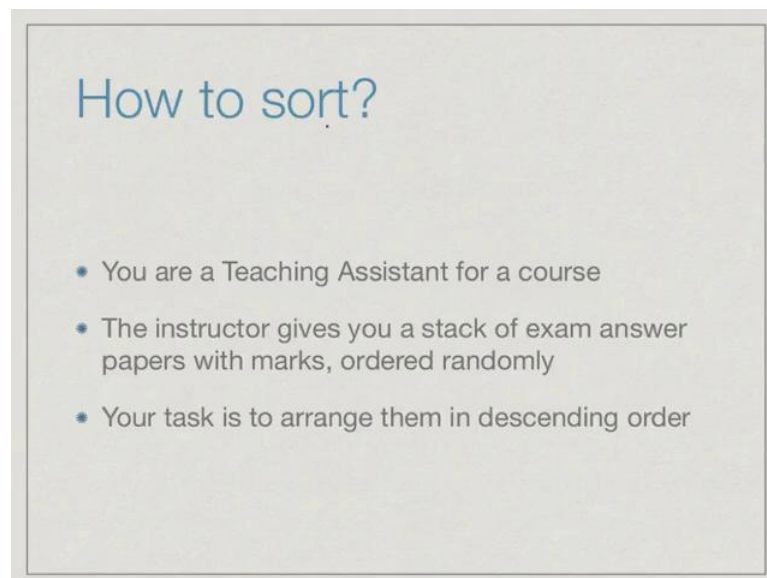


Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 03
Lecture - 07
Insertion Sort

In the previous lecture we saw one natural strategy for sorting, which you would apply when we do something by hand namely selection sort.

(Refer Slide Time: 00:02)

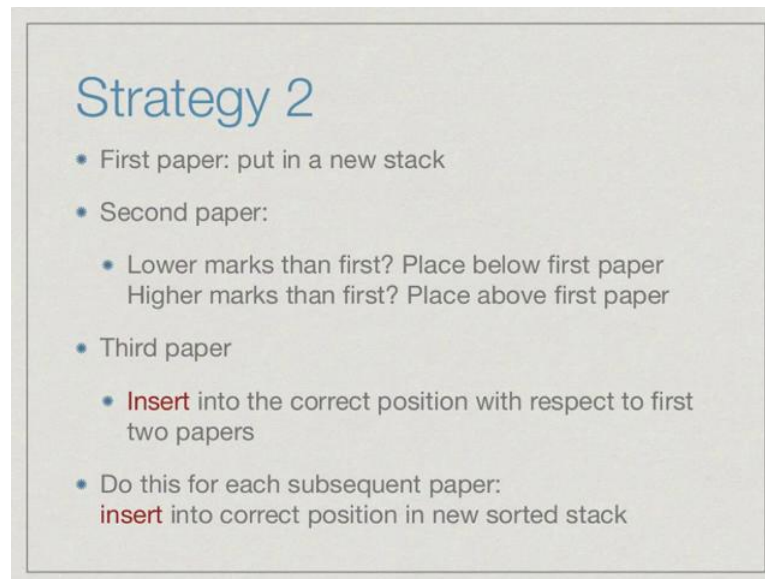


How to sort?

- You are a Teaching Assistant for a course
- The instructor gives you a stack of exam answer papers with marks, ordered randomly
- Your task is to arrange them in descending order

Now let us look at another natural strategy which all of us use at some point. So, the second strategy is as follows:

(Refer Slide Time: 00:17)



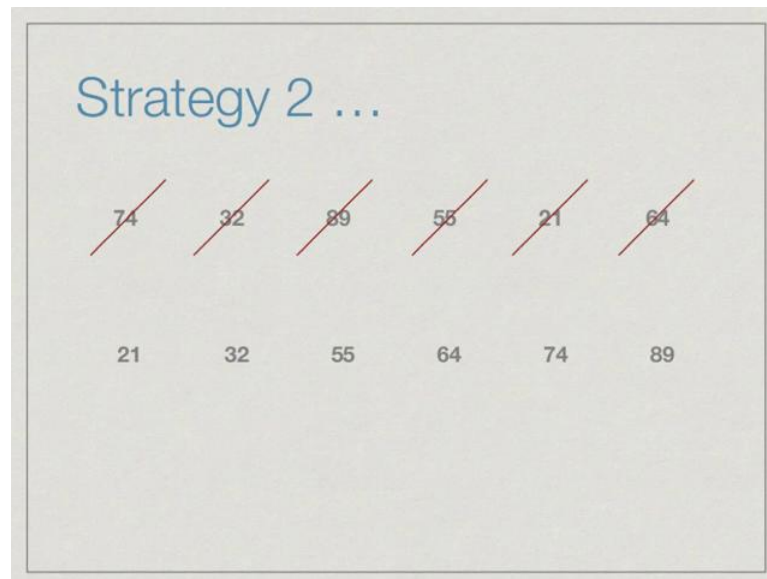
Strategy 2

- First paper: put in a new stack
- Second paper:
 - Lower marks than first? Place below first paper
 - Higher marks than first? Place above first paper
- Third paper
 - **Insert** into the correct position with respect to first two papers
- Do this for each subsequent paper:
insert into correct position in new sorted stack

We have now a stack of papers remember with marks on them and we have to compute a new stack which has this marks arranged in descending order from top to bottom. So, we will take the first paper of the stack we have and create a new stack by definition this new stack is now sorted because it has only one paper. Now we pick the second paper from the old stack and we look at its marks as compared to the first paper that we pulled out. If it is smaller, we put it below; if it is higher, we put it above. So, in this process, we now have the new stack of two papers arranged in descending order.

What do we do with the third paper, **well** the third paper can be in one of three positions; it can either be bigger than the two we saw before. So it can go on top, or it could be in between the two, or it could go below. So, what we do is we scan from top to bottom and so longer if it is smaller than the paper we have seen, we push it down until we find a place where it **fits**. We insert the paper that we pick up next into the correct position into the already sorted stack we **are** building. So, keep doing this for each subsequent paper, we will take the **fourth** paper **and insert** into a correct position among the remaining three and so on.

(Refer Slide Time: 01:31)



This is obviously called insertion sort. So, let us see how it would work. So, what we do with this same list that we had for selection sort is we **will** pick up the first value and move **it** to the new stack saying now I have a new stack which has exactly one value namely 74. Then when I pick up 32, since 32 smaller than 74, I push it to the left of 74. Now 89 is bigger than both, so I keep it on top of the stack at the right end; 55, I have to now look with respect to 89 and 74, so it is smaller than 89. So, it goes to the left of 89 then I look at 74 it is smaller than 74 it goes to the left of that.

So, eventually it settles down as 32, 55, 74, 89. 21, similarly I have to start from the top and say it is smaller than 89 smaller than 74 smaller than 55 smaller than 32, so it goes all the way to the left. And finally, 64 will move down to positions past 84 and 89 and 74, but it will stop above 55. So, this is how insertion sort would build up a new list. You keep picking up the next value and inserting it into the already sorted list that you had before.

(Refer Slide Time: 02:44)

Strategy 2 ...

Insertion Sort

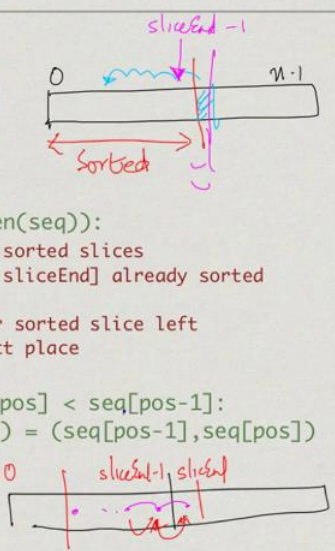
- Start building a sorted sequence with one element
- Pick up next unsorted element and insert it into its correct place in the already sorted sequence

We start building a sorted sequence with one element pick up the next unsorted element and insert it in to a correct place into the already sorted sequence.

(Refer Slide Time: 02:56)

Insertion Sort

```
def InsertionSort(seq):  
    for sliceEnd in range(len(seq)):  
        # Build longer and longer sorted slices  
        # In each iteration seq[0:sliceEnd] already sorted  
  
        # Move first element after sorted slice left  
        # till it is in the correct place  
        pos = sliceEnd  
        while pos > 0 and seq[pos] < seq[pos-1]:  
            (seq[pos], seq[pos-1]) = (seq[pos-1], seq[pos])  
            pos = pos-1
```



We can do this as we did with insertion sort without building a new sequence, and here is a function insertion sort defined in python which does this. So, what we will assume is

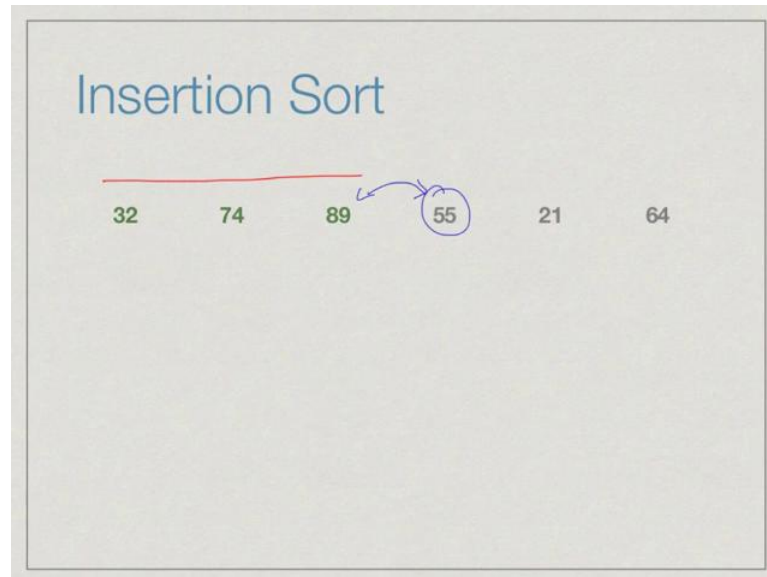
that at any given point, we have our sequence from 0 to $n - 1$ and there are some positions, so that up to this point everything is sorted.

And now what I will do is I will pick up the next element here and I will move it left until I find the correct place to put it, so that now the sorted thing extends to this length right. So, we take a sorted sequence of length i and we extend it to a sorted sequence like $i + 1$ by inserting **in the $i + 1$ th position in the current list**. So, we are going to take this position the slice end right, the slice end is going to be the last position that we have sorted already. So, this is supposed to be slice end.

So, we say sliceend it starts from the value 0 and goes up to the $n - 1$ th position. And at each time, we look at the value at. Actually the slice is up to sliceend minus 1 sorry. So, sliceend is a number of elements that we have sorted. We look at the value immediately after that which will be in the position called sliceend and so long as this position is bigger than 0; and if the value at that position is strictly smaller than the value at the previous position, we exchange these two right. So, what we were doing is that we are saying we draw it again. We have an already sorted slice to from 0 to slice $n - 1$, and we have this position sliceend. We then assume that this is sorted. So, we compare with this value and if this is smaller then we exchange it.

Now if you **have** exchanged it that means, that this value has now gone here. Now, we again compare it to the previous value, and if it is smaller we exchange it. So, again this means that it goes one more position. We just keep going until we find that at this position the value to the left of it is equal to or bigger than this sorry equal to or smaller than this. So, we should not swap it and we have it in the correct position right, so that is what this is doing. So long as you have not reached the left hand end, you compare the value you are looking at now to the value to its left; with the value to its left is strictly bigger, this one must exchange and then you decrement the position.

(Refer Slide Time: 05:29)



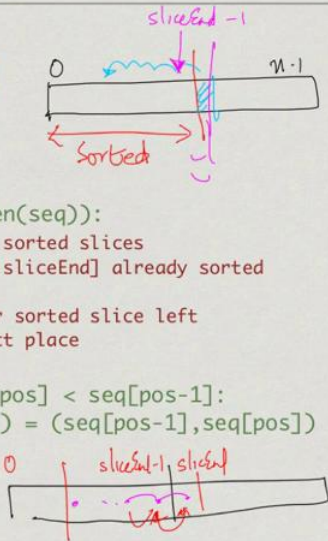
Let us run this the way we have written it on this particular sequence. So, what we do is we initially assume that this thing is unsorted. So, our first thing is here. And so when we sort it, we just get a sorted list of length one which is 74. Then we look at this and we must insert it into this list 74. So since this is smaller than 74, it gets exchanged and we get now new sorted list 32, 74 and now we must insert 89 into this list right and now we see 89 is bigger than 74, so nothing happens. This list now I sorted from 32 to 89, now we try to insert 55 in this. We first compare it with this, and this will say that 55 is smaller than the value to its left, so we must exchange.

Now we will compare 55 again to the value to its left again, we will exchange. Now we will compare 55 to the value to its left and there is no change. Now we have a sorted list of length 4. Similarly, we will take 21 right, and we will compare it to 89; since 21 is smaller than 89, it will swap; since 21 is smaller than 74, it will again swap; since 21 is smaller than 55, it will swap; since 21 is smaller than 32, it will swap, but now the position sorry will swap and now the position is 0. So, we stop not because we have found something to the left which is bigger, but because we have nothing to the left.

(Refer Slide Time: 07:02)

Insertion Sort

```
def InsertionSort(seq):  
    for sliceEnd in range(len(seq)):  
        # Build longer and longer sorted slices  
        # In each iteration seq[0:sliceEnd] already sorted  
  
        # Move first element after sorted slice left  
        # till it is in the correct place  
        pos = sliceEnd  
        while pos > 0 and seq[pos] < seq[pos-1]:  
            (seq[pos], seq[pos-1]) = (seq[pos-1], seq[pos])  
            pos = pos-1
```



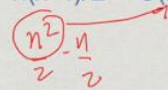
The diagram illustrates the Insertion Sort algorithm. It shows a horizontal array representing a sequence of elements. The left end is labeled '0' and the right end is labeled 'n-1'. A blue arrow points from the right towards the left, indicating the direction of element movement. A red double-headed arrow labeled 'Sorted' spans a portion of the array. A vertical purple line marks the position of 'sliceEnd - 1'. A second diagram below shows the array with a red vertical line at '0' and a purple vertical line at 'sliceEnd - 1', with a red arrow indicating the insertion point.

We have two conditions if you remember that algorithm is said that either pos should be positive, the position should be greater than 0 or we should compare it to the value on its left right. In this case, we have no value to its left, so we stop.

(Refer Slide Time: 07:15)

Analysis of Insertion Sort

- Inserting a new value in sorted segment of length k requires upto k steps in the worst case
- In each iteration, sorted segment in which to insert increased by 1
- $T(n) = 1 + 2 + \dots + n-1 = \frac{n(n-1)}{2} = O(n^2)$



The handwritten formula shows the derivation of the time complexity: $\frac{n^2}{2} - \frac{n}{2}$. A red arrow points from this formula to the $O(n^2)$ term in the list item above.

How do we analyze this? Well, at each round, what are we doing, we are inserting a new

value into a sorted segment of length k . So, we start with the length 0 segment, we insert one value to it, we get a sorted length of sequence of length one, we insert a value into that we get a sorted sequence of length two and so on. Where in the worst case, when we are inserting we have to take the value all the way to the beginning of the segment.

Sorting a segment of length k in the worst case takes k steps, so again we have the same recurrence not expression that we had for selection sort **says** that T of n is 1 plus 2 plus 3 up to n minus 1 which is n into n minus 1 by 2 which is order n square. So, again remember that this is n square by 2 minus n by 2 and so this is the biggest term and that is what we get.

(Refer Slide Time: 08:10)

```
madhavan@dolphinair:~/.eek3/python/insertionsort$ more insertionsort.py
def InsertionSort(seq):
    for sliceEnd in range(len(seq)):
        # Build longer and longer sorted slices
        # In each iteration seq[0:sliceEnd] already sorted

        # Move first element after sorted slice left
        # till it is in the correct place
        pos = sliceEnd
        while pos > 0 and seq[pos] < seq[pos-1]:
            (seq[pos],seq[pos-1]) = (seq[pos-1],seq[pos])
            pos = pos-1
madhavan@dolphinair:~/.eek3/python/insertionsort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from insertionsort import *
>>> l = list(range(500,0,-1))
>>> InsertionSort(l)
>>> !
```

Once again let us see how insertion sort actually works in the python interpreter and we will see something slightly different from selection sort when we run it. First, let us look at the code. This is the code that we saw in the slide. We just keeps scanning segments, keeps taking a value at a position and inserting it into the already sorted sequence up to that position. If we start the python interpreter, and say import this function, then as before if we for example, take a long list and sort it then l becomes sorted. So, before l was in descending order.

(Refer Slide Time: 08:57)

```
88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500]
>>> l = list(range(500,0,-1))
>>>
```

Now we sort it, and now l is in ascending order.

(Refer Slide Time: 09:01)

```
449, 448, 447, 446, 445, 444, 443, 442, 441, 440, 439, 438, 437, 436, 435, 434, 433, 432, 431, 430, 429, 428, 427, 426, 425, 424, 423, 422, 421, 420, 419, 418, 417, 416, 415, 414, 413, 412, 411, 410, 409, 408, 407, 406, 405, 404, 403, 402, 401, 400, 399, 398, 397, 396, 395, 394, 393, 392, 391, 390, 389, 388, 387, 386, 385, 384, 383, 382, 381, 380, 379, 378, 377, 376, 375, 374, 373, 372, 371, 370, 369, 368, 367, 366, 365, 364, 363, 362, 361, 360, 359, 358, 357, 356, 355, 354, 353, 352, 351, 350, 349, 348, 347, 346, 345, 344, 343, 342, 341, 340, 339, 338, 337, 336, 335, 334, 333, 332, 331, 330, 329, 328, 327, 326, 325, 324, 323, 322, 321, 320, 319, 318, 317, 316, 315, 314, 313, 312, 311, 310, 309, 308, 307, 306, 305, 304, 303, 302, 301, 300, 299, 298, 297, 296, 295, 294, 293, 292, 291, 290, 289, 288, 287, 286, 285, 284, 283, 282, 281, 280, 279, 278, 277, 276, 275, 274, 273, 272, 271, 270, 269, 268, 267, 266, 265, 264, 263, 262, 261, 260, 259, 258, 257, 256, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 239, 238, 237, 236, 235, 234, 233, 232, 231, 230, 229, 228, 227, 226, 225, 224, 223, 222, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210, 209, 208, 207, 206, 205, 204, 203, 202, 201, 200, 199, 198, 197, 196, 195, 194, 193, 192, 191, 190, 189, 188, 187, 186, 185, 184, 183, 182, 181, 180, 179, 178, 177, 176, 175, 174, 173, 172, 171, 170, 169, 168, 167, 166, 165, 164, 163, 162, 161, 160, 159, 158, 157, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 146, 145, 144, 143, 142, 141, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 130, 129, 128, 127, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>>
```

(Refer Slide Time: 09:03)

```
>>> l = list(range(5000,0,-1))
>>> InsertionSort(l)
```

Now as before what we said is that if we try to do this for a length of around 5000 then it will be much smaller and much slower right. So, you can see it takes a long time and that is because InsertionSort, it is again order n^2 sort.

(Refer Slide Time: 09:24)

```
, 4634, 4635, 4636, 4637, 4638, 4639, 4640, 4641, 4642, 4643, 4644, 4645, 4646, 4647,
4648, 4649, 4650, 4651, 4652, 4653, 4654, 4655, 4656, 4657, 4658, 4659, 4660, 4661,
4662, 4663, 4664, 4665, 4666, 4667, 4668, 4669, 4670, 4671, 4672, 4673, 4674, 4675, 4
676, 4677, 4678, 4679, 4680, 4681, 4682, 4683, 4684, 4685, 4686, 4687, 4688, 4689, 46
90, 4691, 4692, 4693, 4694, 4695, 4696, 4697, 4698, 4699, 4700, 4701, 4702, 4703, 470
4, 4705, 4706, 4707, 4708, 4709, 4710, 4711, 4712, 4713, 4714, 4715, 4716, 4717, 4718
, 4719, 4720, 4721, 4722, 4723, 4724, 4725, 4726, 4727, 4728, 4729, 4730, 4731, 4732,
4733, 4734, 4735, 4736, 4737, 4738, 4739, 4740, 4741, 4742, 4743, 4744, 4745, 4746,
4747, 4748, 4749, 4750, 4751, 4752, 4753, 4754, 4755, 4756, 4757, 4758, 4759, 4760, 4
761, 4762, 4763, 4764, 4765, 4766, 4767, 4768, 4769, 4770, 4771, 4772, 4773, 4774, 47
75, 4776, 4777, 4778, 4779, 4780, 4781, 4782, 4783, 4784, 4785, 4786, 4787, 4788, 478
9, 4790, 4791, 4792, 4793, 4794, 4795, 4796, 4797, 4798, 4799, 4800, 4801, 4802, 4803
, 4804, 4805, 4806, 4807, 4808, 4809, 4810, 4811, 4812, 4813, 4814, 4815, 4816, 4817,
4818, 4819, 4820, 4821, 4822, 4823, 4824, 4825, 4826, 4827, 4828, 4829, 4830, 4831,
4832, 4833, 4834, 4835, 4836, 4837, 4838, 4839, 4840, 4841, 4842, 4843, 4844, 4845, 4
846, 4847, 4848, 4849, 4850, 4851, 4852, 4853, 4854, 4855, 4856, 4857, 4858, 4859, 48
60, 4861, 4862, 4863, 4864, 4865, 4866, 4867, 4868, 4869, 4870, 4871, 4872, 4873, 487
4, 4875, 4876, 4877, 4878, 4879, 4880, 4881, 4882, 4883, 4884, 4885, 4886, 4887, 4888
, 4889, 4890, 4891, 4892, 4893, 4894, 4895, 4896, 4897, 4898, 4899, 4900, 4901, 4902,
4903, 4904, 4905, 4906, 4907, 4908, 4909, 4910, 4911, 4912, 4913, 4914, 4915, 4916,
4917, 4918, 4919, 4920, 4921, 4922, 4923, 4924, 4925, 4926, 4927, 4928, 4929, 4930, 4
931, 4932, 4933, 4934, 4935, 4936, 4937, 4938, 4939, 4940, 4941, 4942, 4943, 4944, 49
45, 4946, 4947, 4948, 4949, 4950, 4951, 4952, 4953, 4954, 4955, 4956, 4957, 4958, 495
9, 4960, 4961, 4962, 4963, 4964, 4965, 4966, 4967, 4968, 4969, 4970, 4971, 4972, 4973
, 4974, 4975, 4976, 4977, 4978, 4979, 4980, 4981, 4982, 4983, 4984, 4985, 4986, 4987,
4988, 4989, 4990, 4991, 4992, 4993, 4994, 4995, 4996, 4997, 4998, 4999, 5000]
>>>
```

So, though it does it eventually it takes a long time,

(Refer Slide Time: 09:28)

```
>>> l = list(range(0,5000))
>>> InsertionSort(l)
>>> l = list(range(0,100000))
>>> 
```

But there is a small difference here. So, suppose we do it the other way, suppose we take a list which is already sorted, and now we ask it to sort, then it comes back instantly. Why should this be the case **well** think about what is happening now the list is already in sorted order. So, when we try to take a value at any position and move it to the left, it immediately finds that the value to its left is smaller than it, so no swapping occurs. So, each insert step takes only one iteration. It does not have to go through anything beyond the first element in order to stop the insert step. So, actually if we take even a large value like 10,000 or even 100000 this should work.

Insertion sort when you already have a sorted list will be quite fast because the insert step is instantaneous whereas this does not happen with selection sort. Because in selection sort, in each iteration we have to find the minimum value in a cell in a sequence and with no prior knowledge about what the sequence looks like it will always scan the sequence from beginning to end.

The worst case for selection sort, will happen regardless of whether the input is already sorted or not; whereas insertion sort if the list is sorted, the insert step will be very fast, and so you can if actually sort larger things. In that sense insertion sort can be behave much better than selection sort even though both of them technically in the worst case

are order n^2 sorts.