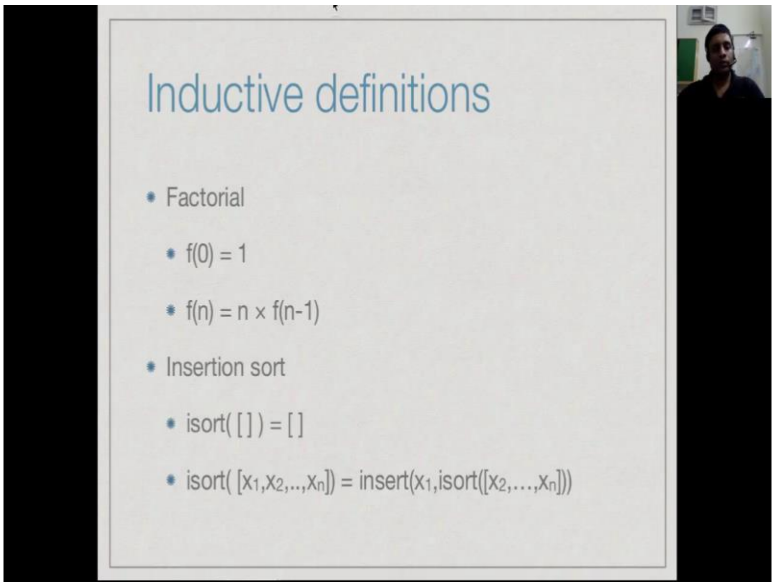
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Module – 02
Lecture - 45
Memoization

Let us continue our discussion of inductive definitions.

(Refer Slide Time: 00:05)

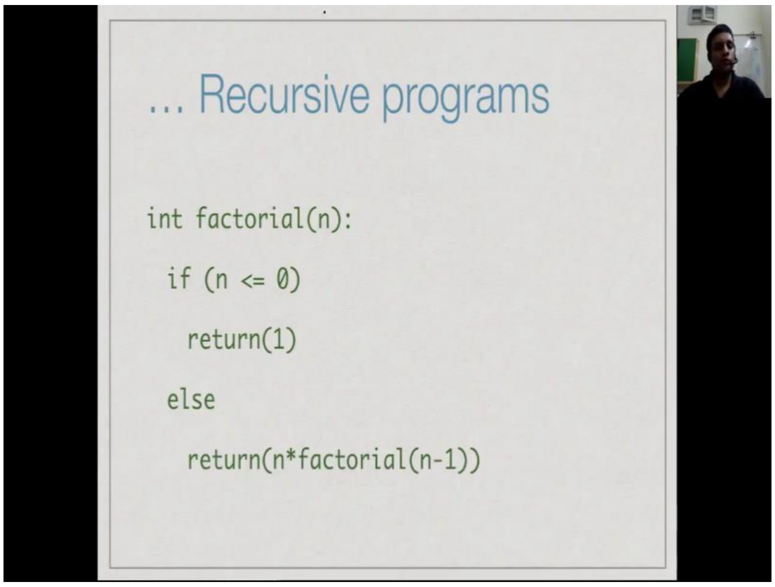


Inductive definitions

- Factorial
  - $f(0) = 1$
  - $f(n) = n \times f(n-1)$
- Insertion sort
  - isort([ ]) = [ ]
  - isort($[x_1, x_2, .., x_n]$) = insert(x_1, isort($[x_2, \ldots, x_n]$))

So, recall that functions like factorial and insertion sort are natural inductive definitions in terms of smaller sub problems.

(Refer Slide Time: 00:16)

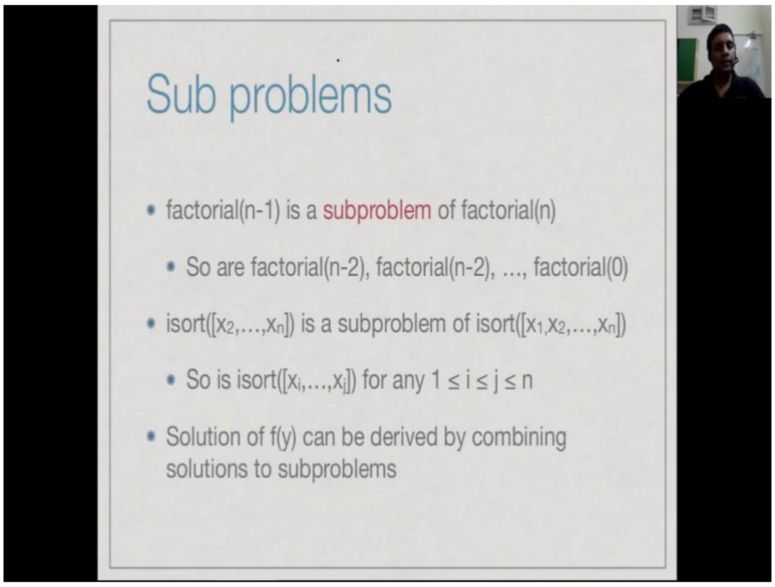


… Recursive programs

```
int factorial(n):
  if (n <= 0)
    return(1)
  else
    return(n*factorial(n-1))
```

And the attraction of looking at inductive definitions is that, we can very easily come up with recursive programs that implement them in a very obvious way.

(Refer Slide Time: 00:25)

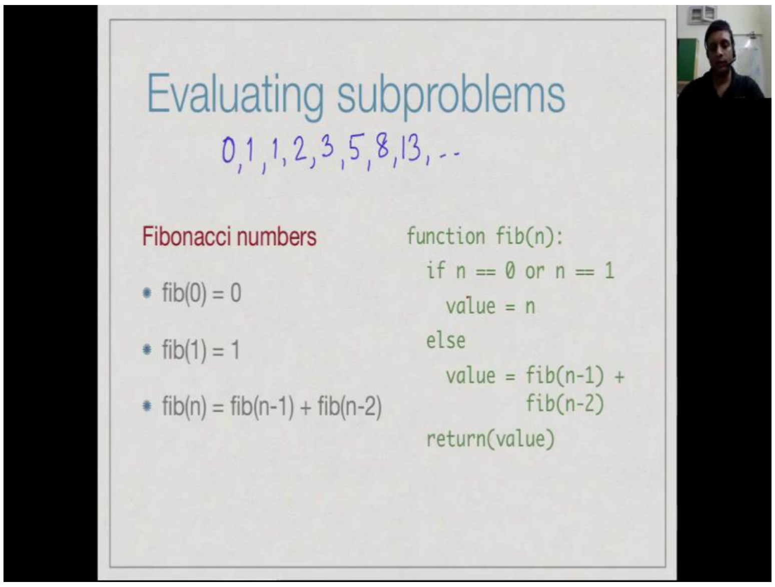


Sub problems

- factorial(n-1) is a subproblem of factorial(n)
  - So are factorial(n-2), factorial(n-2), ..., factorial(0)
- isort([x_2,…,x_n]) is a subproblem of isort([x_1,x_2,…,x_n])
  - So is isort([x_i,…,x_j]) for any $1 \leq i \leq j \leq n$
- Solution of f(y) can be derived by combining solutions to subproblems

But, the challenges is, in identifying the sub problems, I am making sure that they are not overlapped. So, in the case of factorial remember that any smaller input factorial is a sub problem, similarly for sorting you can think of any segment of the list to be sorted as a sub problem. And in general, we are looking at recursive solutions or inductive solutions where the solution to the function f we are trying to define is derived, by combining solutions to the sub problems of this original problem and so we compute f on smaller inputs than y.

(Refer Slide Time: 01:00)



Evaluating subproblems

0, 1, 1, 2, 3, 5, 8, 13, . .

Fibonacci numbers

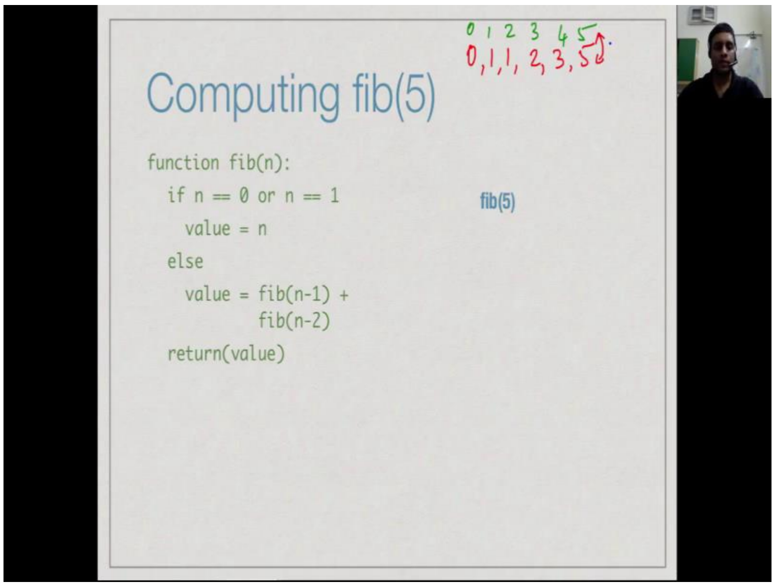
- fib(0) = 0
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

So, let us see how this works with the very familiar series that you may know of by Fibonacci numbers. So, let us define the Fibonacci numbers as follows, the first two Fibonacci numbers are 0 and 1 and then every successive Fibonacci number is obtained by adding these two. Now, it is instructed to see even before you begin how we would numerate the Fibonacci numbers. We know that the first two are 0 and 1 and we know the next one is sum of these two. So, the next one will be 1 again, next one will be 2, for this 1 plus 1, next one will be 3, next one will be 5, next one will be 8, next one will be 13 and so on. So, it is very clear that though these numbers as an inductive definition and there is an obvious recursive function that goes with it which I have just mention in a minute. There is a very efficient way of enumerating these numbers, directly almost in linear time.

So, how does the definition of the function code? Fibonacci of n is just if n is 0, return 0, if n is 1 return 1. So, if n is 0 or 1, the value is n itself. Otherwise, you compute value recursively using this criterion Fibonacci of n minus 1 plus Fibonacci of n minus 2 and finally, whatever value you are computed you return.

(Refer Slide Time: 02:21)

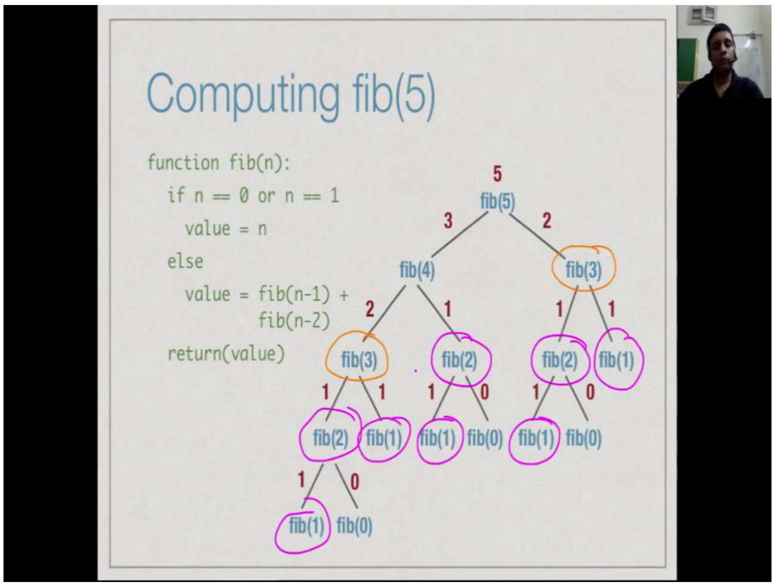


Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

So, where is the catch? So, let us see how this would work on an small input like we have just saw Fibonacci. So, we just saw the Fibonacci of 5, this 5, we know that we start with 0, 1, 1, so these are the arguments. So, the values are 0, 1, 1, 2, 3 and then 5. So, we are trying to compute, if this is a Fibonacci number we computed very fast. But, let us see how this recursive thing would actually got, if I call Fibonacci of 5, recursively it would ask me to compute Fibonacci of 4 and 3, Fibonacci of 4 in turn will ask me to compute 3 and 2.

(Refer Slide Time: 02:55)



Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

Now, 3 in turn will ask me to compute 2 and 1, then 2 in turn will ask me to compute 1 and 0. Now, fortunately for 1 and for 0, I have a base case. So, I will get those values 1 and 0 respectively. So, since I got 1 and 0 and I will now be able to compute Fibonacci of 2 at sum of those and I will get the answer 1. Now, I have turn into Fibonacci 1, again it is a base case, I get the answer 1.

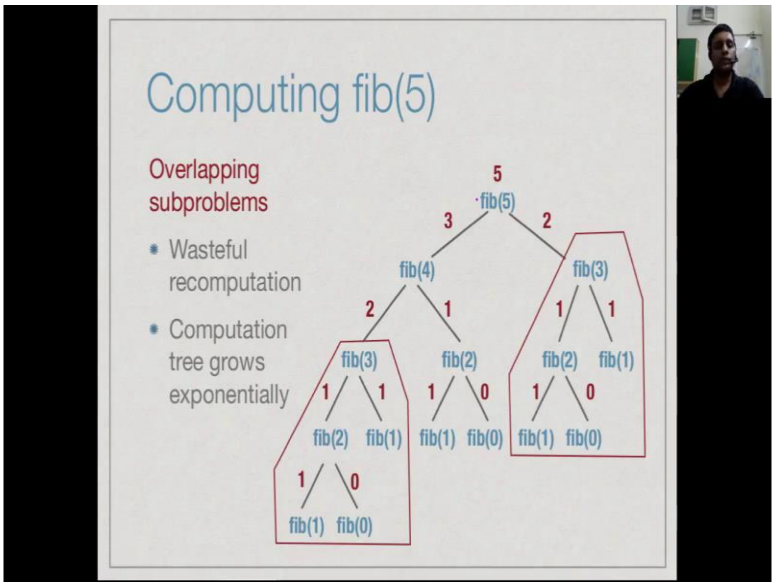
So, now I have both the answers required for Fibonacci of 3. So, I get the answer 2, now I go back and I compute Fibonacci of 2, but remember I have already computed it, but because I did not make note of this in some sense, I blindly doing recursion. Fibonacci of 2 will again ask me to compute Fibonacci of 1 and 0, which will again produce the base case with 1 and 0.

So, I will again compute Fibonacci of 2 as 1 and now will both Fibonacci of 3 and 2 are available, I will get the answer Fibonacci of 4 and 3. And now, I will go back and compute Fibonacci 3 yet again. So, I will do this whole tree again, 3 calls 2 and 1, 2 calls 1 and 0, these return the base case, that give me a value for 2, 1 returns the base case, this give me a value for 3 and finally, after all these, I get the value Fibonacci of 5.

So, the problem we can see is that functions that Fibonacci of 3 have been computed twice in full and full complexity, other functions like Fibonacci of 2 have been computed 1, 2, 3 times and so on. And of course, Fibonacci of 1 towards it is base case have been call several time 1, 2, 3, 4, 5 times and Fibonacci of 0 has been called 3 times.

So, this is the big problem with this recursive computation, but we seem to be computing too many things, the ideally in order to compute Fibonacci of 5 and I know I need 4, 3, 2, 1 and 0. So, I need only six values totally to be computed and I making a lot more and six computation.

(Refer Slide Time: 05:03)



Computing fib(5)

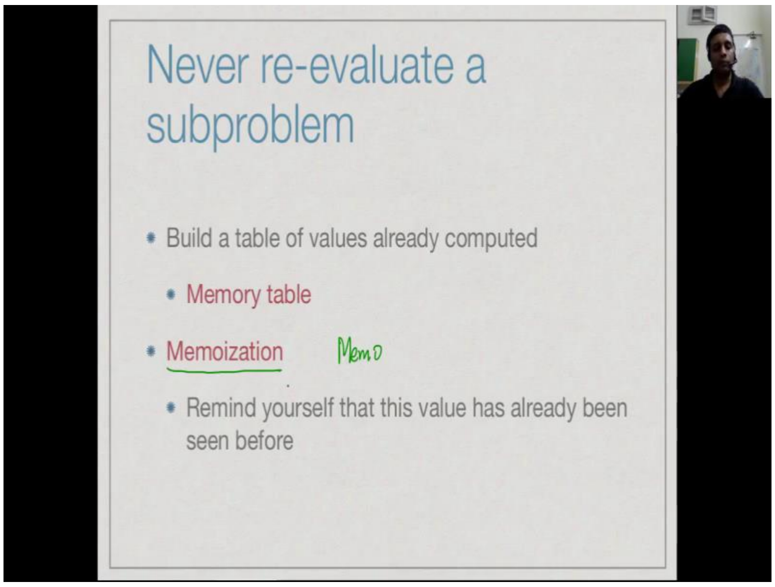
Overlapping subproblems

- Wasteful recomputation
- Computation tree grows exponentially

So, the cracks of the problem is that there are be sub problems which arise in different context within recursive computation, for Fibonacci of 3 is generated both value original called Fibonacci of 4 and the nested recursive called Fibonacci of, so the original called Fibonacci of 5 and the nested recursive called Fibonacci of 4. So, we have this entire tree of computation which is duplicated and because of this kind of waste for recomputation, over all the computation tree grows exponentially.

So, you are in actually find in general, that in order to compute the nth Fibonacci number, you actually do some exponential in n one steps. Even though, we can see that you can just compute nth Fibonacci number in linear time, we just sorted, you just computed as the 0, 1, 2, 3, 4 and so on.

(Refer Slide Time: 05:53)



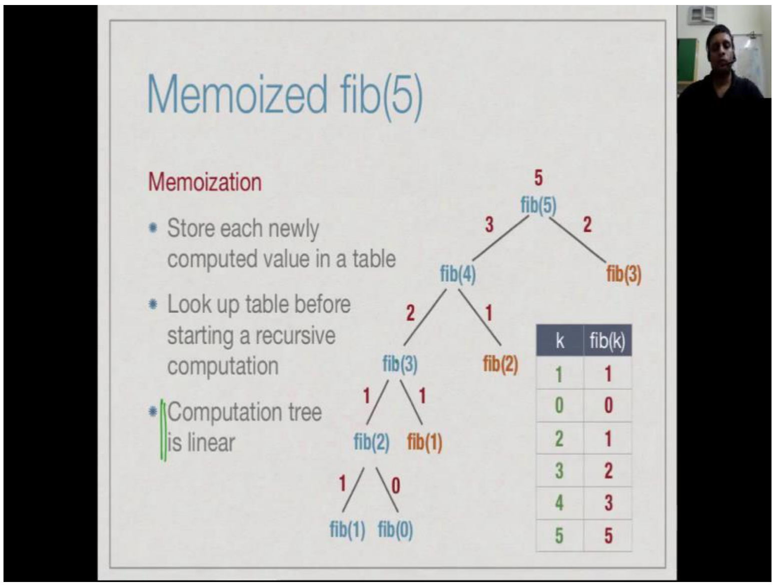
Never re-evaluate a subproblem

- Build a table of values already computed
  - Memory table
- Memoization
  - Remind yourself that this value has already been seen before

So, one way to get around this is to make sure that you never reevaluate a sub problem. So, you build up it table which is sometimes called a memory table and what is the table do it just keeps track of every value for which you have computed the function before. So, it is just the look up in a language like java, it could be a hash map or a language like python, it could be a dictionary. Every time you call a f on a value x, you just store x comma f x in this table.

So, that you can look at up and see we already computed it. So, it is called memoization. So, this term memoization comes from the word memo with some of few may occur. So, memo is a note which remains you are something are remains somebody else or something. So, memoization, memo table or a memory table is supposed to remind you that the value your time to compute has already with computed.

(Refer Slide Time: 06:50)



Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

k	fib(k)
1	1
0	0
2	1
3	2
4	3
5	5

So, how does memoization work? So, we have this memo table here, so this is our memo table, so in this memo table it is just a table where we fit different value of k and fib k has we compute them. We have assumed that we have computed nothing, we do not even know the base case, we just going to apply the recursive definition. But, every time we compute something, we will first look up the table before we computed recursively and if we have to computed recursively, then we will store each newly computed back end of the table.

So, let us start as before we wanted to start computing the Fibonacci of 5. So, the recursive definitions says, that we should call Fibonacci of 4 and 3. Now, we go to 4, where just doing recursion left to right, so 4 will call 3 and 2, 3 will call 2 and 1, 2 will call 1 and 0, at this point we try to evaluate Fibonacci of 1 to the first time. When, we evaluate Fibonacci of 1 for the first time, we get a base case and it tells us that this value is 1. So, we will return this value, but we will also store it in the table.

So, now, we are made up first entry in the table itself, Fibonacci of 1 is 1, then we look at Fibonacci of 0 and again, it is a new value, we got it from the base case, but we are never computed before. So, now, we are computed it by computing the base case and again, we put an entry in the tables side, Fibonacci of 0's sign. So, with this as before we have got Fibonacci of 2.

So, up to this point, we have not saved anything, excepted we have also now, because of be a computed Fibonacci of 2, we have added to, we done new value computed we put it into the table. Now, to continue with Fibonacci of 3 I need to go back to Fibonacci of 1, of course, it is the base case, but what my table thing tells us, tells you that you should look at the table first. So, we both on the table, hopefully it is organized in some efficient way, but in the worst case, let us is to naively scan it, we look have I ever done Fibonacci of 1 before, this is I have done Fibonacci of 1.

So, it will look up that value without calling the base case, we have to looking at the function definition and just return Fibonacci of 1 is 1. So, we are indicated by turning it orange that this call was actually return from the table, it in actual required computation. So, now, that have 1 and 1, Fibonacci of 3 is now 2. So, now I have to continue with Fibonacci of 4. So, now I have to call Fibonacci of 2 again.

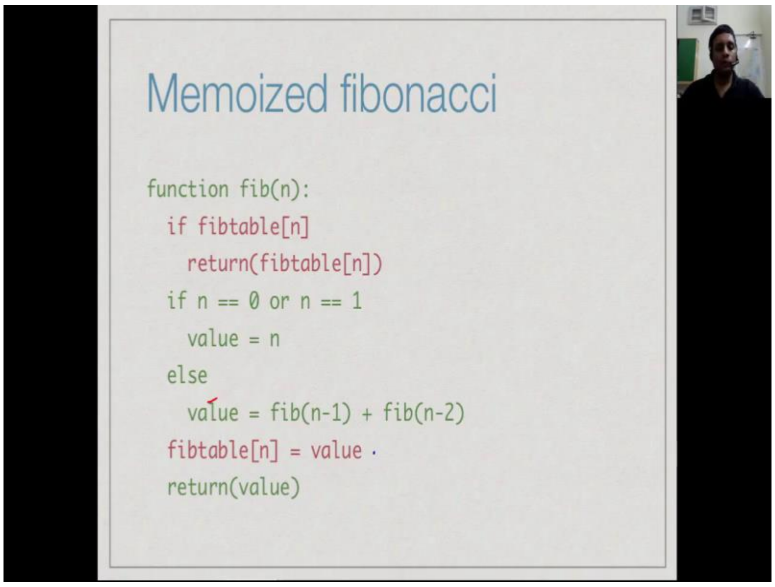
So, now it will go down and say I have ever seen Fibonacci of 2 and say I have and therefore, without doing the re computation, it will look up this memo table and again we mark it as in orange value, because it came from the table, it returns the value 1 which we stored in the table. So, now have I computed this, I got Fibonacci of 4. So, we compute Fibonacci of 4 to be 3 from the recursive definition and there again we store it in the table.

So, every new value that we compute, we store in the table that could be able to use it again or not we do not know, but if we want to use it again, it is good to have it them. So, we just blindly put everything that we ever compute back into the table for future use. now, we come back to the top Fibonacci of 5, we have got the left guns, we need the guns, we go to Fibonacci of 3, earlier we had lineally reproduces the entire 3 at the bottom in order to get Fibonacci of 3, but this time we are smarter, we go down the table and we find that 3 has been computed before and the value is 2, so we get this value of 2.

And now, we have what the answers that we need for Fibonacci of 5 and so we get Fibonacci of 5 is 5 and again, we put it in the table, although in this particular case, we are not going to the user to the computation is over, but it could be part of a bigger computations, so we just keep doing this. So, what you can see in this is that the computation tree which used be exponential is now linear.

Why is it linear? Because, every value which is in blue appears in the 1s and every value which in not in blue is a look up in the table, So, it cost nothing, so therefore, we have now made, so this more or less this memoized Fibonacci is what we do, whether we do it in a different order, when we actually computed by hand.

(Refer Slide Time: 11:01)



Memoized fibonacci

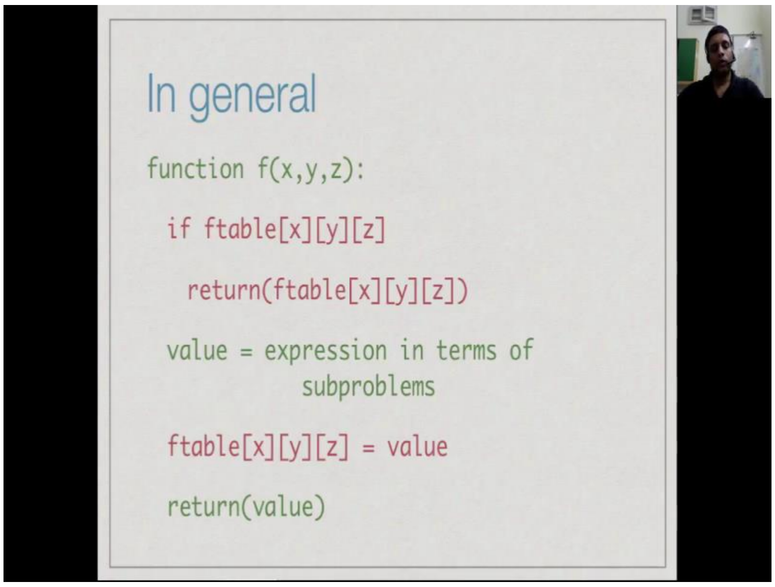
```
function fib(n):
  if fibtable[n]
    return(fibtable[n])
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

So, just to see what the memoized Fibonacci looks like? So, the green code is what we wrote earlier, fib of n, if n is 0 or 1, set the value to n, else set the value to the fib of n minus 1, fib of n minus 2 and return. So, now we have introduced a memo bits which are the red ones. So, we have a table which we call fib table, whose values are indexed by the position.

So, fib of 6, if I compute is going to fib table 6. So, when I get an argument to fill, the first thing I do is I look whether fib table has an entry for that. So, if fib table of n exist return whatever it says, if it does not exist, then we compute it, once we have computed it, before we do anything we put it back in the table. So, that next time we will not computing it. So, every new value it is put back in the table and then, we return it.

So, when we return a value here, it has been computed for the first time, but it has been stored in the table. So, the next time, I get the same in I will come here and I will exist that is what. So, this is our memoized Fibonacci, it just has a very simple check for the table at the beginning and when you compute the value, it puts it back.

(Refer Slide Time: 12:15)



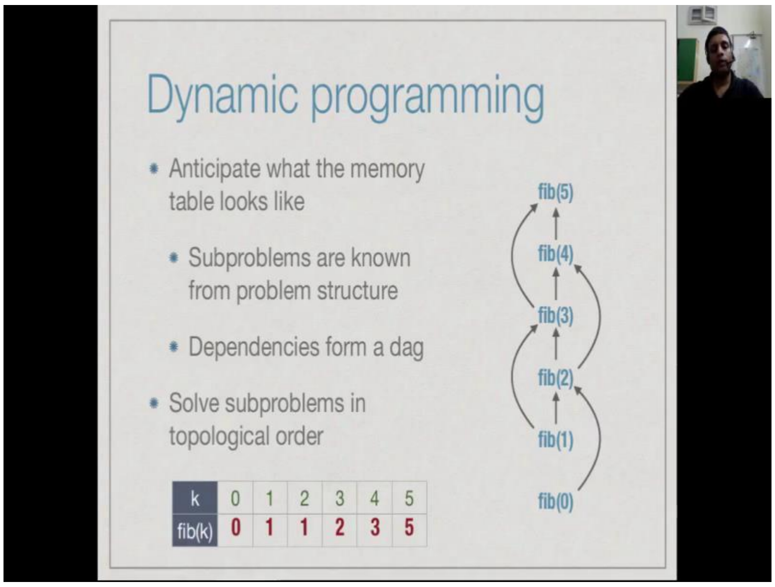
In general

```
function f(x,y,z):
  if ftable[x][y][z]
    return(ftable[x][y][z])
  value = expression in terms of
            subproblems
  ftable[x][y][z] = value
  return(value)
```

So, this is the very simple thing that you can do, you can do this for any functions. So, supposing you had some recursive function or inductive function with three arguments, then you just have a table with three indices. So, let us call itd a f table, so computing f. So, you have some inductive way of computing the value from the sub problems. So, first you look up have an ever seen this x, y, z before, if so just return it.

Otherwise, compute a new value for this given x, y and z in terms of the recursive definition from the inductive structure of the problem having computed it, put it back in the table, that you never had computed explicitly again and that you return. So, this is a generics key which can be applied to any recursive function that you might write, the only thing you have do is you have to make sure that you can design a table with their appropriate look up and look it up it is efficiently. Because, if spend a long time looking up the value of the table, then that is lost. So, if you can make it up end of array look up that is an ideal, so this is memoization.

(Refer Slide Time: 13:17)



Dynamic programming

- Anticipate what the memory table looks like
    - Subproblems are known from problem structure
    - Dependencies form a dag
- Solve subproblems in topological order

k	0	1	2	3	4	5
fib(k)	0	1	1	2	3	5

So, the other term that we introduce in the last lecture is dynamic programming. So, what dynamic programming does is it tries to eliminate the recursive part of evaluating an inductive definition. So, in memoization what we do is, we evaluate the inductive definition recursively exactly as we would normally, you only thing is we keep it table which helps us to avoid having to compute the same thing twice.

In dynamic programming, you anticipate what the table should look like and how the values in the table depended. So, supposing we have computing Fibonacci of 5, then by some simple analysis of Fibonacci, we know that if Fibonacci of I requires anything at all, it requires things smaller than and the smallest thing you can get a Fibonacci of 0. Therefore, from Fibonacci of 5, we immediately know that the sub problem that could be an interest to us or all sub problems between Fibonacci of 0 and Fibonacci of 5.

Fibonacci of 5 is not going to required Fibonacci of 6, 7, 8 mark can be call Fibonacci of the negative number because it is not define. So, there’s a very small infinite set up values which we need to consider. Now, the next observation is that we can compute the dependencies from the problem structure; this is from the inductive definition. So, Fibonacci of 4 is equal to, I mean Fibonacci of n is m minus 1 plus m minus 2. So, 5 depends on 4 and 3.

So, I would say that, there is in order to compute 5, I am as a first computed 4 and 3’s. So, I will drawn arrow from what it depends on to what it is. So, 5 depends on 4, 5 depends on 3. So, I put an array from 4 to 3 and 4 to 5 and 3 to 5, send at there is a dependency. In turn 4 depends 1, 3 at 2 and 3 depends and 2 and 1 and 2 depends on 1 and 0.

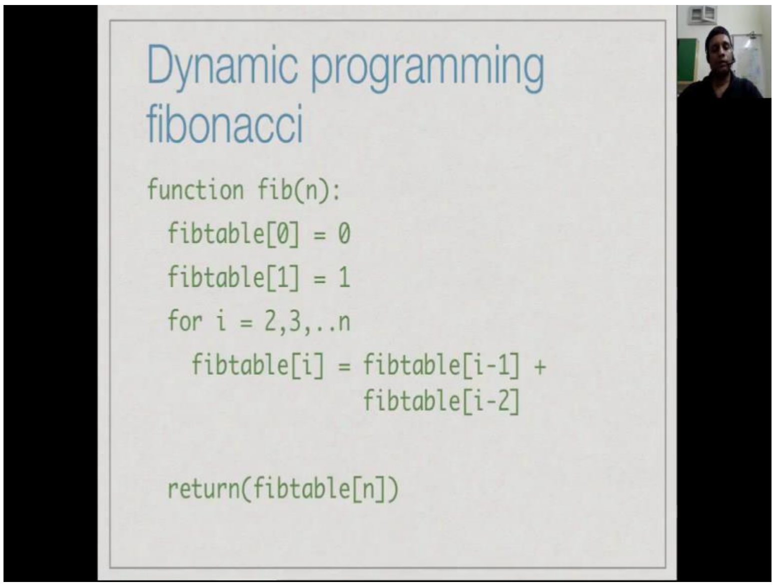
And of course, 1 and 0 are the base cases and they do not depend on anything. So, the dependencies would form a dag, why do their form a dag, well is squired obvious they must from a dag, because if there are cyclic dependencies you cannot compute one before they other. So, it is a very natural thing in any kind of dependency, whether it is dependency between pass that we saw in when we did dag begin with there are many, have do something else it is presides there, it saying that, if I cannot compute Fibonacci of 2, I cannot compute Fibonacci of 3, because has been done first.

Now, that I know these dependencies, I can now enumerate them in any topological order, such that when I reach a value to be computed, everything it depends on is norm 4. So, for an example here a natural log out just be a of course, 0, 1, 2, 3, 4, 5, because both of these are no elements with no incoming ages. So, I can either start mate topological order which 0 or it 1.

So, let us assume as start with 0's, so I know that Fibonacci of 0 depends on nothing, it must be a base case circulate in, likewise I know that Fibonacci of 1 depends on nothing. So, it is a base case, if I that in, so this point have done both of these, since have done both of these, these edges are gone, so there is nothing pointing it 2. So, this is a topological sort, the basic topologic set is also.

So, three still has a dependency namely 2, but 2 has more dependencies any more to a both 1 and 0 are refused I can fib Fibonacci of 2, then Fibonacci of 3, 4 and 5 and this is exactly what we did earlier, when we try to compute Fibonacci by hand. So, what you actually executing, when you are writing of the Fibonacci numbers left to right, where there is looking the previous to and computing the next one is actually what is called dynamic program.

(Refer Slide Time: 17:11)



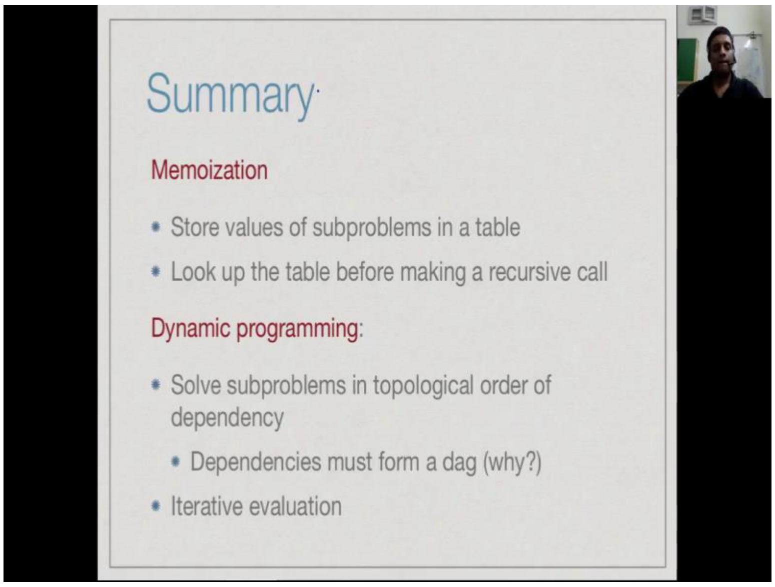
Dynamic programming fibonacci

```
function fib(n):
  fibtable[0] = 0
  fibtable[1] = 1
  for i = 2,3,..n
    fibtable[i] = fibtable[i-1] +
                  fibtable[i-2]

  return(fibtable[n])
```

So, dynamic programming for the Fibonacci function just consist of iteratively filling up the table, we start would the value 0 and 1, then for 2 to n, of course, if n is less than this we will not go through these equation at all for 2 to n, we fill up the ith entry from the i minus 1th entry by i minus 2th entry, but there in as we saw before these two values occurred earlier, so they will already filled. So, I have converted my memo table filling from a recursive computation filling on demand as network to filling blindly from beginning to end, which is dynamic program.

(Refer Slide Time: 17:57)



Summary

Memoization

- Store values of subproblems in a table
- Look up the table before making a recursive call

Dynamic programming:

- Solve subproblems in topological order of dependency
  - Dependencies must form a dag (why?)
- Iterative evaluation

So, to summarize, we have seen two strategies to make recursive computations of inductive functions more efficient. The first is memoization. So, what memoization does is whenever it computes the value of the function in sub problem, it puts it into a deep. And before it calls the function recursively to compute the value, it will look up the table to see if this value as already we computed to the same value is never the same computation never happens twice.

In dynamic programming on the other hand, we analyze problem structure, were identify the sub problems and we know that this sub problem structure satisfies a dag, the dependencies must form a dag, because must be not a dag, then we will have cycles and things could not be able to solve anything. And having done this, what we get from dynamic programming is iteratively evaluation.

So, this is actually in practice a big saving, because in most programming languages there is a hidden cost to recursion because every function call actually requires certain amount of operating system and programming language, bureaucracy, some administrate work, take them to declare some memory on the stack and so on and this is avoided a completely in a dynamic program resolution.

So, it is a big saving to able to remove even though memoization is going to give us an optimum number of recursive calls required for a given problem, if not, going to call anything twice. It will still involve recursive calls and recursive calls can be expansive in their own writing. For sake of algorithms, we typically read function call as kind of unit cost operation, but in practice, it is not the case. So, dynamic programming can be a huge saving, because it actually converts, optimize recursive thing into an iterative scheme.