Introduction to Problem Solving and Programming

Prof. Deepak Gupta

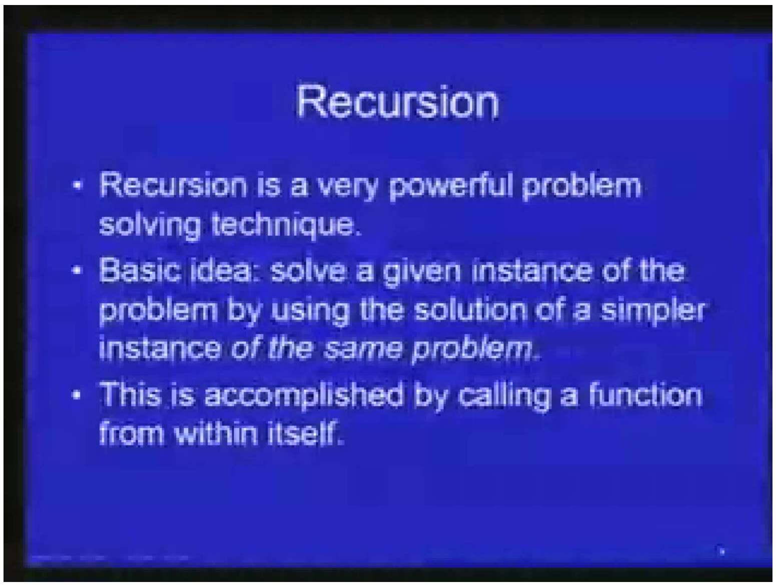
Department of Computer Science Engineering

Indian Institute of Technology, Kanpur

Lecture No. # 19

In today's lecture, we looked at very powerful programming technique that known as recursion. Recursion can be used to obtain in many cases very simply and elegantly solution to, many problems that might be otherwise that it appear at difficult.

(Refer Slide Time: 00:37)



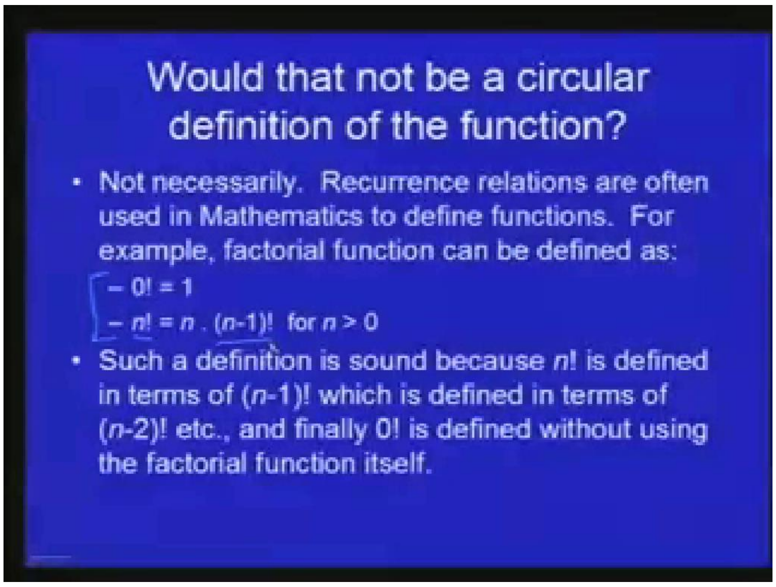
Recursion

- Recursion is a very powerful problem solving technique.
- Basic idea: solve a given instance of the problem by using the solution of a simpler instance *of the same problem.*
- This is accomplished by calling a function from within itself.

The basic idea in recursion is that we solve the given instance of this problem. What do you mean by an instance of a problem is, the problem for the given parameter. For example, finding the factorial is the problem. Finding factorial of the given value of n is the problem instance, so within the recursion what we do is we solve a given instance of a problem by using a solution to be smaller or a simpler instance of the same problem. So, suppose a problem can be defined or expressed in satisfaction that the problem to the given instance of the problem, the solution to the given instance of the problem can be expressed in terms of the solution to a simper instance of the problem. Then we can use recursion to solve that problem very simply.

And essentially, how do we obtain the solution to the simpler instance of the problem - that is the complete back calling it function from within itself. That is the function f can call itself that is essentially what is known as the recursion.

(Refer Slide Time: 01:50)

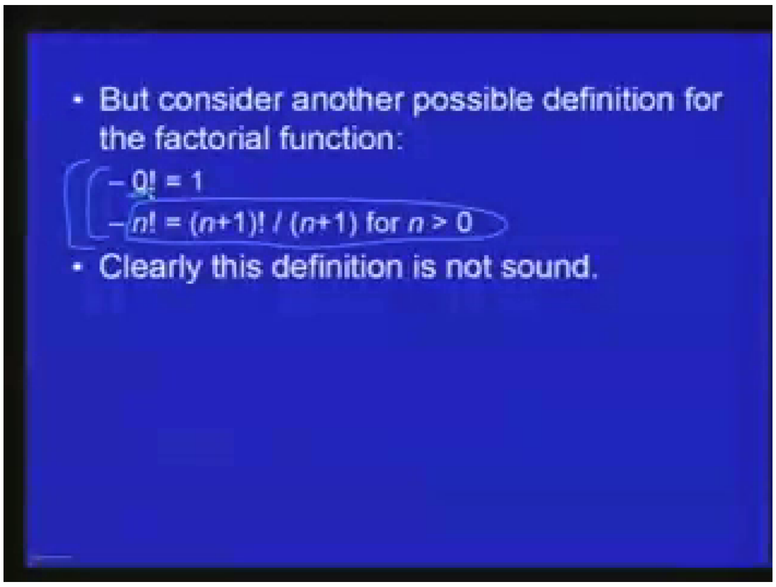


Would that not be a circular definition of the function?

- Not necessarily. Recurrence relations are often used in Mathematics to define functions. For example, factorial function can be defined as:
  - $0! = 1$
  - $n! = n\,.\,(n-1)!$ for $n > 0$
- Such a definition is sound because $n!$ is defined in terms of $(n-1)!$ which is defined in terms of $(n-2)!$ etc., and finally $0!$ is defined without using the factorial function itself.

Now this may appear to be counter (( )) of the loop because one might say that is the function call itself that it has not circular definition of the function. So that not necessarily true. As we know recurrence relation are often using in mathematics to define functions. For example, we moved that this common definition for this factorial function 0 factorial is defined as 1, and n factorial for n greater than 0 is defined as n times n minus 1 factorials, now so, n factorial is defined in terms of n minus 1 factorial, but till this definition is count because the solution to the factorial of n is the finite terms of the solution to a simpler problem which is used to finding the factorial of n minus 1 and which in terms is in term is the finite terms of n minus 2 factorial and so on. And finally, 0 factorial is to define without reference to the factorial function itself. So, the key is to realized that is that any solution to the any instance to the problem must be defined in terms of the solution of a simpler instance of the problem, and finally, there must be braces case in which for the solution to define without reference the function itself.

(Refer Slide Time: 03:03)

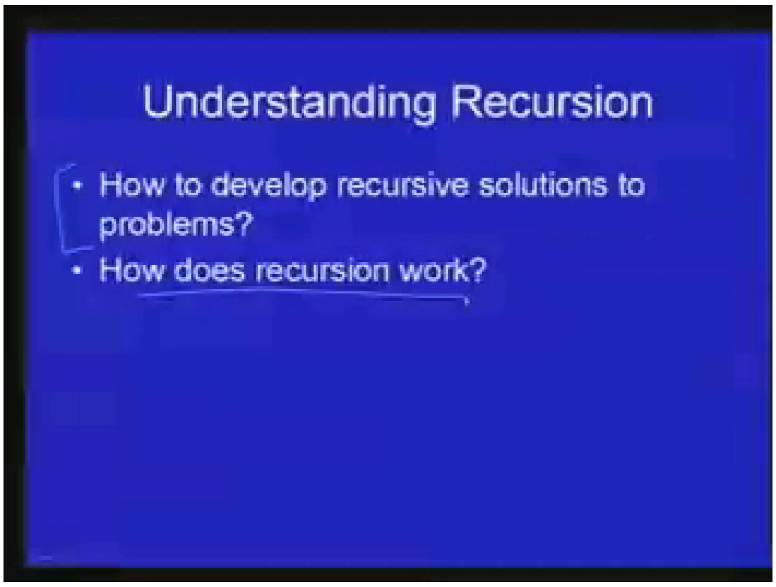


- But consider another possible definition for the factorial function:
  - $0! = 1$
  - $n! = (n+1)! / (n+1)$ for $n > 0$
- Clearly this definition is not sound.

On the other hand, one could also possible think of defining the factorial function this way - that 0 factorial is 1 as before, but by now we are defining n factorial to the n plus 1 factorial divided by the n plus 1 note that as a property this is clearly true. Because follows directly from the definition of the factorial function, but it is the definition this definition is not sound because n factorial is defined in terms of n plus 1 factorial which is defining terms of n plus 2 factorial which is defined in terms of n plus 2 factorial and so on and this never ends.

And so, you never reach the base case in which for which n is 0. So, clearly this definition is not known. So, the basic point is trying to make is that the calling a function from within itself recursively as it known is is not necessarily in correct, it is not, it not only to be correct; it can be extremely useful in certain cases (( )). So, for example, now let see how the factorial function in c can be defined recursively they have already seen that the non recursive definition of the factorial problem is simple enough, but let see how can be done recursively.

(Refer Slide Time: 04:26)

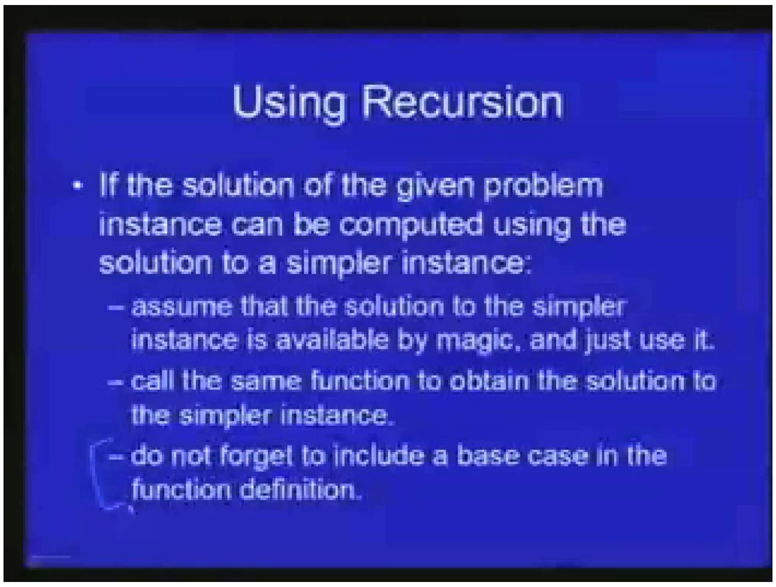


Understanding Recursion

- How to develop recursive solutions to problems?
- How does recursion work?

So, there are two things that will talk about in terms of n in the context of recursion; one is how think about recursion, and how to develop the solution to problem using recursion; and other which is equally important to understand if if to understand how recursion actually works, what what is happening when a function is called itself and so on and so forth, how do iterates the execution of the programs and so on. So, what will do is at in this lecture, we will talk about how to use recursion is a technique for finding solution to a problem and how to write recursive function we will talk about how actually recursion work in the next lecture.

(Refer Slide Time: 05:07)



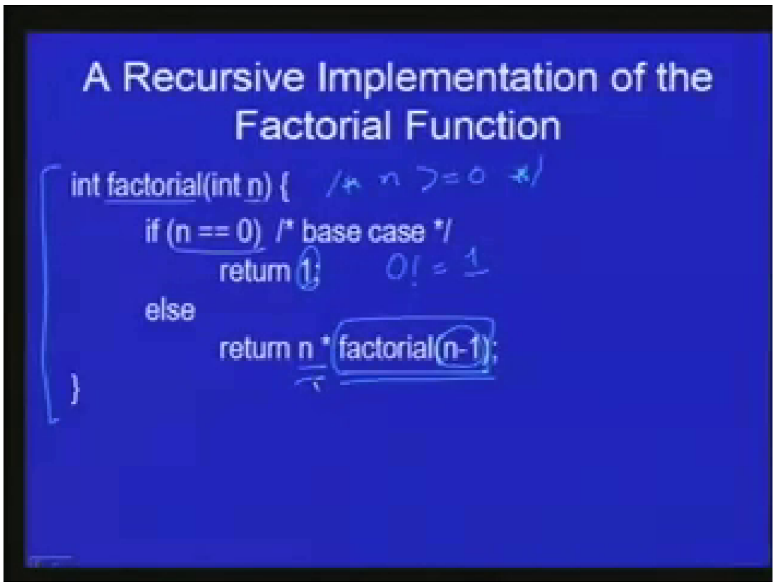
Using Recursion

- If the solution of the given problem instance can be computed using the solution to a simpler instance:
  - assume that the solution to the simpler instance is available by magic, and just use it.
  - call the same function to obtain the solution to the simpler instance.
  - do not forget to include a base case in the function definition.

Where is at the way we use the recursion is the solution to a given problem. Instance can be computed using this solution to a simpler instance of the same problem. Then essentially we assume that the solution to be simpler instance is available by magic. So, that is not really available by magic what to obtain that solution what do we needed to do is to call the same function recursively to obtain the function to a simpler instance and finally, of course, you must have a base case in the function definition where the function is not called recursively and the solution is directly given.

So, let see a recursive definition of the factorial function as that you already seen that it is in the factorial function can be defined non-recursively also without too much trouble, but the recursive definition would be (( )) let us why we are looking at it in many cases we will see that the recursive solution is much simpler then the non-recursive solution.

(Refer Slide Time: 06:12)



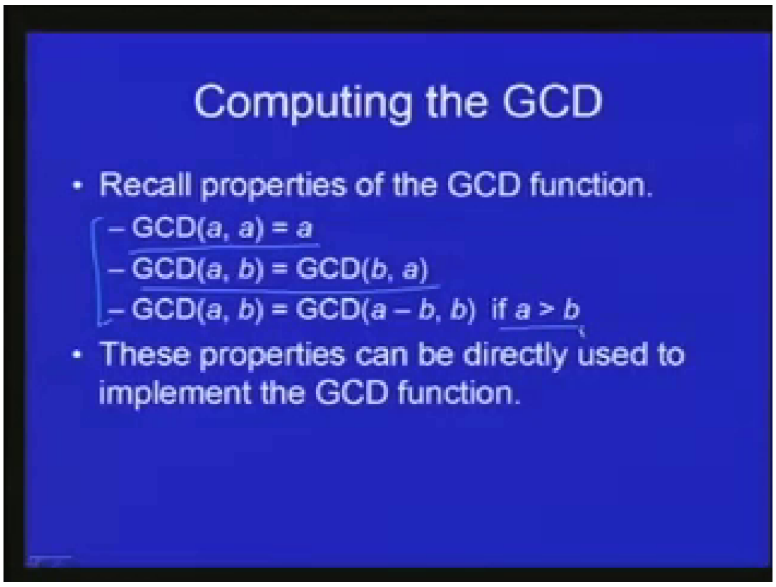
A Recursive Implementation of the Factorial Function

```
int factorial(int n) {   /* n >= 0 */
    if (n == 0)  /* base case */
        return 1;        0! = 1
    else
        return n * factorial(n-1);
}
```

So, here is the recursive definition of the factorial function. So, we are defining a function factorial if which the given n which is assume to be greater than or equal to 0 as an argument and return n factorial. So, first check whether n is equal to 0 or not because that is the base case. So, if n is equal to 0 then we return 1 because 0 factorial as we know is equal to 1, and otherwise if n is greater than 0 we return n in to factorial of n minus 1. Now this is the recursive call the factorial function is calling itself, but with the different parameter.

So, the parameter here is n minus 1 and so the factorial of n minus 1 is the simpler instance of the same problem and we are using the recursion to find the solution to the simpler instance and finally, using the answer to the problem we are computing the answer to the given instances. So, if the factorial of n minus 1 is known then we know this the factorial of n is nothing but n times t factorial of n minus 1.

(Refer Slide Time: 07:22)

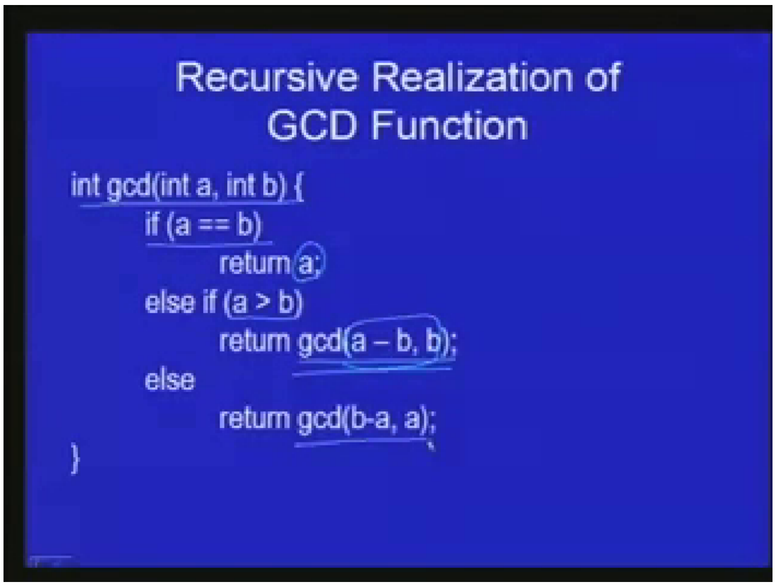


Computing the GCD

- Recall properties of the GCD function.
  - $GCD(a, a) = a$
  - $GCD(a, b) = GCD(b, a)$
  - $GCD(a, b) = GCD(a - b, b)$ if $a > b$
- These properties can be directly used to implement the GCD function.

Let us look at another problem where recursion can be applied, very easily. I called that we are talk early about the GCD function or the greatest common divisor function and I show you some properties of the CGD function and will see that using this property over the CGD function, the CGD function can be implemented very easily recursively. So, this are the property CGD of a and a is a. So, if the two numbers are equal then the CGD is equal to the same number. CGD of a and b is equal to CGD of b and a, so we can trap the two numbers if you wish. And finally, CGD of a and b is same as the CGD of a minus b and b if a is greater than b. So, if a is greater than b then you can subtract the smaller number from the larger number and this difference and this module number here CGD is the same as the CGD of the original two numbers. And as you are seen this property can be directly used to implement the CGD function.

(Refer Slide Time: 08:30)



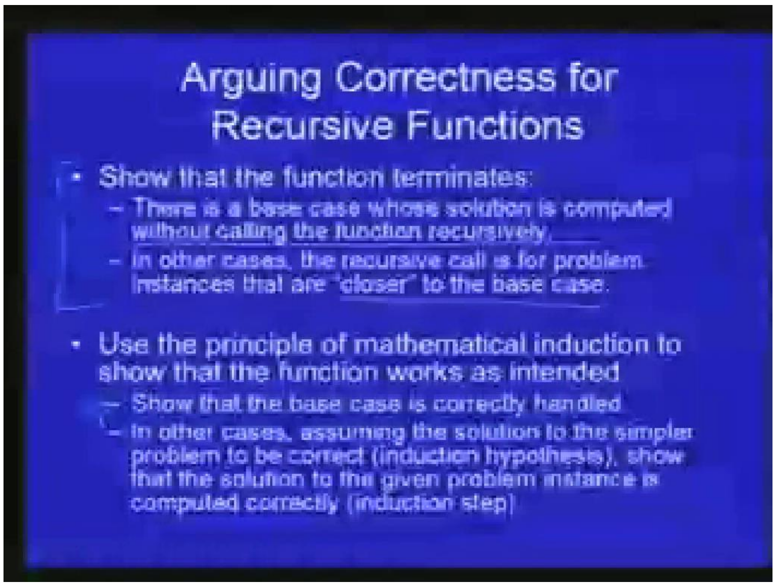
Recursive Realization of GCD Function

```
int gcd(int a, int b) {
      if (a == b)
            return a;
      else if (a > b)
            return gcd(a – b, b);
      else
            return gcd(b-a, a);
}
```

Here is the definition. So, is the CGD function that we are defining, there are two parameters a and b. This is the base case if a equal to b then we return a or return b since both are equal it does not not really matter otherwise if the a greater than b then we return CGD of a minus b comma b. So, this parameter a minus b and b defined the simpler instance of the problem in this case and given the solution to be this simpler instances, the answer to the given instances is to (( )) to compute. This is same as the as the solution to its simpler instances and if a happens to be less than b then we can mentally trap the two numbers and then apply the last property that we talk and that will result in CGD of a and b be equal to CGD of b minus a and a.

So, that is what we return in that case if a happens to be less than b its now having return a recursive function we need to be able to argue for convinced ourselves that the definition is indeed is correct. Now there are two aspects of proving the correctness of a recursive function of for that matter of any program and the first case to show.

(Refer Slide Time: 09:54)



Arguing Correctness for Recursive Functions

- Show that the function terminates:
  - There is a base case whose solution is computed without calling the function recursively.
  - In other cases, the recursive call is for problem instances that are "closer" to the base case.
- Use the principle of mathematical induction to show that the function works as intended
  - Show that the base case is correctly handled
  - In other cases, assuming the solution to the simpler problem to be correct (induction hypothesis), show that the solution to the given problem instance is computed correctly (induction step)

Let the function actually terminate or the program actually terminate. Now for a recursive function this means that we must show or argue that there is a base case whose solution is completely without calling the function recursively. So, for example, in the factorial function the base case are n equal to 1 or n equal to 0 in the case of CGD function with the base case function for when the two given numbers are equal.

So, we have to ensure that there is a base case in which recursive function is not called, and in the other cases when the recursive call is made then the recursive call is made for one or more problem instances that are closer to the base case. So, for example in the factorial function, the recursive call is made for n minus 1 factorial. Now n minus 1 is closer to the base case which is the n is equal to 0 then the given value n clearly. And so this process will finally, end.

So, we have to argue that the recursive call is been made for a problem instance or multiple problem instances that are all closer to the base cases then the given instances. So, once we have argue that the function actually terminates then what we need to show gives that whatever answer it does that that the function computes for a given instance for the problem is the correct answer. And we get to argue that for recursive function is very simple you use the well known and familiar to the principle of mathematical induction to show that in the base case the answer is computed correctly.

And in the other cases, what do we do in proving something by induction gives us to assume an induction hypothesis and then show in the induction steps that the solution to the given instances of the problem is correct. So, I been induction hypothesis you would assume that the solution to the simpler instances of the problem for which we are using the recursion. They result in the right answer and if the result is in the right answer then the induction step we need to show that the collision to the given instances of the problem is computed correctly. So, let us look at the factorial function and the CGD functions can again and try to apply this kind of arguments. So, for the factorial function are being made the function terminates the e v we can see that there is a base case is the base case.

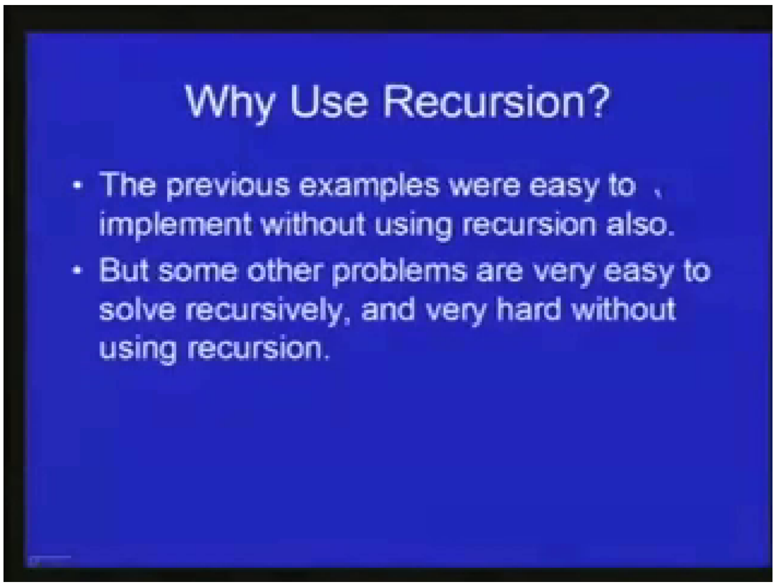
And in the other cases where so in the base case there is no recursive call within the function is made and in the other cases the call is made for a smaller value of n. And so which is which is closer to the base case then the given value of n, and therefore the function will certainly terminate and the other things that are needed to argue is that when the function terminates is return to the right value. Now again we use the principal of mathematical induction for n is equal to 0 which is the base case the answer is correct, because by definition n factorial 0 factorial is equal to 1 and of course, for other cases which is also correct because if we assume that the factorial n minus 1 is computed correctly then it is able to see that the factorial of n which is computed as n times factorial of n minus 1 is also computed correctly. So, this happens to be particularly simple example we are argue correctness is quite easy let us look at the CGD function next.

Now it is easy to argue that the result computed by the function is correct because we are directly using the property of the CGD function that we known to be correct. It is the little harder to argue that the function indeed terminate because. So, we need to show that when we make the recursive call the recursive call is for a problem instances which is closer to the base case. Now for the base case, which look at the magnitude of the difference of the two numbers given that is 0; because in the base case a is equal to b. In the other cases, you can see that the recursive call is made for a instances in which the absolute value of the difference of the two number actually is less than the distance of the two given number.

So, let say a is greater than b; if a is greater than b then the distance of the two numbers in this instance of this problem that we are calling recursively is a minus b minus b is equal to a minus 2 b and which is less than a minus b. right And so we have got we have gone closer to the base case because the difference between the two numbers has actually reduced. And similarly, if a happens to be less than b then the absolute value of the difference between the two numbers is b minus. It was originally b minus a and which is greater than the new distance which is b minus a minus minus a which is equal to b minus 2a.

And so again from the difference b minus a we have gone to the difference b minus 2a. So, the differences between the two numbers are absolute values differences between the two numbers are decreased. And for finally, the difference will become 0 because in each step it is certainly decreasing and therefore, finally it will become 0 in which case will have to reached the base case. So, that is how you argue that the correctness of recursive functions. We will see in the next lecture how the recursive function are work, but even without understanding (( )) function how the recursive works we should be able to use its principle to develop and argue the correctness of recursive function.

(Refer Slide Time: 16:22)

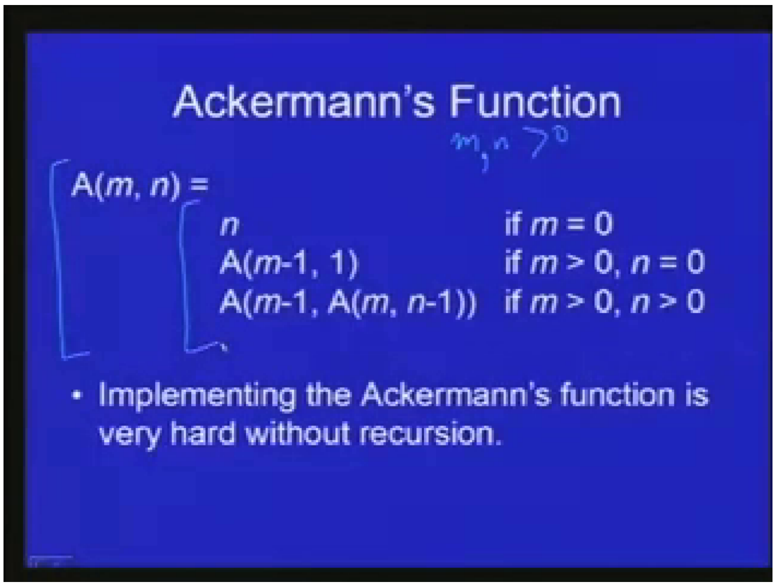


Why Use Recursion?

- The previous examples were easy to implement without using recursion also.
- But some other problems are very easy to solve recursively, and very hard without using recursion.

Now the question is why do you want to use recursion at all. So, as I said in in many cases, the problems are very difficult to solve without using the recursion, but are quite simple to solve by using recursion. The examples that we just saw the factorial functions and the CGD function these are not surely very good example of the recursion, because as we know these functions can be these functions can be implemented quite easily without using the recursion of also, but we will see some other problems which are very easy to solve recursively and comparatively harder to use comparing comparatively harder to solve without using the recursion.

(Refer Slide Time: 16:52)



Ackermann's Function

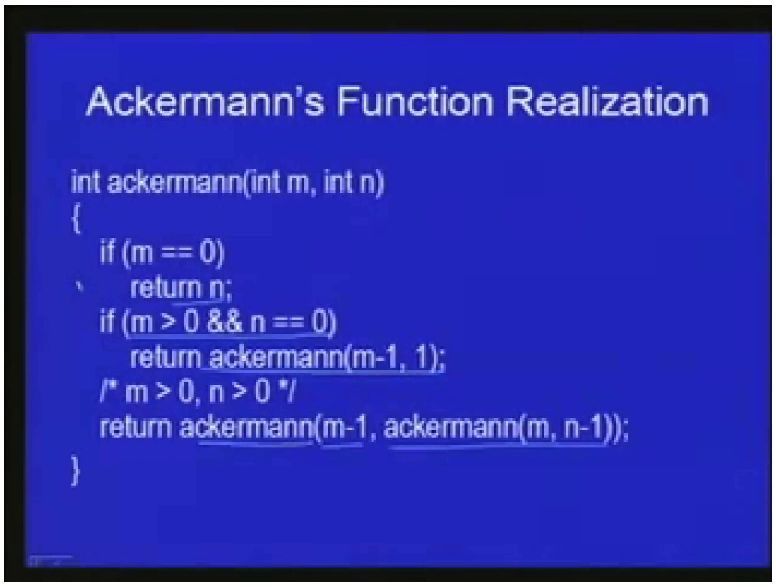
$m, n > 0$

$$A(m,n) = \begin{cases} n & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0, n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0, n > 0 \end{cases}$$

- Implementing the Ackermann's function is very hard without recursion.

So, let see an example. This is the famous Ackermann's function which is defined as follows. So, it is defined for two parameter m and n which are both greater than 0 and a of m n is equal to n; if m is equal to 0; if equal to a of m minus 1, 1; if m is greater than 0 and n is equal to 0 and if it is equal to a of m minus 1 comma a of m and n minus 1 if both m and n are greater than 0. So, as an exercise you can try to implement the Ackermann's function without using the recursion will find that there is extremely hard, but as you can see if we directly use the definition using the recursion the function is extremely easy to define.

(Refer Slide Time: 17:42)



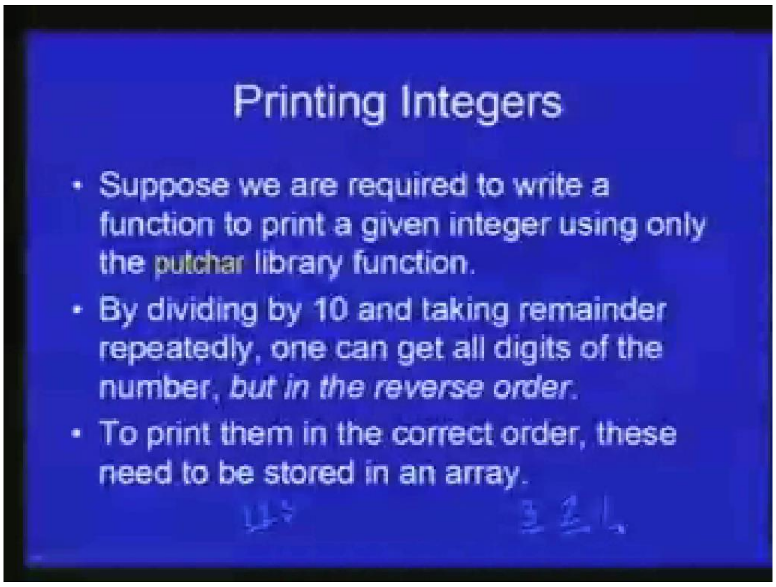
Ackermann's Function Realization

```
int ackermann(int m, int n)
{
   if (m == 0)
      return n;
   if (m > 0 && n == 0)
      return ackermann(m-1, 1);
   /* m > 0, n > 0 */
   return ackermann(m-1, ackermann(m, n-1));
}
```

So, here is the possible implementation for the Ackermann's function if m is equal to 0 return n if m is greater than 0 and n is equal to 0 return Ackermann's of m minus 1 comma 1 otherwise it return Ackermann's of m minus 1 comma Ackermann's of m comma n minus 1 as you can see that this. So, the implementation of the function follows directly from the definition of the function itself and within have to be hint at all how do we implement this. On the other hand, if we want to implement the non recursive solution to this problem that extremely hard even though in principle can be done.

So, let us now look at another example of using recursion. Suppose you did not have useful print f function to print integer for us, and let say we did not want to print an integer. And let say you assume only (( )) function for printing something on the screen is the putchar function which can print a single character given its ascii values.

(Refer Slide Time: 18:52)



Printing Integers

- Suppose we are required to write a function to print a given integer using only the putchar library function.
- By dividing by 10 and taking remainder repeatedly, one can get all digits of the number, *but in the reverse order.*
- To print them in the correct order, these need to be stored in an array.

[illegible] [illegible]

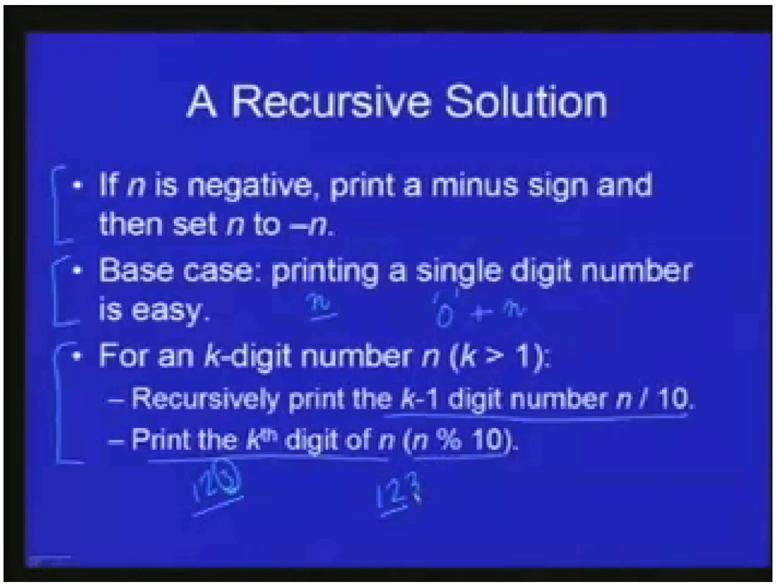
So, essentially is the problem is to write a function which given an integer n print certain integer on the screen using only the putchar library function. Now how do we go about printing an integer, you can print only one character at a time. So, what we need to do is to find all the digits of the number successively one by one, and then print them using the putchar function.

Now so the problem if the number is the single digit number. So, for example, if n is happened to be six, all we have to do is to call putchar with the ascii value of the character h. But if the number is larger than 10 then it is successively divide by 0 and in every step take the remainder we are repeatedly divide the number by 10 and every step take the remainder and if the 1 digit of the number and this process has to be repeated till the number becomes equal to 0.

Now this process gives you all the digits of the number, but in the reverse order. So, for the example if the number happen to be 123 then you divided first by 10 and take the remainder, you get 3, and then if you then the number remaining is 12 in then if you take again the remainder with ten then you get 1 then you get 2 and the number left is 1 and that of course, is less than 10. So, you get all the 3 digit, but in the reverse order. So, if you print them as soon as you.

Take the remainder you will get the wrong result because it each will print will be print in the reverse order and you want to print them in the correct order then what we need to do is to store this digit as you obtain in an array and the letter print the elements of the array in the reverse order. By using the recursion we can actually make this problem simpler. So, let us think about this problem recursively.

(Refer Slide Time: 20:48)



A Recursive Solution

- If n is negative, print a minus sign and then set n to $-n$.
- Base case: printing a single digit number is easy. n '0' + n
- For an k-digit number n ($k > 1$):
  - Recursively print the k-1 digit number n / 10.
  - Print the k^{th} digit of n (n % 10).

120 123

For negative integer, of course, are not difficult to handle. If n is negative it print it all we have do is to print minus sign the minus character and then set to set n to minus n. So, we do not have to worry too much about this negative number. Therefore, let assume that the number is greater than or equal to 0. Now the base case is when the number is the single digit number in the other words in other words n is less than 10. Now when n is less than ten which easy to print the number you can putchar. So, if the number is n then what we need to do is to find the ascii values of the character of the character n.

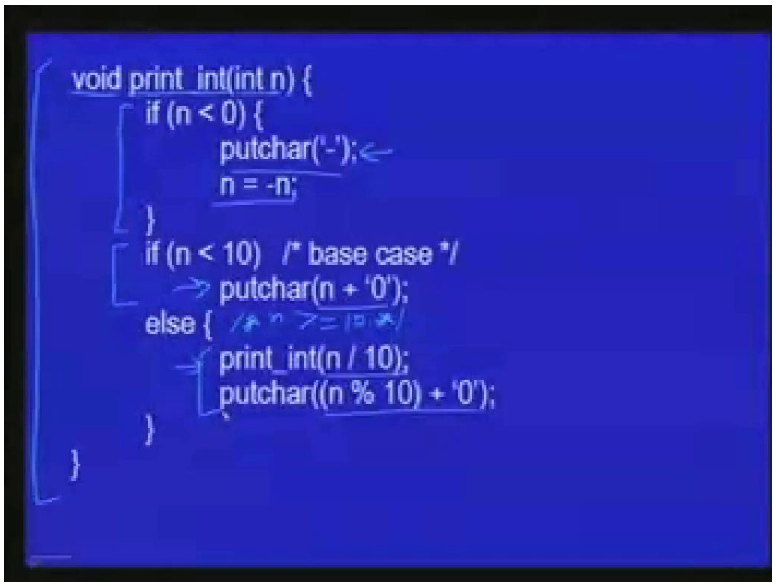
Now how we do that let us quite easy if you take the ascii value of 0 and add n to it that will give the ascii value of the character n. So, I (( )) of course, to be less than 10, so for example, if n happen to be 6 then the ascii value of 0 plus the integer 6 will give you the ascii value of the character 6 and that is because if you recall the ascii value of all digits from 0 to 9 or contiguous to each other and it is in the increasing order.

So, the ascii value of 6 is actually 6 more than the ascii value of the character 0 and so on. So, in the base case printing the single digit number is easy, now suppose we have given a number which is not a single digit number. Let us assume, if k digit number n where k is less is greater than 1, now how do we go about printing this number well we can see that we need to first print the value n by 10, now how you print n by 10?

n by 10 in general n by 10 is always been a k minus 1 digit number and k minus 1 will also be more than 1. So, how do you print this k minus 1 digit number? When we use recursion to solve this simpler instance of the same problem, this is the simpler instance of the same problem because the number of digits have reduce by 1, and the base cases when the number of digits is left 1. So, having printed recursively the number n by 10 then we print the last digit of n which is nothing but n percent 10 - that is the remainder left when n is divide by 0.

So let say again the number happen to be 123. So, let us assume that n by 10 can somehow be print it. So, n by 10 is in this case is nothing but 12, and by making the recursive call to the same function. Let assume the number 12 - that is printed somehow and once that is done always to be not sense the last digit of the number and it is nothing but n percent 10 and you know how to print the single digit since we can do that.

(Refer Slide Time: 23:50)



```
void print_int(int n) {
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n < 10)   /* base case */
        putchar(n + '0');
    else {   /* n >= 10 */
        print_int(n / 10);
        putchar((n % 10) + '0');
    }
}
```

So, let us look at the implementation of the function now. So, here is the implementation. So, I am calling the function print underscore int the return type is void because it does not return any value it gives the print the given integer n and n is the parameter which is the even integer which is need to printed. So, if n is less than 0 then you print minus sign using the putchar function and set to minus n. So that is (( )).

Now this is the base case. If n is less than 10 then we know what to do all of you do is to compute the ascii value of the digits that represent 10 and print that using the putchar. So that is what we do here otherwise n is greater than equal to 10 which means that n has at least two digit. So, we recursively print the number n by 10 which is guaranteed to have one digit less and the original number n and finally, after doing that this thing the last we are obtain which is nothing but n percent 10. So we obtaining the ascii value by adding the ascii value of 0 to it and printed.

So, again you can argue the correctness and the termination property of the function in which similar to what we have argued earlier we can see the function terminates, because every time when you make recursive call the number of digits in the number decreases by one strictly decreases by one. And finally the number of digits in the number will reduces to one in which case the number is printed without calling the function recursively and of course, you know that to print the k digit number all we do that is first print, the first k minus 1 digit which are represented by n by 10 and then print the k eth digit. So, assuming that the recursive call to print the k minus 1 digit the number works correctly it is to see that the given k digit number will also be printed correctly. So, we stop the lecture here today and in the next lecture we talk about how recursion actually works and that we give us the better inside also in to how to use recursion in our problem solving exercise.