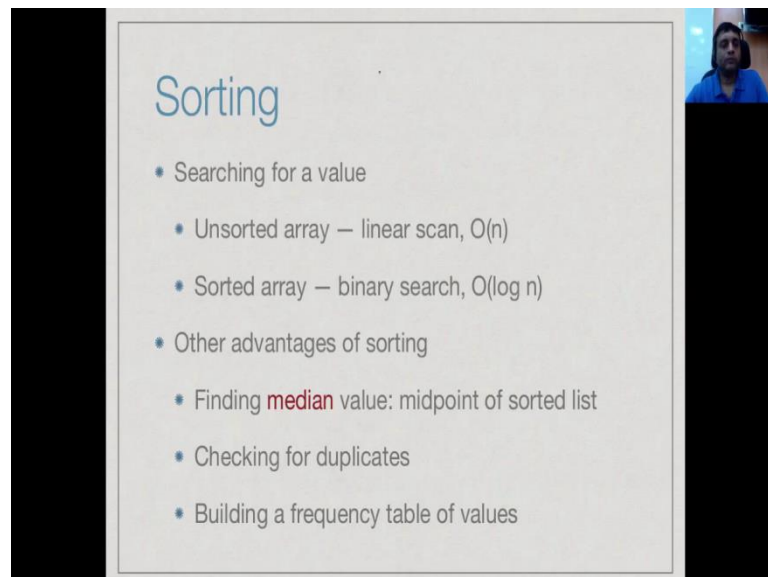**Design and Analysis of Algorithms**
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 02
Module - 04
Lecture – 12
Insertion Sort

So, let us continue with a discussion of sorting, and look at another simple sorting algorithm.
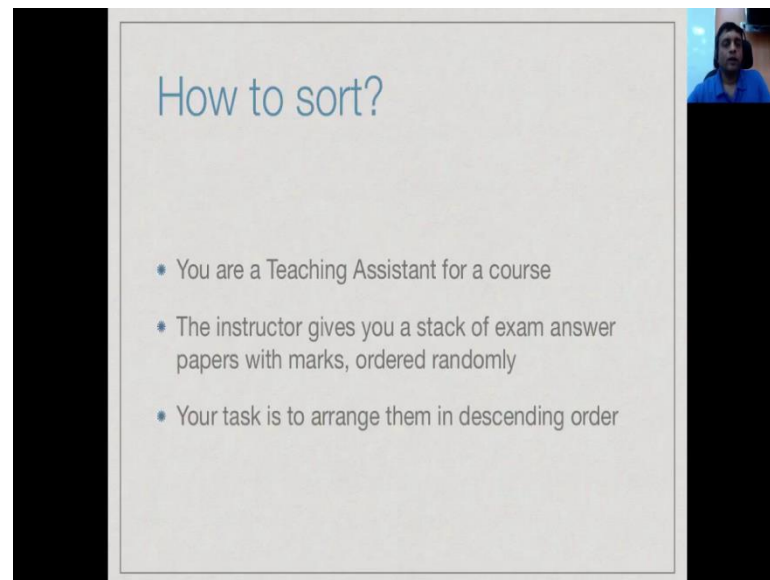
(Refer Slide Time: 00:06)

So, as we said before that are many more motivation for sorting; starting from searching, to removing duplicates, to computing some statistical properties, such as frequency tables, and so on.
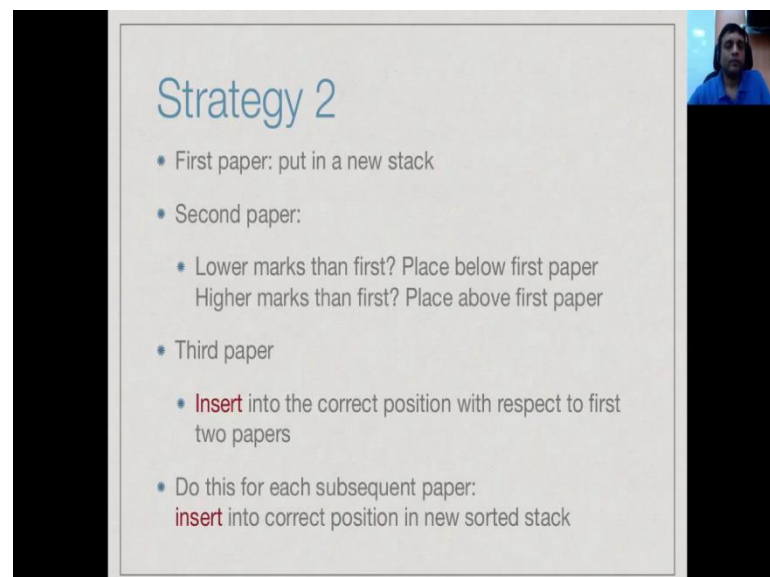
(Refer Slide Time: 00:18)

And the example that we have in hand, is one way we are ask to sort a bunch of exam papers in descending order of marks.

(Refer Slide Time: 00:28)

So, second strategy to sort this bunch of a exam papers would be the following. So, you take the top most paper in this stack that you have, and create a new stack that. Now you take the second paper and compare it to the first paper. If the mark is bigger, you put it about, because you wanted it in descending order. If the mark is smaller, you put it below. So, after this step, you have two stacks of papers in descending order. Now the

stack of two papers rather. And now you take the third paper, and now you see where it fits with respect to the first two; either it goes all the way to the bottom or it goes between the two or the all the way go on . In this way at each point you pick up the top most paper in the unsorted stack, and you inserted it into is correct position, in the sorted stack that you have building up.
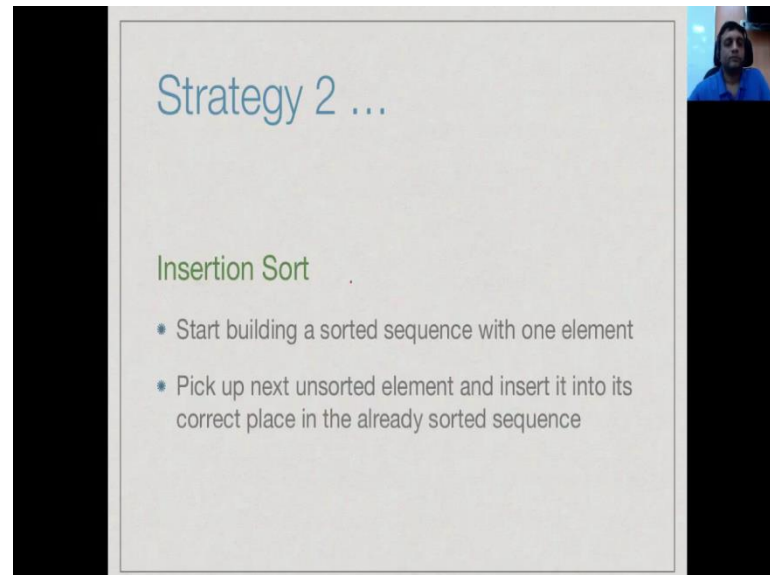
(Refer Slide Time: 01:18)
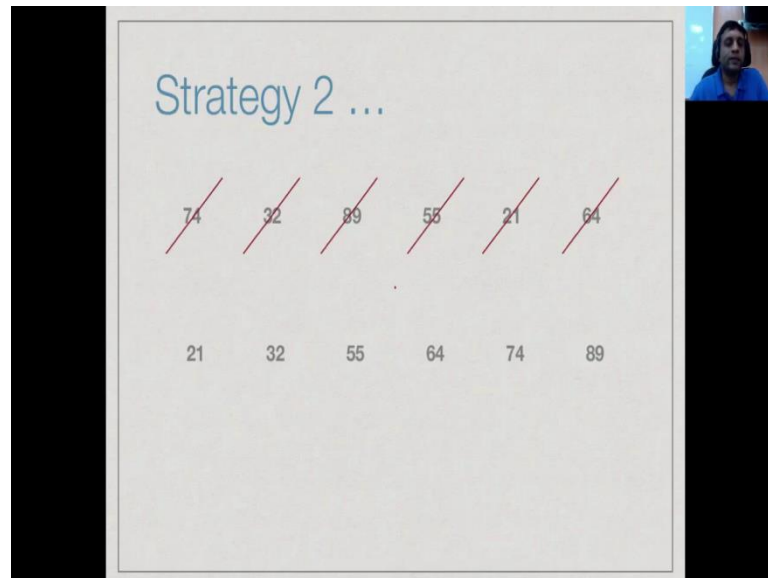
So, let see how should work. So, supposing we have a rolled array of the unsorted elements. So, what we do is, we start with the very first element namely 74. So, we take 74, and we start a new stack. Now we have to take. Now the top most elements at this point is now 32, to the left most in this case. So, now we have to take 32, and since it is smaller it should go to the left of 74. So, we do this, so we now get this array. Now we look at the next element; namely 89. And once again it must go with the appropriate position with respect these two, so it must go to the right of the 74, so we get this array. Now we take 55, and we have to find out where it goes. So we can start at end and say it is not here, it must go to the left of this.

Then we finally, find that this is a correct place for it to go. So, this is the insertion step. We take each element, and we walk down to the point where we want to insert. So, 55 comes between 32 and 74. Now where it is 21 go. Well if we try it insert it, it turns out, it must go all the way to beginning, because it is smaller then everything that we have so far. So, this is the next step. And finally, when we do 64, it will come between 55 and 74.

So, at each step we pick up next element, and we insert it into the already sorted segments that we have created with all the previous elements.

(Refer Slide Time: 02:42)
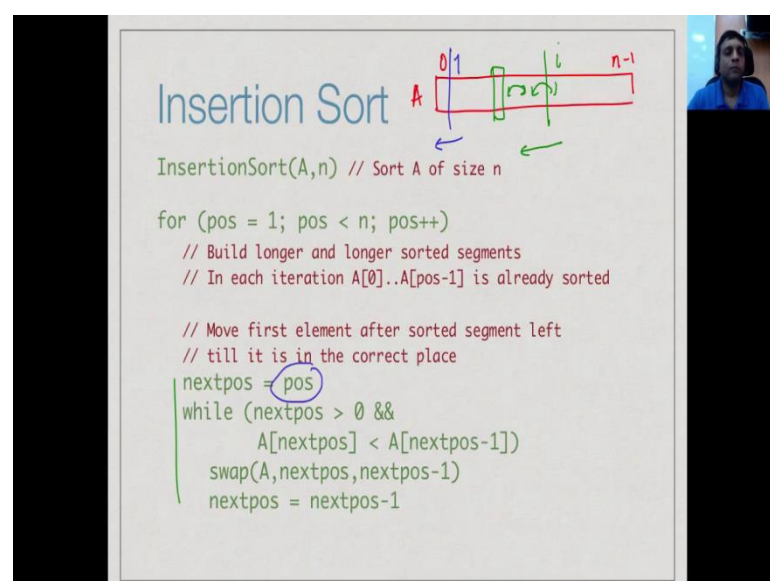
So, we start by building a sorted sequence with one element. So, this is called insertion sort. And we insert each unsorted element into the correct place in the already unsorted sequence. So, it is because of the insertion step, that every element is inserted into its correct place. This is called insertion sort.

(Refer Slide Time: 03:05)

So, this is a very straight forward iterative implementation again. So, what will do is, we

have an initial array a, and position 0 to n minus 1. So, what will do; of course, is at the beginning we have to nothing. So, we cannot assume that we actually start with position 1. So, we start with position 1, and then we look backwards. So, we start with a current position, and we work backwards, and so long as value that we are looking at, is smaller than the value to its left. We keep moving. So, we have assumed that we have this swap operation, which basically takes two positions, an array and exchanges them. So, swap a nextpos nextpos minus 1, means take the value a nextpos, take the value of a nextpos minus 1, and swap them. As we expand at the beginning, we can assume that such things are basic operation is convenient for us to express algorithm like it, as adding this only adds a constant factor to the number of overall operation that we have to do, which we can ignore an ((Refer Time: 04:06)) complexity.

So, we swap each element with element on a left, so long as it is, the element on a left smaller, and we stop when we come to a position where the value on the left, is greater than are equal to the current value, at this point we stop. So, this brings the value that we started with, is in correct position. So, in general, if we had; say done up to sum i and then I start walk in backward. Then so long as i is smaller than i minus 1, I will exchange. I will keep exchanging, until I reach a position, where a find that the values on the left, are smaller than this value, and I will stop. So, this is a basic loop, and I do this for all elements. So, for each element I have to insert it. So, first time I have to insert it in the smaller segment. As I go along, the segments we have to insert it in, become longer and longer.
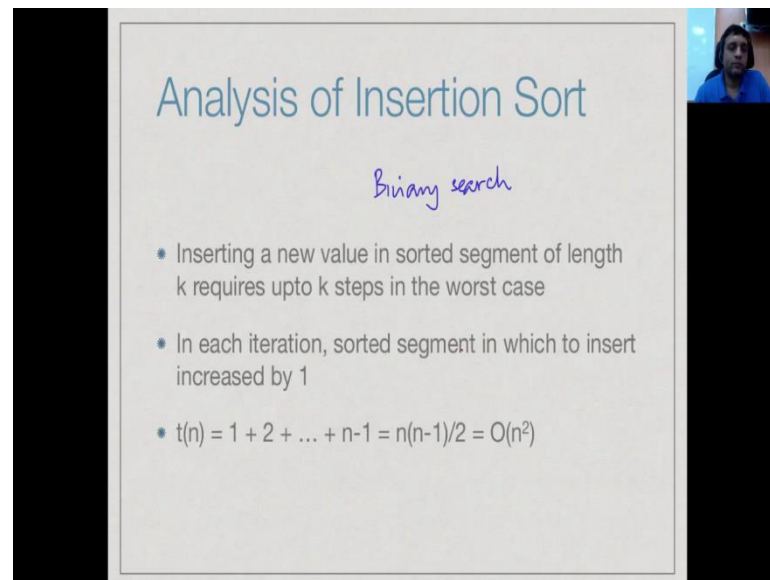
(Refer Slide Time: 04:54)
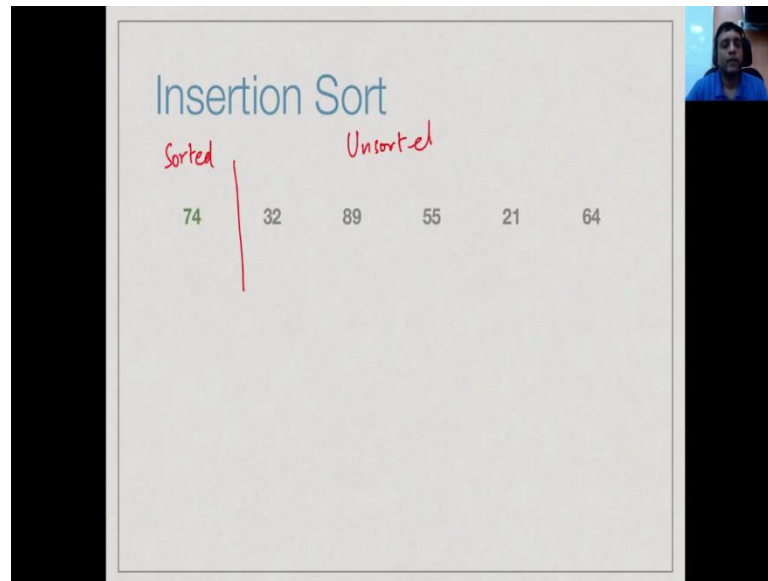
So, we can see now how this works, on an array given to us. So, we first of all start with, looking at this segment. So, this is my initial configuration. So, this is sorted, and this is unsorted. So, now, I will immediately see the 32, is bigger than 74, so I will exchange, and then because it reaches the beginning. So, one of the condition in that loop, is that if I reach beginning of the loop, so nextpos is equal to 0, I will also stop, if I found it to the left most position then the loops terminates . So, having done that, then I have this. So, now I must try to insert 89 into this. So, I will find that 89 is already bigger than 74, nothing is to be done. So, the first nontrivial step that happens, is with 55. So, when I do 55, I compare it to the 89. I find that 89 is bigger than 55, I will exchange them. Now I will compare 55 with element once left 74.

And then again it is the wrong way, so I will exchange. Finally, having found that 55 is now bigger than 32, I will stop. So, this is now the end of the space. Now I will look at the next element to do, which is 21. So, I will look at 21. So, 21 will get exchanged with 89. Then 21 will exchange, so it will get exchanged all the way to the left, so I will exchange 21 which 74. Then I will exchange 21 with 55. Then I will exchange 21 and 32. And now once again I will stop, because every (( )) left most position, there is nothing to the left. And the last round I will take 64. So, now, this part is all sorted. So, I will take 64 and try to insert it here, so it will swap with 89. It will swap with 74, and then stop. This is how insertion sort works. This is a very intuitive sort. If you take a pack of card and try to sort it, typically this is how you would sort.

(Refer Slide Time: 06:59)

So, if you look at the analysis, it is quite similar to selection sort that we sort before. So, inserting a value means we have to walk down to that segment till the very end. Now of course, one made argue, that to find the position to insert, you can use binary search. We need not go one element at a time. If you want to actually find the position where it must go, we can use binary search, but even if you find the position where it must go in logarithmic time. You need to shift all the elements, and that is what really takes linear time. You need to actually make space for this. So in the worst case, you might have to put it in the left most position.

So, all the k elements which are already there must be shifted right by 1, and that takes case steps. So, binary search, although it can help as find the position faster, does not really help us to implement insert any faster. So, insert takes linear time for a segment, and this segment keep growing. Initially I want to insert a 1 into segment of size 1, then a 2 into segment of size 2 and so on. So, I have 1 plus 2 plus 3 up to n minus 1; finally, a n minus 1 must be inserted in this segment a 0 2 a n minus 2, which is a length n minus 1. So, again I have t of n is 1 plus 2 up to n minus 1, and this is just variation of this summation we have seen many times before, is n into n minus 1 by 2. So, this is again an order n square search.
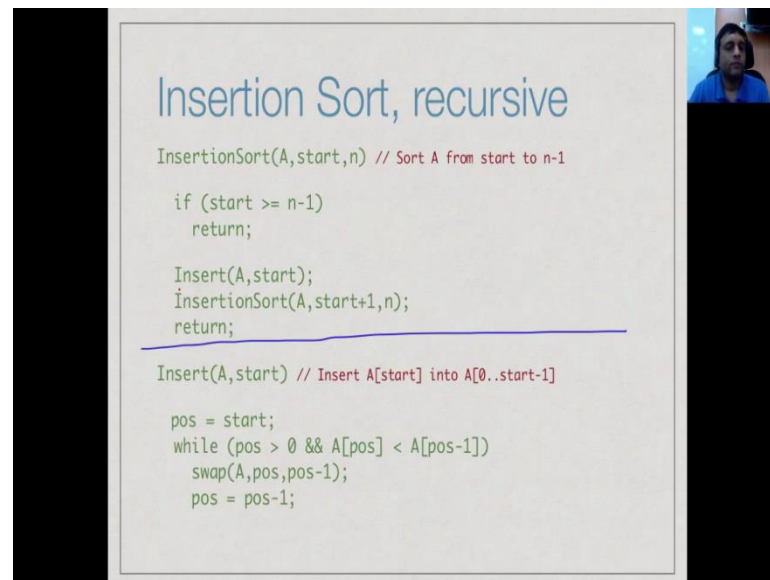
(Refer Slide Time: 08:16)

So, once again as we saw for selection sort. ==There is a natural way to think of insertion sort as a recursive thing.== We sort part of the array, and then we insert an element into that to grow it. So, in this case we think of the array as being into components. So, we have a sorted portion, and an unsorted portion. So, what we do is, we take the first element here and insert it and then we recursively apply the algorithm to the rest of the unsorted portion. So, if a 0 to i minus 1. So, this is position i and this is position i minus 1. So, i minus 1 is the last sorted position, i is the first unsorted position. Then we insert a i into the sorted portion. And then we recursively sorted a i plus 1 onwards. And once again when i is the actually here, at n minus 1, then we do not have to do anything, so we can just trivially return.

(Refer Slide Time: 09:17)

So, now we have recursive formulation in two parts. So, we have insertion sort itself, which says sort the unsorted segment from start to n minus 1. So, if start is already in at n minus 1 return; otherwise insert the value at position start into the rest of a, which you will see if below. And then recursively sort the rest of the array from sort plus 1 onwards. So, what is the insert do? Well it starts if the position start and tries to insert it into the segment 0 to start minus 1. So, it works backwards exactly as we have done it in iterative thing. It finds the first position such that the value on the left, is at least as smalls as the value currently looking for and stops there. So, this insert is basically what is the body of the iterate loop, but insert doing an outer loop, we do it recursively. So, how much time does this take.
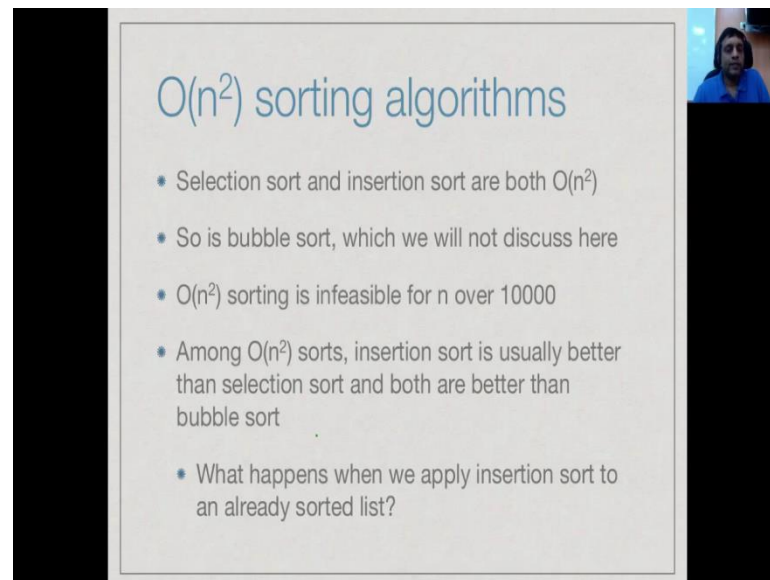
(Refer Slide Time: 10:10)

Once again this is just give you practice in looking at recursively algorithms and writing down the analysis. So, whenever we have a recursively algorithm we write a recurrence; that is, we write of formulation of t of n in terms of smaller values of t. So, let us try to analyze this recursively algorithm. So, we will analyze it, looking at a slightly differently from the way we have formulated the algorithm. So, if you want to sort a 0 to a n minus 1, then what we are doing is, we are recursively sorting this segment, and then inserting this value. So, it takes time t n to sort the entire thing. This breaks of it taking time t n minus 1 to sort the first part up to n minus 2, and then n minus 1 step to the insert right. So, we get same recurrence as we did for selection sort t of n is t of n this is the insert space, and plus t of n minus 1 which is the recursive phase.

And if we expand this out exactly as we did before, we get n minus 1 plus n minus 2 down to n, which is n into n minus 1 by 2, which is order n square. So, again there is no different between time for the recursive and iterative version; except the one should keep in mind in general that, recursive calls are more expansive then iterative loops. We will come back to this point a little later, but otherwise if we count recursive calls calling a function is a basic operation, then there is no difference between the recursive and the iterative version. The recursive version is sometimes easier to conceptually understand and to code.

So, what we have seen is that, the two natural algorithms that we would typically applied when we do something manually; selection sort and insertion sort are both order n square. There is another algorithm which you may come across which we will not discuss in this course called bubble sort, which not something that we would do naturally, but it is seems to be a favorite way of describing how to do the things on a computer. So, bubble sort basically does this insertion kind of swapping, except it takes the maximum or minimum value depending on which you want to do. Say it takes the maximum value and moves it to one end of the array. So, then you have like selection sort, the largest value at one end. Then you go back to the beginning, and again you keep looking at adjusting values, and take the second largest value to the second largest position and so on. So, bubble sort is also N Square, but as we have seen n square algorithms are not really feasible for large values of n. So, if we have n above about 10000, somewhere between 10 to the 4 in 10 to the 5, then n square is going to take several hundred seconds to execute, and therefore, this is not going to be useful for large bodies of data.

But this is not to say that these algorithms are all equally bad. In particular it turns out that if actually look at experimental evidence, then n insertion sort usually behaves better than selection sort, and both of them are better than bubble sort. And to think about why insertion sort behaves well, just imagine what happens when we applied insertion sort to an already sorted list. Whenever we want to insert something into the already sorted list,

we will find that is already in that correct position. So, the bottle neck, which is the insert phase, is the bottle neck. The insert face will basically terminate after one comparison. So, insertion sort will essentially become linear time, if you applied to on already sorted it. So, this is while many situation; insertion sort actually behaves a little better than other order n square sorts, but in general order n square sorting algorithms are not acceptable, and we would like to loop at remote clever ways of sorting a data, because in order to sort large volumes of data this is impractical.