



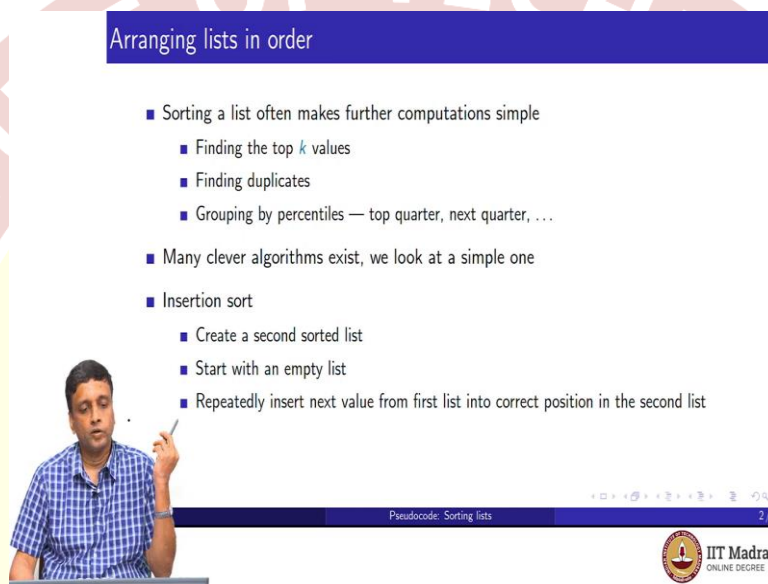
IIT Madras

ONLINE DEGREE

Computational Thinking
Professor Madhavan Mukund
Department of Computer Science
Chennai Mathematical Institute
Professor G Venkatesh
Indian Institute of Technology, Madras
Pseudocode for insertion sort and ordered list

So, now we come to one of the most important things that we need to do with a list which is to sort it.

(Refer Slide Time: 0:18)



The slide is titled "Arranging lists in order" in a blue header. It contains a bulleted list of points:

- Sorting a list often makes further computations simple
 - Finding the top k values
 - Finding duplicates
 - Grouping by percentiles — top quarter, next quarter, ...
- Many clever algorithms exist, we look at a simple one
- Insertion sort
 - Create a second sorted list
 - Start with an empty list
 - Repeatedly insert next value from first list into correct position in the second list

In the bottom left corner of the slide, there is a small video inset showing a man in a blue checkered shirt speaking and gesturing with his hand. At the bottom of the slide, there is a navigation bar with the text "Pseudocode: Sorting lists" and a page indicator "2/5". The IIT Madras logo and "ONLINE DEGREE" text are in the bottom right corner.

So, sorting a list means arranging the elements in order either in ascending order or in descending order assuming of course that the elements in the list can be compared as bigger or smaller than each other. And this is useful because we can use it later on to make other things very simple. Supposing I wanted to find out the top k , we looked at the top 3 marks for instance in a class. So, if we sort the list by marks, then the first 3 values, the highest 3 values in that sorted list will be the top 3 marks.

So, once you have got it sorted you can take the top k values by just looking at the first k or the last k depending on whether it is in descending order or ascending order. So, this is something will become very easy once you have arranged them in order. Another thing that you can find very easily if you arrange things in order is if there are duplicates because if things are in order, then the equal values will come next to each other. So, automatically for instance, if we are looking for 2 students who have the same birthday and we get the list of all the birthdays in the class and we arrange them in order for January to December in ascending order.

Then, if 2 students have the same birthday, they will appear next to each other in this sorted list. So, finding duplicates becomes very easy once you have sorted a list. And finally one thing which we will see in the next example we consider after this lecture is if we want to group elements according to some kind of criterion with respect to the length of the list.

So, for instance, if we want to award grades according to percentiles, we want to say that the first one fourth of the class gets an A, the next one fourth gets a B and so on, then if we have arranged the marks in descending order, then we can take the top one fourth marks and assign an A, the next one fourth assign a B and so on.

So, there are many very natural situations where if you have the list in sorted order, it becomes very easy to continue with the processing. So, as you can imagine sorting is an extremely important problem in computing and there are many many very clever algorithms to do it which we will come across in later courses in this program. But right now let us just look at a most straight forward and naïve thing that we do all the time when we sort by hand. And this is called insertion sort.

So, imagine for instance, that you have a pile of exam papers which have got marks on them and they are being graded and of course, in the process of grading they could have been in any jumbled order. You, the students will be graded in the order in which they submit the marks. So the marks are not going to be in any particular order and now you are asked to sort this. You want to put them say in ascending order or descending order of marks.

So, one way to do this is to take the top most mark, answer paper in the pile and put it aside. Now, this right hand side is going to be my sorted pile. So, as I move things aside, I will make sure that this remains sorted. So, I will, first one is going to be sorted because it is only one paper, every one paper is always sorted. The second paper will be either more or less so depending supposing I want it in ascending order. That is I want the lowest mark on the top and the highest mark on the bottom.

So, then if the mark that I see now is higher, I will put it below, if it is lower I will put it above. Now, I have two things which are sorted, a lower mark and a higher mark. I see the third one, it could come in any of 3 position, so I start in the top and I go down until I find the correct position and I insert it. So, this is why it is called insertion sort. So, you walk from the beginning of the list that is sorted, find the correct position where it should go and then you insert it in that point.

And if you keep repeatedly taking the first pile and inserting it into the second pile, then as you go along the second pile will grow in a sorted fashion and finally your second pile will be a sorted version of the first pile. So, this is called insertion sort. So, now let us see how to implement insertion sort using our pseudocode.

(Refer Slide Time: 3:55)

Inserting into a sorted list

- We have a list l arranged in ascending order
- We want to insert a new element x so that the list remains sorted
- Move items from l to a new list till we find the place to insert x
- Insert x and copy the rest of l
- Be careful to handle boundary conditions
 - l is empty ✓
 - x is smaller than everything in l
 - x is larger than everything in l

Procedure SortedListInsert(l, x)

```

newList = []
inserted = False
foreach z in l
  if (not(inserted)) {
    if ( $x < z$ ) {
      newList = newList ++ [x]
      inserted = True
    }
    newList = newList ++ [z]
  }
if (not(inserted)) {
  newList = newList ++ [x]
}
return(newList)
End SortedListInsert
  
```

Handwritten notes: "Sum", "to insert", "l = [1, 3, 4, 5]", "x = 2.5", "z = 1", "1, 3, 4, 5".

Pseudocode: Sorting lists 3/5 IIT Madras ONLINE DEGREE

So, the main operation in insertion sort is to take this item from the first pile and put it into the second pile, so the second pile is sorted. So, if we have a sorted list and if we have a new element that we want to insert into it, how do we insert it? So, we have a list l arranged let us assume in ascending order, we could also do it in descending order that is very easy as we will see at the end. So, we want to take a new element x and put it into this list which is already in ascending order so that now x is part of the list, but the list is still in ascending order after inserting x . So, x must be put at the correct position.

So, the way we will do this is intuitively to create yet another list. So, we will start moving, basically we have to go down this list to find the position to insert, but as we go down this list we have to kind of hold the things that we have seen so far somewhere. So, the easiest way is to make a new list.

So, I keep putting these into a third pile and then when I reach the position that I want my x to be in, I have moved x to the third pile and then I (re) move the rest. So, I basically find the, I process the list until I find a position where x should be. So, how do I know where x should be?

Well, it has a value and if I see a value which is smaller than x, then I see a value which is bigger than x, then x must be between these two values. So, that is how I check. So, I keep passing all values which are because x has to come in ascending order so any value which is smaller than x must be above x so I can bypass it. And when I come to a value which is bigger than x, then at that point I must first put x and then continue. So, I insert x and then I copy the rest of the list.

Now, there are a couple of things we have to be a bit careful about, what if I am doing it for the first time? When we started our insertion sort of this mark sheet, we had nothing on the right, so we had an empty list which is also sorted by definition because there is nothing in that list.

So, when we take an empty list and we insert it do we insert it properly? Secondly, we have to look at the two boundary cases where x is either smaller than everything in this list, so it must be the very first item to be inserted so I should not accidentally look at the first item only and then look at x or it could be the very last item.

So, what if I finished processing everything and I still have not found where to put x, I should make sure that I get it right. So, these are the 2 things that we have to be, 3 things we have to be a little careful about. So, we have to make sure that our code handles these 3 things carefully. So, here is the code. So, we will call it SortedListInsert. So, we take 2 arguments, so the first argument in this is just the list so, this is our sorted list and this is the value to insert. And we have no idea what x means in terms of how big it is compared to what is there in it.

So, as I said we are going to do this by creating a new list which will have l and the new element x but it will continue to be in sorted order. So, for this it will be convenient to keep track. Remember, we had this thing which is that we insert x and then we copy the rest of l. So, once we have put x in, we can stop checking, we just have to move everything. So, to keep track of that fact whether we have copied, whether we have inserted x or not, we will keep track of this Boolean value called inserted. So, initially we will say we have not yet inserted x, so inserted is false.

So, now we will do our usual foreach loop. So, for every z in our sorted list l, if we have not yet inserted x, we have to check whether or not we are reached a point where we should insert x. So, so long as z is smaller than or equal to x we can keep going. So, only if x is smaller than z so imagine let us just do a complete example supposing l is 1, 3, 4, 7, 9 and I

want to insert x is equal to 5. So, when I first look at z, z is first 1, so now since 1 is smaller than 5 I do not have to worry I can move this out. So, I can start a new list to 1 and this one is gone.

Now, I look at 3 and then I see again 3 is smaller than 5, this is so 3 is smaller than 5 so I move 3 here. then I look at 4, again 4 is smaller than 5 so I look here now I pick up 7 and at this point 5 which is x is smaller than my z so now I must put 5 here and then continue with 7 and 9 so that is the idea.

So, if x is smaller than z and we first x into the new list before we put z. so, finally we are always going to put the z that we are looking at into the new list but whether we put it immediately or wait for x depends on whether we are choosing x at this point or not, but we choose x only once.

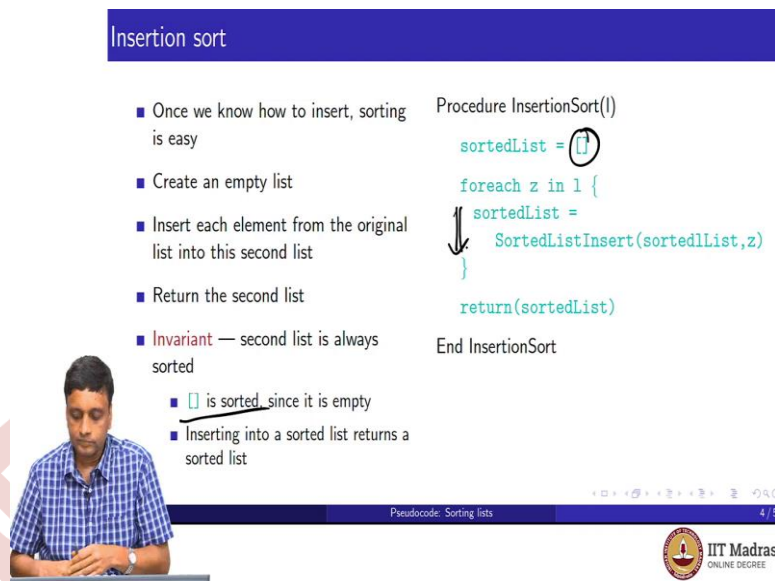
So, as soon as we choose x we say that inserted is true. And that means if inserted is true the next time we will skip this entire thing we will just directly move z, copy the rest as it were. So, this means it will be start inserted is false we will keep examining each element and after we examine it we may decide to move it when we decide we have to put x then we will set inserted to true move x and then the rest we will just blindly copy. So, this is how it works.

Now, we have to check what happens in these various conditions. So, the first condition is that l is empty, now if l is empty then this foreach loop here does not execute at all, so we directly come to this statement. At this statement, it says okay if you have not yet inserted x, then you insert it now. So, basically it says that the empty list will then get appended by x and we are fine.

If x is smaller than everything else in l, then it is no problem because the very first z that you see will be bigger than x and we will do it. So, that is not a problem. And if x is larger than everything in l, it will be similar to the empty list case. We will process this entire foreach loop, we would have moved every z and never moved x because every z that we saw was smaller than equal to x.

So, finally at the end of the foreach loop, found will still be false. So, that means we still have not moved x and that means x must come at the end. So, again this if at the end which says if you have not yet inserted, now insert it. So, this is our insertion procedure which takes a sorted list and puts a new value in at the correct place.

(Refer Slide Time: 10:08)



Insertion sort

- Once we know how to insert, sorting is easy
- Create an empty list
- Insert each element from the original list into this second list
- Return the second list
- **Invariant** — second list is always sorted
 - is sorted since it is empty
 - Inserting into a sorted list returns a sorted list

```
Procedure InsertionSort(l)
    sortedList = []
    foreach z in l {
        sortedList = SortedListInsert(sortedList, z)
    }
    return(sortedList)
End InsertionSort
```

Pseudocode: Sorting lists 4/5

IIT Madras
ONLINE DEGREE

So, once we have this were in business, because once we have know how to insert, then we just have to take a list and as we saw in the beginning with that marks list, so we will take the unsorted marks, the answer papers and we create an empty list which is sorted and we keep shifting from here to there and inserting at every point.

So, we start with a sorted list which is empty and then for every list z in my unsorted list, I just insert it into the sorted list and update the sorted list with that new copy. So, one by one I am inserting into the sorted list and replacing the sorted list by the new sorted list. So, that is all that is happening.

And at the end, the claim is that the new list that I have created is the final answer and I will return that as my outcome. So, that is the sorted list. So, it is perhaps a little bit instructive to argue why this is true? So, what technically why this is true is because we are maintaining what is called an invariant. So, that second list that we have, we have an unsorted list and we are updating a second list.

Now, in that second list we are using this insert procedure so the claim was that an insert procedure will take a sorted list and add an element and return a sorted list with the element added. For that to happen the list that we have should be sorted to begin with. So, fortunately we start with this empty list and the empty list is always sorted. So, to begin with that list is sorted. Since the insert procedure preserves the sorting at each stage when we do this calling the sorted list insert, it takes a sorted list and gives back a new sorted list.

So, therefore, this sorted list is an invariant. The fact that the second list is sorted is an invariant. So, you actually have to establish that by arguing that this sorted list insert procedure that we wrote in the previous slide actually preserves the sortedness property. Once you have that then it is very clear that this does an insertion sort because it produces systematically a new list which has all the elements because you have done a foreach. Every z has gone into that list. So, it has all the elements and they are sorted.

(Refer Slide Time: 12:08)

Summary

- Sorting is an important pre-processing step
- Insertion sort is a natural sorting algorithm
- Repeatedly insert each item of the original list into a new sorted list
- We assumed that the list is sorted in ascending order
- Reverse the comparisons to sort in descending order

Inserting into a sorted list

- We have a list l arranged in ascending order
- We want to insert a new element x so that the list remains sorted
- Move items from l to a new list till we find the place to insert x
- Insert x and copy the rest of l
- Be careful to handle boundary conditions
 - l is empty ✓
 - x is smaller than everything in l
 - x is larger than everything in l

Procedure SortedListInsert(l, x)

```

newList = []
inserted = False
foreach z in l
  if (not(inserted)) {
    if ( $x < z$ ) {
      newList = newList ++ [x]
      inserted = true
    }
  }
  newList = newList ++ [z]
if (not(inserted)) {
  newList = newList ++ [x]
}
return(newList)
End SortedListInsert

```

Handwritten notes:

- $l = [1, 3, 4, 5]$
- $x = 2$
- Resulting list: $[1, 3, 4, 5]$ with 2 inserted at index 1.

So, what we have seen is that sorting is an important pre-processing step so we can find the k largest elements, we can for example, do this duplicate detection and all that. And concretely we have looked at insertion sort. So, insertion sort as a human being is a very natural algorithm, it turns out computationally is not a tribally efficient algorithm, there are cleverer

algorithms that do it faster than this, but that is not the focus of what we are doing now, we are not really interested in that.

But we are really interested in making sure that we have an effective way to sort a list when we need to and the way there insertion sort works is by repeatedly inserting an element into an existing sorted list starting with an empty list. So, the other thing that I had mentioned is that and our examples we said that the list is sorted in ascending order so that is not a big issue, all we have to do to make it sort in descending order is actually, in the in this condition.

So, we check whether x is less than z and then we decide to move it into the new list. When we insert, instead if we check when x is greater than z and move it, then it will actually sort, you can check in the reverse order. So, all we need to do is invert that condition inside the insert procedure and if you reverse that comparison, then the resulting sorting order will go from ascending to descending.

