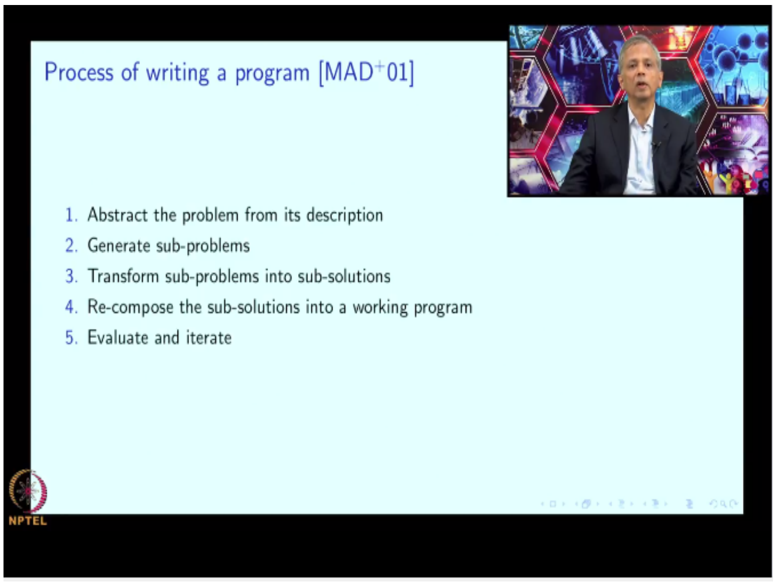
So for example requirement analysis is an important part of this big definition. Now requirement analysis in my opinion requires a lot of experience and maturity which perhaps introductory programming students do not have. So in that case perhaps it makes sense for us to give them the requirements and give the requirements in a fairly crisp, fairly precise formal manner.

This is the input, this is the output not this is the grand problem that I want to solve if you design what the input and output should be. So that is probably too difficult for introductory programming students. Designing algorithms including those that require domain knowledge is also a little bit tricky in my opinion. This is because algorithm design is actually a specialized course in the third year in most universities.

So we have to be a little bit careful in deciding exactly what we want over here. Coding is certainly going to be a big part of introductory programming courses. Software engineering, well it is mentioned in the list above but we should not really focus too much on it because it is going to require experience and maturity again just like requirements analysis.

So the point is going to be that rather than give superficial knowledge of everything, we should really give detailed solid knowledge of only a small number of things and other topics could be relegated to subsequent courses.

(Refer Slide Time: 05:08)



Process of writing a program [MAD$^+$01]

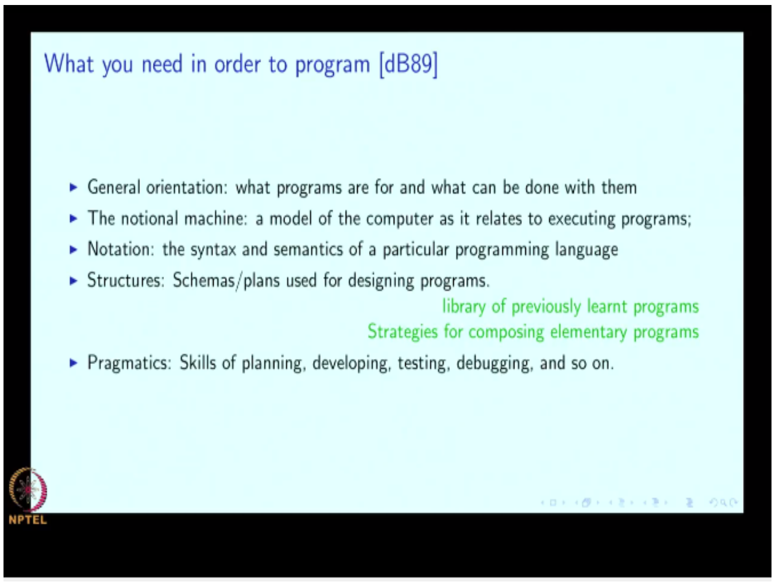
1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose the sub-solutions into a working program
5. Evaluate and iterate

Now let us look at the process of writing a program and let us try to see what researchers have said about this. So this is from a paper by a McCracken and coauthors and it is one of the influential papers. So here is what they say is involved in writing a program. The first thing to do is to abstract the problem from its description. What is that mean? Well typically the problem is going to be given to us in English language.

So we should decide, we should make somewhat more formal description out of it. So I do not mean to make this into requirements analysis but it is like what we do when we solve problems in mathematics or physics. The problem is given to us in English and we are supposed to decide what are the variables okay, what are the quantities that the problem is asking as to compute and so on, so that is the first step.

The second step is generated sub-problems. So you may be asked to solve the problem and what you really usually end up doing is breaking it up into small pieces. Then, you transform the problems into solutions or resort problems into sub-solutions and we re-compose the sub-solutions into a working program and at the end of it we again go back and ask ourselves did we do the right thing, is this solving the problem exactly as we wanted or could we have simplified things. So this is what in some sense we would like to teach.

(Refer Slide Time: 06:48)



What you need in order to program [dB89]

- General orientation: what programs are for and what can be done with them
- The notional machine: a model of the computer as it relates to executing programs;
- Notation: the syntax and semantics of a particular programming language
- Structures: Schemas/plans used for designing programs.
  library of previously learnt programs
  Strategies for composing elementary programs
- Pragmatics: Skills of planning, developing, testing, debugging, and so on.

Now Du Boulay had an interesting take on this. So he says that if we are going to a program of course we should do the steps that I have mentioned earlier in the previous slide but the programming course should really be teaching some of these things. So it should be giving a general orientation. So what that means is the student should be made to understand what programs are and what can be done with programs.

What can programs do for you, what is the capability of programs, what is the range of things that a program can do? Then, when we are talking about programming the student needs to know something about the machine. They may not need to know all the details, of course they will not be able to know all the details but just as when you are driving a car you need to have some model of what a car is even though you do not know how internal combustion engines work.

You need to know what happens when you break things like that. So some kind of a notional machine must be described and must be told to the students. Then, of course the syntax and semantics of the programming language to be used must be discussed and that is called notation by Du Boulay. Then, here is an interesting requirement called structures. So this term structure is meant to be used in the sense of mental structures.

So Du Boulay and some other researchers believe that programmers have in their mind set of can plans for designing programs. So they may have a notion that look this program is going to go over all the input and therefore this is what we should be doing. So that is the kind of language that our students must be given okay and in addition to that we should show them a lot of examples and essentially in their mind there should be a library of previously learnt programs as well.

So when they come upon a new program, we can go back to their schemers or plans or the library or the programs, the case studies that they have done before okay. That is not enough because with the new program that you want to write may require you to take elements from different programs. So you must have some strategies and some techniques as well for composing two programs or the ideas in two programs into a single program.

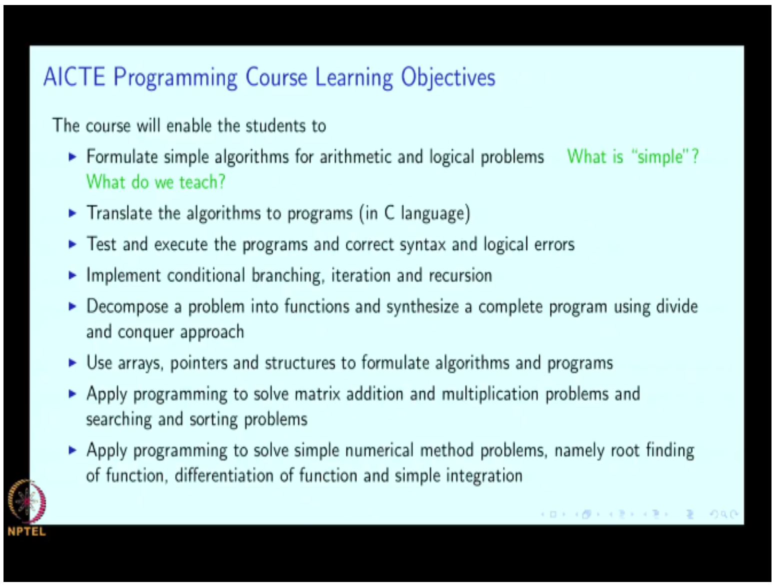
And then there are the practical aspects, so you need to develop the programs, so you will have to edit it, you will have to test it, debug it and things like that okay.

(Refer Slide Time: 09:54)

The standard approach to introductory programming

Now that was what I can say is sort of the description of the terms, so what is involved, what do people think is involved in programming okay, what people think is required to be taught to the introductory programming students and now we are going to turn to the standard approach to introductory programming. Let me remind you, by this I mean the approach that 90% to the courses use.

(Refer Slide Time: 10:29)



AICTE Programming Course Learning Objectives

The course will enable the students to

- Formulate simple algorithms for arithmetic and logical problems What is "simple"? What do we teach?
- Translate the algorithms to programs (in C language)
- Test and execute the programs and correct syntax and logical errors
- Implement conditional branching, iteration and recursion
- Decompose a problem into functions and synthesize a complete program using divide and conquer approach
- Use arrays, pointers and structures to formulate algorithms and programs
- Apply programming to solve matrix addition and multiplication problems and searching and sorting problems
- Apply programming to solve simple numerical method problems, namely root finding of function, differentiation of function and simple integration

So I am going to start by looking at the AICTE programming course on problem solving and programming and you can get, you can see what I am going to show next on their web page. I have just got the material from their web page. So here are their learning objectives. So the first learning objective is formulated simple algorithms for arithmetic and logical problems.

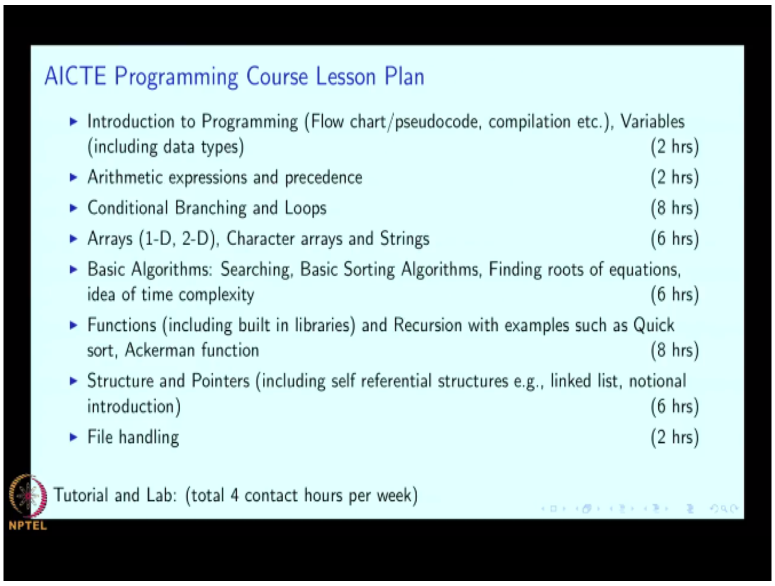
Now unfortunately we have not said what is simple and exactly what do we teach so that students can formulate simple algorithms. Translate the algorithms to programs in the C language, test and execute programs and correct syntax and logical errors. So this is kind of the pragmatics that Du Boulay was mentioning. Implement conditional branching, iteration and recursion.

So these are the language elements that are expected to be studied. Decompose a problem into functions and synthesize a complete program using the divide and conquer approach. Now this is also a little tricky. How do you decompose? It is easy to say divide it into pieces but studies indicate that dividing into pieces is not really that easy. Where do you break, how do you break? These are important questions.

Use arrays, pointers and structures to formulate algorithms and programs. Again these are language elements which are well understood but it is also understood that they must be taught to students. Apply programming to solve matrix addition and multiplication problems and searching and sorting problems. So these are fairly precisely stated specific algorithms and may be the course designers have in mind some slight variations of these which could be asked to students.

Apply programming to solve simple numerical method problems namely root finding of functions, differentiation of functions and simple integration.

(Refer Slide Time: 12:30)



AICTE Programming Course Lesson Plan

- Introduction to Programming (Flow chart/pseudocode, compilation etc.), Variables (including data types) (2 hrs)
- Arithmetic expressions and precedence (2 hrs)
- Conditional Branching and Loops (8 hrs)
- Arrays (1-D, 2-D), Character arrays and Strings (6 hrs)
- Basic Algorithms: Searching, Basic Sorting Algorithms, Finding roots of equations, idea of time complexity (6 hrs)
- Functions (including built in libraries) and Recursion with examples such as Quick sort, Ackerman function (8 hrs)
- Structure and Pointers (including self referential structures e.g., linked list, notional introduction) (6 hrs)
- File handling (2 hrs)

Tutorial and Lab: (total 4 contact hours per week)

Now AICTE also gives a programming course lesson plan and I am going to go over it just so that it is clear what emphasis they have in mind. So introduction to programming, flow chart, pseudocode, compilation etc and variables including data types and all this is expected to be taking our 2 hours. Arithmetic expressions and precedence about 2 hours, so 2 lecture hours, that is conditional branching and loops 8 hours okay. Arrays 6 hours, so notice that everything except for the first 2 hours are really language elements.

And even within the first 2 hours, variables including data types will take up at least an hour from those two hours. So so far we have really been talking about language elements. Basic algorithms; searching, sorting, finding roots and ideas of time complexity together 6 hours. Functions including built in libraries and recursion with examples such as Quick sort, Ackerman function.

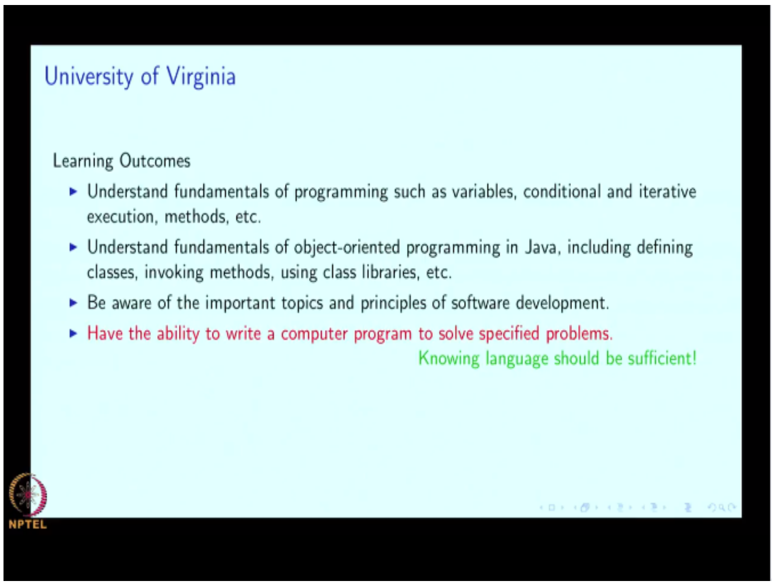
The Ackerman function seems a little bit too hard but that is okay for some reasons they have decided to put it over here. All of these will be taking about 8 hours. Structures and pointers including self-referential structures and linked list and I guess the notional introduction is only as far as the linked list is concerned or maybe all of it but there are 6 hours, so it cannot be all notional, file handling 2 hours.

So notice one thing that in all of this there is not that much discussion of how do you actually write programs. There are some specific programs that are expected to be taught but there is no discussion say there are no hours allocated to say thinking about or teaching how do I take a problem and break it up into smaller problems. Preferably, the expectation is that it will not be taught explicitly but when we discuss searching then that will naturally happen.

So searching may involve breaking the elements to be searched into 2 parts so that will be the divide and conquer aspect but it is still a little tricky because divide and conquer is sort of a very classical algorithm design strategy and it is taught and very non-trivial algorithms can be designed out of it. So there is still some question here as to what exactly the authors of this course have in mind as far as divide and conquer and breaking problems into parts is concerned.

And there will be these lecture hours and it is to be expected that there will be 4 contact hours per week for tutorials and labs, so perhaps one tutorial and 2 hours or 2 each. So that is a lot of effort that is going on and the course does look quite a good course actually except for the remarks I made about not being clear on what exactly is to be taught as far as divide and conquer is concerned, what does it mean to say what are simple programs and you will see that that is the problem with many courses.

(Refer Slide Time: 16:17)



University of Virginia

Learning Outcomes

- Understand fundamentals of programming such as variables, conditional and iterative execution, methods, etc.
- Understand fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries, etc.
- Be aware of the important topics and principles of software development.
- Have the ability to write a computer program to solve specified problems.

Knowing language should be sufficient!

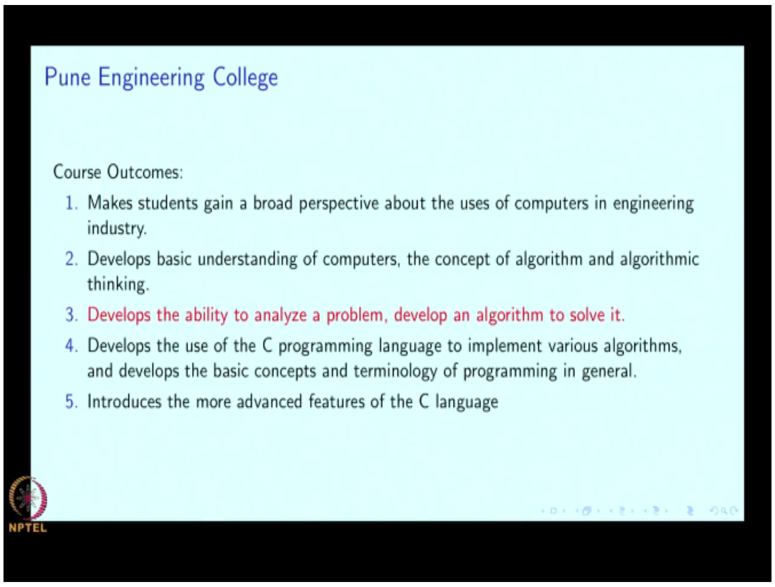
Here is a course from university of Virginia, their learning outcomes are understanding fundamentals of programming such as variables, conditions, conditional and iterative execution methods in object-oriented programming, fundamentals of object-oriented programing in Java, defining classes, invoking methods using class libraries. Be aware of the important topics and principles of software development.

So this to be look sort of heading towards software engineering. I am not sure to how much justice can be done to this but maybe it is just a pragmatic suspect maybe it is just they wean that people should be aware of how to debug and how to keep versions and things like that. Then, there is this outcome, have the ability to write computer program to solve specified problems.

Now there has been no discussion about what else to be taught okay. So it might see in that just if you teach the language, people will have that ability to write computer programs to solve specified problems and what are the specified problems is also not very clear. Are they very hard? They cannot be very hard, so what kind of and what kinds of problems are you expecting students to be able to solve is also not very clear.

We able to use the Java SDK environment to create debug and run simple Java programs.

(Refer Slide Time: 17:50)



Pune Engineering College

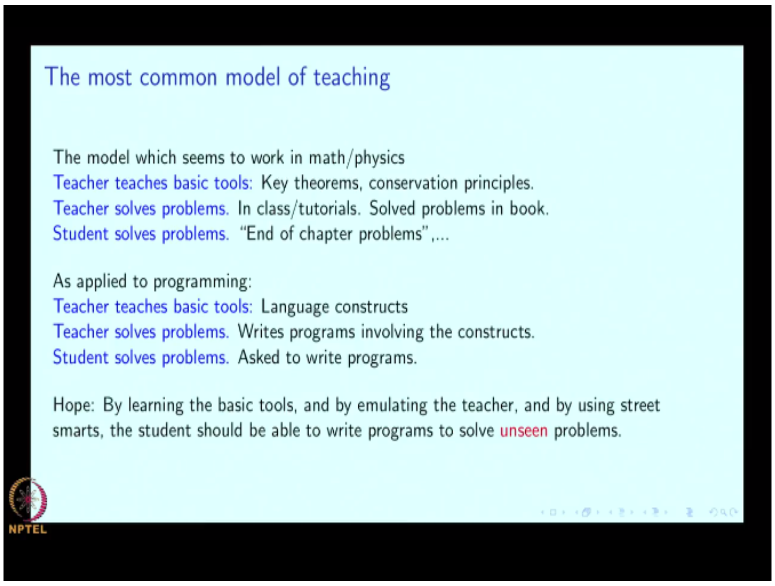
Course Outcomes:

1. Makes students gain a broad perspective about the uses of computers in engineering industry.
2. Develops basic understanding of computers, the concept of algorithm and algorithmic thinking.
3. Develops the ability to analyze a problem, develop an algorithm to solve it.
4. Develops the use of the C programming language to implement various algorithms, and develops the basic concepts and terminology of programming in general.
5. Introduces the more advanced features of the C language

Here is a course from Pune Engineering College, make students gain a broad perspective about the uses of computers in engineering industry. This is an interesting outcome, it is indeed essential that students know that computers are used very commonly in all of engineering and the power of computers should be stressed on the minds of students, develops basic understanding of computers, the concept of algorithm and algorithmic thinking, develops the ability to analyze a problem, develop an algorithm to solve it.

Now this is a very broad statement, what kind of problems, what kind of algorithms, may be something simple but then they should say something simple. Develops the use of C programming language to implement various algorithms and develops the basic concepts and terminology of programming in general, introduces the more advanced features of the C language okay.

(Refer Slide Time: 18:47)



The most common model of teaching

The model which seems to work in math/physics
Teacher teaches basic tools: Key theorems, conservation principles.
Teacher solves problems. In class/tutorials. Solved problems in book.
Student solves problems. "End of chapter problems",...

As applied to programming:
Teacher teaches basic tools: Language constructs
Teacher solves problems. Writes programs involving the constructs.
Student solves problems. Asked to write programs.

Hope: By learning the basic tools, and by emulating the teacher, and by using street smarts, the student should be able to write programs to solve unseen problems.

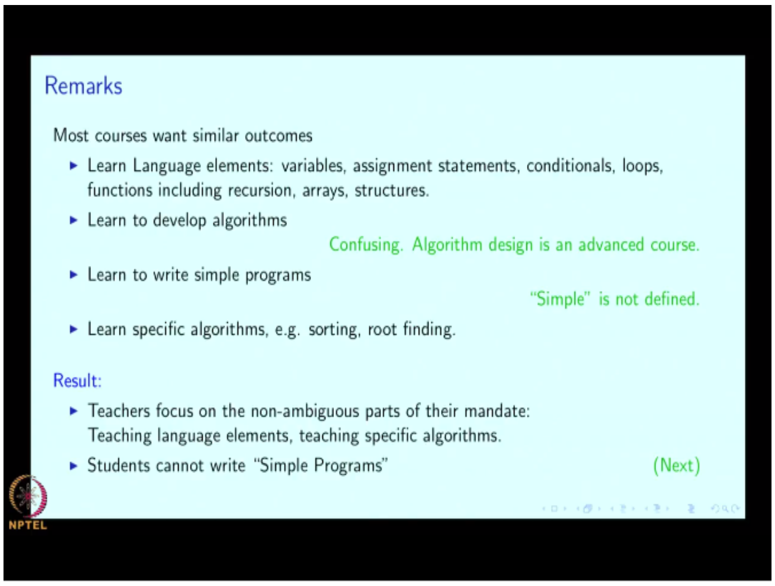
So it seems that the most common model of teaching is pretty much the model which seems to work in math and physics and what is that model, well the teacher teaches the basic tools. So in the case of math and physics, the teacher may teach the key theorems, conservation principles and then the teacher solve some problems, maybe there are tutorials. So teacher will solve problem in the board, maybe after asking the students to think about them first.

And similarly the books will have solved problems and finally the students are asked to solve problems and the book may have problems at the end of the chapter which are unsolved and which are expected to be solved by the students based on what they have learnt regarding the basic tools and whatever experience they have gained if the problem solved by the teacher. As applied to programming, what they seem to be, how this seems to have played out is the teacher teaches basic tools.

And as we have seen in the courses that we have talked about, the basic tools really only mean the language constructs, how to actually design algorithm has not been really discussed, teacher solves problems so the teacher writes program using the constructs and here there might be some cleverness or some ideas that the teacher may show; however, there is no specific time allotted for discussing the construction of programs as such. Specific time seems to be allotted only for teaching the language constructs.

And finally the student has to solve the problems. So the student is asked to write programs okay. So the hope is that by learning the basic tools and by emulating the teacher and by using some kind of cleverness or street smartness, the student should be able to write programs to solve unseen problems that seems to be the hope, okay so a few remarks.

(Refer Slide Time: 21:07)



Remarks

Most courses want similar outcomes

- Learn Language elements: variables, assignment statements, conditionals, loops, functions including recursion, arrays, structures.
- Learn to develop algorithms
  Confusing. Algorithm design is an advanced course.
- Learn to write simple programs
  "Simple" is not defined.
- Learn specific algorithms, e.g. sorting, root finding.

Result:

- Teachers focus on the non-ambiguous parts of their mandate: Teaching language elements, teaching specific algorithms.
- Students cannot write "Simple Programs" (Next)

Most courses want similar outcomes, so they want you to learn language elements, they want you to teach rather, they want you to teach language elements so variables, assignment statements, conditionals, loops, functions possibly including recursion, arrays and structures. They want you to teach students or they want your students to learn how to develop algorithms and while they may say so this is a little bit confusing.

It is confusing because the details have not been given and algorithm design is an advanced course. So teachers are being left to infer what exactly they should be teaching but they know that they should not be teaching what is there in the algorithm design course which is a advanced course and which is often considered a difficult course and most courses wants students to learn to write simple programs and unfortunately simple is not really nicely defined.

Most courses want students to learn specific algorithms like sorting, root finding and this is of course a very, very appropriate requirement okay. So what is the result of all of this, in my opinion and in the opinion of many researchers in the educational literature, teachers focus on the non-ambiguous part of their mandate which is teaching language elements, teaching specific algorithms.

And the second result is that perhaps because how to write new programs is not taught, how to develop algorithms is not taught, students cannot write simple programs okay.

(Refer Slide Time: 22:53)

Experience with the standard approach

So this is going to be the next part. Will stop here and take a break for a few minutes and then continue.