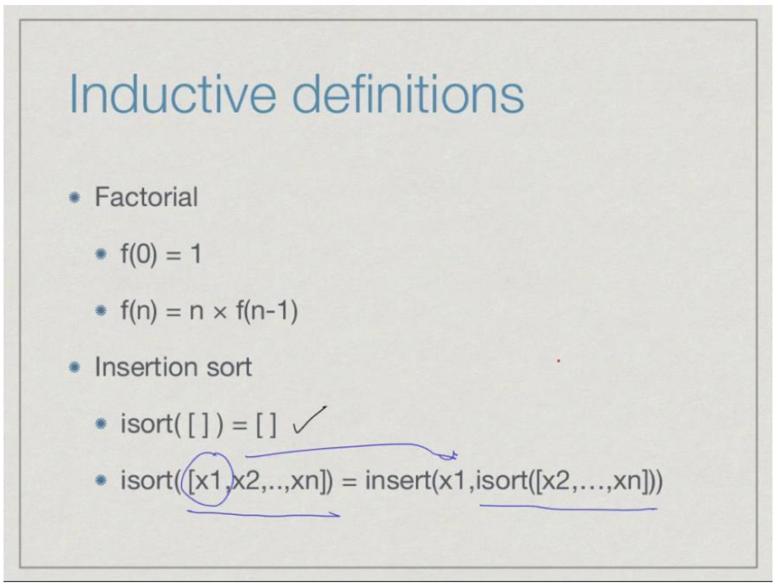
Programming, Data Structures and Algorithms in Python

Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 08
Lecture - 01
Memoization and Dynamic Programming

(Refer Slide Time: 00:02)



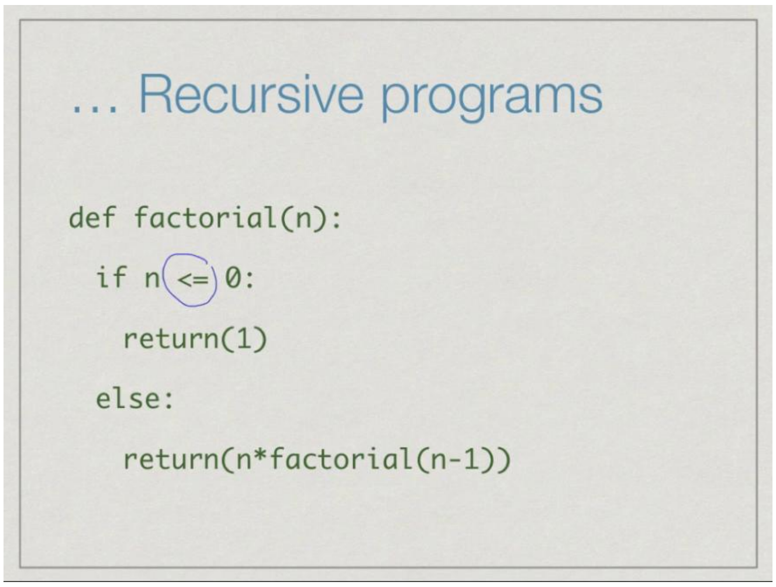
Inductive definitions

- Factorial
  - $f(0) = 1$
  - $f(n) = n \times f(n-1)$
- Insertion sort
  - isort([ ]) = [ ] ✓
  - isort([x1,x2,..,xn]) = insert(x1,isort([x2,...,xn]))

We saw earlier that inductive definitions often provide a nice way to get a hold of the functions we want to compute. Now we are familiar with induction for numbers. For instance, we can define the factorial function in terms of the base case f of 0, and in terms of the inductive case saying f of n is n times factorial of n minus 1. What we saw is we can also define inductively functions on structures like lists, so for instance, we can take as the base case an empty list, and in the inductive case, we can separate the task of sorting of list into doing something with the initial element and something with the rest.

Insertion sort can be defined in terms of insert function as follows. So, isort of the base case for the empty case just gives us the empty list. And then if you want to sort a list with n elements, we pull out the first element right and then we insert it into the result of inductively sorting the rest. This is a very attractive way of describing the dependency of the function that we want to compute the value that we are trying to compute on smaller values, and it gives us a handle on how to go about computing it.

(Refer Slide Time: 01:12)

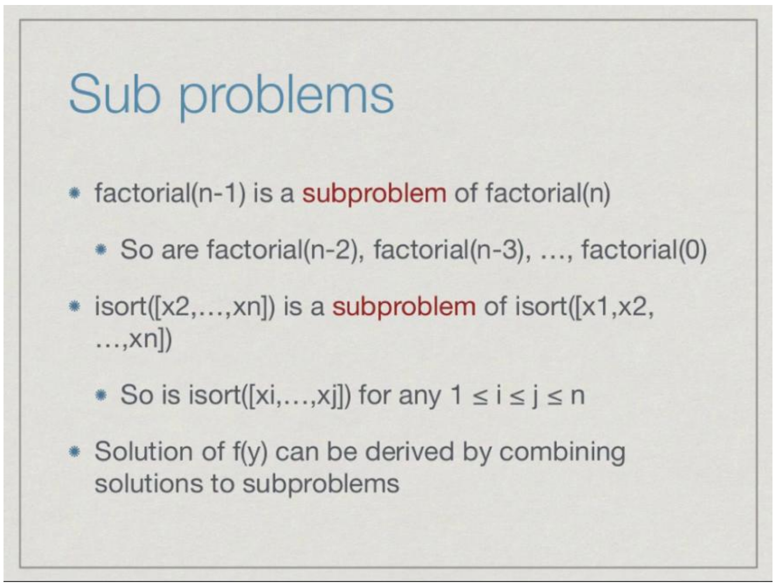


… Recursive programs

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

The main benefit of an inductive definition is that it directly yields a recursive program. So, we saw this kind of a program for factorial which almost directly follows a definition saying that f of 0 is 1 and f of n is n into f n minus 1. So, we can just directly read it talk more or less and translate it. The only thing we have done is we have taken care of some error case, where if somebody feeds a negative number, we will still say 1, and not go into a loop.

(Refer Slide Time: 01:40)



Sub problems

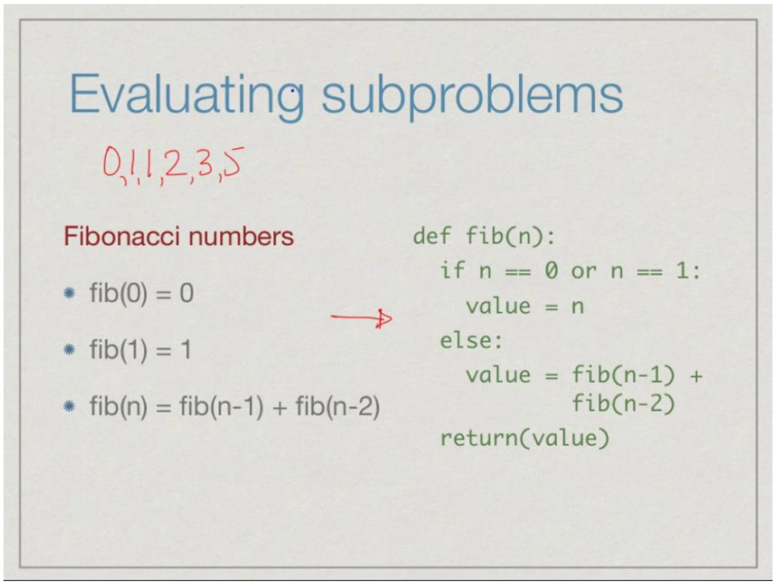
- factorial(n-1) is a subproblem of factorial(n)
  - So are factorial(n-2), factorial(n-3), …, factorial(0)
- isort([x2,…,xn]) is a subproblem of isort([x1,x2,…,xn])
  - So is isort([xi,…,xj]) for any 1 ≤ i ≤ j ≤ n
- Solution of f(y) can be derived by combining solutions to subproblems

In general, when we have such inductive definitions, what we do is we have sub problems that we have to solve in order to get to the answer we are trying to reach. So, for instance, to compute factorial of n, one of the things we need to do is compute factorial of n minus 1. So, we call factorial of n minus 1 as sub problem of factorial n.

Now in turn factorial of n minus 1 requires us to compute factorial n minus 2, so actually if you go down the chain, the factorial n sub problems are all the factorials for values smaller than n. Similarly, for insertion sort, in order to sort the full list, we need to sort all the elements excluding the first one what is called the tail of the list, and in turn we need to sort its tail and so on.

In general, when we do insertion sort we will find that we need to sort things a segment of the list. So, we can in general talk about x i to x j. And in all these cases, what the inductive definition tells us is how to compute the actual value of f for our given input y by combining the solutions to these sub problems; for instance, in factorial we combine it by multiplying the current input with the result of solving it for the next smaller input. For insertion sort, we combine it by inserting the first value into the result of solving the smaller input that is the tail of the list.

(Refer Slide Time: 03:05)



Evaluating subproblems

0,1,1,2,3,5

Fibonacci numbers

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

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

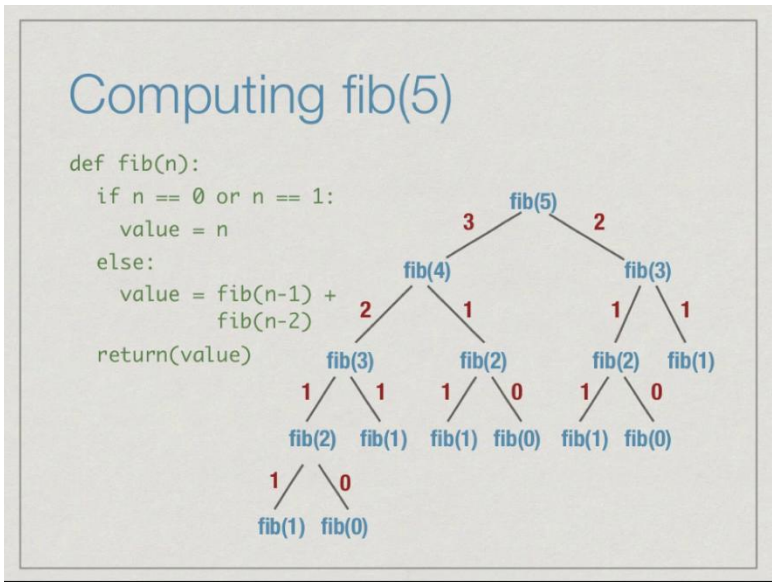
Let us look at one particular problem, which will highlight an issue that we have to deal with when we are looking at inductive specifications and naively translating them into programs. The Fibonacci numbers are a very famous sequence which were invented by

Fibonacci, and they occur in nature and they are very intuitive and most of you would have seen them. The Fibonacci numbers are 0, 1 and then you add. So, 1 plus 0 is 1, 1 plus 1 is 2, 3, 5 and so on.

So, you just keep adding the previous two numbers and you go on. The inductive definition says that 0th Fibonacci number is 0; the first Fibonacci number is 1; and after that for two onwards, the nth Fibonacci number is obtained by sub adding the previous two. The Fibonacci number 2 is Fibonacci 1 plus Fibonacci 0. As before we can directly translate this into an inductive into a recursive program. We can just write a python function fib which says if n is 0 or n is 1, you return the value n itself. So, if n is 0 return 0, if n is 1, we return 1.

Otherwise, you compute the value by recursive to recursively calling Fibonacci on n minus 1 and n minus 2, add these two and return this value. Here is the clear case of an inductive definition that has a natural recursive program extracted from it.

(Refer Slide Time: 04:27)



Computing fib(5)

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

Let us try to compute a value and see what happens. So, supposing we want to compute Fibonacci of 5 using this definition. So, Fibonacci of 5, we will go into the else clause and say we need to compute Fibonacci of n minus 1 namely 4 and n minus 2 namely 3. As Fibonacci of 5, leaves us with two problems to compute, Fibonacci of 4 and Fibonacci of 3. So, we do these in some order; let us go left to right. So, we pick

Fibonacci of 4, and this in turn will require us to compute Fibonacci of 3 and Fibonacci of 2 by just applying the same definition to this value.

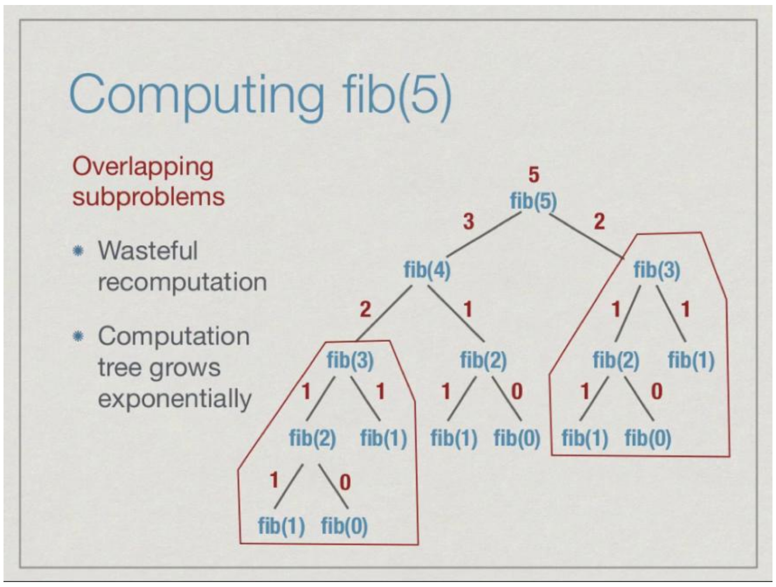
Similarly, we go to the left of the two sub problems Fibonacci of 3 requires 2 and 1; 2 requires 1 and 0. Now for 1 and 0 fortunately we can exit without making a recursive call; if n is equal to 0 or n is equal to 1, we just return the value. So, we get back Fibonacci 1 as 1 and Fibonacci 0 as 0. So, with this, we can complete the computation of Fibonacci 2, we get value is 1 plus 0 in other words 1. So, Fibonacci of 2 is 1.

Now, we are back to Fibonacci of 3. So, we have computed for Fibonacci of 3, the left case Fibonacci of 2. Now we have to compute the right case. And once again we find the Fibonacci of 1 being a base case it gives us 1, and now we can combine this and get Fibonacci of 3 is 2.

Now, we are back to Fibonacci of 4. And we have computed the left side of Fibonacci of 4. So, we need to compute the right side. And now what happens is we end up having to compute Fibonacci of 2 again, even though we already know the value. We naively have to execute Fibonacci of 2 call 1 and 0, again propagate the values 1 and 0 back up add them up and get 1. Now, we can compute Fibonacci of 4 is 2 plus 1 3.

And now we are finally, back to the original call where we had to compute Fibonacci of 4 and Fibonacci of 3. So, we are done with 4. Now, we want to do Fibonacci of 3. Notice that we have already computed Fibonacci of 3, but this will blindly require us to call this function again. So, we will again have to execute this full tree, go all the way down, go all the way up and eventually Fibonacci of 3 will of course, give us the same answer namely 2, which we already knew, but we would not take exploit or we would not take advantage of the fact that we knew it. And in this way, we get 3 plus 2 and therefore, Fibonacci of 5 is 5.

(Refer Slide Time: 06:45)



Computing fib(5)

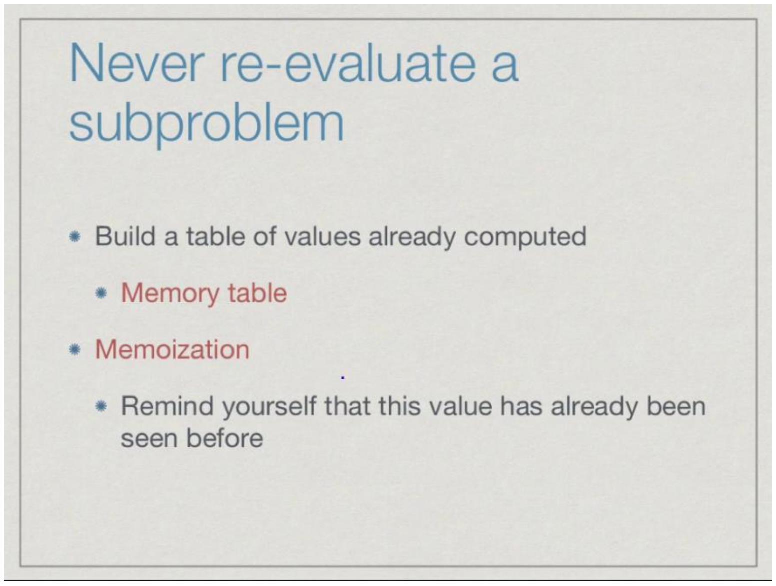
Overlapping subproblems

- Wasteful recomputation
- Computation tree grows exponentially

The point to note in this is that we are doing many things again and again. In this particular computation, the largest thing that we repeat is Fibonacci of 3. So, as a result of this re-computation of the same value again and again, though we in principle only need n minus 1 sub problems.

If we have fib of 5, we need to fib of 4, fib of 3, fib of 2 and so on. N subproblems, if we don't include fib of 0, but some of these sub problems like in this case Fibonacci of 3, we compute repeatedly in a wasteful way. As a result, we end up solving an exponential number of solve sub problems even though there are only order n actual problems to be solved.

(Refer Slide Time: 07:26)



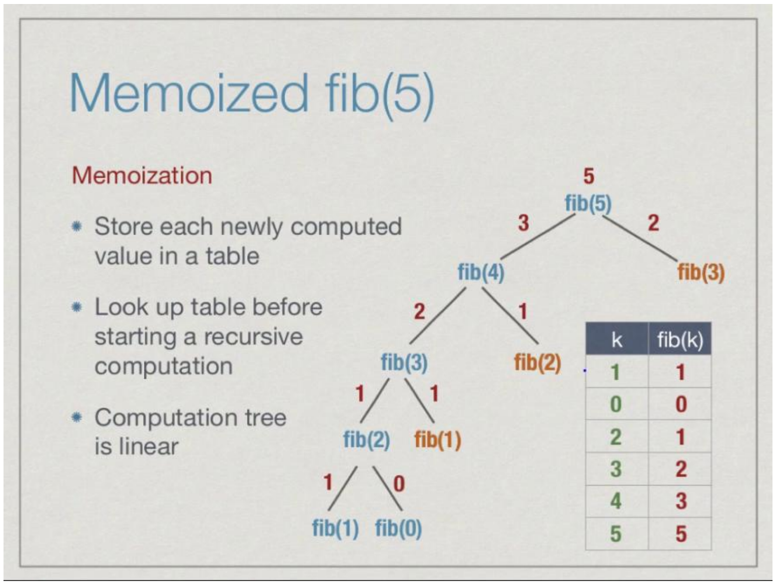
Never re-evaluate a subproblem

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

So, what we want to do is move away from this naive recursive implementation of an inductive definition, and try to work towards never reevaluating a sub problem. This is easy to do, if we could only remember the sub problems that we have solved before, then all we have to do is look up the value we already computed rather than recompute it. So, what we need is a kind of a table, a table where we store the values we have computed and before we go and compute a value, we first check the table. If the table has an answer, we take the table's answer and go ahead.

If the table does not have an answer then we apply the recursive definition compute it, and then we add it to the table. This table is normally called a memory table to memorize; and from this, we get this word memoization. It is actually memo and not memorization; memoization in the sense of write yourself a memo, memo is like a reminder, write yourself a reminder that this has been done before. Memoization is the process by which when we are computing a recursive function, we compute the values one at a time, and as we compute them we store them in a table and look up the table before we recompute any.

(Refer Slide Time: 08:41)



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

Here is how a computation of Fibonacci of 5 would go, if we keep a table. This is our table here right. So, we have a table where in some order, it does not really matter for now; in some order as and when we find Fibonacci of k for some k, we just record it. And notice that this table is empty, even though we know the base case of Fibonacci of 0 and Fibonacci of 1 are 0 and 1 respectively, we do not assume you know it, because it will come out as the first time we hit the base case it will come out of the recursive definition.

Let us see how it goes right. So, we start Fibonacci of 5 as usual, it says do 4 and 3, 4 says do 3 and 2, 3 says do 2 and 1, 2 says do 1 and 0. And now from our basic case, the base case in the function, we will get back that fib of 1 is 1. So, we store this in the table, this is the first value we have actually computed. Notice we did not assume we knew it, when we came to it in the base case; we put it into the table. Same way, fib of 0 is 0 we did not know it before; we put it in the table.

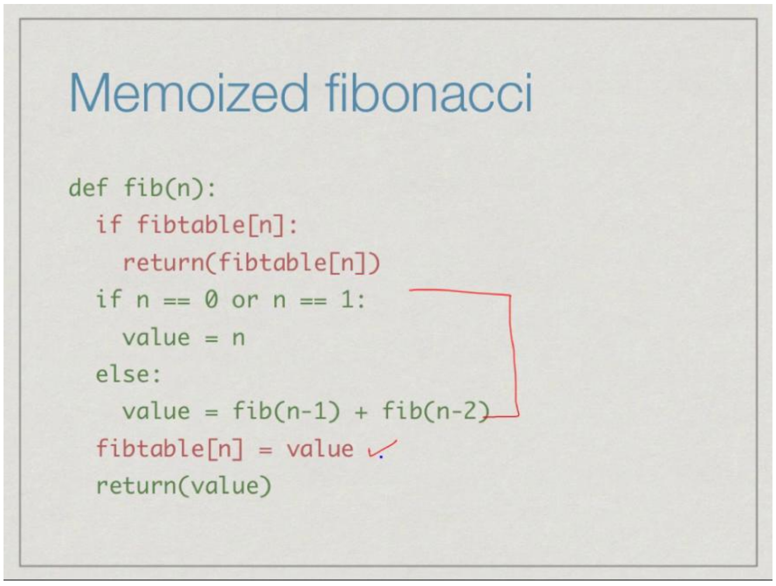
Now we come up and we realize that fib of 2 is now available to us, it is 1 plus 0 is 1. So, we store that in the table. We say for k equal to 2, fib of k is 1. Now we come back to fib of 3, and now we go down and it asks us to compute fib of 1 again. Now although this does not take us any work, because it is a base case we do not actually exploit that fact. We first look in the table and say is there an entry for k equal to 1, yes there is and so we pick it up.

We highlight in orange the fact that this value was actually not recomputed, but looked up in the table, so from 1 plus 1, we now have Fibonacci of 3 is 2. Now we go back up to Fibonacci of 4, and it asks us to compute the second half of its sub problems, namely Fibonacci of 2. Once again we find that there is an entry for 2 in our table. So, we mark it in orange, and we just take the value from the table without expanding and computing the tree again as we had done before when we did the naive computation. Now, 2 plus 1 is 3, so we have Fibonacci of 4.

So, we have to now go back and compute the other branch Fibonacci of 3, but once again 3 has an argument k is in our table. So, we have an entry here for 3. So, we can just look up Fibonacci of 3 and say oh, it is two. So, once again we mark it in orange. And so now, we have Fibonacci of 5 is 3 plus 2 and that is 5, and then now this is a new value, so we enter that.

Notice therefore, that every value we computed, we expanded the tree or even looked up the base case only once according to the function definition. Every subsequent time, we needed a value we just looked it up in the table, and we can see that the table grows exactly as many times as much as there are sub problems to be solved and we never solved a sub problem twice in the sense of computing it twice. We solved it by looking at the table.

(Refer Slide Time: 11:32)



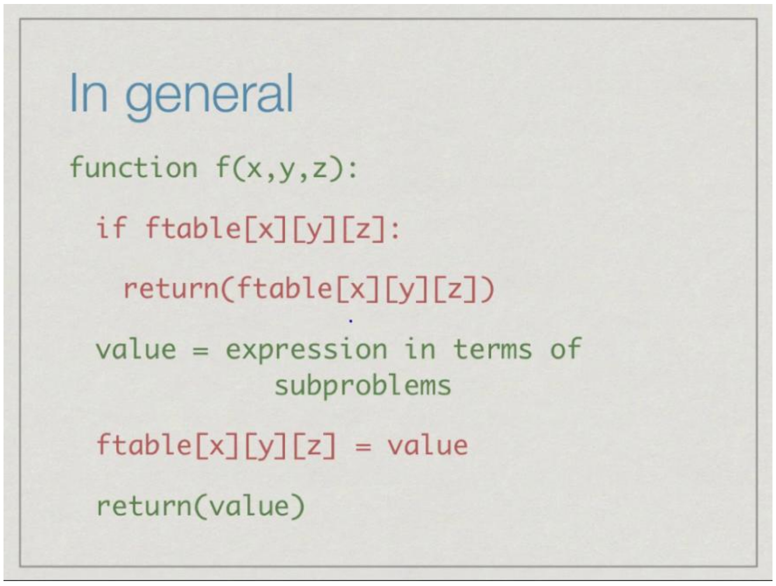
Memoized fibonacci

```
def 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)
```

This is a very easy step to incorporate into our Fibonacci table, the Fibonacci functions. So, we just add this red code. The green lines are those ones we have already had before. Now what Fibonacci says is, the first thing you do when you get a number is try and look up the table. If there is a value Fibonacci of n, which is defined then return that value; otherwise, we go through the recursive computation. This is the usual computation which will make a recursive call and eventually come up with a new value which is the value for this particular n. So, before we return this value back as a result of this function, we store it in the table.

Henceforth, if this value n is ever invoked again, we never have to look up the thing we never have to compute it; it will be in the table right. It is very simple as we said when you get an argument n for which you want to compute the function, you first check the table, if it is there in the table you do not do anything more you just return the table value. If it is not in the table, you apply your recursive definition to compute it, just like you would normally. Having computed it you first store it in the table, so that future accesses to this function will work without having to do this recursion and then you return the value you got.

(Refer Slide Time: 12:49)



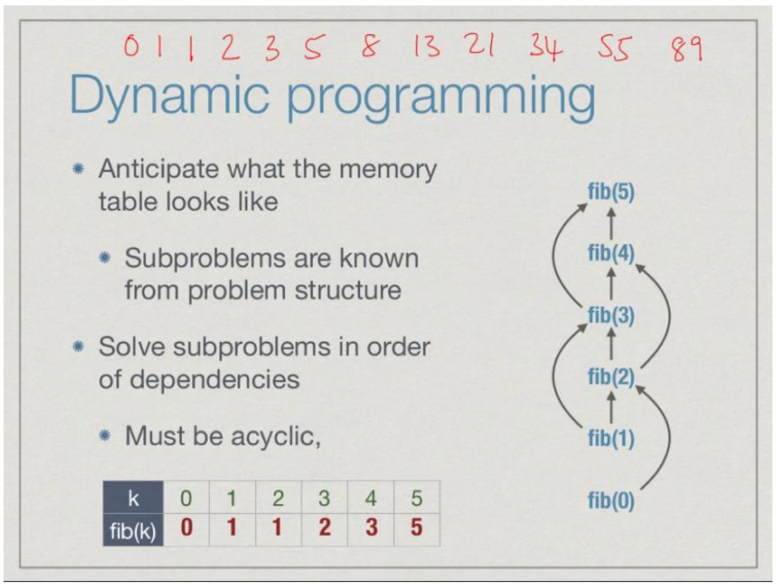
In general

```
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)
```

This can work for any combination of arguments. You just need a table, in python terms; you just need a dictionary for every combination of arguments if that value has been computed before that key will be there in our table. This table in general in python would be a dictionary and not a list because the arguments could be any particular values. They could some could be string, some could be numbers, they need not be continuous that is all.

We basically for given the particular combination of arguments, we look up whether that combination of keys is there in the dictionary if so we look it up and return it. Otherwise, we compute a new value for this combination, store it and then return it. So, we have glossed over little few things for instance typically where if you want to really write this properly in python way we have to use some exceptions and all that, but this is more or less the skeleton of what we need to do.

(Refer Slide Time: 13:42)



0 1 1 2 3 5 8 13 21 34 55 89

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic,

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

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

This brings us to the main topic that we want to do this week in a couple of lectures which is called dynamic programming. So, dynamic programming is just a strategy to further optimize this memoized recursion. This memoized recursion tells us that we will store values into a table as and when they are computed. Now it is very clear that in an inductive definition, we need to get to the base case and then work ourselves backwards up from the base case, to the case we have at hand. That means, there are always some values some base values for which no further values need to be computed; these values are automatically available to us.

We have some problems which have sub problems, and some other problems which have no sub problems. If a problem has a sub problem, we cannot get the problem, we cannot get Fibonacci of five unless we solve its sub problems, Fibonacci 4 and 3. But if we have a base case that Fibonacci of 1 or Fibonacci of 0, we do not have any sub problems, so we can solve them directly. This is a kind of dependency. So, we have to solve the sub problems in the dependency order, we cannot get something which is dependent on something else until that something else has been solved, but a little thought tells us that this dependency order must be acyclic, we cannot have a cycle of dependency.

So, if one value it is like 5 depends on 3, 3 depends on 1, and 1 again depends on 5, because there will be no way to actually resolve this right. There will always be a starting point, and we can solve the sub problems directly in the order of the dependencies instead of going through the recursive calls. We do not have to follow the inductive structure, we can directly say ok, tell me which are all the sub problems which do not need anything solve them, which are all the ones which depend only on these solve them and so on. This gives as an iterative way.

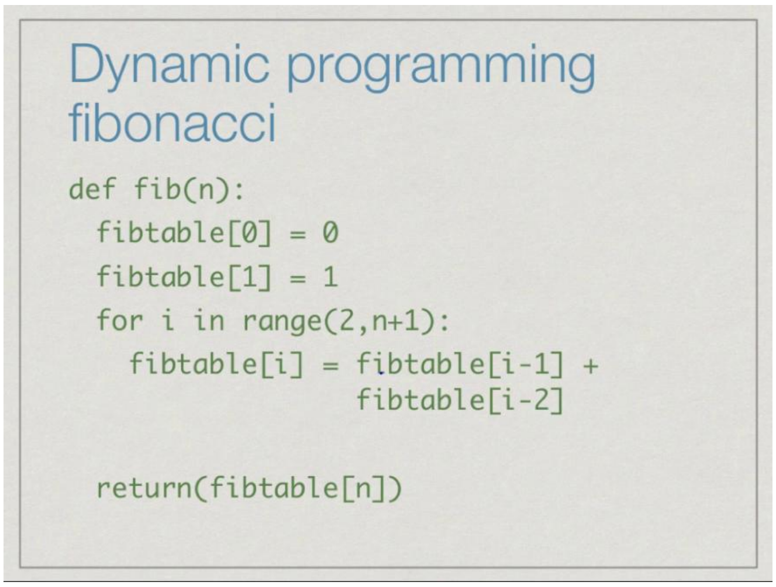
If we look at the sub problem for Fibonacci of 5, for example, it says that it requires all these sub problem but the dependency is very straightforward; 5 depends on 4 and 3; 4 depends on 3 and 2; 3 depends on 2 and 1; 2 depends on 1 and 0; and 0 and 1 have no dependencies. So, we can start at the bottom, and then work ourselves up, we can say Fibonacci of 0, needs no dependencies, so let me write a value for it. Fibonacci of 1 needs no dependency, so let me write a value for it.

Now we see that for Fibonacci of 2 both the things that it needs have been computed. So, I can write fib 2 in the table directly without even going to it from 5, I am not coming down from 5, we are just directly filling up the table, just keeping track of which values depend on which values. So, assuming that we know the function, we can calculate this dependency and just compute that values as and when the dependencies are satisfied.

Now, we have 1 and 2. So, we can compute Fibonacci of 3. So, we just compute it. We have 3 and 4, so we can compute I mean 2 and three. So, we can compute Fibonacci of 4, so we compute. And finally, we have 2 and fib 3 and fib 4, so we can get fib 5. This value as you can see becomes now a linear computation, I can just walk up from 0 to 5 and fill in the values. In fact, this is what we do by hand.

When I can you ask me for the tenth Fibonacci number, I can write it out. I can say 0. 0 plus 1 is 1, and then 1 plus 1 is 2, 2 plus 1 is 3, 5, 8, 13, 21, 34, so clearly it is not a very complicated process as it seems to be when we have this exponential recursion. I can do it on the fly more or less, because all I have to do is keep generating the values in a sequence, and this is the virtue of dynamic programming. It converts this recursion into a kind of iterative process, where you fill up the values that you would normally fill up in the MEM table by recursion to fill them up iteratively starting with the ones which have no dependencies.

(Refer Slide Time: 17:25)



Dynamic programming fibonacci

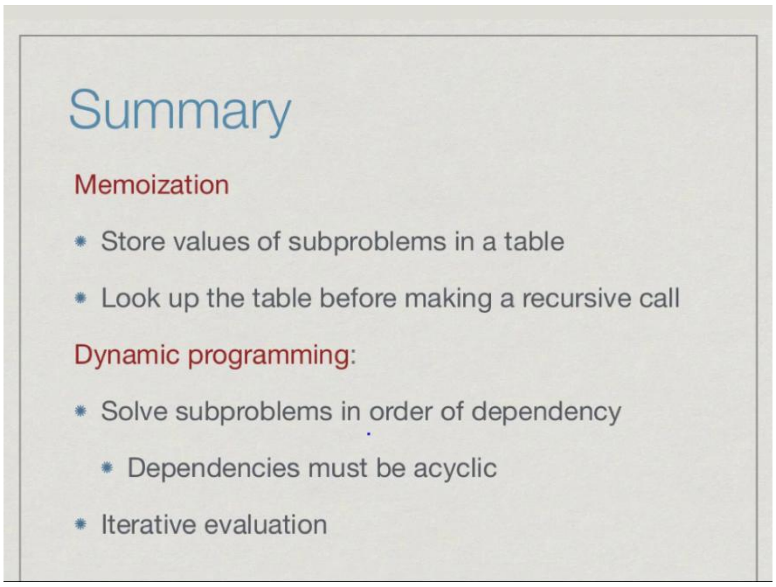
```
def fib(n):
  fibtable[0] = 0
  fibtable[1] = 1
  for i in range(2,n+1):
    fibtable[i] = fibtable[i-1] +
                  fibtable[i-2]

  return(fibtable[n])
```

Then the dynamic programming version of Fibonacci is just this iterative blue. So, it says that you start from with value 0 and 1 to be the values themselves what we earlier said was that if n is 0 or 1 then you return n. So, here we just store it into the table directly; we say fib table of 0 is 0, fib table of 1 is 1.

And then we walk from 2 to n, so in python notation the range end is n plus 1; and at each stage, we just take the ith value to be the sum of the i minus 1, i minus 2 values which we have already computed because we are going in this particular order because we have recognized the dependency order goes from 0 to n. And finally, the answer we need is the nth entry, so we just return that.

(Refer Slide Time: 18:10)



Summary

Memoization

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

Dynamic programming:

- Solve subproblems in order of dependency
    - Dependencies must be acyclic
- Iterative evaluation

To summarize, the basic idea to make naïve recursion more efficient is to never compute something twice. And to never compute something twice, we store the values we compute in what we call a memo table this is called memoization. And we always look up the table before we make a recursive call.

Or this can be further optimized, so we avoid making recursive calls altogether and we just directly fill in the table in dependency order which must be acyclic otherwise this sub problem will not be solvable. And this converts the recursive evaluation into an iterative evaluation, which is often much more efficient.