Programming, Data Structures and Algorithms

Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute Technology, Madras

Module – 10 E
Lecture – 24
Content
Example: factorial of a number
Non-recursive versus recursive functions
Stepping through the recursive calls
Stacks and activation records
Example: Compute power(m, n)
Why and where to use recursion?

Hi, Welcome back. We looked at functions and we looked at a function being called by the main program and so on. So, there is nothing which actually stops functions being called by other functions also. But there is an important and interesting class of functions, which are called recursive functions. So, these are functions which actually call themselves. And this is actually a natural thing that happens in lots of mathematical equations and so on. So, it is an interesting thing to learn. So, I want to talk about the notion of recursion in this module.

(Refer Slide Time: 00:51)

Factorial (n)

- n! = 1 * 2 * 3 * .... * (n-2) * (n-1) * n

Iterative version

```
int fact(int n)
{
   int i;   int result;
   result = 1;
   for (i = 1; i <= n; i++)
        result = result * i;
   return result;
}
```

In practice *int* may not be enough!

So, let us start with a very simple example. Let us look at this notion of factorial of n. So, n factorial is defined as the product of the first n terms. So, formally it is; n factorial is

one times, 2 times, 3 times so on up till n. So, for example, 3 factorial is one times 2 times 3, which is 6; 5 factorial would be one times 2 times 3 times 4 times 5 that is, 120 and so on. So, if you want to write a small program to calculate factorial of a number it is not very hard. So, you have seen loops and you know how to do it.

So, let us look at this little function here called fact. It takes one parameter n and it has a result, variable, which is initialized to one and there is a loop iterator called i. So, the loop iterator runs from one to n; we can see that. And the result is just multiplied by itself. So, you take one and then multiply with one. So, that is in the result in iteration one. Then i becomes 2. You take one times 2; that is temporarily stored in result and so on. So at the end of n iterations, this loop will terminate and the variable result will have the corresponding factorial. At that point, you are ready to return it. So, this is a fairly straight forward and simple code.

And so we have seen this notion of function name, the formal parameter n, the return type int and actual return result and so on. So, one small issue. This is result; can only accommodate certain values. So, integers in the real world are unbounded, whereas integers in C programming is bounded to a certain large value. So, if you give a large enough n, this program may actually give you incorrect result. So, I suggest that you go and try something like 40 factorial and fifty factorial and so on and see what the result is. So, you would be surprised at the result that you get, but this is because the integer variables cannot accommodate results of indefinite size.

(Refer Slide Time: 03:14)

Factorial(n) – recursive program

$n! = n \times (n-1)!$

```
int fact(int n)
{
    if (n == 1)
      return(1);
    else
      return(n * fact(n - 1));
}
```

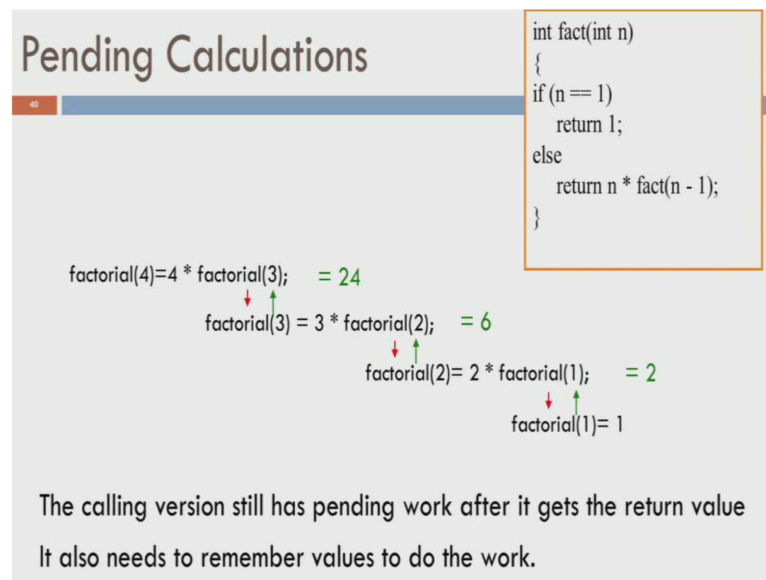
- Shorter, simpler to understand
- Uses fewer variables

So, there is one way in which you can look at this program. In which, this problem you can look at n factorial as the product of n times n minus 1 factorial. So, this is a very natural and recursive way of defining n factorial. So, this is something that you might have seen in your school days. So, n factorial is defined as n times n minus 1 factorial for appropriate n. So, clearly n cannot be; so if n goes to negative, so this thing keeps going on forever and so on. So, we will have to be careful about it. But for positive numbers, for positive integers n, this seems to be working nice. So, let us say I want to take this idea and convert that to a program as the idea shows.

So, I would like to do something like this; return n times fact of n minus 1. So, whenever I am going to call this factorial with variable n, I am going to return n times factorial of n minus 1. So, that seems to faithfully do what this mathematical description wants. However, as I said we cannot go on doing this indefinitely because at some point let us say I start with 5, 5 will go down to 4, 4 to 3, 3 to 2, 2 to 1 and so on. And at some point it will become negative. And then what do we do? So, we have this extra check; if n equals one, we return one. So, that is the base case for the recursion. Even here we need a base case, which defines one factorial as one. So, this is clearly very short. And if you understand how functions can call themselves, it is easy to understand also and it definitely uses fewer variables.

So, the nice thing about that is it becomes very readable. But, the slightly messy thing here is that so you have fact. And we looked at the control flow of programs. So, if main call fact, we know what it does. But then there seems to be a call to fact, within fact itself. So, you need to understand how this is going to happen.

(Refer Slide Time: 05:24)



So, let us look at this setup first. So, I have this program on the right side just for clarity sake. And let see we want to look at. Let us say we want to look at how this is going to work. So, I want to do factorial of 4. Let us say the main function called fact with 4. So, this local, this formal parameter n will copy the value 4. So, n is now 4. So, factorial of 4, you go and check. If n equals one, at this point n equals 4. So, this condition is not true. So, you have to return n times factorial of n minus 1. However, you have 4; which is n factorial of n minus 1 is not known yet. So, at this point you have to calculate factorial of n minus 1 which is factorial of 3 and then do this product 4 times that and only then you will be able to return the value. So, it seems logical. We have 4 times some value that we need, but we have not computed that yet. But once it is computed, I can multiply that with 4 and I will have the result of factorial of 4. I will be able to return the value. Right

So, but now that I have this fact of n minus 1, let us see how to do that. So, fact of 4 in turn calls fact with the parameter 3. At that point, factorial of 3 again you could. So, n takes the value 3 now. And this multiplication is pending and this value 4 has to be

remembered. So, we will look at that in little more detail later. But remember that this star, this multiplication, is pending as of now. We cannot do it yet.

So, by that time we get the factorial of 3, we can then multiply it. So, to find out factorial of 3 we call fact with 3 with in fact itself. And again you check if n equals one, at that point n equals 3. So, 3 times factorial of 2 is required. So, we want fact of 2. Again at that point, I should remember that n is 3. And I need to do multiplication. So, there is a pending calculation and there is a value with which you have to do it. But, right now I am going to just go and look at how to compute fact of 2. So again if we go one step further, it says take 2 and multiply with factorial of one. But finally when you call this with factorial of one, at that point n equals one. And this condition is true if n equals one return one; which means, you are actually going to return from the function. This else clause will not to be looked at anyway because n equals one. You are now ready to return from the function. And what are you returning? You are returning the value one. So, factorial of one is one. So, we have touched the base case of recursion.

At this point, we know fact of one. So, there was a pending multiplication. It was waiting on this factorial of one to be calculated. So, we return the value one. And this one, when you multiply by 2, factorial of 2 is 2. So, what are all the pending things at factorial of 2? It was remembering 2 and then this product was pending. Once the product is ready, once the other factor is ready, so you have n times factorial of n minus 1. Let us say n equals 2. Once factorial of one is ready, you have to compute this product. And that product, remember, it has to be return. So, now we have computed the product as 2. And this; once that is computed as 2, you are now ready to return it. And there was this version of fact where n was 3. It was waiting on factorial of 2 to be return back. So, factorial of 2 is now 2. You are now ready to do the product, which is 6. And you are now ready to return it back. So, if we continue doing this, at some point we will go back to the very first function call that was made to fact.

So, we called with 4. So, 4 times; now I have factorial of 3 ready, which is 6. So, 4 times 6 is twenty 4. We do the product and we are now ready to return the product. So, twenty 4 actually gets return back to the caller of fact of 4. So, this is how this works. So, this may look like a little bit of magic right now. But, we will see in the next slide in detail how this actually works.

So, the calling version; whenever it has pending work, it will just as though it will suspend itself and it makes a; so it passes the control to the new function. Once the return value comes back, it will do the pending computation. But during that time, the caller needs to remember the values. So, once the return happens, you get a value from there. You have to remember what computation has to be done and on what value you have to do this computation. So, this is something we will look at in detail in the next slide.

(Refer Slide Time: 11:03)

Recursive Function Calls

- New automatic local variables and formal parameters are created for functions for each call
  - true of both recursive and non-recursive functions
- These are stored in a memory area called ***stack***
- Each new function call **pushes** a new ***activation record*** on the stack
- These values are used in calculations
- When a function call returns, its activation record is removed from the top of the stack

n: 0
result: 1

n: 1
result:

n: 2
result:

n: 3
result:

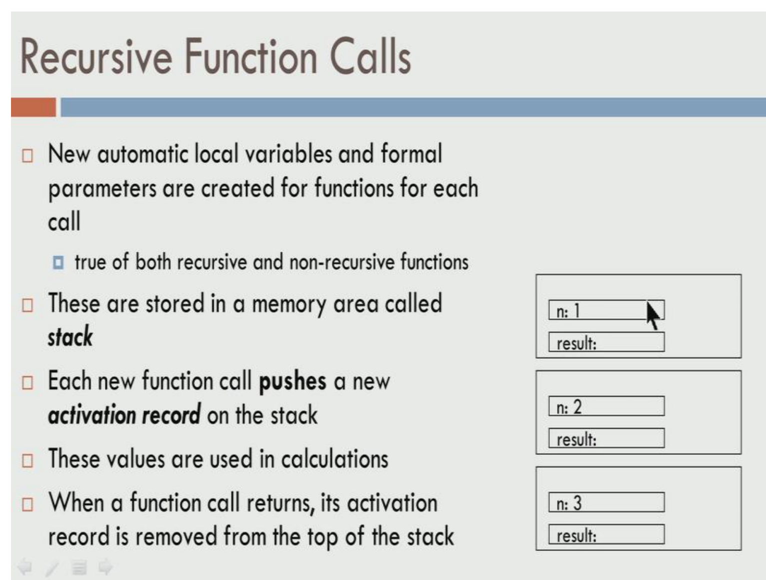
So, let us see how recursive function calls are actually implemented. So, we already know with respective functions that all automatic local variables and formal parameters are created every time we call a function. So, we saw this in an earlier module that every time a function is called, you have a new avatar of the variables. They get used up during the function. And when you return from the function, all these actual parameters and automatic local variables are all destroyed also. So, this is not just true of non-recursive functions; it is also true of recursive functions. So, even for a functions like fact this notion is true. Let us see what the implication of that is.

So, whenever you have these automatic variables and formal variables, they are actually stored in an area called stack. So, this is something which is actually a region in memory. And every function call will push what is called an activation record on the stack. So, the activation record contains what are the different variables that are local to the function and so on. And the activation record gets pushed on to the stack, you pass the control to

the caller, passes the control to the callee and the callee does various calculations. And when a function call returns, the activation record is removed from the top of the stack. So, solve this; may sound like theoretical.

Let us see pictorially what it is doing. So, let us say I called factorial for n equals 3. So, I call fact of 3 and the fact as a function creates an activation record. So, at that point we have n equals 3 and we do not know the result yet. So, this is the state in the beginning. Now, we call n times fact of n minus 1. So, n is already saved. So, that is there in your record. The result is unknown. We will have to come and update the result later. But, now you make a call to fact of 2. So to do that, it actually creates another activation record with n equals 2. So, remember it is not over writing the value 3 here in the current activation record, it creates a new record. And for the new record you know the value of n because fact of 3 called fact of 2. So, n equals 2; that fact of 2 the result is unknown. So, it is pending. Let us say at some point you called fact of one, you create another activation record. So, you have this activation record n equals one. And at this point the result is known. So, this activation record assumes that we go all the way down to 0. Zero factorial is also one. So, instead of n equal to 1, if you have checked n equal to 0 and return one; this example shows that.

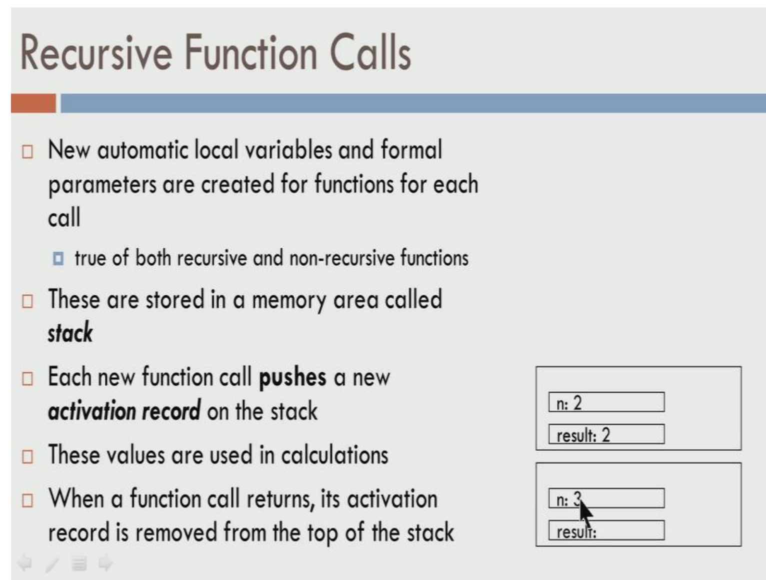
(Refer Slide Time: 14:25)



So, n equals 0 returns one. And when that returns n equals one times, whatever return from the previous activation record, the result was one. So, one times one is saved as the

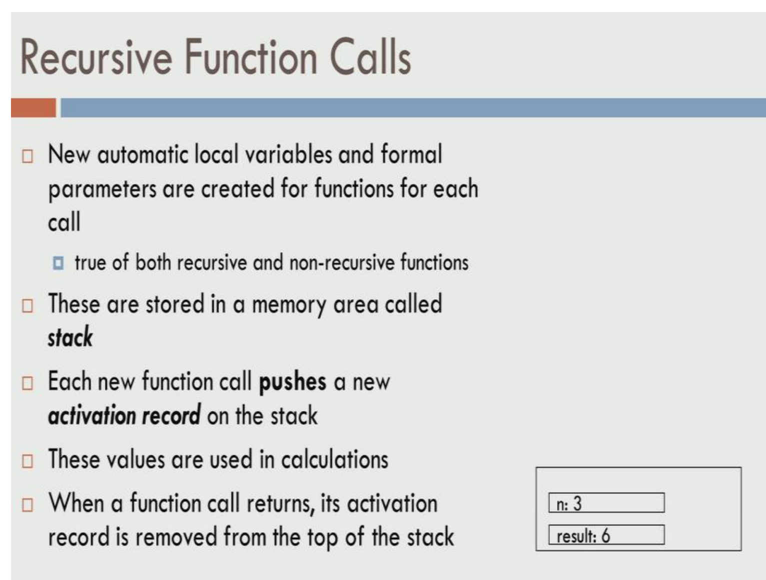
result for fact of one and this returns to its caller. Its caller is expecting to compute its result. This caller is expecting the result form the callee and it has this variable 2. It has to multiply the result from the callee and the 2 and put that here.

(Refer Slide Time: 11:58)



So, you have that. And now the callee is going to return 2; the caller is waiting with another variable n which is having a value of 3. It will take that, multiply it and put it in the result.

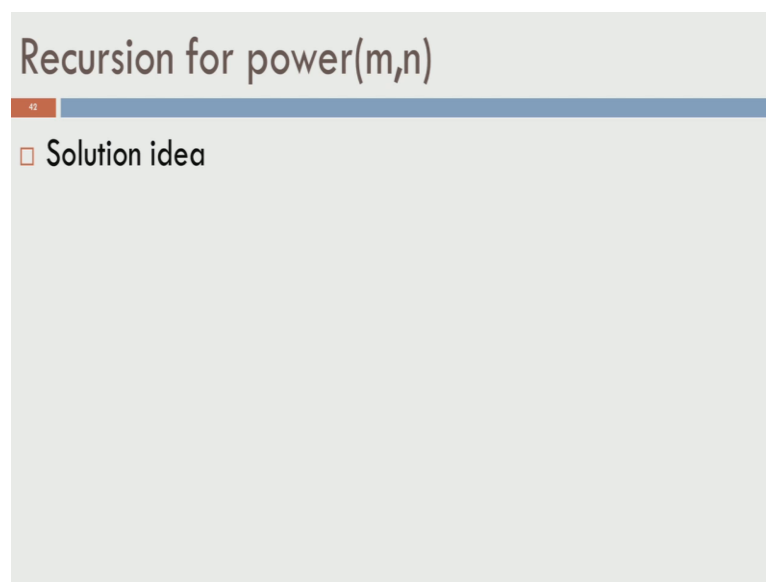
(Refer Slide Time: 15:12)



And finally whoever called fact of 3, will return with the value of 6. So, that is how it works. So, the basic premise in this thing is you have what are called activation records. So, the activation records are just copies of all the variables that are local to a function.

If a function calls itself, the activation record is kept and then you create a new activation record with new variables n and result. And you compute things there and when you return, you destroy the activation record. And the result, the return value is the only thing that is passed on to the caller. So, I hope this set up was clear.

(Refer Slide Time: 15:20)



So, we will use this notion of recursion in solving this other problem. So, I want to look at a recursive way of solving power of m, n. You have already done this using an iterative set up. Let us say I want to think about it recursively.

(Refer Slide Time: 16:20)

So, let us do this on a piece of paper. So, what we are going to do is we are going to look at a recursive way of doing power of m, n. So, we call this base and n earlier. So, I am going to call it m for now. So, earlier what we did was this. So, to compute m power n we did something like this. We did m times m times m times so on till m. And this, you need n of those. So, actually what you are doing is you are doing n minus 1 multiplications. So, this is the key thing. You are actually doing n minus 1 multiplications.

So, let us take a little while and think about whether we really need n minus 1 multiplications. So, there is this nice recursive way of doing power of m, n. So, I will take a specific example and show how this thing works. So, let us say I want to compute 3 raise to the power of thirteen. So, I could always look at computing 3 power twelve and then multiply by 3 or 3 power eleven and then multiply by 3 twice and so on. But, one nice way to do this is take 3 power thirteen and split thirteen into half. So, thirteen by 2 is 6 point 5. Let us look at the smallest integer. So, the integer which is lesser than n by 2. So, n is thirteen. Let us look at the integer that is just less than n by 2. If n by 2 is an integer, we will keep that itself. So, what is that? Thirteen by 2 is 6 point 5. The integer that is smaller than that is 6.

We will start with 6. Let us say I have able to compute 3 power 6. Right. I have to do some computation. It is not going to come jump right into our lap. We need to compute 3

power 6. But then if I have 3 power 6 and I can multiply that with another copy of 3 power 6. And if I now multiply that by 3, this is actually 3 power thirteen. So, this is correct. So, 3 power 13 is 3 power 6 into 3 power 6 into 3. So, what have we really done? We still have 2 multiplications. So, and instead of looking at 3 power twelve into 3, 3 power one right, we have it as 3 power 6 into 3 power 6 into 3. So, what is the big deal? So, what we are going to do is we are going to compute 3 power 6. But, if I am going to compute 3 power 6, this product, we can think of it as a pending calculation that we have to do.

So, I am going to compute 3 power 6. Somehow when I get that computed, I still have to multiply that by something else. So, there is pending computation. So, we will do that pending computation later. So, we have 3 power 6. Now, how do you compute 3 power 6? So to compute 3 power 6, I am going to write it as 3 power 3 into 3 power 3. So, I have used the same idea. I have taken 6 divided that into 2 divided that by 2. So, I have 6 by 2 which is 3. And I am going to calculate 3 power 6 as the product of 3 power 3 and 3 power 3. Now, again I do not have 3 power 3. So, I will keep this as a pending calculation. I am going to compute 3 power 3. So, to do that I will have 3 power one. So, this exponent here; 3, if I divided that by 2 it is one point 5. I look at the integer smaller than that; which is one 3 power one into 3 power one and this time the exponent is odd. So, it is not. So, you have to take care of the fractional part also. So, remember 3 by 2 is one point 5. If I only do 3 power one by 3 power one, I get 3 squared; not 3 cube. So, I still need to do a multiplication by 3. And to compute 3 power one I am going to look at how to do that.

So, we know that any number raise to the power one is n itself. So, m raise to the power one is m. Therefore, we have 3. So, now this gives us in some sense a very nice and recursive way of doing it. So, where is the recursion here? So, the recursion comes from the fact that if I have 3 power 6, I am going to call. So, for computing 3 power thirteen, I am going to call 3 power 6; to compute 3 power 6, I am going to call 3 power 3; to compute 3 power 3, I am going to call 3 power one to compute 3 power one. I am like; it is just any number raise to the one, I do not have to call the same function once more. Instead I can have this base case that m power one is m. That is what we have used here.

Now, let us say we actually did that. So, to compute 3 power 3 we wanted 3 power one, but we got that by making another function called factorial. It returned 3. So, we got 3

now. Now, what do we do this 3? We got 3 power one. We still have some pending work. At this point what is the pending work? I have to calculate 3 power one and 3 need not be calculated. So, 3 power one and 3; we have to take a product of that, multiply that with the current product that I got, which is 3 itself. But, one nice thing is that instead of making of function call to compute 3 power one, we just got 3 power 1. We just got 3 power one. While we made a function call for 3 power 1, we got the return result as 3. Why do not we actually use that right?

So, what happens now is instead of making another call to compute 3 power one, since we just got the value 3 for 3 power one, I will not use 3 power one. I will instead use the copy of this; which is 3 itself. Now, I have to do 3 times 3 times 3. The result is twenty seven. And this 27 gets return back to its caller. So, 3 power 3 becomes 27.

And I have to do some pending computation. The pending computation is to compute 3 power 3 and multiply that with whatever value I get. But, I just computed 3 power 3. Now, I will not call the function once more to get it. I will just use a copy of this, which is twenty seven. I still have to do this pending multiplication. So, I have twenty seven times twenty seven; that could be 729. And that gets returned here. So, 3 power 6 is now seven twenty nine. I want 3 power 6 again. But, again I will not do some computation. I will just computed 3 power 6. I will use seven twenty nine here and I still have these pending multiplications, this and this to be done. So, eventually I will do seven twenty nine times seven twenty nine times 3. And that is the value of 3 power thirteen. So, let us look at the number of multiplications that we did.

(Refer Slide Time: 24:28)

So to do 3 power thirteen, we did 3 power 6 into 3 power 6 into 3. So, that requires 2 multiplications. If I know 3 power 6, I have do only 2 multiplications to do 3 power 6. We did 3 power 3 into 3 power 3. So, that required only one multiplication. To do 3 power 3, we did 3 power one into 3 power one into 3. So, that required 2 multiplications. And to do 3 power one, we do not have to do any multiplication. We will use the base case of recursion that m power one is m. We will use that directly. So, that requires actually 0 multiplications. So if we add all of the sub, 2 plus 1 plus 2; that is actually 5. We only did 5 multiplications. I am supposed to doing it as 3 power thirteen. If I had done it as 3 into 3 and so on, right, this would have required twelve multiplications. So, clearly 5 multiplications is better than twelve multiplications.

So, I want you to go and think about what is really happening here. If it is instead of thirteen, if I had use twenty, if I use twenty 5 and so on, I want you to think about what number of multiplications you will need. If you did it using this way verses what number of multiplications you would need if you are done something like this. So, go and think about it. You will also see how to analyze this algorithm and how many steps it takes and so on in a later lecture. But, let us get back to how to write a program for this because that is the thing that we wanted now. How do you write a program to compute power of m, n?

(Refer Slide Time: 26:18)

Recursive Version of power(base, n)

```
int power (int base, int n) {
    int p;
    if (n= = 1) return base;
    p = power(base, n/2);
    if (n % 2 = = 0) return p*p;
    else return p*p*num;      }
```

The base case exp = 1
Guarantees termination

power (3, 13) $3^6*3^6*3 = 729*729*3 = 1594323$
power (3, 6) $3^3*3^3 = 27*27 = 729$
power(3,3) $3^1*3^1*3 = 27$
power(3,1) $= 3$

So, I written this simple program here to take care of this. I written a function called power. It takes base and n as a 2 parameters as it was before. And we have int p. We had this earlier also. So, instead of writing an iterative way of doing this, we have a recursive way of doing this. If n equals one, return base. So, this base case takes care of the fact that m power one is m. right. Otherwise, what I am going to do is I am going to compute p as power of base, n by 2. And if n is odd, for example, in thirteen n is odd, so I need to take 3 power 6 and 3 power 6 and multiply that by 3 once more. That is what you get here. So, this is p times p times num. If n is even, then you have only p times p. For example, in here for to compute 3 power 6, it is enough to have computed 3 power 3 and reuse the 3 power 3. You would get 3 power 6. So, one thing that you have to probably recollect is that n by 2. If n is an integer it will truncate the decimal value, it returns only an integer. So, thirteen by 2 is only 6. It is not 6 point 5. So, thirteen integer divided by 2 integer is 6. It is not 6 point 5.

So, now let us see how this whole thing would have worked. Power of 3, 13. So, n equals thirteen. So, this check would not have been true. p would get power of 3 power 6 because n by 2 is 6. You call power of 3 power 3, 6. Power of 3, 6 would transfer control to power again with n equal to 3, sorry, n equal to 6. So for n equal to 6, n is not equal to 1. That should have called power of 3, 3. Power of 3, 3 would have called power of 3, one. So when n equals to one, you have this return base. So, it would have just returned

3. So, that is going to come back where n was 3. It will come back here. So, you would have computed 3 power 1.

Now, you check if 3 percentage 2 is 0. Three percentage 2 is not 0. It is a odd number. We are looking at this exponent. For this exponent, it is an odd exponent. So, I have to take 3 times 3 power one times 3 power one times 3. So, that gives twenty seven; it returns back here. p becomes twenty seven. For the case, where n equal to 6. Right. So, 6 percentage 2 is actually equal to 0. So, you do 3 power 3 into 3 power 3, which is 729. And then you return to the case where actually n was 13. When n was thirteen, we called power of base, 6. So, you would have received 3 power 6 here. You take that as p, multiply that by p and by one more value of num that gives you 1 5 9 4 3 2 3. So, this is the basic idea of recursion.

So, clearly for every recursive function at some point you should not call the function any more. So, you have if n equals to one you return just the base without calling the function anymore. However if n is greater than one, you will call the function atleast once more. So, this is the basic idea behind recursion. So, you can think of it.

As in some sense even for induction, we do this. For proof by induction, we say this is the base case for induction. And from there on, we keep building things. You can think of recursion in a similar way. So, there is this base case and then you are building something on top of it. If you do not have a base case is program would be incorrect. So, we saw 2 kinds of recursive functions namely factorial and power. And this is a very powerful setup, because once you know recursion, there are several things that you can do very easily and you can write programs very easily. You would not have to think about doing them in an iterative manner.

(Refer Slide Time: 30:37)

A Few Points About Recursion

- Recursion must end at some point of time
  - Should be careful to have the termination condition put in
  - Think about what would happen if the check n==1 is not present in fact(n)
- Sometimes loops are straightforward to use
- Generally recursive solutions are **more elegant** but **less efficient**

$$^{n}C_{r} = {}^{n}C_{r-1} + {}^{n-1}C_{r-1}$$

But, you have to remember a few things about recursion. The first thing is recursion must end at some point of time. If function f calls itself and it calls itself and so on, it cannot go on forever. If you do that, then the program is never going to terminate. So, it can be a problem. So, you should have some condition inside any recursive function to terminate recursion. You should not call the function once more from the base case. So for that, go and think about what would happen if the check for n equal to one was not there for factorial. We just did return n times fact of n minus 1. If you have done that, see where you will stop.

So, it is not that recursion is a silver bullet. It is not going to be useful every time. Sometimes loops are very straight forward to use. So, you should see where to use loops and where to use recursion. So, in general what you have is recursive functions are actually quite elegant. It is very simple to write. You can take any mathematical formula. It is usually recursive. You take that and write it down. It is very elegant, but in general it is less efficient. So, it will recursive functions usually take more time. And if you are not careful it can take a lot more time than this writing loops to do the same thing. And of course in recursive functions you have to take care of the base case.

So, there are several other things that you can write using recursion. So, one classical example is you can do what is called n choose r. So, let us say I want to n choose r. Recursively, I can use n choose r minus 1 plus n minus 1 choose r minus 1. This is a

recursive definition for n choose r. So, out of n objects if you want to choose r objects, what are the number of ways in which you can choose them? So, this is clearly recursive. So, n choose r uses n choose r minus 1 plus n minus 1 choose r minus 1. So, all these different things can be actually done with recursion. So, that brings us to the end of this module.

Thank you very much.