IIT Madras

ONLINE DEGREE

Computational Thinking

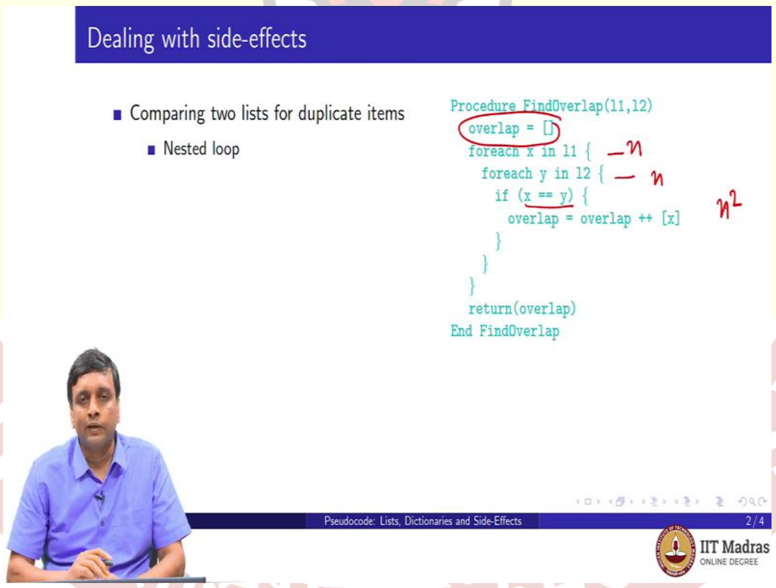
Professor. Madhavan Mukund
Department of Computer Science
Chennai Mathematical Institute
Professor. G. Venkatesh
Indian Institute of Technology

Side effects in pseudocodes for lists and dictionaries

So, when we introduced procedures we had talked about the issue of side effects, So, we have talked about in the context of cards or tables where we said that sometimes when we pass a stack of cards or we pass a table to a procedure, inside the procedure there may be some changes to this table or the stack of cards which will be reflected outside and this change which may or may not be required or may not be intended is called a side effect.

So, now that we are dealing with lists and dictionaries, this problem is something that we need to be aware of and we need to see when it is undesirable and sometimes when it is desirable and make a conscious decision whether we are going to use side effects or not.

(Refer Slide Time: 0:54)



Dealing with side-effects

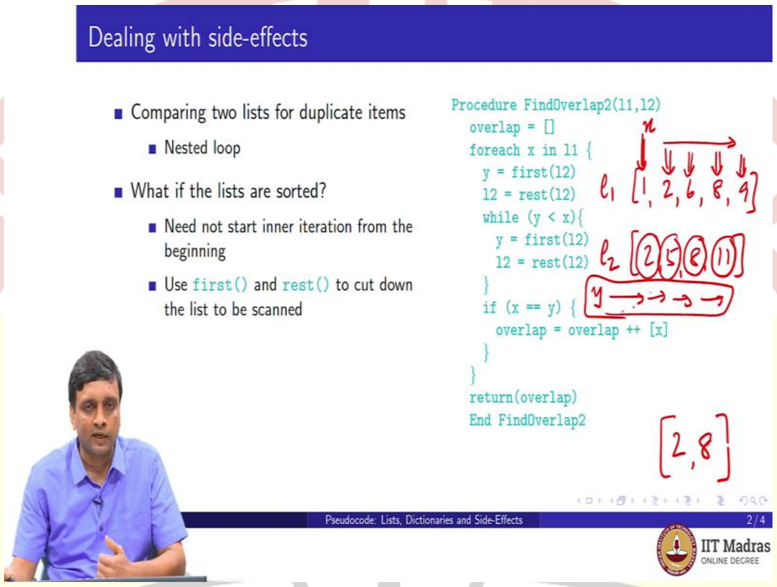
- Comparing two lists for duplicate items
  - Nested loop

```
Procedure FindOverlap(l1,l2)
  overlap = []
  foreach x in l1 {         — n
    foreach y in l2 {       — n
      if (x == y) {                $n^2$
        overlap = overlap ++ [x]
      }
    }
  }
  return(overlap)
End FindOverlap
```

So, let us look at a very simple problem involving list, so supposing we have two lists and we want to check wheather there are items which are duplicated across this list. So, here is a very simple piece of pseudo code for this, so what it says is that we start with the outer list and then we trade through all that outer list and all the elements in a list and we are accumulating a new list call overlap.

So, overlap has all the items which we found which are both in l1 and l2, so every time we see an x and y and x in l1 and y in l2 which are equal we append it to overlap. So, this is a very simple nested loop and what we know is that if l1 and l2 have n elements each, then the outer loop will run n times and the inner loop will also run n time so we have something which will run n square times. So, the question is whether we can do better than this?

(Refer Slide Time: 1:51)



Dealing with side-effects

- Comparing two lists for duplicate items
  - Nested loop
- What if the lists are sorted?
  - Need not start inner iteration from the beginning
  - Use first() and rest() to cut down the list to be scanned

```
Procedure FindOverlap2(l1,l2)
  overlap = []
  foreach x in l1 {
    y = first(l2)
    l2 = rest(l2)
    while (y < x){
      y = first(l2)
      l2 = rest(l2)
    }
    if (x == y) {
      overlap = overlap ++ [x]
    }
  }
  return(overlap)
End FindOverlap2
```

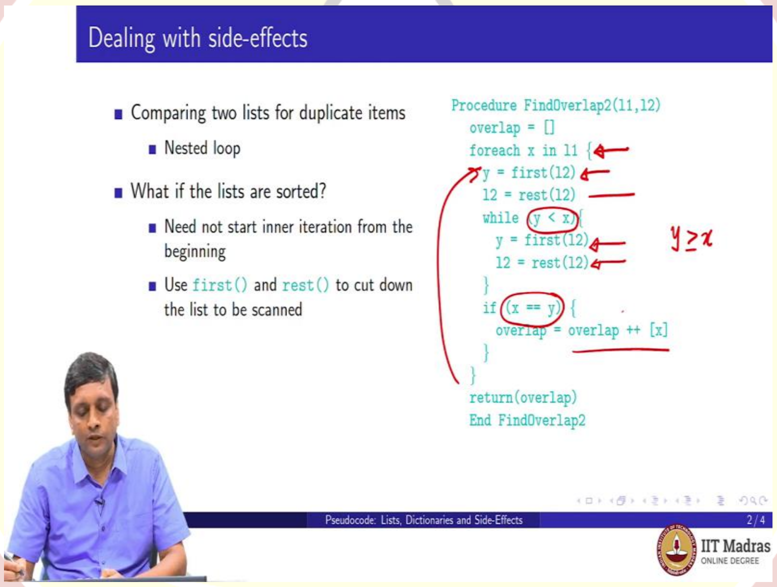
For instance, supposing we know that the list are sorted. So, the list are sorted, so supposing we have list of the forms say 1, 2, 6, 8, 9 and 2, 5, 8, 11. So, supposing so let us assume that this is myl1 and this is myl2, so now what we can do when we are looking at the first x in l1, then we can scan l2 until we find something which is bigger than l1 bigger than or equal to in general because once we sit hit to we know that nothing beyond that can be smaller can be equal to 1 because everything after 2 is going to be bigger.

So, we can kind of optimize our loop by moving x as before but we move y only once, so initially we move it and supported to then we move x to position 2 and y is still at 2, we check that they are equal and we pull out this 2 into our list of common elements or the overlapping elements. Now once we have found it we can move x to 6, now we keep moving y so long as we are smaller than 6, so if it is 5 then we know that 5 is not equal to 6 but there could be something to the right of 5 which is equal to 6, so we go one more step and we stop at 8.

Now we have overshot 6 so we know that 6 is not there, so now we move x down one more level and now we keep it at 8 in the bottom one, so we now know that 8 is a common element. So now, we shift to 9 on the top then we shift here to 11 we have overshot 9, so 9 is not there and since we have finished the outer list, we have computed the overlap completely just 2, 8.

So, what we are trying to do is do this optimization on the second list, the second list where we do not go and start at the beginning each time but we start from where we left off, so we kind of stop the scan of the second list when we reach a number which is bigger than or equal to the current number we are looking at in the first list. And then at that point after checking whether it is equal to we will proceed. So this is what this pseudo code is doing here. So the way it is doing this is basically by walking down the second list using this first and rest.

(Refer Slide Time: 4:09)



Dealing with side-effects

- Comparing two lists for duplicate items
  - Nested loop
- What if the lists are sorted?
  - Need not start inner iteration from the beginning
  - Use first() and rest() to cut down the list to be scanned

```
Procedure FindOverlap2(l1,l2)
  overlap = []
  foreach x in l1 {
    y = first(l2)
    l2 = rest(l2)
    while (y < x) {
      y = first(l2)
      l2 = rest(l2)
    }
    if (x == y) {
      overlap = overlap ++ [x]
    }
  }
  return(overlap)
End FindOverlap2
```

$y \geq x$

Pseudocode: Lists, Dictionaries and Side-Effects 2 / 4

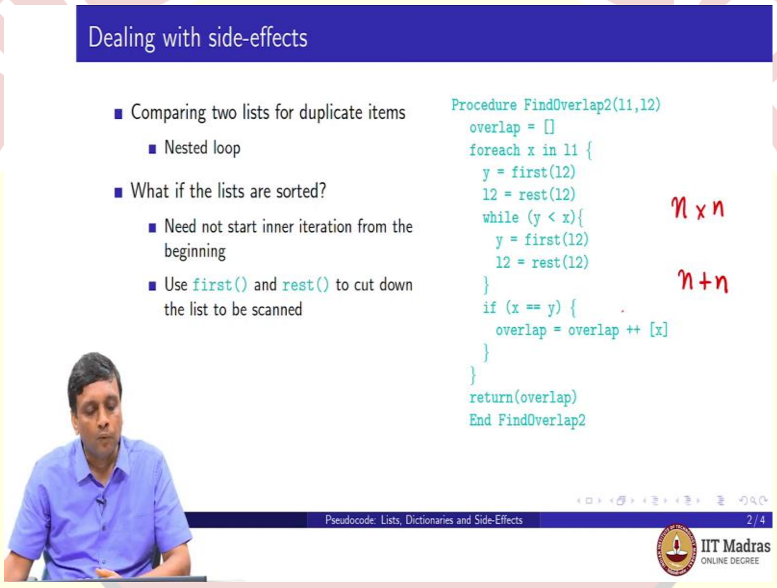
IIT Madras ONLINE DEGREE

So what we do is, we do to the outer loop as before for each x in l1, now we do not an explicit iterating loop through the second list, what we do we pick up the first element and be update l2 to to be the rest, so now l2 has a second element onwards. So now we check whether this first element that we saw is smaller than x, if it is smaller than x, then we need to discard it and proceed to the next element it.

So, we do the same thing again be pickup the first element in now what is the rest of the list and we move l2 down one more step and we keep doing this until we hit a y which is bigger than or equal to x. And once we get a y which is bigger than or equal to x we check whether it is actually equal to x, if it is actually equal to x, then we update our overlap and then we go back here we go back here and now we have process this x either we have process this x by finding a y that was equal or finding a y that was greater than.

So, now we keep y at that position and go back and go back to the next x, so that is exactly what we did in our code before. So, what we are using here now is the fact that updating the second list using this first and rest we are essentially walking down the second list only once so we take any steps to the first list and while we are doing this n steps of the first list we also do n steps cumulatively through the second list. So, each list has seen only once.

(Refer Slide Time: 5:28)



Dealing with side-effects

- Comparing two lists for duplicate items
  - Nested loop
- What if the lists are sorted?
  - Need not start inner iteration from the beginning
  - Use first() and rest() to cut down the list to be scanned

```
Procedure FindOverlap2(l1,l2)
  overlap = []
  foreach x in l1 {
    y = first(l2)
    l2 = rest(l2)
    while (y < x){
      y = first(l2)
      l2 = rest(l2)
    }
    if (x == y) {
      overlap = overlap ++ [x]
    }
  }
  return(overlap)
End FindOverlap2
```

$n \times n$

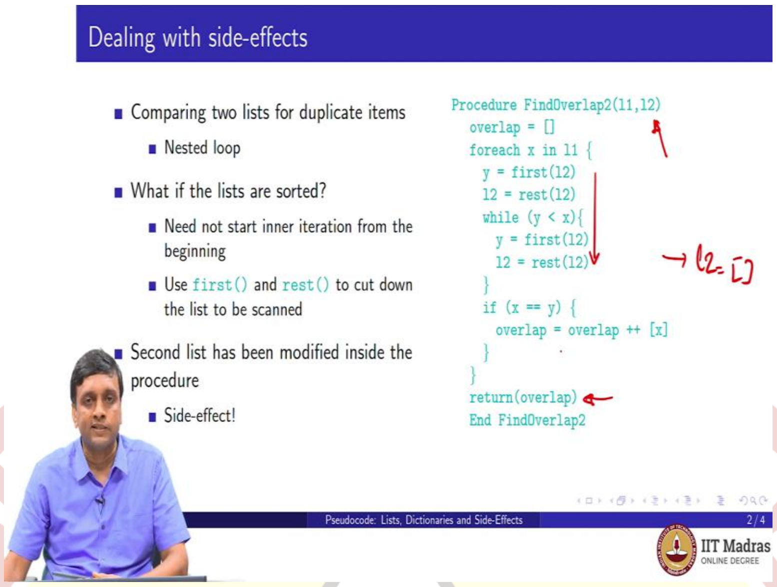
$n + n$

Pseudocode: Lists, Dictionaries and Side-Effects 2 / 4

IIT Madras ONLINE DEGREE

So, instead of doing n times n which is the nested loop we are doing roughly n plus n which is much smaller. So, what is the problem if any with this?

(Refer Slide Time: 5:38)



Dealing with side-effects

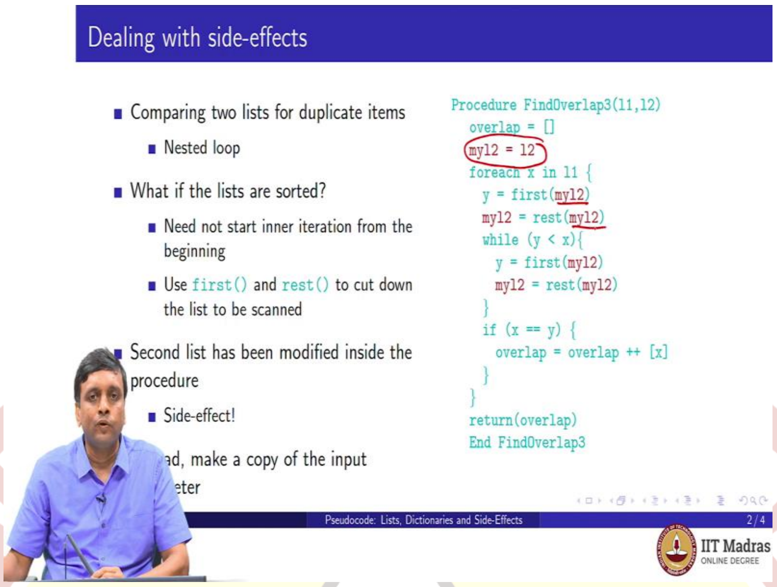
- Comparing two lists for duplicate items
  - Nested loop
- What if the lists are sorted?
  - Need not start inner iteration from the beginning
  - Use first() and rest() to cut down the list to be scanned
- Second list has been modified inside the procedure
  - Side-effect!

```
Procedure FindOverlap2(l1,l2)
  overlap = []
  foreach x in l1 {
    y = first(l2)
    l2 = rest(l2)
    while (y < x){
      y = first(l2)
      l2 = rest(l2)
    }
    if (x == y) {
      overlap = overlap ++ [x]
    }
  }
  return(overlap)
End FindOverlap2
```

Well, the problem is that in the process of doing this repeatedly replacing l2 by the rest of l2 we are consecutively removing the first element until we end up with l2 being the empty list more or less, I mean it might be not quite empty list if we run out of x much before that but in general we have removed certain elements from l2 and l2 was what was given to us when the procedure was called.

So, we are returning in some sense an overlap which is what we wanted to get back but in the process of returning this overlap, we are also an intentionally destroying the second argument and this is a side effect. So, how do we avoid this? Well, this because I presume that when somebody gives me two list, they want those two list to remain intact, there is no reason why finding an overlap should modify this list in anyway. So, it is not expected that there is a side effect, so if I produce a side effect in my procedure it is definitely an unintentional and not desirable.

(Refer Slide Time: 6:38)



Dealing with side-effects

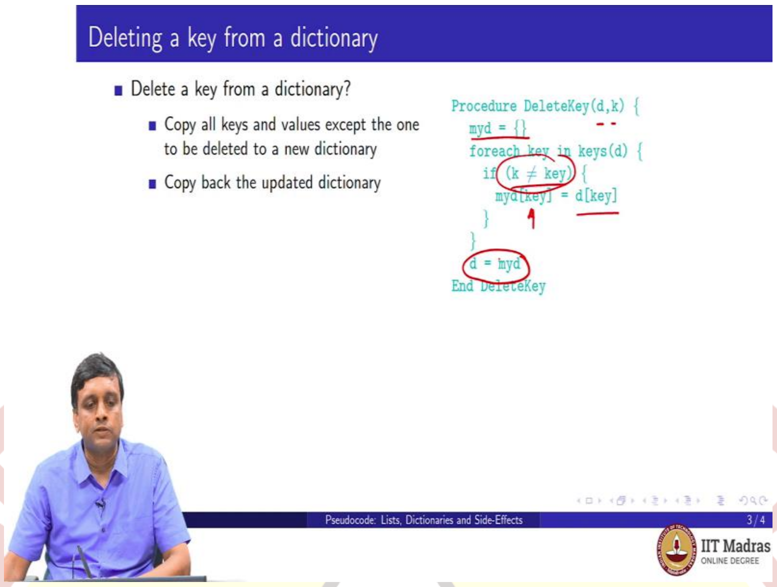
- Comparing two lists for duplicate items
  - Nested loop
- What if the lists are sorted?
  - Need not start inner iteration from the beginning
  - Use first() and rest() to cut down the list to be scanned
- Second list has been modified inside the procedure
  - Side-effect!
- ad, make a copy of the input eter

```
Procedure FindOverlap3(l1,l2)
  overlap = []
  myl2 = l2
  foreach x in l1 {
    y = first(myl2)
    myl2 = rest(myl2)
    while (y < x){
      y = first(myl2)
      myl2 = rest(myl2)
    }
    if (x == y) {
      overlap = overlap ++ [x]
    }
  }
  return(overlap)
End FindOverlap3
```

So, one simple way to do this is whenever you write a procedure which would benefit from doing a side effect would benefit from updating the argument you just make a copy of the argument. So here, we start off by copying l2 into a local list which I will call myl2 and now all the updating that I did earlier on l2 I now do on myl2, so I will take the first element of myl2 I update myl2 to the rest of my myl2 and so on.

So, whatever we were doing with l2 the original argument earlier now we are doing without private copy of the arguments so l2 is untouched so the outside person who was called this procedure does not see any change but in side were able to exploit this fact that by walking down l2 in this manner we can do it more efficiently.

(Refer Slide Time: 7:25)



Deleting a key from a dictionary

- Delete a key from a dictionary?
  - Copy all keys and values except the one to be deleted to a new dictionary
  - Copy back the updated dictionary

```
Procedure DeleteKey(d,k) {
  myd = {}
  foreach key in keys(d) {
    if (k ≠ key) {
      myd[key] = d[key]
    }
  }
  d = myd
End DeleteKey
```

Pseudocode: Lists, Dictionaries and Side-Effects 3 / 4

IIT Madras ONLINE DEGREE

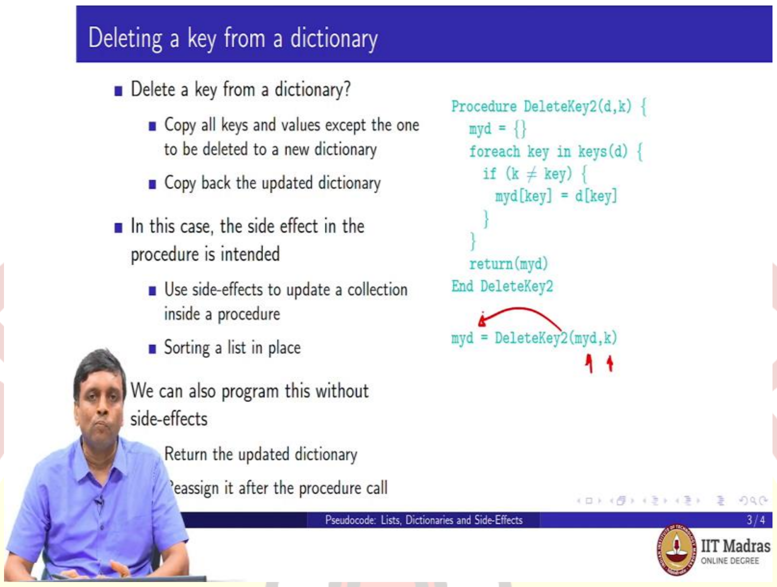
So, let us look at another example. In the previous example where we are trying to find duplicates in two sorted lists, we saw that computationaly it was better to destroy the second list and work it down using first and rest but this was an undesirable side effect from the point of view of the person calling the procedure, but sometimes the procedure may be intended to perform a side effect.

So, consider a thing that we have not seen so far which is how do we remove a key from dictionary, we have seen that if we create a new key by assigning it it will get added to the dictionary implicitly but we do not really have a way to remove a key from the dictionary so let us write a procedure for this. So, here is one procedure for removing a key, so you get the d a dictionary and k a key to be removed.

So, what you do we create a new dictionary called my d, so my d is a new dictionary which is initially empty and now for every key in the dictionary d you check whether it is the same as a key to be deleted or not and if it is not the key to be deleted then you create a key in my d and you assign it the same value that the key has in d. So, we are really copying d into my d except that was keeping the case where keys k is equal to key.

So, k is the value that that I am asked to delete, so as long as that is not the value that I am iterating over in the, when I get all the keys from d, I am done but here at the end I need to update this d, so what I do is a copy back the updated dictionary back into d. So, this is a procedure with a side effect but perhaps this is what the person intended. So, if I wanted to create a procedure which would actually update a dictionary d by deleting a key then it is expected in some sense this should be done a side effects so perhaps this is okay.

(Refer Slide Time: 9:15)



Deleting a key from a dictionary

- Delete a key from a dictionary?
  - Copy all keys and values except the one to be deleted to a new dictionary
  - Copy back the updated dictionary
- In this case, the side effect in the procedure is intended
  - Use side-effects to update a collection inside a procedure
  - Sorting a list in place
- We can also program this without side-effects
  - Return the updated dictionary
  - eassign it after the procedure call

```
Procedure DeleteKey2(d,k) {
  myd = {}
  foreach key in keys(d) {
    if (k ≠ key) {
      myd[key] = d[key]
    }
  }
  return(myd)
End DeleteKey2

myd = DeleteKey2(myd,k)
```

Pseudocode: Lists, Dictionaries and Side-Effects 3 / 4

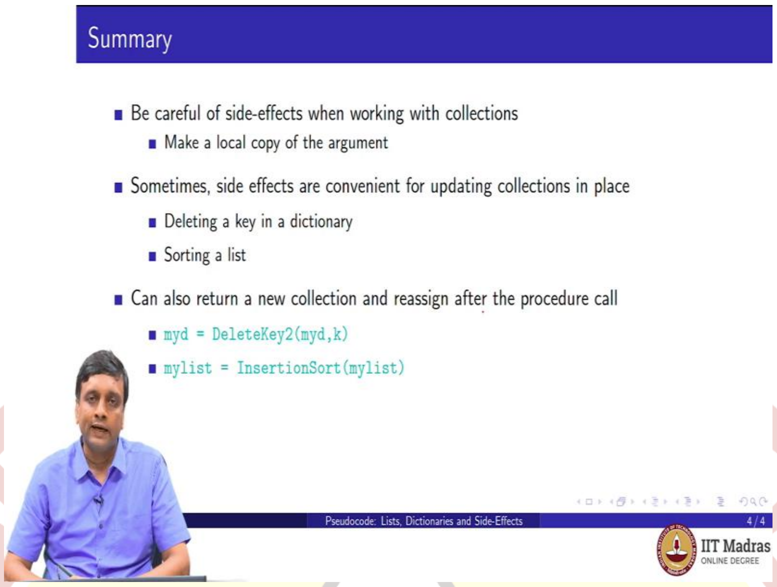
IIT Madras ONLINE DEGREE

So, in some situations like this, if it is documented as the behaviour, I mean so you have to tell the person that this is how this should be used you give me a dictionary and a key to be deleted and I will update that dictionary in place. So, deleting a dictionary another example would be sorting, so there could be a procedure which takes a list and just sorts it by updating it, it does not create a new sorted list.

So, there are situations where the side effect could be intended but you must make sure that the procedure is written in, I mean the documentation of the procedure records this. Now of course you could do this deletion of the key without a side effect so in this situation, what you would do is you would not written the old dictionary after updating but you would return the new dictionary, you return my d which is the dictionary with k deleted.

And now outside the way you would use it is to call the function with your dictionary that you want to update with the key you want to remove, take the value that you get back from the procedure, so the value that you have sent is not by default changed and you reassign it to the value that you started with. So, this is a very common way of writing this kind of things where the procedure is not going to perform a side effect but is going to return a new modified object, then you just reassignment.

(Refer Slide Time: 10:31)



Summary

- Be careful of side-effects when working with collections
  - Make a local copy of the argument
- Sometimes, side effects are convenient for updating collections in place
  - Deleting a key in a dictionary
  - Sorting a list
- Can also return a new collection and reassign after the procedure call
  - `myd = DeleteKey2(myd,k)`
  - `mylist = InsertionSort(mylist)`

Pseudocode: Lists, Dictionaries and Side-Effects 4 / 4

IIT Madras ONLINE DEGREE

So to summarize, we saw side effects in the context of tables or entire piles of cards but now that we are working with collections like list and dictionaries, we should be careful about side effects and we should be aware when a side effect is happening if we want to use a side effect for our computation but we do not want it to be actually reflected in what the users sees, then we can always make a local copy of the argument before performing the update, this way the argument that we get untouched and we just work with a local copy, that is what we saw when we looked at this thing of finding overlaps between two sorted list.

It could be the side effect is intended like updating or deleting a key or sorting a list but if this is the intended behaviour, then we should make sure the procedure is documented in such a way that the person knows that the argument will be updated in place but for such situations you can also create a new collection which is the updated collections, we will take an unsorted list you create a new sorted list you take a dictionary, you want to delete a key you create a new dictionary without that key and then you pass it back and then the person who is calling the procedure can reassign it to the collection that they were using to get the update.