

**Programming, Data Structures and Algorithms**  
**Prof. N. S. Narayanaswamy**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Module - 01**

**Lecture - 28**

**What is an algorithm?**

**Example: Fast arithmetic-raise a number to the power n**

**Raise to power n by direct method**

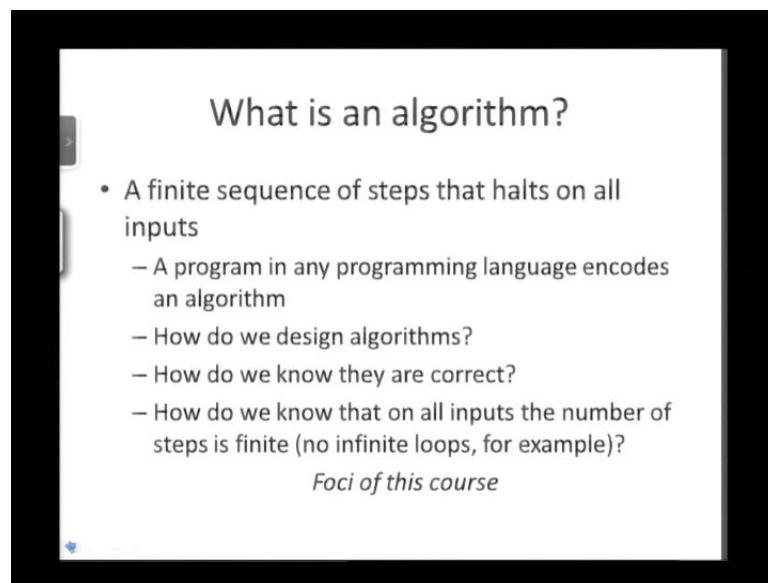
**Correctness and efficiency**

**Raise to power n by repeated squaring**

**Efficiency of repeated squaring**

Welcome to this first lecture of the algorithms course, and I am Narayanaswamy from IIT, Madras.

(Refer Slide Time: 00:23)



So, the course is about designing and analyzing algorithms and which instructive to actually start off with what an algorithm is. Algorithm is a finite sequence of steps that halts on all inputs. We are familiar with algorithms. If you think of it, a recipe to cook a dish is an algorithm. The multiplication, the sequence of steps to multiply 2 numbers is an algorithm.

And if you look at any program written in any programming language, it is desired that it encodes an algorithm. I say it is desired, because it is a fairly challenging exercise to

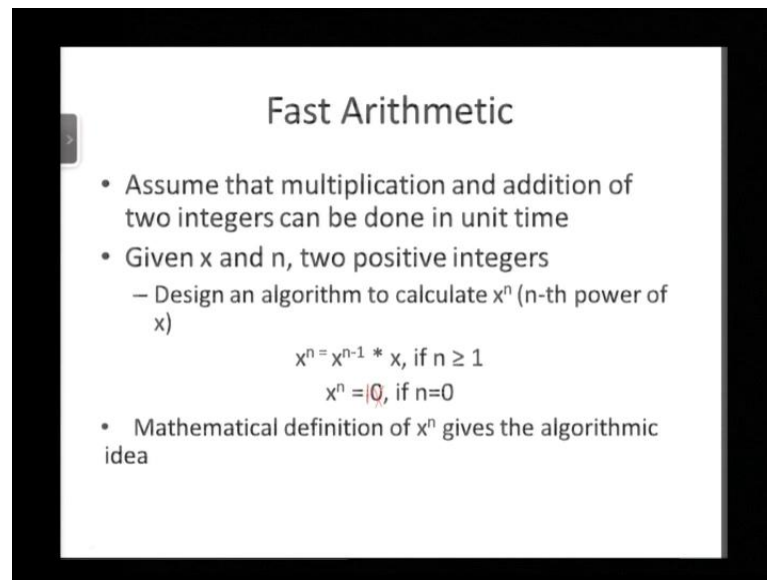
ensure that it halts on a finite sequence of steps, halts on finite number of halts in a finite number of steps on all inputs.

Now, the question that we address in this course is how do you design algorithms, how do you ensure that algorithms are correct, and how do you we also ensure that it halts on all inputs in a finite number of steps. And for those of who may already know some programming it should not have infinite loops for example. And we also address the issue of minimizing the number of steps on every input. This is not a very well stated goal. And this is one of the things we will formalize as we go through this course. These are the main foresight of this course.

Outline of the course is that we are going to have 10 one hour lectures. Every week there will be 2 of them; and every lecture would be broken into approximately 4 modules each of 15 minutes. And at the end of each module there will be some exercise for the learner to go out and try before the next module. The contents of the course are as follows. During the first week we will focus on fast arithmetic, during the second week we will work on algorithms for searching and sorting, the third week we will work on greedy algorithms, during the fourth week we will move slowly towards advanced techniques for designing algorithms and one of them to address quite a few problems as the dynamic programming approach to design algorithms.

And finally, we will come to challenging exercises which have means practical applications like string matching and identifying shortest paths in a network like a road network or any transportation network for example.

(Refer Slide Time: 03:31)



**Fast Arithmetic**

- Assume that multiplication and addition of two integers can be done in unit time
- Given  $x$  and  $n$ , two positive integers
  - Design an algorithm to calculate  $x^n$  (n-th power of  $x$ )

$$x^n = x^{n-1} * x, \text{ if } n \geq 1$$
$$x^n = 1, \text{ if } n=0$$

- Mathematical definition of  $x^n$  gives the algorithmic idea

So, let us start off with the contents of the first lecture which is all about fast arithmetic. Now, in these issues we make some minor assumptions which are often verifiable as being satisfied by most processors. We assume that multiplication and addition of 2 integers can be done in unit time. And most processes this is not a completely valid assumption because it is either a 32 bit integer or a 64 bit integer addition, or 128 bit integer addition or a finite number of bits addition that happens in unit time.

But, for the purposes of understanding the challenge of designing algorithm to do fast arithmetic we relaxes, sometimes we assume that any 2 integers can be multiplied or added in unit time. So, now, here is the simplest of exercises which we can do very efficiently as human beings and let us see how to convert this into a computer program. To convert it into a computer program we need to ensure that we understand the underlying mathematics first, we come up with an algorithm, and then you can choose your favorite programming language to convert the algorithm into a program. This is how algorithms and programming are very closely connected.

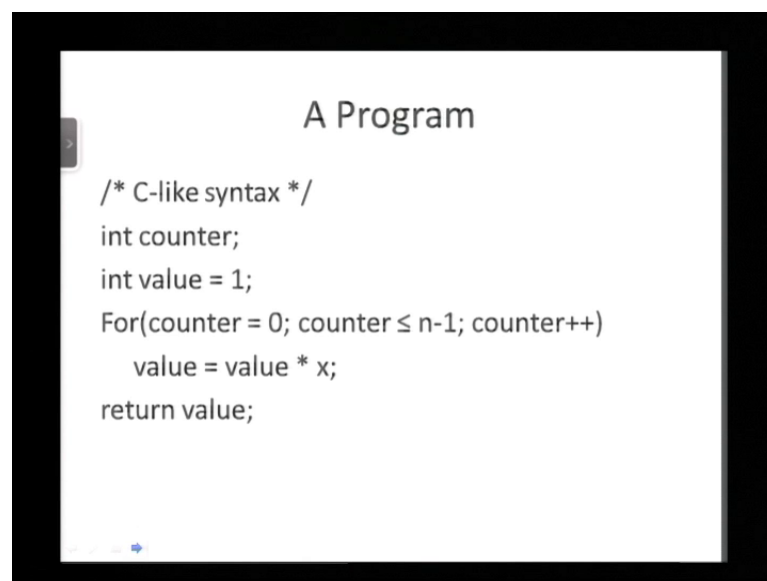
So, whenever you want to solve a problem, when you want to write a program to solve a problem, we go out and try to design an algorithm. When we try to design an algorithm we try to go out and understand the kind of inputs that we process and identify the appropriate mathematical logical steps to that part up that play an important role in the algorithm. And then ensure that the algorithm is a good algorithm; in the sense, that it terminates on all inputs, and it uses as few as steps possible; it uses as few resources as

possible. These are all the parameters that we will try to quantify as we progress through this course.

Coming back to fast arithmetic, let us consider the simplest of problems; let us consider the question where the input to a program is 2 positive integers,  $x$  and  $n$ . And the goal is to design an algorithm that will calculate  $x$  to the power of  $n$  that is the  $n$ th power of the given integer  $x$ . Now, let us look at a bit of mathematics, right; very simple mathematics which all of us are very familiar with, that the definition of  $x$  to the power of  $n$  is  $x$  to the power of  $n$  minus 1 that is  $n$  minus 1 to the power of  $x$ , multiplied by  $x$ .

And there is always a boundary condition that  $x$  to the power of  $n$  is 1 if  $n$  equals 0, right. So, this requires a bit of correction. There is an error there; this must be 1. So, this is the number 1 and not the 0. This mathematical definition of  $x$  power  $n$  gives us a desired algorithmic idea. And this is implemented in any programming language using a loop control structure which our programming language it is, we can use a counter control loop.

(Refer Slide Time: 07:29)



```
A Program

/* C-like syntax */
int counter;
int value = 1;
For(counter = 0; counter ≤ n-1; counter++)
    value = value * x;
return value;
```

As in this case where we use almost c like a syntax where you see a variable called counter which is of integer type. And we initialize the value which is going to store the variable value to contain the value 1. And then here is a loop which has an initial value which sets counted to be equal to 0 and runs for  $n$  iterations. At the end of each iteration it increases counter by 1, and value is multiplied by  $x$ ; at the end of it value is returned.

(Refer Slide Time: 08:19)

Work slide

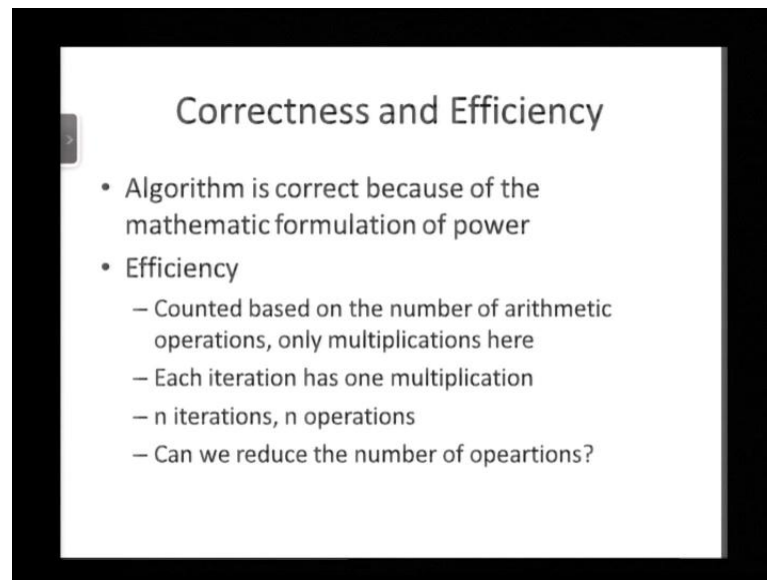
$4^3$	value = 1				
4	Counter	0	1	2	3
	Value	4	16	64	exit

Value = 64

Now, this is quite a simple program, and let us run a small example, right. So, for example, let us say I want to evaluate 4 to the power of 3. So, this is done in 3 iterations; initially value is equal to 1, and counter takes a range of value which is 0, 1 and 2. Now, when control enters the iteration for the first sign counter is equal to 0, and then so this is the table of values, value becomes 4.

At the end of this counter is incremented by 1, then value becomes 16, and value becomes 64 in the third iteration that is when counter is equal to 2. At the end of this counter becomes 3, control exits from the loop. And the final value is returned and that is 64. As you can see, the correctness of this algorithm comes from this formal mathematical definition which is exactly what is implemented in the loop in the program. And it is clear that this algorithm is indeed correct. However, we are not interested just in the correctness, but we are also interested in how many arithmetic operations are performed.

(Refer Slide Time: 10:21)



The slide is titled "Correctness and Efficiency" and contains the following text:

- Algorithm is correct because of the mathematic formulation of power
- Efficiency
  - Counted based on the number of arithmetic operations, only multiplications here
  - Each iteration has one multiplication
  - n iterations, n operations
  - Can we reduce the number of opeartions?

And that is really the focus of the next part on this small example. As we just mentioned, the algorithm is correct because of the mathematical formulation of power of a particular number. As well as efficiency goes, we measure efficiency as the number of arithmetic operation which had performed. And in this case, there are only multiplications. And it is quite easy to see that every iteration there is a single multiplication that is performed. Therefore, for any iterations there are n operations.

So, let us just go back to our program. You look at this. There are n iterations. Every iteration has an iteration number which is captured by counter. Counter takes a range of values between 0 to n minus 1. And in every iteration the value is updated by a single arithmetic operation which is a multiplication. Therefore, the number of multiplications that are associated with this program is exactly n where n is the power of x that we want to compute.

There are also a couple of additions which happen. So, where counter gets increased at every step; that is one addition that happens at every iteration. But, we are not interested in counting that because that is more a structural exercise, and we are interested just counting, interested in counting the number of multiplications. Let us test a question as how to reduce the number of arithmetic operations which are performed in computing the power.

(Refer Slide Time: 12:11)

**Repeated Squaring**

$x^n = (x^2)^{(n-1)/2} * x$ , if n is odd  
 $x^n = (x^2)^{n/2}$ , if n is even

Calculate the following sequence  
 $x, x^2, (x^2)^2, x^4, x^8, x^{16}$ , so on..

Calculate the square of the previous term  
The k-th term in this sequence is  $x^{(2^k)}$

For what value of k is  $2^k = n/2$   
For  $k = \log_2(n) - 1$

Handwritten notes in red:  
 $\frac{2(n-1)}{2} = n-1$   
 $2^{n-1} \times 2$   
 $2^{2^k}$   
 $(x^2)^{n/2}$

So, here is the next approach which is very interesting. And this is the technique that you can use even to compute the higher powers of matrices, not just numbers. So, in this case, we want to compute the nth power of x, on other word we want to compute x power n. And the observation that we make is that x power n is given by x square raised to the power of n minus 1 by 2 if n is odd, and multiplied by x.

It is very easy to do the arithmetic - 2 times n minus 1 by 2; 2 times n minus 1 by 2 is equal to n minus 1 that is what happens in the exponent. And then there is a multiplication by x. Observe that when n is odd, n minus 1 is even; therefore, n minus 1 is then divisible by 2. So, therefore, the left hand side is an integer; you get the value n minus 1. And then you multiply by x. Therefore, this is basically x power n minus 1 multiplied by x when n is odd; n is even is very clear, it is x square whole power n by 2.

So, here is a trick which is recursive repeated squaring. It is repeated squaring of what? Let us see that now. Remember, that initially we have the value x, and earlier in the first iteration we computed, in the iteration number 1 we computed the value x square. And remember that in the next iteration we computed the value x cube, instead we do something which is slightly cleverer and compute x power 4 which is x square whole power 2 which is x power 4.

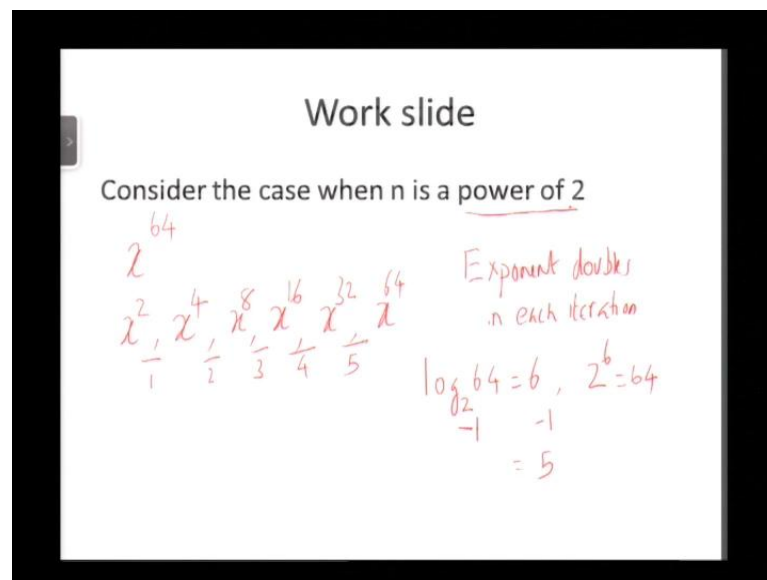
And then we compute, we square the value of x power 4 in the fourth iteration, right. There is the iteration number 3 that is zero; the first iteration, the second iteration and the

third iteration, we compute  $x$  power 8. The fourth iteration we square the value computed the previous iteration. So, we get  $x$  power 16, and so on. So, what is the whole idea? The whole idea is very simple. We compute the square of the previous term; in other words, a keep term in the sequences  $x$  power  $2$  power  $k$ .

So, let us write this term in the  $k$ th iteration. We would have computed  $x$  power  $2$  power  $k$ . And for what value of  $k$  is; so what we want to compute is  $x$  square whole to the power of  $n$  by  $2$ . So, what we do is we compute, we perform this operation of repeated squaring of the value computed in the previous iteration till  $2$  power  $k$  and  $n$  by  $2$ , or  $1$  in the sense, at which a point starting from  $x$  square we would have of computed  $x$  square to the power of  $n$  by  $2$ .

That is, we calculate the value of  $x$  power  $n$  by repeated squaring, and the termination condition is when  $2$  power  $k$  where  $k$  is the iteration number becomes equal to  $n$  by  $2$ , assuming that we start with  $x$  square. For what value of  $k$  is  $2$ , for what value of  $k$  is this equation satisfy, that  $2$  power  $k$  becomes equal to  $n$  by  $2$ ? So, for  $k$  equals  $\log$ in to the base  $2$  minus  $1$ . For the value  $k$  equals  $\log$ in to the base  $2$  minus  $1$ ,  $k$  equals  $n$  by  $2$ .

(Refer Slide Time: 17:06)



Let us do a small exercise here to get an idea as to what we are talking about. Consider the case when  $n$  is a power of  $2$ . So, let us assume that the number that is given is  $x$ ; consider the case when the number is  $x$ , and we want to compute, see,  $x$  to the power of  $64$ . So, let us write down the sequence. We start off with  $x$  square, then we compute  $x$



power 4, then we compute x power 8, then we compute x power 16, we compute x power 32, and finally we compute x power 64. Observe that at every step the exponent keeps getting doubled.

The most important concept here is that the exponent doubles at every step, in each iteration, and we started off with x square, and we wanted x to the 64, x to the power of 64, and observe that the number of iterations is 1, the second iteration, the third iteration, the fourth iteration and the fifth iteration. And it is very clear that log 64 to the base 2 is 6 because 2 power 6 is 64; minus 1 because we started off with x square, and minus 1, this is 5. Therefore, there are 5 iterations.

In the zeroth iteration is x square, in the iteration number 1- x power 4, iteration number 2 it is x power 8, iteration number 3 it is x power 16, iteration number 4 it is 32, iteration number 5 – 64, x power 64 is calculated; at the end of it for powers of 2 in the log n to the base 2; minus 1 steps; the correct value is calculated; then very importantly n is a power of 2.

(Refer Slide Time: 19:44)

**Exercise**

- Write a C-program to implement the repeated squaring algorithm
- How do we handle this when n is not a power of 2?

$n = 65$       in binary       $\begin{matrix} 65 & 64+1 \\ x & = x \\ \cdot & = x \cdot x \end{matrix}$   
 $= 1000001$        $\cdot = x \cdot x$

So, this brings us to the exercise here. As a student it is very instructive to write a c program to implement the repeated squaring algorithm. In particular, the challenge here is to write down the loop when n is not a power of 2. And this is very interesting. So, let me give you a hint as to how to go about this. Let us assume that n is, instead of 64, let us assume that n is 65, and let us write 65 in binary. It will be 1 0 0 0 0 followed by 1 is

65, right. So, this is contributes the value 1, this is  $2^0$ ,  $2^1$ ,  $2^2$ ,  $2^3$ ,  $2^4$ ,  $2^5$ ,  $2^6$ .

So, this is 64 plus 1, in binary this is 65. What we wanted to do is we want to compute  $x^{65}$ . And if you write this carefully, this is equal to  $x^{64} + 1$ ; this is  $x^{64}$  multiplied by  $x^1$ . And we already know how to handle the case when the exponent is a power of 2; here also the exponent is a power of 2; recall that 1 is  $2^0$ . And we know how to handle this. And this is a hint for you to complete the algorithm when  $n$  is not a power of 2 and implemented in the C programming language.