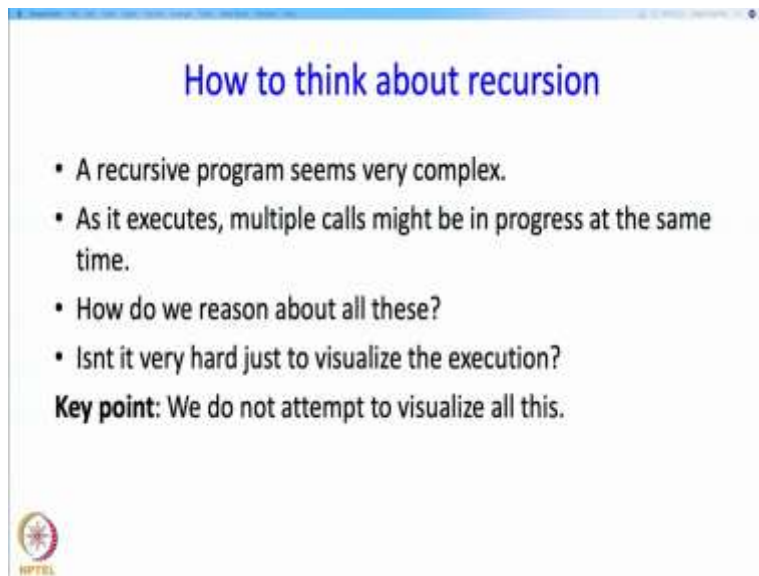


**An Introduction to Programming through C++**  
**Professor Abhiram G. Ranade**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology Bombay**  
**Lecture No. 11 Part - 3**  
**Recursion**  
**How to think about recursion**

Welcome back, in the previous segment we discussed recursive objects. In particular we discussed trees and how to draw them. And for this we used a recursive function. In this segment, we want to say something about how to think about recursion. And after that we will conclude this (segment of) sequence of segments.


(Refer Slide Time: 0:53)



**How to think about recursion**

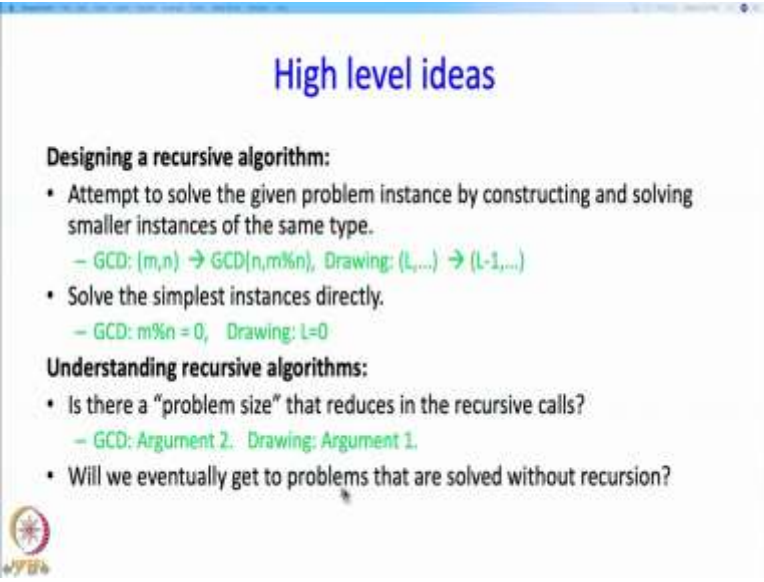
- A recursive program seems very complex.
- As it executes, multiple calls might be in progress at the same time.
- How do we reason about all these?
- Isn't it very hard just to visualize the execution?

**Key point:** We do not attempt to visualize all this.



How do you think about recursion? On the face of it recursive a program seems very complex when it executes typically multiple calls are in progress. And how do we reason about all of them? How do we even visualize what is going on? So the good news is that we do not need to do any such visualization, there is a much more direct simple way of thinking about recursive algorithms.

(Refer Slide Time: 1:20)




**High level ideas**

**Designing a recursive algorithm:**

- Attempt to solve the given problem instance by constructing and solving smaller instances of the same type.
  - GCD:  $(m,n) \rightarrow \text{GCD}(n,m\%n)$ , Drawing:  $(L,\dots) \rightarrow (L-1,\dots)$
- Solve the simplest instances directly.
  - GCD:  $m\%n = 0$ , Drawing:  $L=0$

**Understanding recursive algorithms:**

- Is there a “problem size” that reduces in the recursive calls?
  - GCD: Argument 2. Drawing: Argument 1.
- Will we eventually get to problems that are solved without recursion?



So, let me begin with some high level ideas. And let me first consider the case when you are trying to design the recursive algorithm. Now, you may think that designing recursive algorithms is tricky, and indeed Euclid's algorithm is pretty tricky. However, if you are dealing with recursive objects, then recursive algorithms are quite natural. And you may well find yourself designing such algorithms.

So, how do you do this? Well, you first attempt to solve the problem given to you by constructing and solving a smaller problem of the same type or what you might more accurately call a smaller instance of the same type. And I want to observe that this is exactly what we did for GCD as well as for drawing. So, when we wanted to solve, when we wanted to find the GCD of M and N we instead requested to find the GCD of N and M mod N and why did we do this? Because, these numbers in fact smaller than the given numbers. So, in some sense, we are reducing, we are simplifying the problem or we are reducing its size. So when I say size, I mean it in a metaphorical sense. So, we are reducing its size in some sense.

Even in drawing, when we were when we were asked to draw a tree which had L levels, we reduced it and we said oh let us first draw trees which have just L minus 1 levels. So, again this is a simpler problem in the sense this is a smaller tree that we are going to draw, it has fewer number of levels. So, if you can do this ok. So, if you can say that I want to solve this big problem, but I want I can break it up into problems which are of the same type, but smaller in

some sense, then you have already, already taken the first solid step towards designing the recursive algorithm. And then you should be able to solve the simplest instances directly. So, that is the second step, ok. So, for example, in the case of GCD this was the simple instance if  $M$  was divisible by  $N$ , then you can directly say that  $N$  is the GCD or in the case of drawing if you wanted to draw a tree with 0 levels, then you can directly do it, well actually in this case, that is nothing to be done. But that is trivially doing it.

Now, I want to say something about how do you think of a recursive algorithm that is given to you by somebody? How do you understand it? So, the first step is sort of the analog of this step over here. So, can you sort of interpret or is the designer telling you that look, this is the problem size that is reducing in successive calls. So, that is the first question. And let me make this discussion more precise. So, for the GCD, you can argue that this second argument is always reducing, as you execute the program. It does not have to be the second argument. It could be some complicated function of the arguments in general, but in this case it happens to be the second argument. And for drawing, it is in fact the very first argument. So, here it was  $L$  then we said that this tree could be drawn by drawing several trees with  $L$  minus 1 levels. So in fact, here you have you can think of the second argument and the first argument as the problem size. So, if I give you a recursive algorithm, then you should look for a problem size in it, something that is reducing as you do the recursion. And then the question that you should ask is, will be eventually get to problem sizes to problems that are solved without recursion?


(Refer Slide Time: 5:45)

**Some terminology**

**Top level recursive call:**  
The call made from the main program, or the first call made to the recursive function.  
• gcd: the call  $\text{gcd}(m, n)$       Drawing: the call  $\text{tree}(L, rx, ry, H, W)$

**Level 1 recursive calls**  
Calls made directly while executing the top level call.  
• gcd:  $\text{gcd}(n, m\%n)$   
• Drawing:  
     $\text{tree}(L-1, rx-W/4, ry-H/L, H-H/L, W/2)$   
     $\text{tree}(L-1, rx+W/4, ry-H/L, H-H/L, W/2)$

**Base cases:**  
Input values for which the top level call returns without recursing.  
• gcd:  $m, n$  such that  $m\%n == 0$       Drawing:  $L = 0$

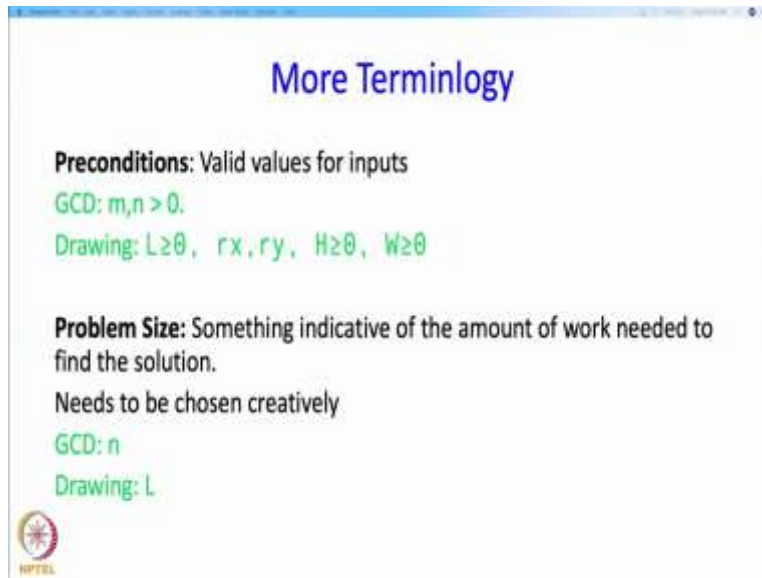


All right, so now we are going to look at these things in a little bit more detail. And so, for that, I will need some terminology. So I am going to define something called a top level recursive call. So this is the call made from the main program or the first call made to the recursive function. So again taking examples, for the case of GCD the top level call is the GCD of M, N or this is in fact you can think of this as identified by the signature of the function, and for the drawing it is this call, so this would be the call that could be that in the signature of the function, then there are other level one recursive call, so, these are the calls which are made directly while executing the top level call. So for GCD the next call was GCD of N and M mod N. So this is the level 1 recursive call for this top level call and for drawing, for drawing this tree, the recursive call was this and also this, so these are the two level 1 recursive calls, ok. So, this is just notation could define level 2 recursive calls in a similar manner, but I do not need to. So, the whole point of this discussion is that we can just think about one level of recursion to understand what is going on.

Then there is a notion of a base case. So, these are simply the input values for which the top level call returns without recursing. So in the case of GCD if M and N are such that M is divided perfectly by N then that is a base case, because in this case you can return N directly as your GCD. So M, N such that M mod N is 0 are the base cases for the GCD, GCD function. Then for drawing similarly, if L 0 then you are not going to recurse, you are just going to return in this

case without doing anything but that is that is how you that you are going to return. So L equal to 0 is the base case; L equal to 0 and other arguments, whatever they might be.


(Refer Slide Time: 8:07)



**More Terminology**

**Preconditions:** Valid values for inputs  
GCD:  $m, n > 0$ .  
Drawing:  $L \geq 0$ ,  $r_x, r_y$ ,  $H \geq 0$ ,  $W \geq 0$

**Problem Size:** Something indicative of the amount of work needed to find the solution.  
Needs to be chosen creatively  
GCD:  $n$   
Drawing:  $L$



Some more terminology. So we have defined this earlier actually, but I am just going to restate it. So for every for every problem for every call really, there is a notion of a precondition. So this simply says what are valid values for the inputs. So in the case of GCD, the two arguments have to be 0 only then this is a reasonable call, only then is the GCD define.

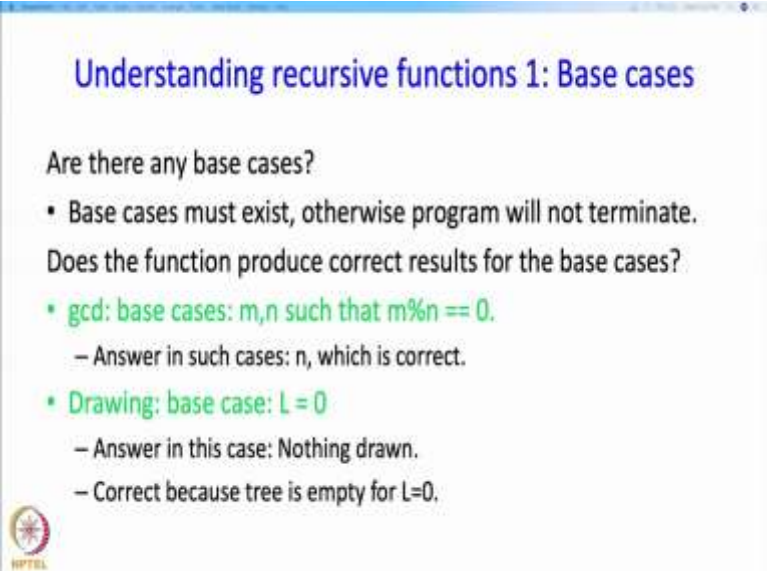
And similarly for drawing, the number of levels must be greater than or equal to 0. And the height and the width have to be greater than 0, greater than or equal to 0. And RX and RY, the coordinate where, the position where, the route is to be placed, well they could be negative, ok. So, that might just force the drawing of the screen. Or if you want it inside a screen then you should say ok I want I want the tree to be inside the screen. So, I will require these also to be positive and in fact maybe smaller than the size the so, they should be smaller than the width of the screen, they should be smaller than the height of the screen, ok. But we are just ignoring those conditions for now.

Then, there is a problem size and we gave an example of this, but let me just say a little bit more. So this is something which is indicative of the amount of work needed to find a solution. Or this is something which suggests how difficult or how complex this problem is. So we said that the number of levels is indicative of how difficult the problem is - the more the levels, the more

work period and similarly, for GCD the values of these numbers, the magnitude of these numbers are indicative of the problem size, but more specifically, we looked at the second argument this N. So, this we said was the problem it could be considered the problem size.

And they should be chosen creatively. And as I said, this is how you might choose. So there is no, there is no obvious way of what, what it should be, but as a designer you would have to say that look, I want to reduce this, because if I reduce this then I can I can recurse.

(Refer Slide Time: 10:30)




**Understanding recursive functions 1: Base cases**

Are there any base cases?

- Base cases must exist, otherwise program will not terminate.

Does the function produce correct results for the base cases?

- gcd: base cases:  $m, n$  such that  $m \% n == 0$ .
  - Answer in such cases:  $n$ , which is correct.
- Drawing: base case:  $L = 0$ 
  - Answer in this case: Nothing drawn.
  - Correct because tree is empty for  $L=0$ .

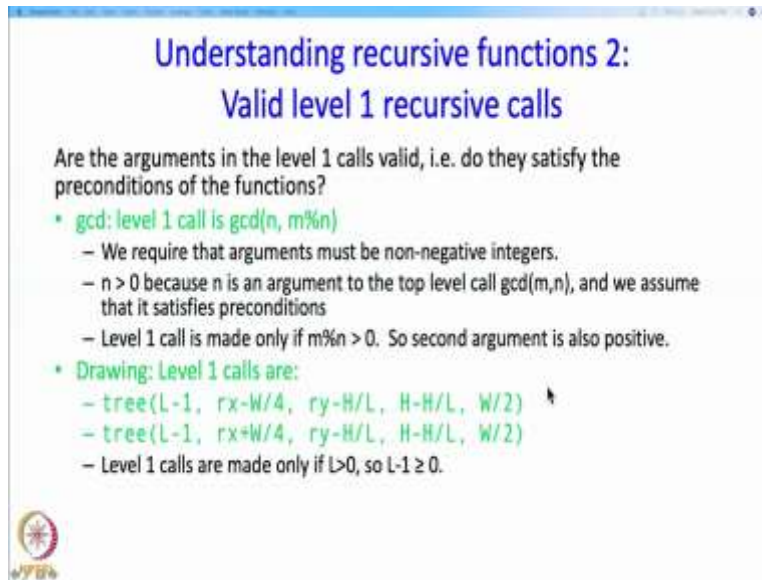


So now, I want to say what you need to do to verify that a given recursive function is a reasonable function. So the first thing that you need to check is whether there are any base cases. So if there are no base cases, then clearly your program will not terminate and therefore, base cases must be there. And then, for those base cases, function must produce the correct results. So, that is one check that you have to make. So, for the GCD, the base cases were values of M, N such that  $M \bmod N$  equal to 0. So in this case, the answer was N, is that correct? Well, yes, if M is divisible by N, then N must be the GCD. So, which is being returned and so the program is working correctly for the base case base cases.

For drawing, the base case was L equal to 0 or maybe I should write  $L == 0$  over here. But in this case, what did the program do? It did nothing, ok. Was that correct? Well, if the tree has 0 levels, then it is an empty tree and so nothing should be drawn. So even here, for both of these

programs, this is how you would check that the base cases that there are base cases and they are being correctly handled.

(Refer Slide Time: 12:02)



The slide is titled "Understanding recursive functions 2: Valid level 1 recursive calls". It asks, "Are the arguments in the level 1 calls valid, i.e. do they satisfy the preconditions of the functions?". It then lists two examples:

- **gcd: level 1 call is  $\text{gcd}(n, m\%n)$** 
  - We require that arguments must be non-negative integers.
  - $n > 0$  because  $n$  is an argument to the top level call  $\text{gcd}(m,n)$ , and we assume that it satisfies preconditions
  - Level 1 call is made only if  $m\%n > 0$ . So second argument is also positive.
- **Drawing: Level 1 calls are:**
  - $\text{tree}(L-1, rx-W/4, ry-H/L, H-H/L, W/2)$
  - $\text{tree}(L-1, rx+W/4, ry-H/L, H-H/L, W/2)$
  - Level 1 calls are made only if  $L > 0$ , so  $L-1 \geq 0$ .

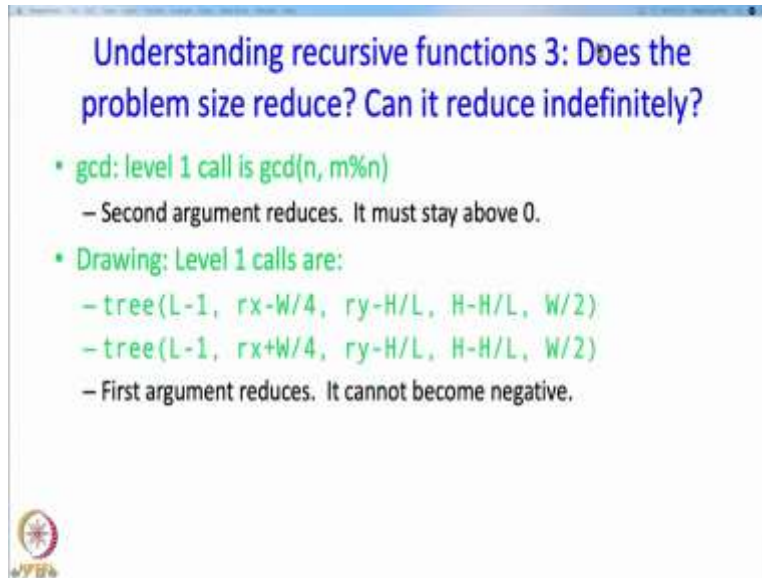
Then we turn our attention to the level 1 recursive call. So, first thing we should check is whether they are valid? So, are the arguments in the level one calls valid? Or in other words, do they satisfy the preconditions of the functions? So, in the GCD the top level call was GCD of M and N and the level 1 call is GCD of N and M mod N. So, what are the preconditions here? So, we require that the argument must be non-negative integers, so these must be non-negative integer.

So, can we argue that these are non-negative integers? Yes, N must be bigger than 0. Why? Because N is an argument to the top level call also. And the top level call we are assuming satisfies the preconditions. So assuming the top level calls satisfies the preconditions the level 1 call must satisfy the precondition. So, which is what we can argue over here. And what about the second argument? Well, if you go back to the code you will see that this level 1 call is made only if M mod N is bigger than 0, because if it is equal to 0 then we just return N, so only if M mod N is bigger than 0, then we make this call and therefore, this argument must also be positive. So, in other words, we have verified that the preconditions are correct.

What about drawing? The level 1 calls are these - so the original call, the top level call was had L levels, here we have L minus 1 levels. So, are these are these arguments satisfying the preconditions? Ok, well level one calls are made only if L greater than 0. So if L is greater than 0

than  $L - 1$  must clearly be greater than or equal to 0 or in other words  $L$  must be bigger than or equal to 1. So, so, this argument must also be bigger than 0. So, we have checked that this argument must be bigger than or equal to 0 sorry, bigger than or equal to 0. And you can check and I am not going to go through it right now, but you can check that these other arguments will also satisfy the required preconditions.

(Refer Slide Time: 14:35)



Understanding recursive functions 3: Does the problem size reduce? Can it reduce indefinitely?

- gcd: level 1 call is  $\text{gcd}(n, m\%n)$ 
  - Second argument reduces. It must stay above 0.
- Drawing: Level 1 calls are:
  - $\text{tree}(L-1, rx-W/4, ry-H/L, H-H/L, W/2)$
  - $\text{tree}(L-1, rx+W/4, ry-H/L, H-H/L, W/2)$
  - First argument reduces. It cannot become negative.

Then, we come to the next check that you should make. So, the next check is that does the problem size reduce and can it reduce indefinitely? If it cannot, if it can reduce indefinitely there is a problem, if the problem size does not reduce there is a problem. So, the answer to this first question should be yes, the problem size does reduce, but can you reduce indefinitely? No, it should not, it should not be possible to reduce the problem size indefinitely.

Alright, so the GCD level 1 call is  $\text{GCD } N, M \bmod N$  and top level call was  $\text{GCD } M$  and  $N$ . And we said that this second argument is the problem size So, second argument has reduced, why? Because originally it was  $M, N$  so it was  $N$  and now it has become  $M \bmod N$ , and we said that this was the problem size and  $M \bmod N$  is certainly smaller than  $N$ , so the second argument has reduced. So, that is a good thing.

Now, can it reduce indefinitely? Well, the remainder has to stay above 0. So it means that it, it cannot reduce indefinitely, ok. So this essentially saying that look, you cannot have unlimited indefinite recursion, you cannot just have one call after another because in every call, this

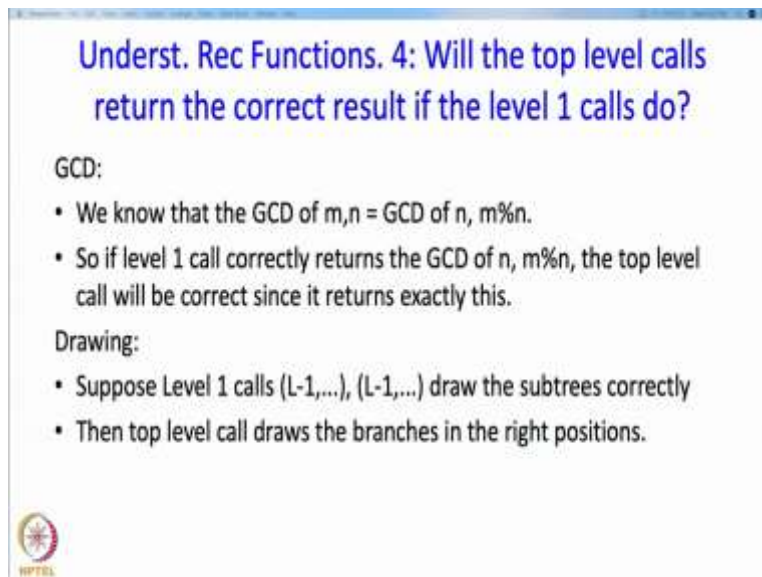


argument is going to reduce. But if it reduces, then eventually it is going to go down to 0 which is not possible. So, the answer to this question is indeed yes and no for GCD, so that is a good sign.

Drawing: So, the level one calls are these ok. And we said that the first argument was the problem size. So has that reduced over here? Well, in the original or in the top level call it was  $L$  and it has become  $L$  minus 1. So it has indeed reduced, can it reduce indefinitely? Well, it cannot become negative, because we are checking if  $L$  equal to 0 then we do not make a recursive call.

And so this cannot become negative. And therefore, again, we have checked that this function is a good function. The problem size is reducing but it cannot reduce indefinitely.

(Refer Slide Time: 17:18)




**Underst. Rec Functions. 4: Will the top level calls return the correct result if the level 1 calls do?**

**GCD:**

- We know that the GCD of  $m,n = \text{GCD of } n, m\%n$ .
- So if level 1 call correctly returns the GCD of  $n, m\%n$ , the top level call will be correct since it returns exactly this.

**Drawing:**

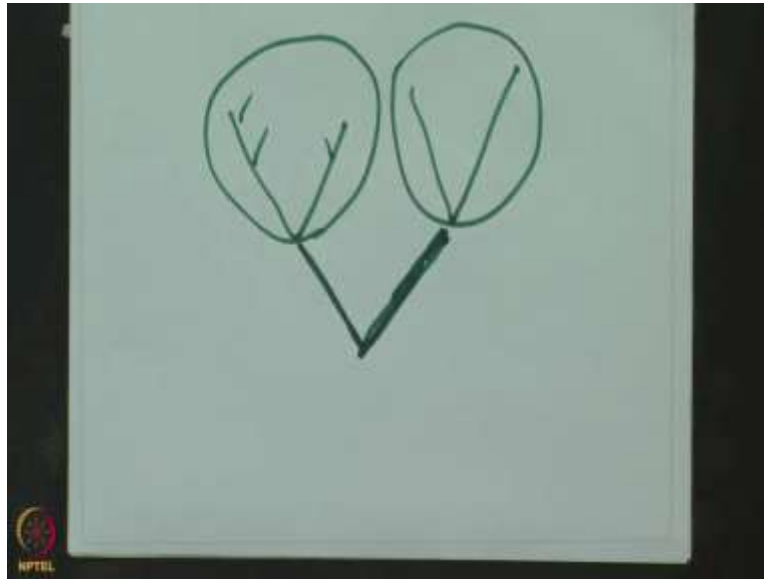
- Suppose Level 1 calls  $(L-1, \dots), (L-1, \dots)$  draw the subtrees correctly
- Then top level call draws the branches in the right positions.



The last requirement to check is will the top level calls returned the correct results if level 1 calls do? So GCD, we know that the GCD of the arguments to the top level call  $M$  and  $N$  is the same as the GCD of the top level calls to the level 1, the GCD of the, the level one recursive call which are  $N$  and  $M \bmod N$ . So the GCD of the arguments to the level 1 recursive call is the same as the GCD of the arguments to the top level call. And now what does the function do? So the function is going to return the GCD  $N M \bmod N$ , ok, if it is, if it does this correctly, ok, then the top level call will be correct. So, we are not checking whether it is going to correctly return this but it is returning this, ok. If this is correct then this will also be correct because they are exactly the same thing. So, we have checked this for GCD.

In the case of driving the level 1 calls ask for a small smaller trees to be two smaller trees to be drawn. Ok. And we had to check that the two smaller trees are drawn in the right position. So, what happened there? So yeah, so, so we asked we wanted to draw a level L tree and we said that oh if you want to draw such a level L tree, then we can do that by doing by doing these small L minus 1 level trees, ok. So, if those were correct, then our program or this, this function simply drew the branches in the right position.

(Refer Slide Time: 19:22)



So, what it was doing was so, we wanted to draw a tree starting from this point. So it said look, you first draw two smaller trees over here. And then in the top level call, you just do this we just drew these branches. So, if these trees got drawn correctly, then we know that oh there, there are just these two branches to be drawn and the top level call is drawing them correctly. So therefore, the total thing must be correct.


(Refer Slide Time: 19:55)

## Summary

To check if a recursive function is correct we should check

1. There are base cases and correct results are obtained for the base cases.
2. The level 1 recursive calls satisfy the preconditions.
3. The problem size reduces but cannot reduce indefinitely.
4. If the level 1 calls work correctly, the top level call will work correctly.

- We do not need to argue that the level 1 calls work correctly.
- We don't even need to think about calls made by level 1 calls.
- 1,2,3 ensure that the computation will terminate eventually.
- 4 ensures that the correct result will be returned.

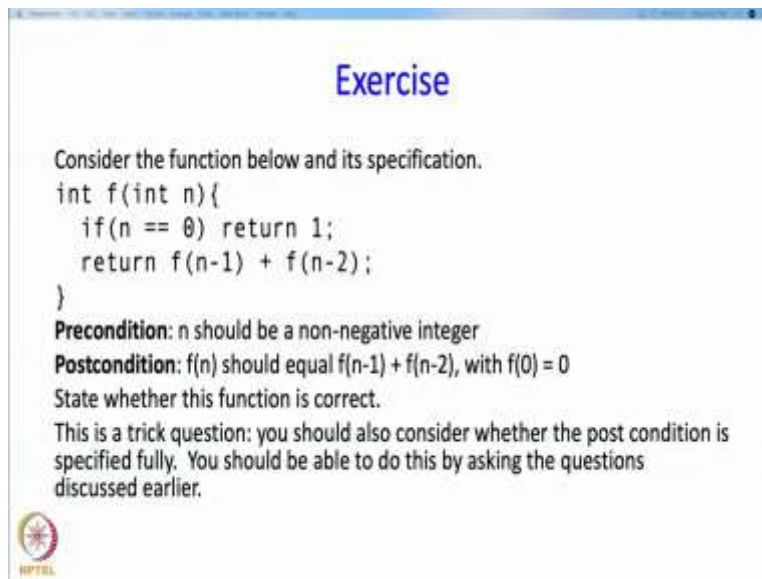


Alright, so let me summarize this part. So how do we check if a recursive function is correct? So, what should we check in order to do that? First we should check that there are base cases and correct, correct results are obtained for the base cases, then we should check that the level 1 recursive calls satisfy the preconditions, then we should check that the problem size reduces, ok, that there is such something called the problem size and that it reduces, but the code is such that it will not it is it is not possible that it can reduce indefinitely. And finally, we should check that if the level 1 calls work correctly, then the top level call will also work correctly that is it.

In particular, what is it that we do not need to argue? So, we do not need to argue explicitly that the level 1 calls, in fact work correctly. Ok? We do not need to do this. We do not even need to think of what the level 1 calls do. Are they going to make recursive calls? Are they just going to return the result directly? We do not have to worry about it. So, basically what is going on is that these first three points are ensuring for us that this computation is going to terminate eventually.

And this last point is going to say that look, if the recursive calls were correct, then we will return the correct answer. Ok. And this really this whole thing together says that effectively, not only does the top level recursive calls call works correctly, but it effectively proves that the level 1 call works correctly level 2 call works correctly, because this argument really is applicable to all the calls, ok? And we do not really but we do not really have to worry about that we do not have to think about all the calls, ok.

(Refer Slide Time: 22:15)



**Exercise**


Consider the function below and its specification.

```
int f(int n){
    if(n == 0) return 1;
    return f(n-1) + f(n-2);
}
```

**Precondition:** n should be a non-negative integer  
**Postcondition:** f(n) should equal f(n-1) + f(n-2), with f(0) = 0

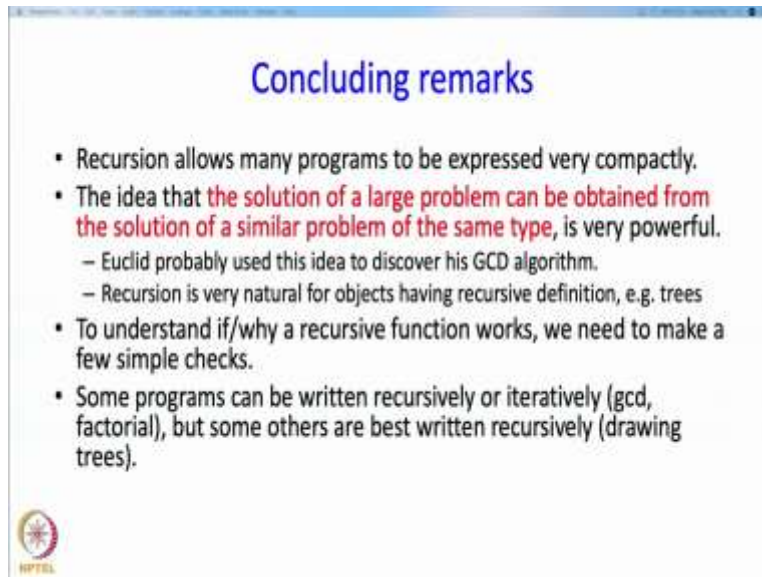
State whether this function is correct.

This is a trick question: you should also consider whether the post condition is specified fully. You should be able to do this by asking the questions discussed earlier.




Alright, here is an exercise for you. So, I give you this function, and there are it has its precondition and the post condition that is what it is supposed to be returning, ok. So, I want you to tell me whether this function is correct, ok, or whether there is a problem over here. Now, you should apply what we have discussed in this segment and tell me where things go wrong, ok. This is a bit of a trick question. But the questions the strategy we decided enable you to say what is wrong, to figure out something is wrong and say what is wrong.

(Refer Slide Time: 23:10)



### Concluding remarks

- Recursion allows many programs to be expressed very compactly.
- The idea that **the solution of a large problem can be obtained from the solution of a similar problem of the same type**, is very powerful.
  - Euclid probably used this idea to discover his GCD algorithm.
  - Recursion is very natural for objects having recursive definition, e.g. trees
- To understand if/why a recursive function works, we need to make a few simple checks.
- Some programs can be written recursively or iteratively (gcd, factorial), but some others are best written recursively (drawing trees).



That brings us to the end of this lecture segment as well as the sequence of segments. So, I want to make some concluding remarks about recursion in general. First, I want to observe that recursion allows many programs to be expressed very compactly. For example, GCD recursive GCD is certainly more compact than iterative GCD. The idea, that the solution for large problem or a problem whose problem size is large can be obtained from the solution of a similar problem, a similar smaller problem of the same type. I should really have the word smaller problem here. This idea is very powerful, ok, it is.

Euclid probably use this idea to discover his GCD algorithm. He probably did say it himself that look, I want the GCD of these large numbers. Is it sufficient if I find the GCD of some smaller numbers instead? But, the moment you take this single step you can, you are implicitly developing a capability of applying it again and again. And that is exactly what recursion gives you.

And recursion is also very natural, in fact, much easier to understand and much more natural, much less clever for objects which have a recursive definition, say for example, trees. And here we saw tree drawing as an example. To understand, why a recursive function works, we need to make a few simple checks outlined earlier and I just want to observe that some programs can be written recursively and iteratively.

So for example, GCD or the factorial, but some others are best written recursively. So for example drawing trees, it will be pretty complicated to write this program using iteration or rather I should say write this program without recursion. So, that concludes our discussion of recursion. It concludes this sequence of segments. Thank you.