Introduction to Problem Solving and Programming

Prof. Deepak Gupta

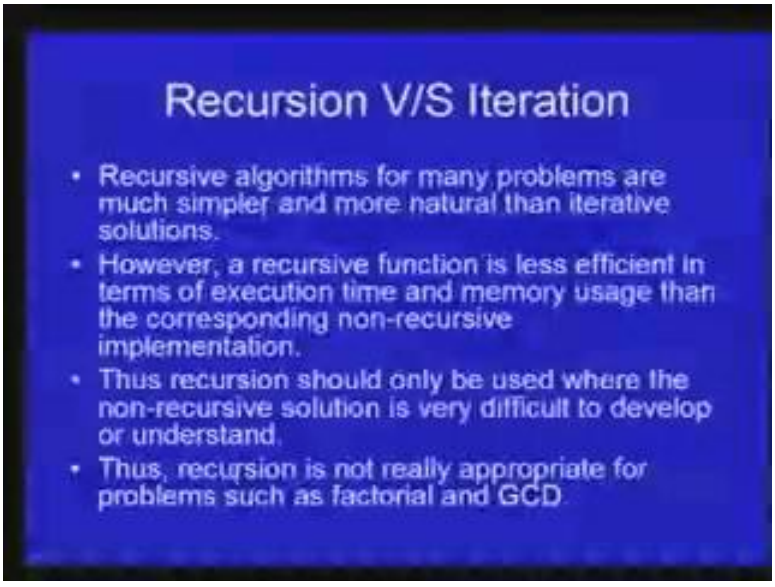
Department of Computer Science Engineering

Indian Institute of Technology, Kanpur

Lecture No # 21

In the last two lectures, we had talked about recursion, and we saw that several problems can be solved easily and very elegantly using recursion; and in the last lectures, we saw how recursion actually works, and what happens at runtime when recursion recursive functions executes and so on. So, in today's lectures, we will talk about how and when not to use recursion, and we see certain many cases recursion is not appropriate; and also in many cases if we in if you carelessly use recursion, we can actually end up with extremely inefficient programs. We will also introduce some notation for analyzing the performances of programs, which is very important to understand, because our programs should not just work, but we should also try to make the major efficient as possible.

(Refer Slide Time: 01:02)



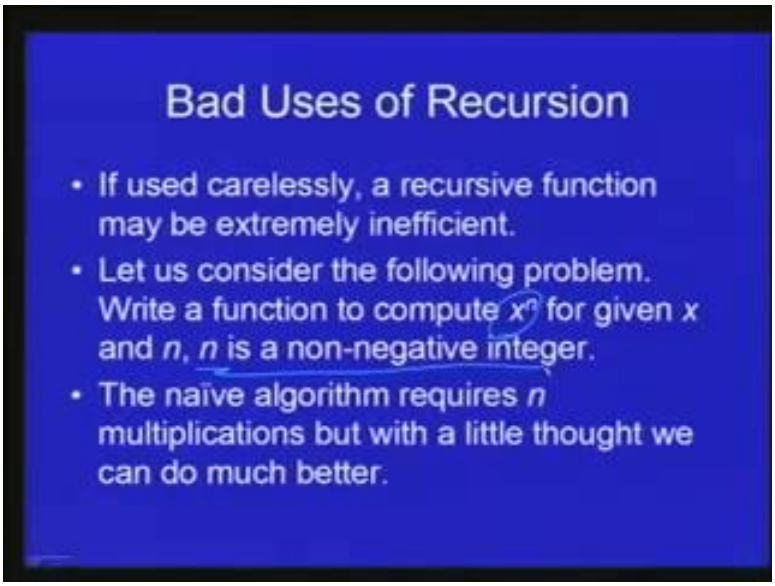
So, I say already seen recursive will got them for many problems are much simpler and more natural than iterative solutions, then we have to seen number of examples in the last two lectures. But when we discuss how recursive function works would recall that when a number of functions call that happens if very large depending on the depth of the

recursion. And each function call and return has a substantial over head, because return address and local variable etcetera have to be created space for local variable and return it has to be created etcetera the parameters have to be initialized and so on so forth.

And this overhead can become substantial in certain cases especially with recursion a very deep. So, and also in terms of usage of memory, recursion is more expensive, because we have to create (( )) for all the simultaneous invocation of the recursive function. So, there is a trade of between using recursion and writing an equivalent non-recursive or iterative function solving same problem. So, the ideal solution is at where the problem is very simple and the non-recursive solution can be implemented as well as understood easily, then we should always prefer not using the recursion, because recursion as we have seen as a substantial overhead.

However, in certain cases, the problem itself may be very difficult to solve or a solution may be very difficult to understand if recursion is not used for example, the Tower of Hanoi problem, let we solve last time. And such cases certainly we should use recursion, but what problem such as factorial and GCD, which can be easily solved and solution can be easily understood, even if it is not required for solution, recursion is not really appropriate for this kind of problems.

(Refer Slide Time: 02:57)

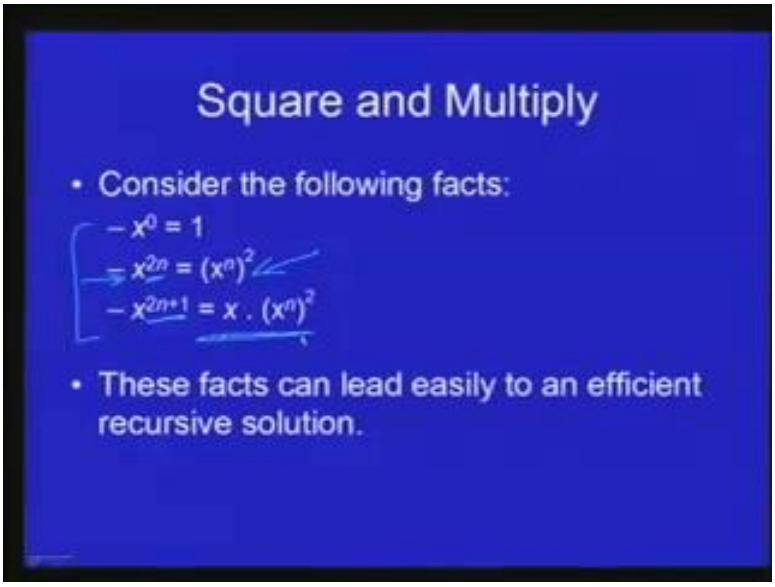


Let us now see how careless uses of recursion can lead to a extremely inefficient program, and while considering this will also introduces analysis of program for

analyzing, how much time the program take to solving a given program as a function occupies of the problem instant; and to do all that, let us considering the following problem as an example.

You have to write the function to compute x to the power n for given values of x and n. So, x can be a floating point number or an integer it does not really matter, but n is a non negative integer that is the another integer, which is greater than or equal to zero. Now hardly compute x to the power n? The n eth simplest solution is, start with one and keep multiplying the answer by x till you do it n times. So, start with one and multiply it by x n times. So, this obviously required n multiplication, and the time taken by the algorithm would obviously, proportional to the number of times the multiplication operation is performed. Now, I will be think little about this problem will find that we can actually do substantially better than n multiplications.

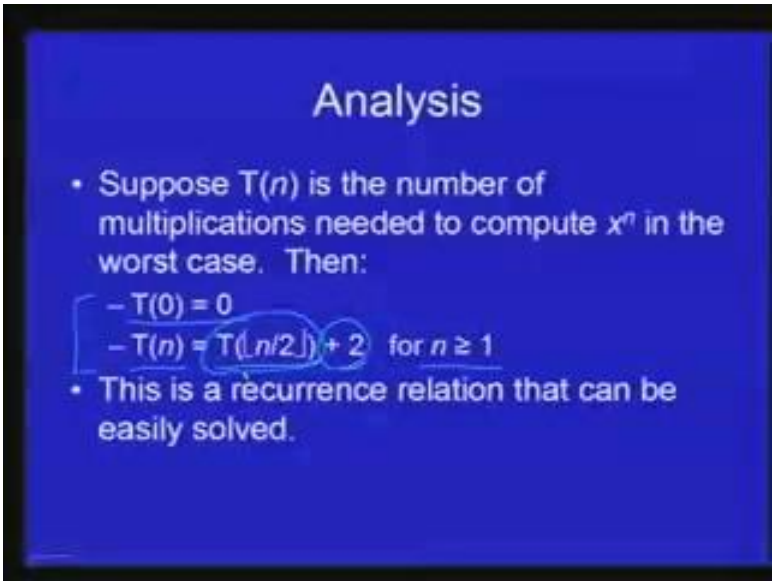
(Refer Slide Time: 04:17)



So, let us see how; substitute the following facts will all of us know about exponentiation x eth power 0 is 1 for any value of x; x to the power 2 n is same as x to the power n square, and similarly x eth power 2 n plus 1 is equal to x time x to the power n square. And we can use this facts to very easily come up to their efficient recursive solution to this problem of exponentiation; and to see what this algorithm will be we have to simply utilize this property such as we solved. If this exponent value that is given to us that is the power that we have today x 2 if it is happen to be even, then what we can do is we

can real x 2 half that power and then simply square the result. And if the power is out, then we half the power, obtain the result, square it and then multiply by x that what is happening in this case. And that can be very easily lead to a a recursive solution to this problem, we will see the recursive solution in a little while. But let us try to analysis how many multiplications should be required, if we do that.

(Refer Slide Time: 05:42)

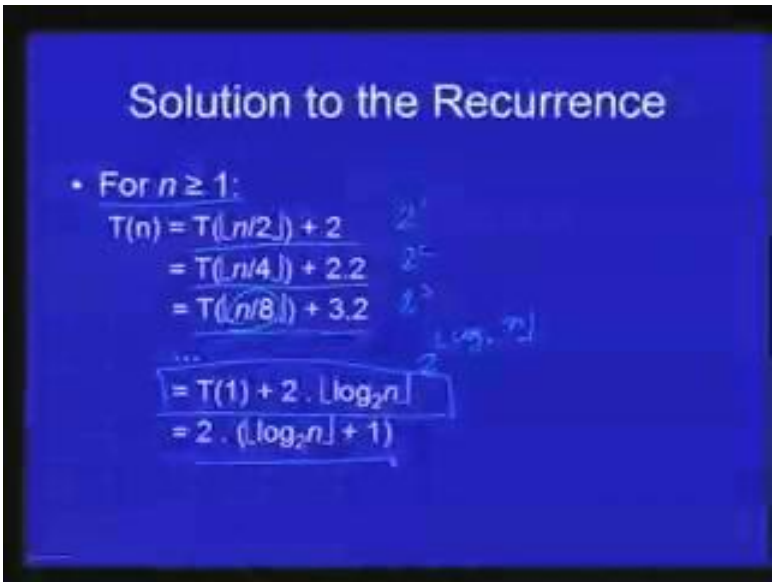


So, let us say if we denote the number of multiplications needed to compute x to the power n in the worst case as T n, where T is the function of n; clearly the number of multiplications required will be a function of n. And we have to find, what the function T n is? So, clearly you can see that T 0 is 0, because x to the power 0 is 1, and we can compute that without using any multiplication. And to compute T n, what we need to do is to compute T to power… To compute x to the power n, what we need to compute is x to the power n by 2, and then square it, and if n happen to be even that is the answer, and if n happen to be odd, then we have to multiply it once again by x.

So, how many multiplications are needed? To obtain T to power n by 2, the time taken would the number of multiplication could be T of n by 2, we are using floor, because of n by 2 would give us only by the integer components, which will be the floor of n by 2 in general. And then we have to square that one more multiplication; and then possibly we have to multiply the result again by x. So, in the work case, we required two more multiplication.

And so T of n is T of n by 2 plus 2 for all values of n greater than equal to 1. So now, this function T has been x just as recurrence relation, and we can actually evaluate all this recurrence relation by keeping one (( )) became the T of terms n by 2 writing that in terms of T of n by 4 and so on so forth. So, let us do that exercises.

(Refer Slide Time: 07:25)

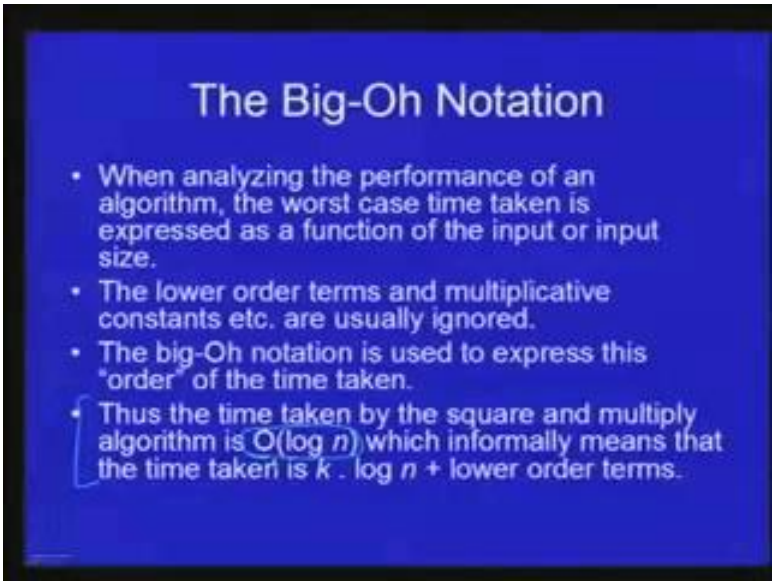


So, for n equal to 1 and greater than equal to 1 as we just for T n is equal to T n by 2 plus 2, and then in the next step T of n by 2 using the same relation, you can write T of n by 4 plus 2 and so that becomes T of n by 4 plus 2 times two. And in the next step, it becomes equal to T of n by 8 plus 3 times 2 and so on so forth. So now, every step you have to write down further dividing and further by 2 and finally, these values n by 2, 4, 8, 16, 32 etcetera, finally this value becomes one. And how many steps will it become one? Since, we are dividing by 2 to the power 1 2 to the power 2 2 to the power 3 and so on and. So, so finally when we divide by 2 power log 2 times n, then this value will becomes one. And so the point is that at in that expression will get T 1 plus 2, how many times log n times? Log n time, the floor of log n time; and so T 1 be of course, note to be again T n by the same relation if T 0 plus 2. So, we had to one more times. So, finally, the number of multiplication turns out to be two times if floor of log n plus 1.

Note that this would be logarithmic function of n whereas, in the Naïve algorithm, we have to perform n multiplications which is the linear function of n whereas, using this square and multiply algorithm, we we required only the number of multiplications

required is only multiple of the logarithmic of n. And naturally for large values of n, the even two times logarithmic of n is always going to be less than n. So, that how we analyze the running time of an algorithm and in fact, for essentially what we do is to find the runtime of the algorithm in in the worst cases, because that we are usually interested in.

(Refer Slide Time: 09:36)



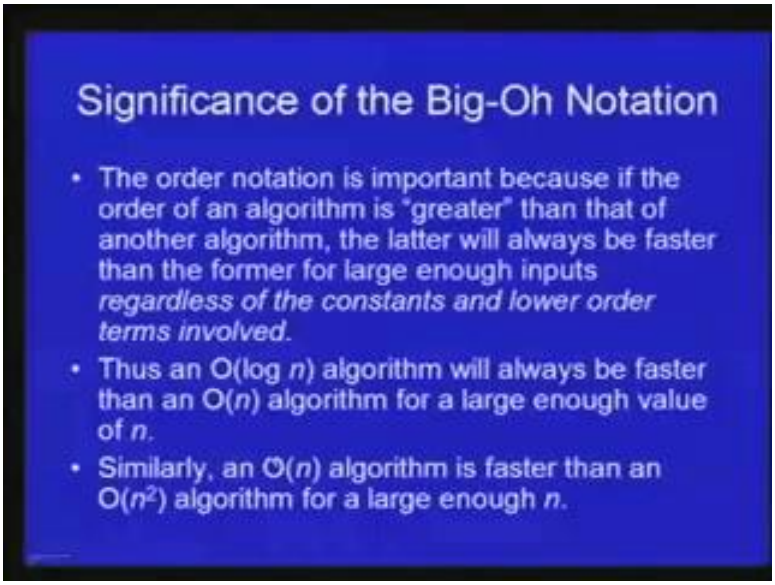
So, the worst case time taken is expressed as a function of the input of the input or of the input size. So, in this case, we have expressed n, which is the number of multiplication put really be the running time of the algorithm would also would also to be proportional to T n. Note that we are not really bothered about the actual running time of the algorithm on a particular machine on any given machine, we actually anytime of a particular program will depend on so many factors like.

How exactly the algorithm was implemented, and what was the complier used, and how fast it the machine, how much memory it has and so on so forth; number of factor which cannot really analyzed very easily, but this will take… So, this notation is such a way of analyzing the running time without really worrying about constant. So, that is big over notation that we are going to talk about. So, the running time in the running time expression that we are obtain using that kind of (()) that we just talked. Usually the multiplicative constants are ignored, because they would vary from different machine to

different machine, it goes that really important, and we cannot predict those constant really.

And also the lower order terms are not so important for reason that will just see. So, in the in the previous example, we saw that the running time will be constant times will be log n plus constant. So, both be the value of constant and the additive constant both are really unimportant in the big O notation. So, the the big O notation used to express the order of the time taken or in the general of any function. So, in in this particular example, the time taken by the square in multiply algorithm in the big O notation, if order log n, which is return in the fashion big O and within the bracket function log n. And this in formally means that the time taken is some constant times log n plus plus sums which are lower order than log n. Now the big O notation is very significant, because it abstracts the way the constant and lower order terms; and it is important to understand that if the order of an algorithm is greater than that of another algorithm.

(Refer Slide Time: 12:01)

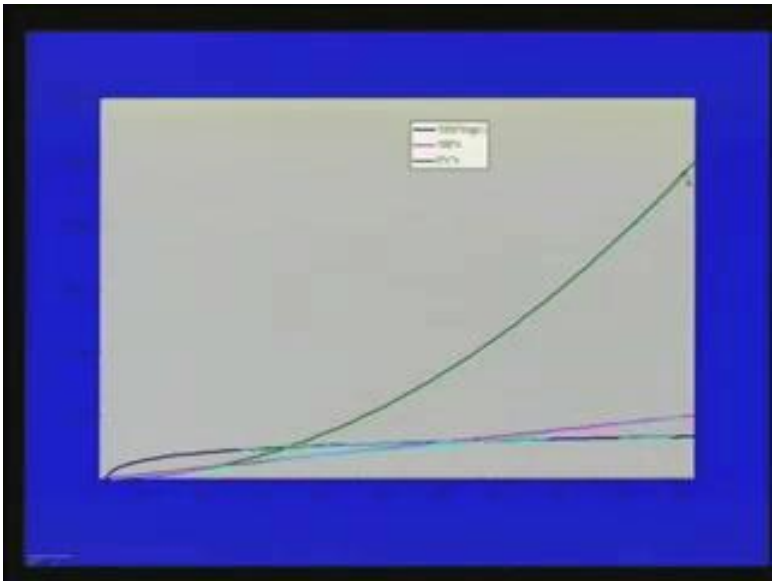


So for example, order n is greater than order log n, then the second algorithms, which are the smaller order will always be faster than the first algorithm which has the higher order for large enough input value, regardless of the values of the constants and the lower order terms involved. So, for examples and order log n algorithm will always be faster than the order n algorithm for large enough values of n. So, what does mean says that we have an order of log n algorithm in the running time of the algorithm, if some constant

times log n plus lower order terms; and for the order n algorithm, the running time is some constant times n plus lower order terms.

Now regardless of what the constant and the lower order terms are, we can always find the value of n, beyond which the time taken by the order log n algorithm will be less than the time taken by the order n algorithm.

(Refer Slide Time: 13:09)

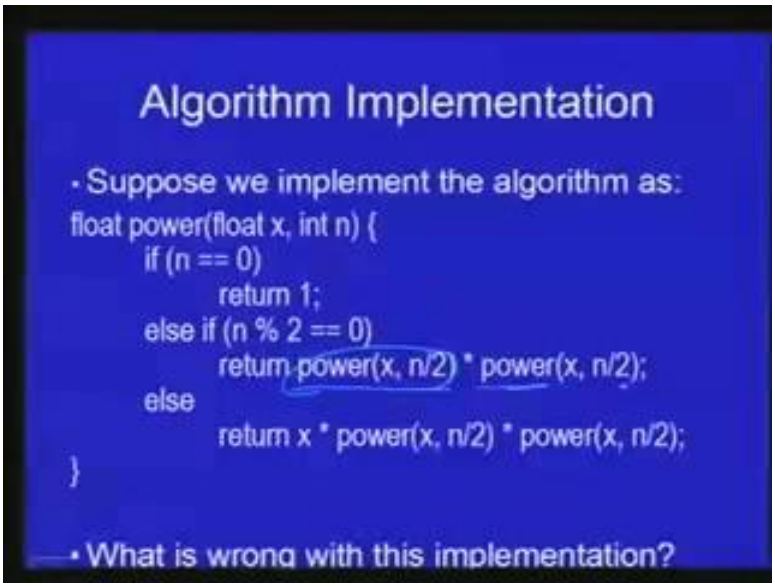


So, let us see this in terms of we see this by plotting some of this functions. So, this plots shows three curves; the blue curve plots 1000 times log x, so this is in order log x function, note there is multiplicative constant is 1000; the pink curve is the linear function of x is 100 times x. Note that even though the constant to 10 times smaller in this case, but still the linear function that is the straight line always over take this log x curves. And this will be true regardless of what the relatives, what the constant, what the magnitude of these constants are. And the third curve we have to plot is parabola is with 5 x square, and again the multiplicative constant is very, very small compared to the others.

And it will also always over take the linear straight line as well as log x curve, regardless of the relative magnitude of the constants involved. And that is why you know when we talk about the running time of an algorithm or the complexity of an algorithm; we talk about really the asymptotic complexity as it is called, where we talk about very large values of the input sizes. And for very large values of input sizes as as if curve show, the

actual values of constants and lower order terms are not really important, it is really the most (( )) or the greatest order term, which is really important that is what the big o notation really represent.

(Refer Slide Time: 14:45)



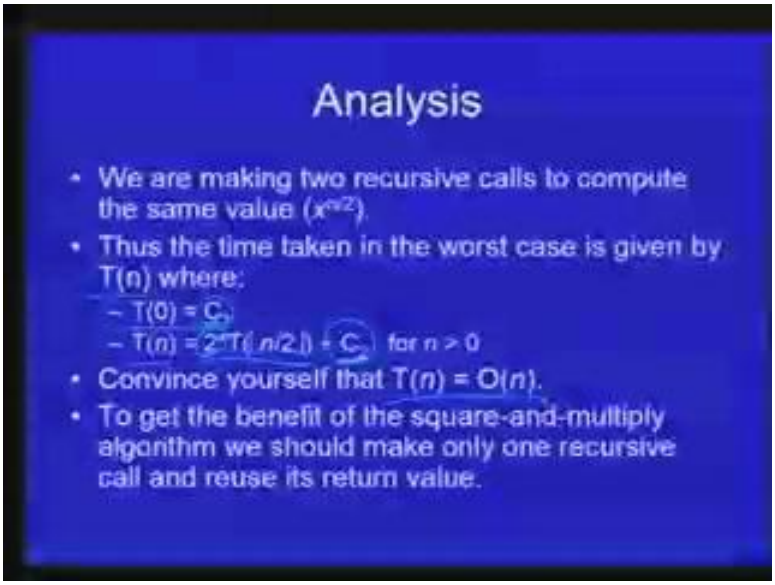
But so, we got an efficient algorithm for exponentiation, but suppose we implement this algorithm in recursive fashion as shown in this code. So, x and n are given as parameter in the function if n is equal to 1 return 1 as if n is even, then return x to the power n by 2 in to x to the power n by 2, otherwise end result and so we return x into x to the power n by 2 in to x to the power n by 2.

So, we correctly computes the values of x to the power n, but what is the really wrong with this implementation, what is wrong with this implementation is that we are computing the same values twice. So, if you look at for example, this statement we are computing x eth power n by 2 one here, and then we are again calling the same function because simply with the same parameter once again. So, we are recomputing x to the power n by 2. So, this is obviously inefficient, because we need to compute these values are one, and then we could have re-use this value to be multiplied with itself.

So, what are seen is that x to the power n by 2 is being computed twice, now for each computation of x to the power n by 2 will have to compute x to the power by 4 twice, which means the overall x to the power n by 4 will be computed 4 times; it is extremely

really x to the power n by 8 will get computed 8 times x eth power n by 16 will get computed 16 times and so on so forth. So, clearly this is an extremely inefficient.

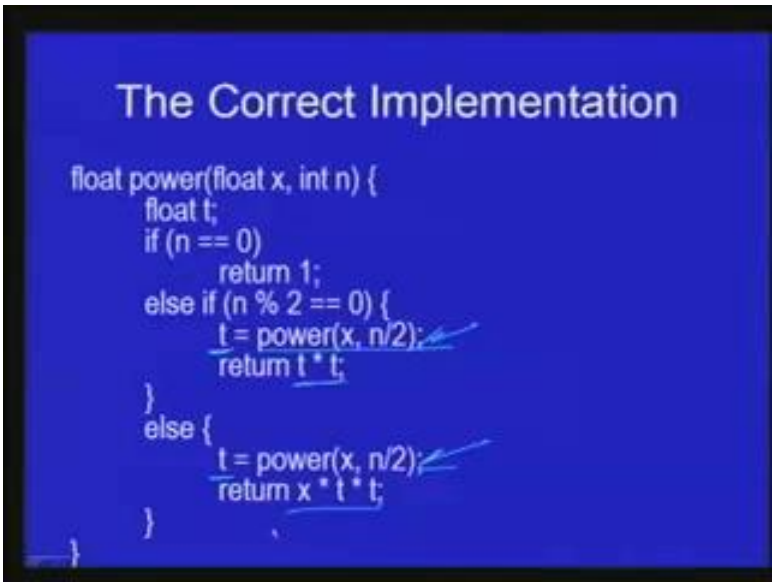
(Refer Slide Time: 16:22)



Let see (( )) how inefficient it is by trying to analysis the time taken by this particular implementation. So, let us denotes time taken to be T n for where the n is the exponent. So, when n is 0 clearly the time taken is a constant from some constant C 1, and for n greater than 0, T n is 2 times T of n by 2, because we are calling the function power x n by 2 twice plus a constant. And so this constant would be the same thing to multiply x in to power of x to the power n by 2 in to x to the power n by 2 and so on and so forth. So, I will say the actual values of a constant C 1 and C 2 are not really important. So, what does deals from the previous calculation? In the previous calculation this factor of two was not present, but because of if factor two has some, suddenly this time taken by the algorithm has gone that from being a logarithmic function of x of n to a linear function of n. So I leave it as an exercise for you to show that T n is actually order n.

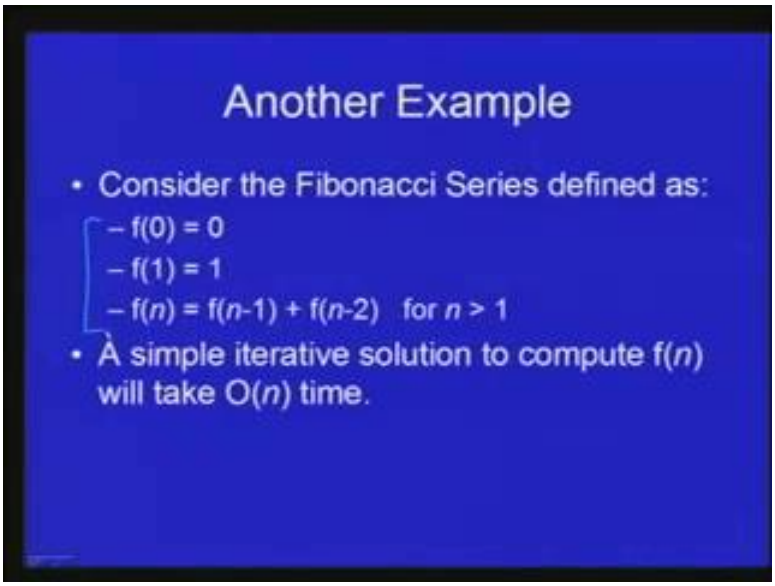
So, this is not really good implementation of (( )) multiply algorithm, the algorithm is good, but we are not really implemented it properly, but we are not used recursion carefully enough. And if we really want to get benefit of the speed up that the square and multiply algorithm (( )) then we have to make sure that we from within the function, you make only one recursive call to the function, and the reuse the return value; that quite easy to do.

(Refer Slide Time: 18:07)



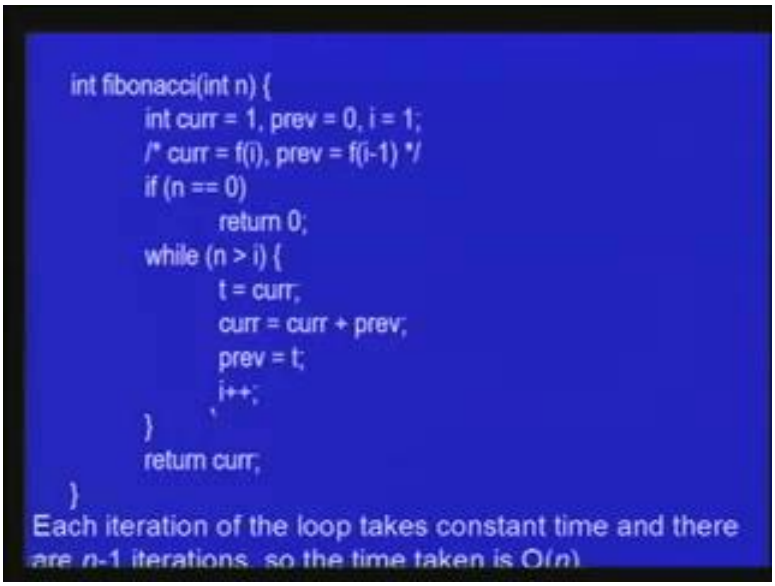
So, here what is we are doing is we if any reason, we call power again to compute x to the power n by 2, and save the return value in variable T, and then multiply T by T to compute x to the power n by 2 times x to the power n by 2. And similarly if n is odd, we compute, we save x eth power n by 2 in the variable T, and then computes x times T times T. So, there are in this function, still there are 2 cross 2 the power function, but for a given values of n, it is easy to see that either this call will be made or this call will be made both of them cannot be simultaneously even, because clearly n can be either even or odd its can be both. Of course, it could have we could have call power from where outside this statement in both cases at a common places, and then then use the values both the, but that could not be really has been any different. So, this this particular implementation of the power function, now actually correctly and efficiently implement square and multiply algorithm for exponentiation, and it does give the benefit and has the running time of water log n.

(Refer Slide Time: 19:20)



So, let us now take another example to see how recursion could not to be used. So, let us consider the computation of fibonacci number. The fibonacci series is defined as follows; f of 0 is 0, f of 1 is 1, and for n greater than 1 f of n is f of n minus 1 plus f of n minus 2. So, essentially any term in this series is the sum of the last two terms, and suppose you want to write the function to compute the enough fibonacci number that is the compute f of n. We can see that we can write very simple iterative algorithm to compute f of n, all we have do is we have to remember the last two terms in the series, and keep adding he last two terms to obtain the next term. And in the next iteration with the newly obtain term and the last term become the last two terms, and we keep going this still we have obtain f n.

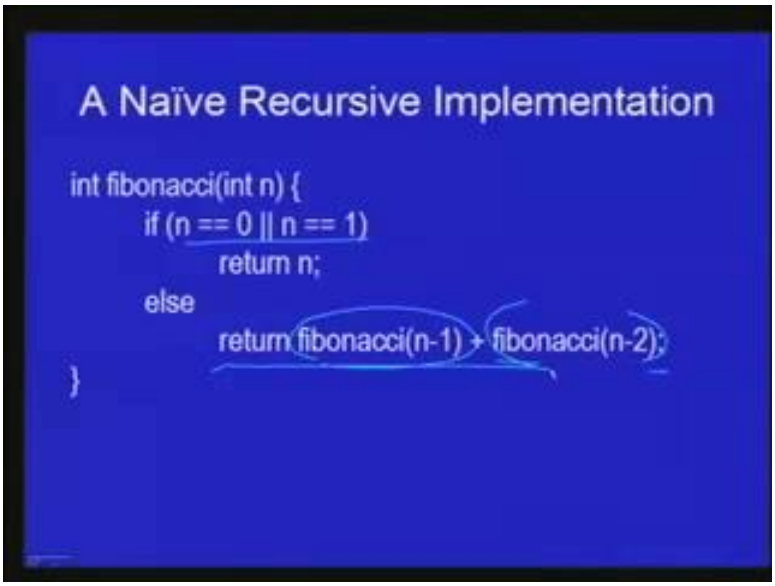
(Refer Slide Time: 20:20)



So, here you see implementation of the simple iterative algorithm that we discussed just now; curr and perv are the last two terms that has been computed in the fibonacci series. And we always maintain i in such that curr is f of i and prev is f of i minus 1. So, if n is equal to 0, we return 0, otherwise as long as the n is greater than i that means, we have not yet computed f of n, because the last term that we computed was the f of i. So, we added the two terms and that becomes a new values of curr and that the value of prev is the old value of curr, which is saved in t, and then (( )) related to previous. And then we implement i because now one more term in the prev has been computed. And finally, when n becomes equal to i we return the value of curr, because that will be the value of f of n.

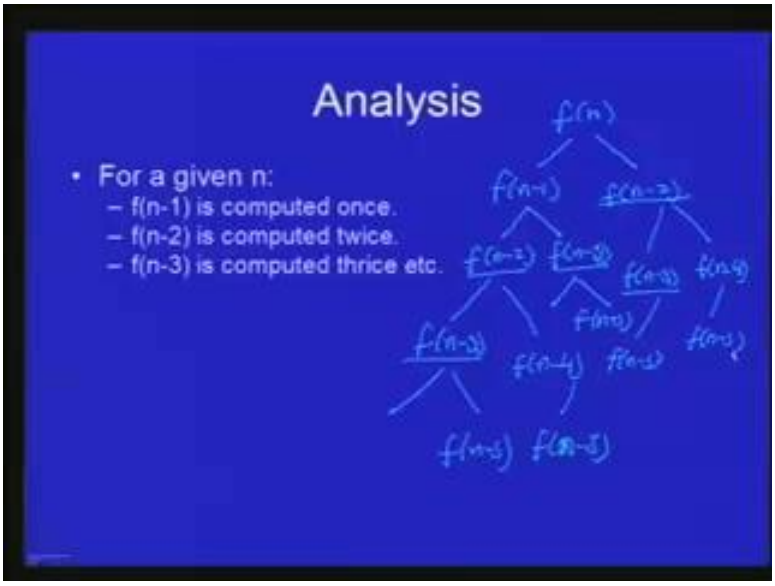
Now, so that simple enough, but looking at this looking at this recursion relation one might be tempted to implement the algorithm recursively, by just adding up the values of f of n minus 1 and f of n minus 2, and use the recursion to compute this two values; and we just see that would be extremely inefficient, but let us first analysis the iterative algorithm that we have told and we how much time it take. So, note that each iteration of this loop take the constant amount of time, it does not really depend on the value of n, and the number of iteration is n minus 1. And so, therefore, is the total time taken by this function is order n, where n is the parameter applied to be constant.

(Refer Slide Time: 22:17)



So, let us say we ride the straight forward naïve recursive function to implement fibonacci. So, we should really to write that it as it turns out to be extremely inefficient, and we will see why it is inefficient. So, if n is equal to 0 or 1, will return the value of n, because f of 0 is 0 and f of n is 1. Otherwise this function simply makes two recursive calls to itself, computes fibonacci of n minus 1 and fibonacci of n minus 2, and returns their sum

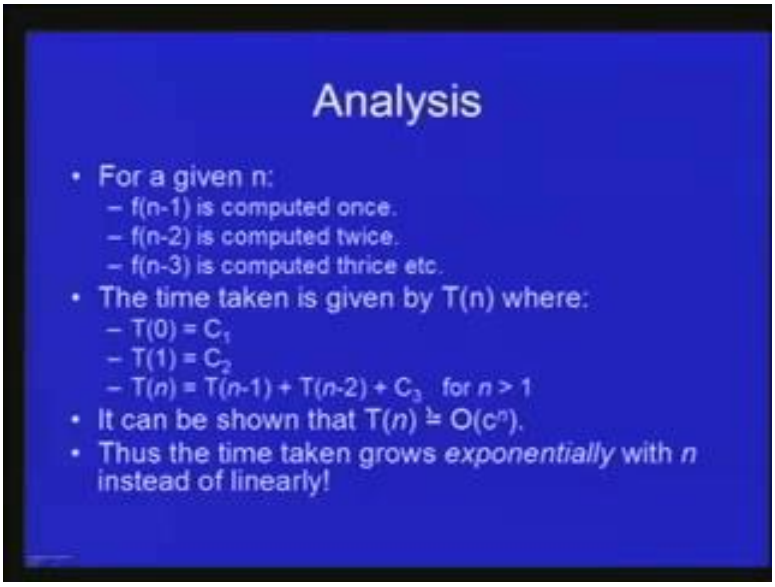
(Refer Slide Time: 22:53)



Now why is this implementation really inefficient? If it is inefficient, because of same values are again being computed multiple multiple times; f of n minus 1 is computed only once, but f of n minus 2 is computed twice, because when f of n is called, it calls f of n minus 1 and f of n minus 2, and f of n minus 1 again calls f of n minus 2 and f of n minus 3. So, if we try to see how the recursive calls to f are happening, if we get f of n calls f of n minus 1 and f of n minus 2, and this calls again f of n minus 2 and f of n minus 3, and this calls f of n minus 3 and f of n minus 4.

So, see that f of n minus 2 is being called twice and f of n minus 3 will actually get called three times, because f of n minus 2 will again call f of n minus 3. So, f of n minus 3 is calls once here, once here and once here; and we look at f of n minus 4 that will be actually called five times, because it will be called from here, and this is called from here, will be called from here, from here and from here.

(Refer Slide Time: 24:34)



And so as the value of i in previous f of n minus i is called an increasing number of times this is obviously, gross the inefficient. And if we write the recurrence relation to understand the time taken by this implementation, you find that for n is equal to 0 and n is equal to 1 clearly the time taken to be constant, but for n greater than 1, T n is T of n minus 1 plus T of n minus 2 plus constant. We will not try to mathematically solve this statement of relation, but it can be shown that T n T of n is actually order a constant times a constant space to the power of n. So what that means is that time taken by this

algorithm, but this implementation of the fibonacci series computation, the time taken growth exponentially as a function of n rather than as a linear function of n as in the case of the simple iterative solution. So, that is extremely inefficient. This is the end of today's lecture.

In the next lecture, will talk about couple of very important and frequently encountered problems; these are the problems of searching and sorting, and will try to develop from some simple algorithms to solve this problems and try to analyze this algorithm to find their time complexity.