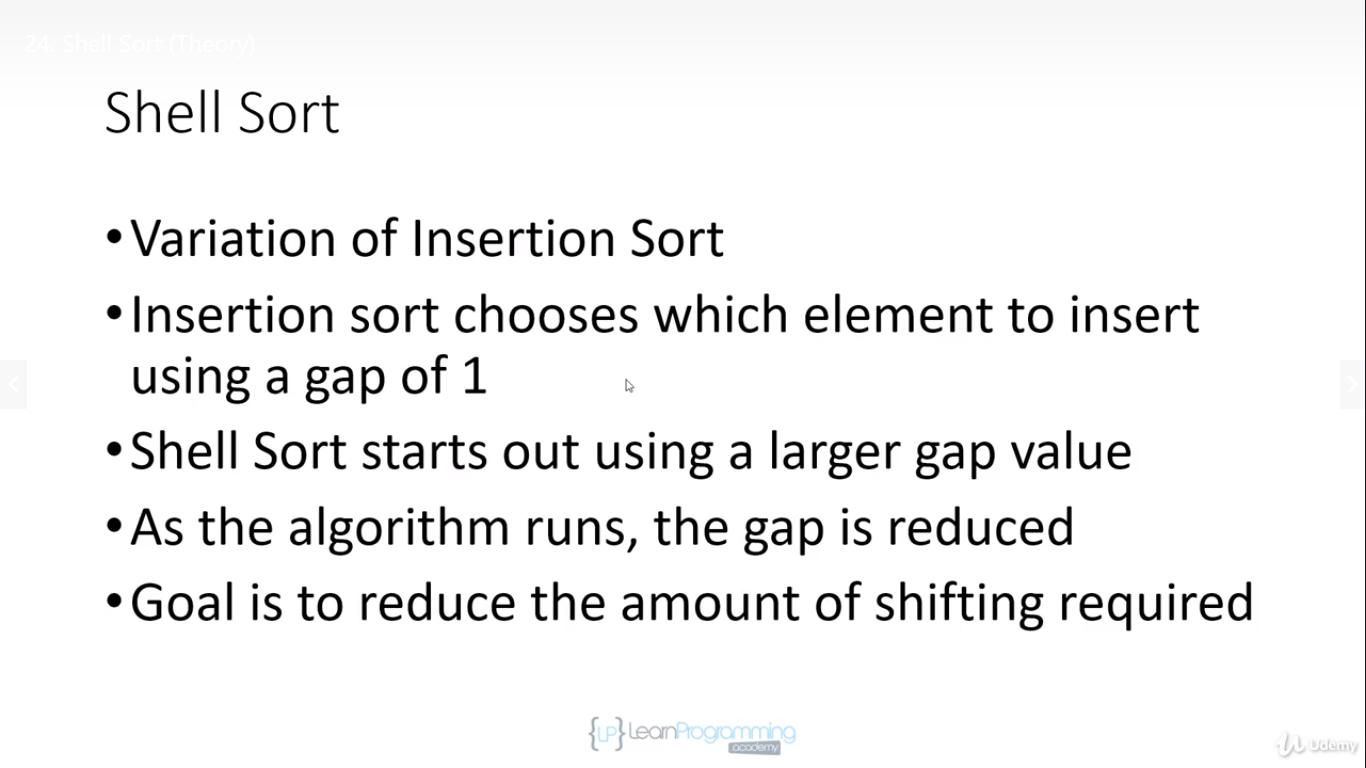
**Shell Sort**

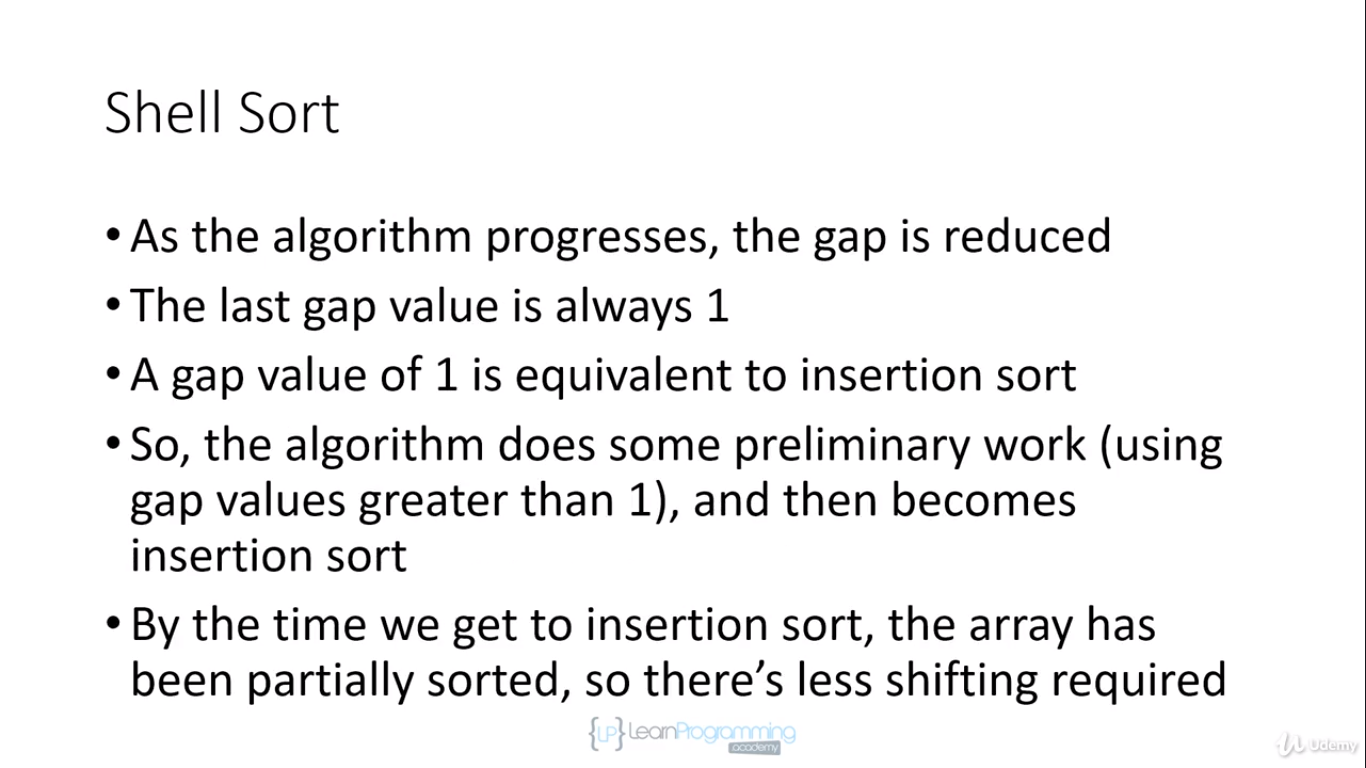
All right, we learned the insertion sort algorithm and you saw the implementation or at least one implementation, and I said that insertion sort is a quadratic algorithm and it is, but if the sequence of values that we're sorting is nearly sorted, then insertion sort runs in almost linear time and it does that because it doesn't have to do as much shifting. Think about it. If most of the values are already sorted, then only a few values will have to be inserted into the sorted partition and the amount of shifting will be reduced. Now a computer scientist named Donald Shell realized that if we could cut down on the amount of shifting that insertion sort would run a lot faster and so he came up with something that is known as the Shell sort algorithm



**So how does Shell sort work?**

Well, it's a variation of insertion sort and what it does is insertion sort chooses which element to insert using a gap value of 1. So every time insertion sort runs, it picks off the first unsorted value and then it compares that value to its neighbour and it keeps shifting the neighbours to the right until it finds the correct insertion point for the element that it's inserting.

Shell sort starts out using a larger gap value,so instead of comparing elements to their neighbours,it compares elements that are farther apart from each other in the array. And then as the algorithm runs, it reduces the gap that it's using. And the goal is to reduce the amount of shifting that's required.



So as I said, as the algorithm progresses, the gap value is reduced. So Shell sort traverses the array with a certain gap value and after it's done its first traversal with the initial gap value, it decreases the gap and it does it again and it does this and this is very important, it keeps reducing the gap value until the gap value is 1.

Now when the gap value is 1,we're essentially doing an insertion sort. So the last iteration of the gap value will actually perform an insertion sort. But at that point, the array will be more sorted than it was at the beginning. And so essentially what Shell sort does is it does some preliminary work using gap values that are greater than 1 and that preliminary work puts the elements in the array and perhaps closer to their sorted positions and then at the very last iteration when the gap value becomes 1,it does an insertion sort.

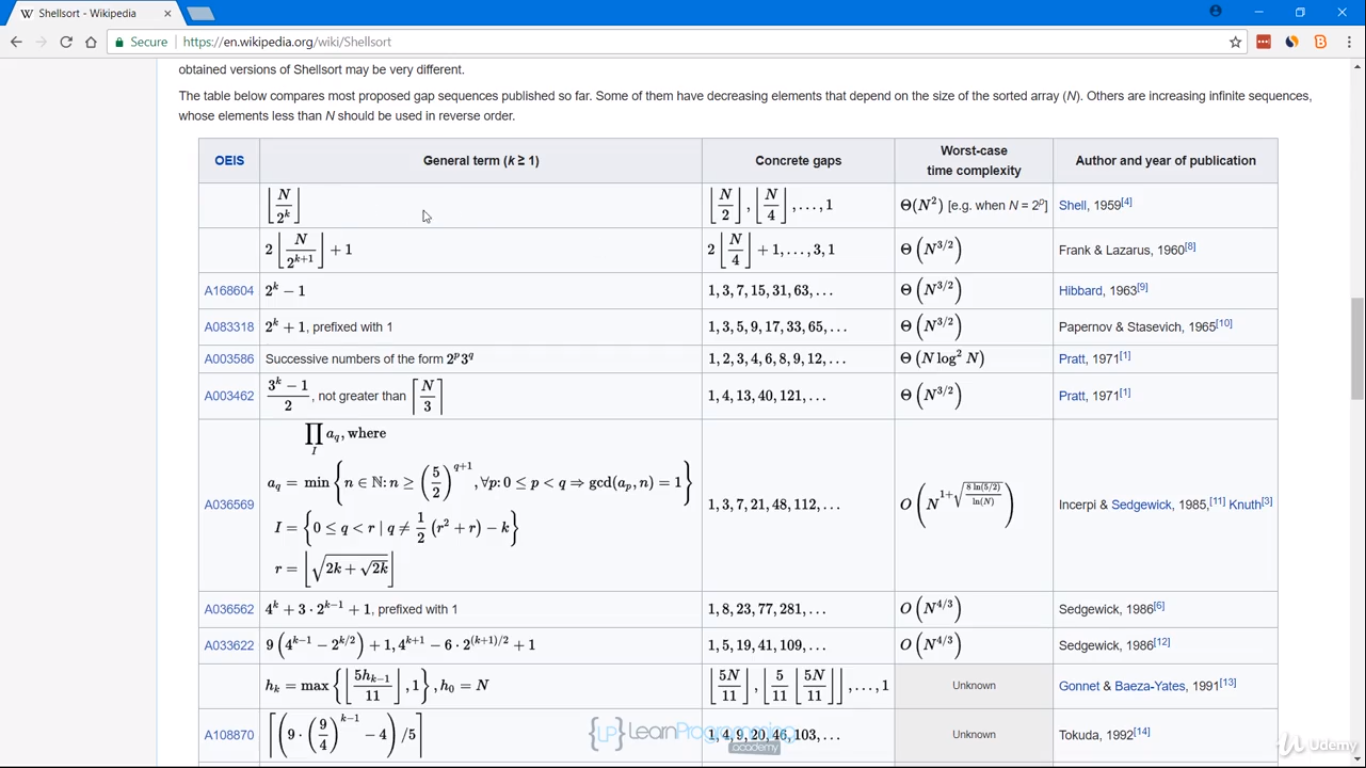
But that final insertion sort will be working with values that have had some preliminary sorting done on them. And because of that, there will be a lot less shifting required.So one question is well,

what do we use for the gap value?

How do we figure out what to start with and how to reduce it?

And you're going to see that there is a tonne of theories about this.

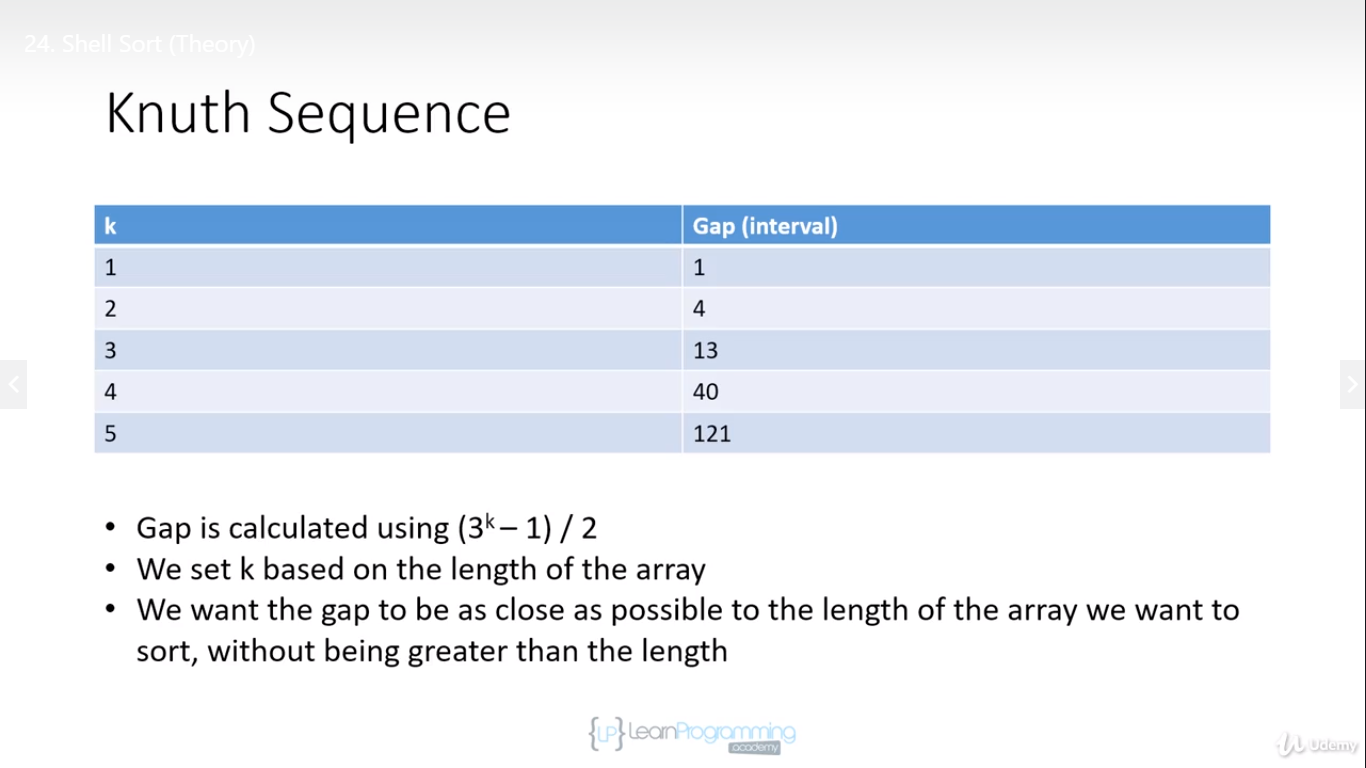
We're going to go out now to the Wikipedia article about Shell sort.



So here we are at the Wikipedia article that I've linked to in the resources and here's a table with different initial gap values and how to reduce the gap, what sequence of gap values to use. And as you can see, there's quite a number of them.

The important thing to note is that the way that you calculate the gap can influence the time complexity. And so here we have a time complexity column and depending on what gap we're using, the time complexity is different.We have a logarithmic time complexity here. Here's a quadratic time complexity. And so the gap value you choose can influence how many steps the algorithm requires so keep that in mind.

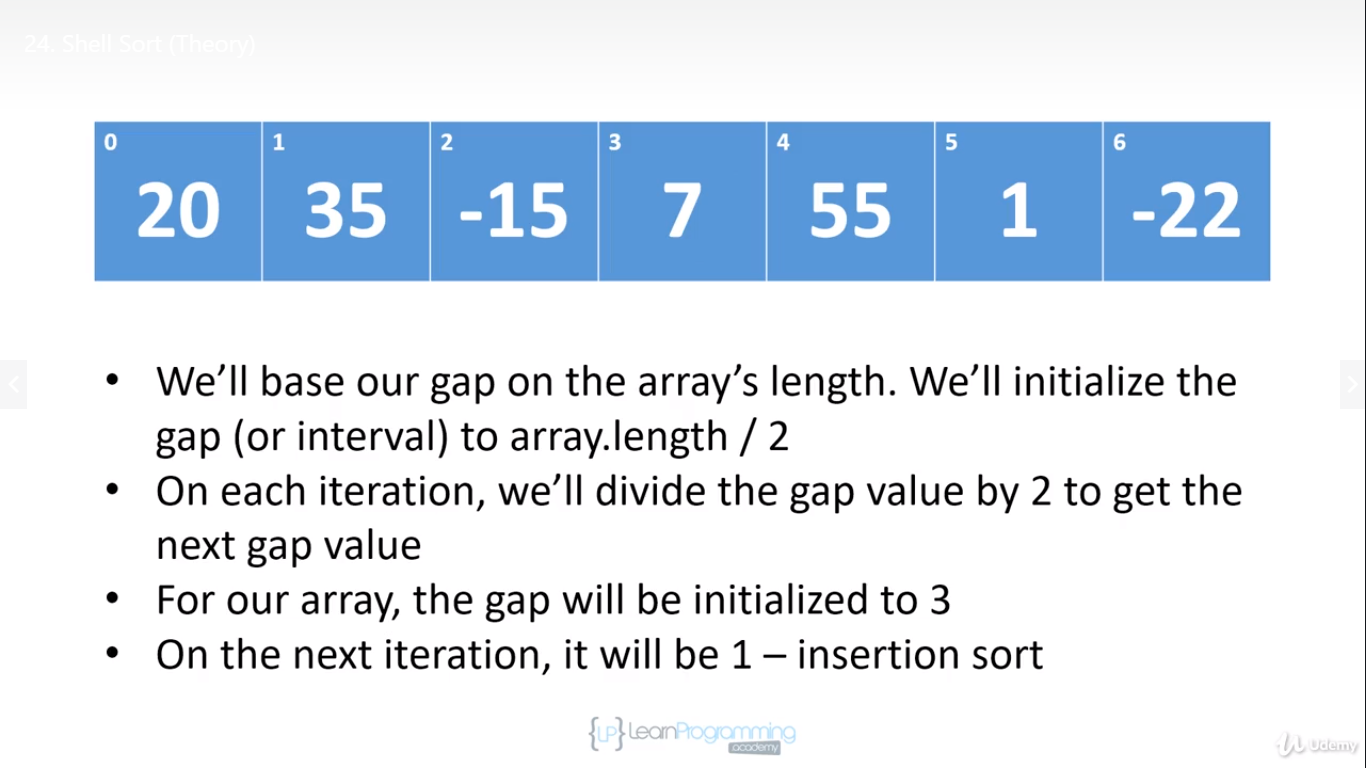
Okay, so there are a number of ways we can calculate the gap value. One really common sequence that's used for the gap value and the gap is also called the interval is the **Knuth sequence**.



* This says that we calculate the gap using (3k-1)/2.
* And the initial value that we want to use is based on the length of the array.
* What we want is we want to use the value of k that's going to calculate to a value that's as close as possible to the length of the array without going over.

And so if we had an array of let's say 20 elements, we would wanna start with k equal to 3.If we were to start with k equals 4,the gap would be 40 and that's greater than the length of the array so that won't work. And so when you want to use this sequence, you want to start with the value of k that's going to result in a gap value that is as close to the array's length as possible without going over.

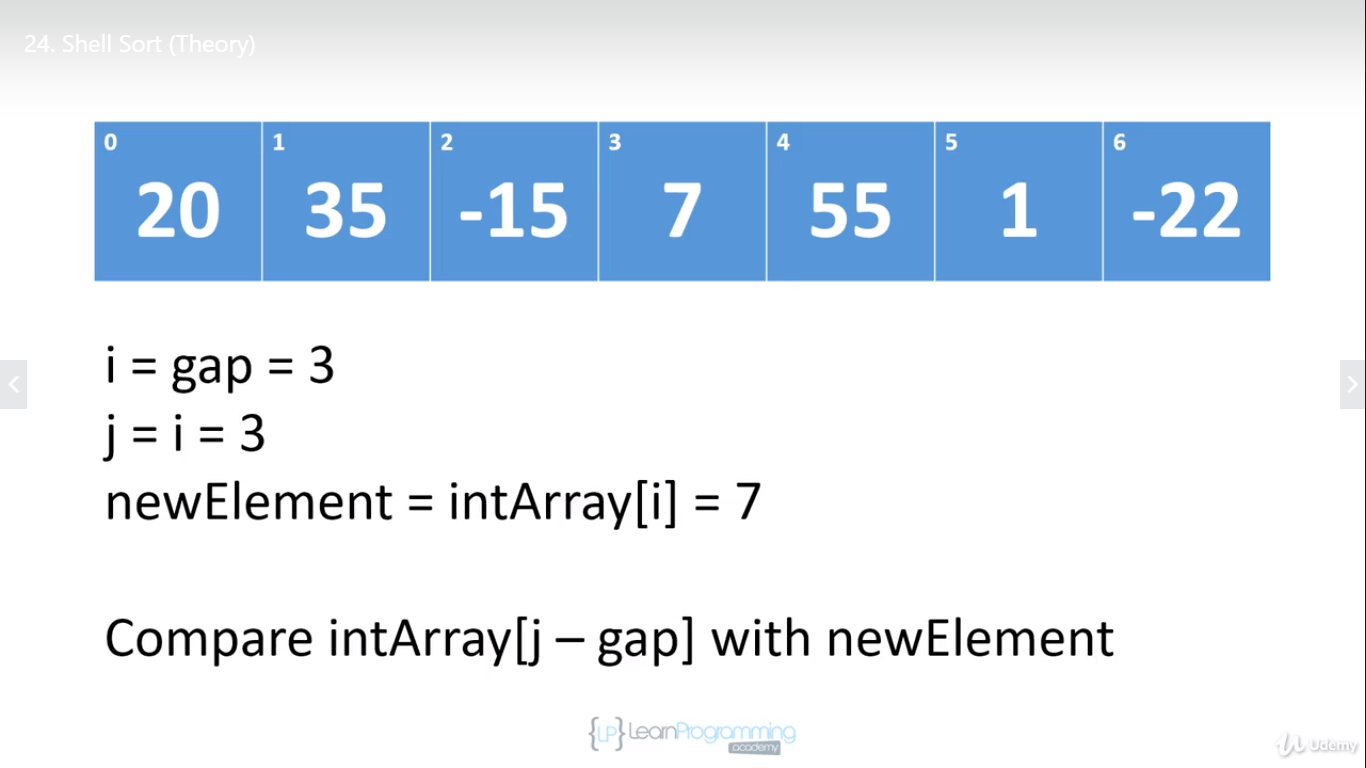
Now in the implementation I'm going to show you, we're not going to use this sequence, but it's a common way of calculating the interval or the gap so I thought I'd show it to you. We're going to do something simpler than that.



So what we're going to do is we're going to base our gap on the array's length and we're going to initialize the gap to array. Length/ 2.

And then on each iteration, we're going to divide the gap value by 2 until we hit 1. Now our array only has 7 elements in it and so the first gap value is going to be 7 over 2 which is 3 and then the second gap value will be 3 over 2 which is 1 so we're actually only gonna have 2 iterations for this array.

So on the first iteration, we'll use a gap value of 3. And then on the final iteration, and this is always true for Shell sort, the gap value will be 1. And at that point, we'll be doing an insertion sort. But because we've done a previous iteration and we've moved some of the elements around, some of the elements will be closer to their sorted positions.

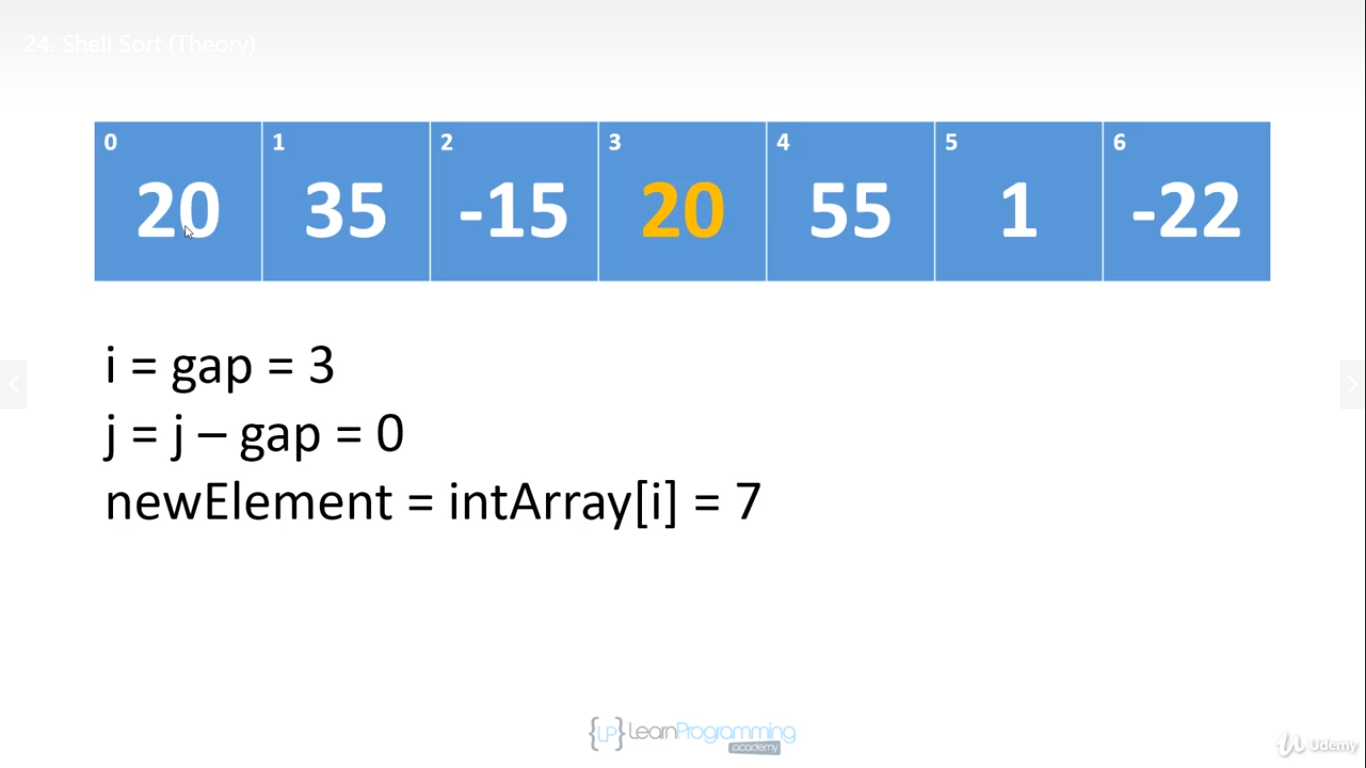


So we're gonna start with a gap value of 3 because we're gonna use array.length/ 2.

We're gonna initialise i = gap and j = i.

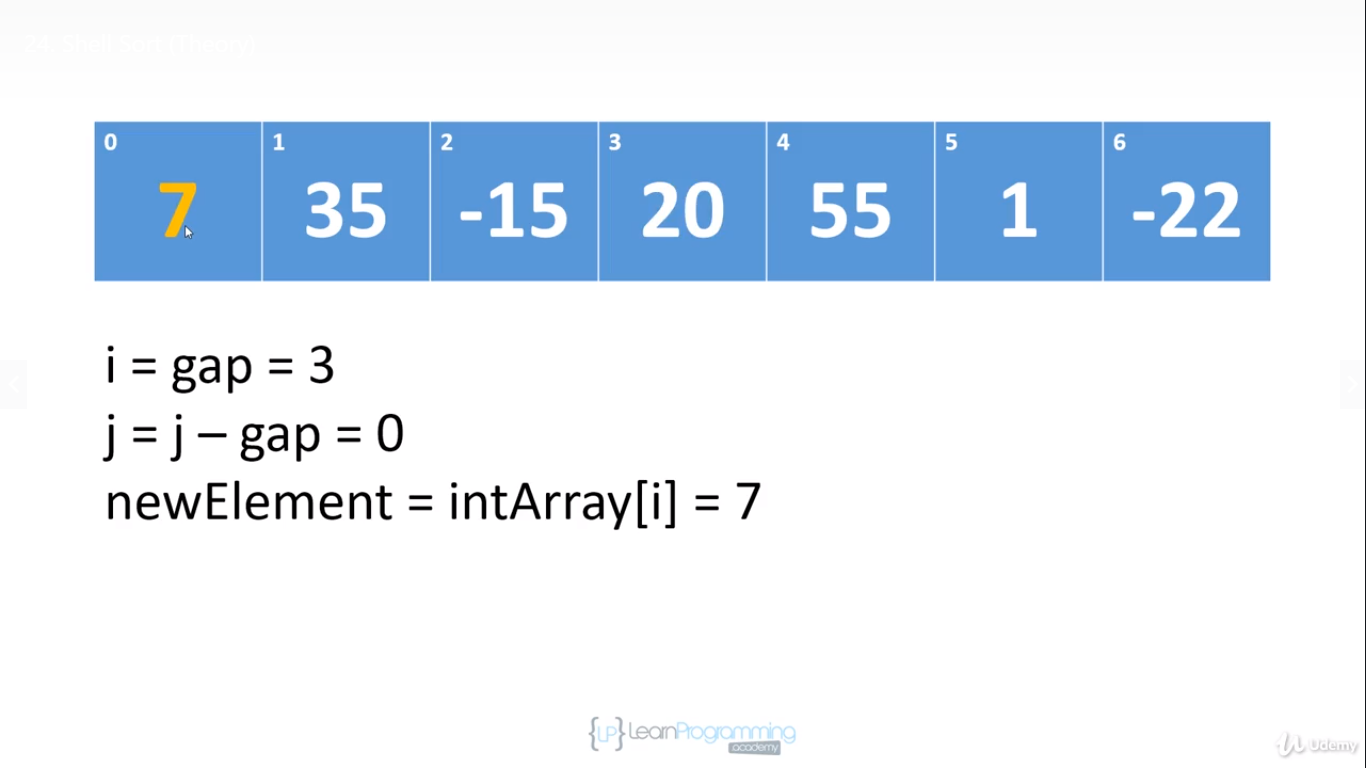
And as we did before with insertion sort, we're gonna save the value that we want to work with into newElement. And then what we do is we compare the element at index j-gap so that will be 3 - 3 is 0 with newElement.

So our gap is 3 and we're starting with element 3 so we basically wanna compare it. Because the gap is 3,we wanna compare it to the element that's 3 positions away and so we compare 7 with 20.7 is less than 20



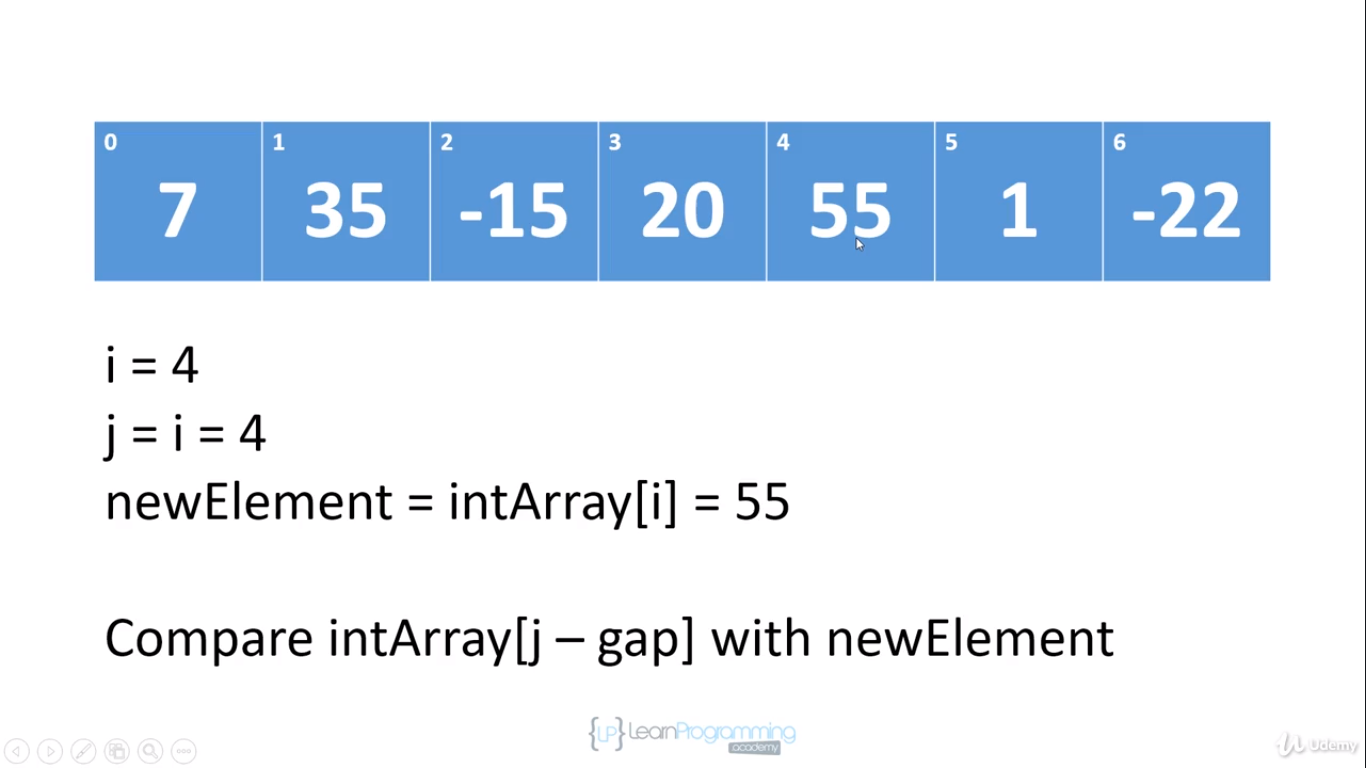
and so what we're going to do is we're going to assign 20 to where 7 was.

So instead of doing what we were doing with insertion sort, which is we're comparing to the neighboursand shifting up 1, we're comparing using a gap of 3 and we shift by 3 and so 20 has been shifted up 3 places.And then we're going to set j = j-gap which is 0

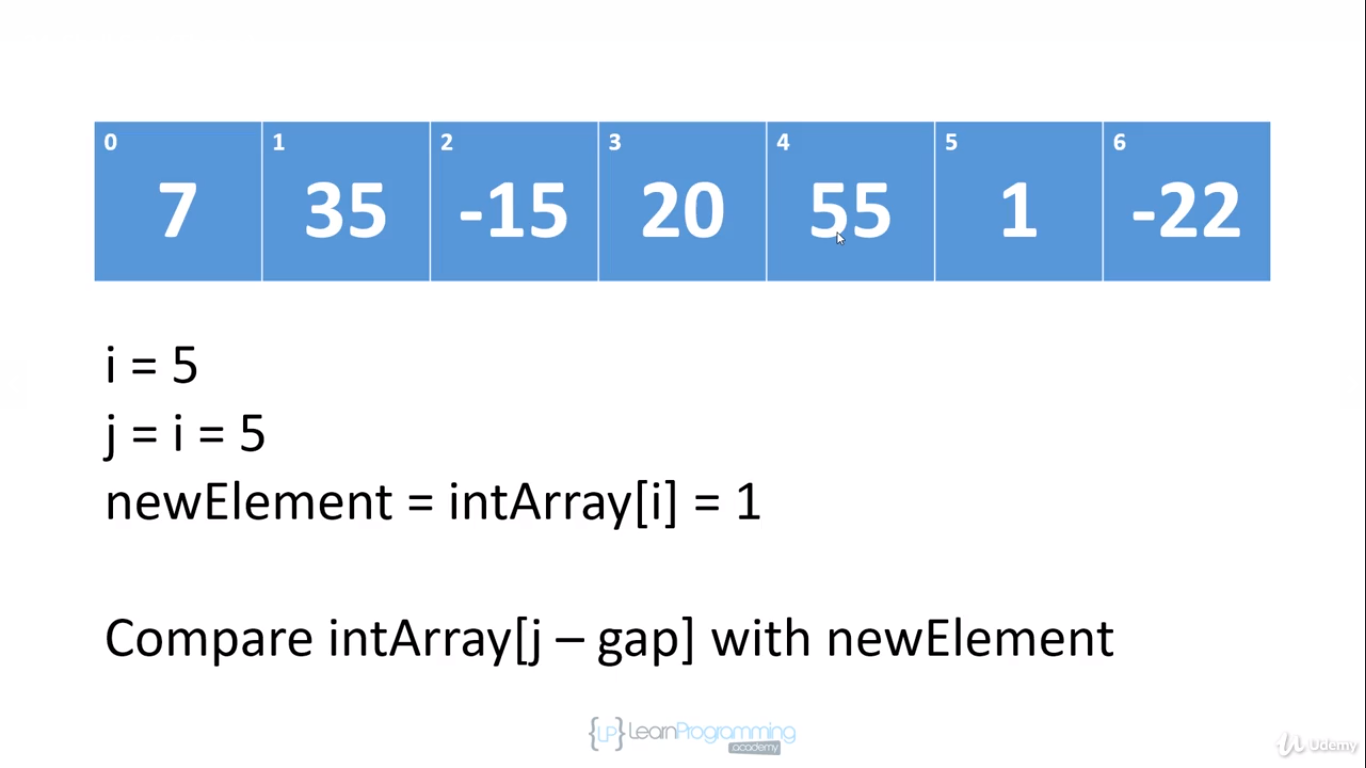


and we've hit the front of the array at this point and so what we're going to do is assign newElement to position 0.

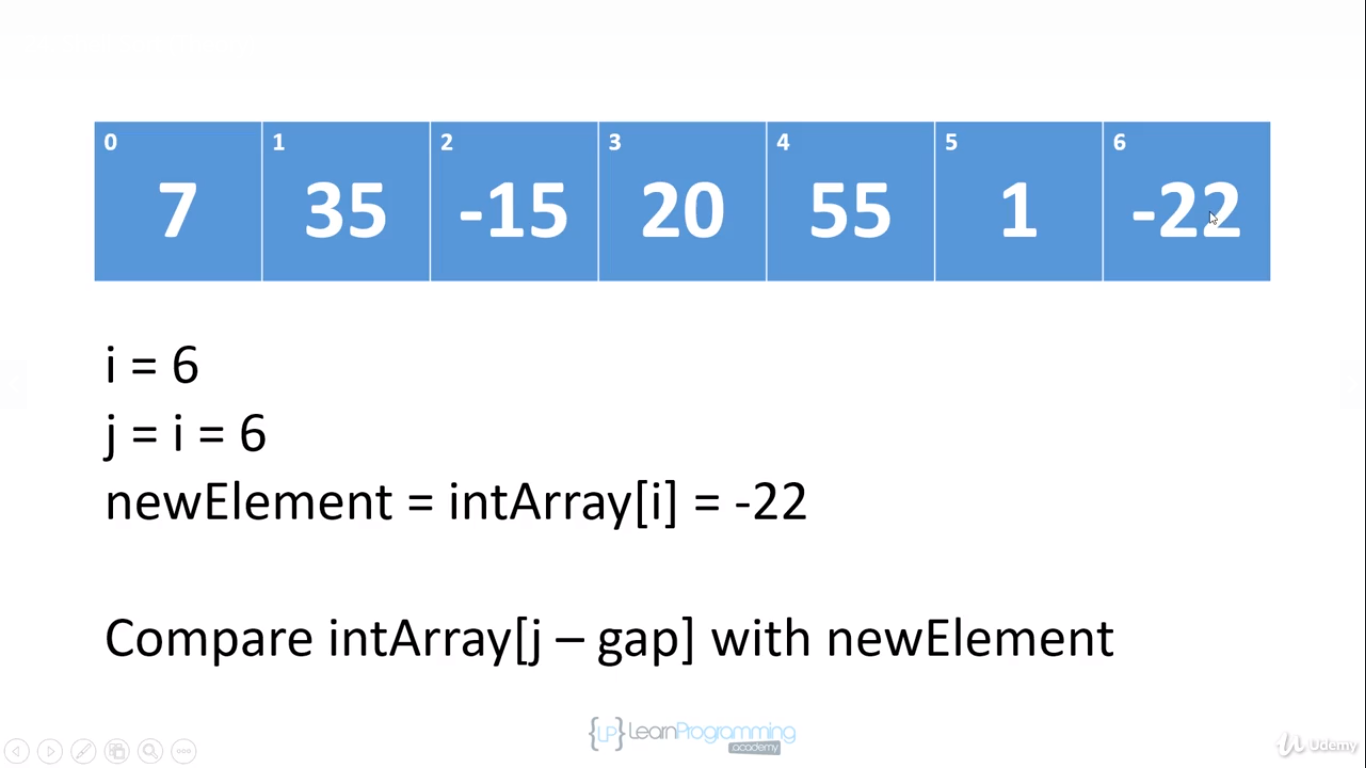
So what we've basically done is we're kind of doing an insertion sort but we're using a gap of 3.



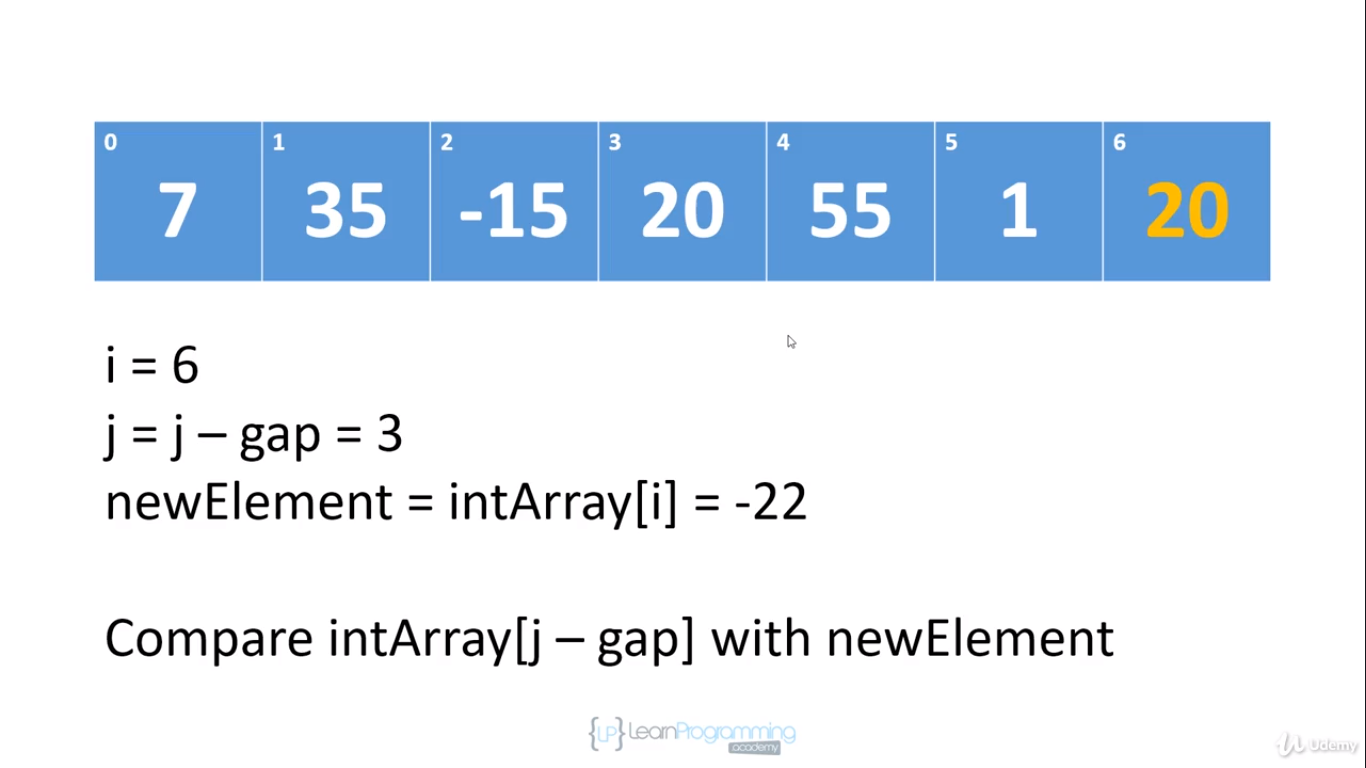
So what we're gonna want to do next now is move i to 4 and j becomes i which is 4, the newElement is 55 and we're gonna compare 55 to 35 because that's 3 elements away. 55 is greater than 35 so we don't do anything.Everything remains as it is.



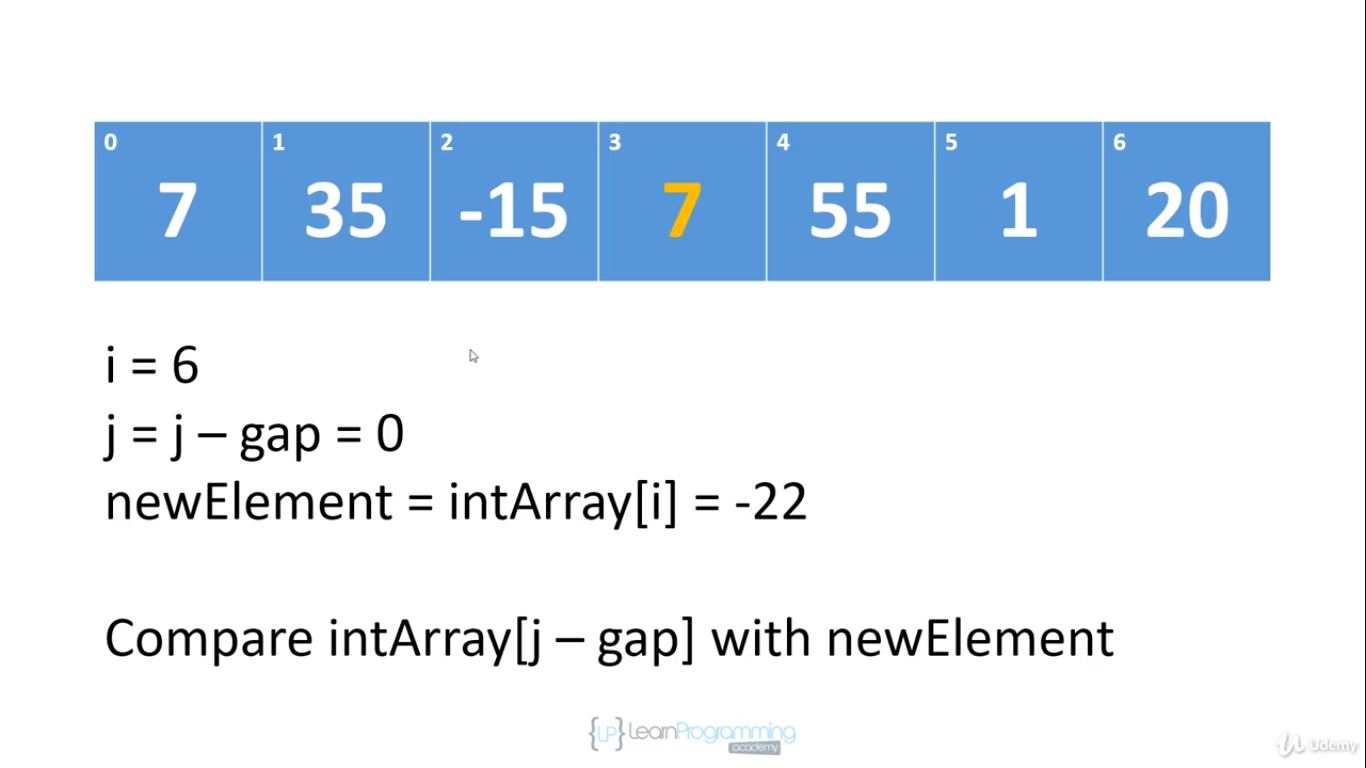
And now i will be 5, j will be 5,we're going to compare 1 to -15 'cause -15 is 3 elements away.Okay, so there's nothing to do because 1 is greater than -15



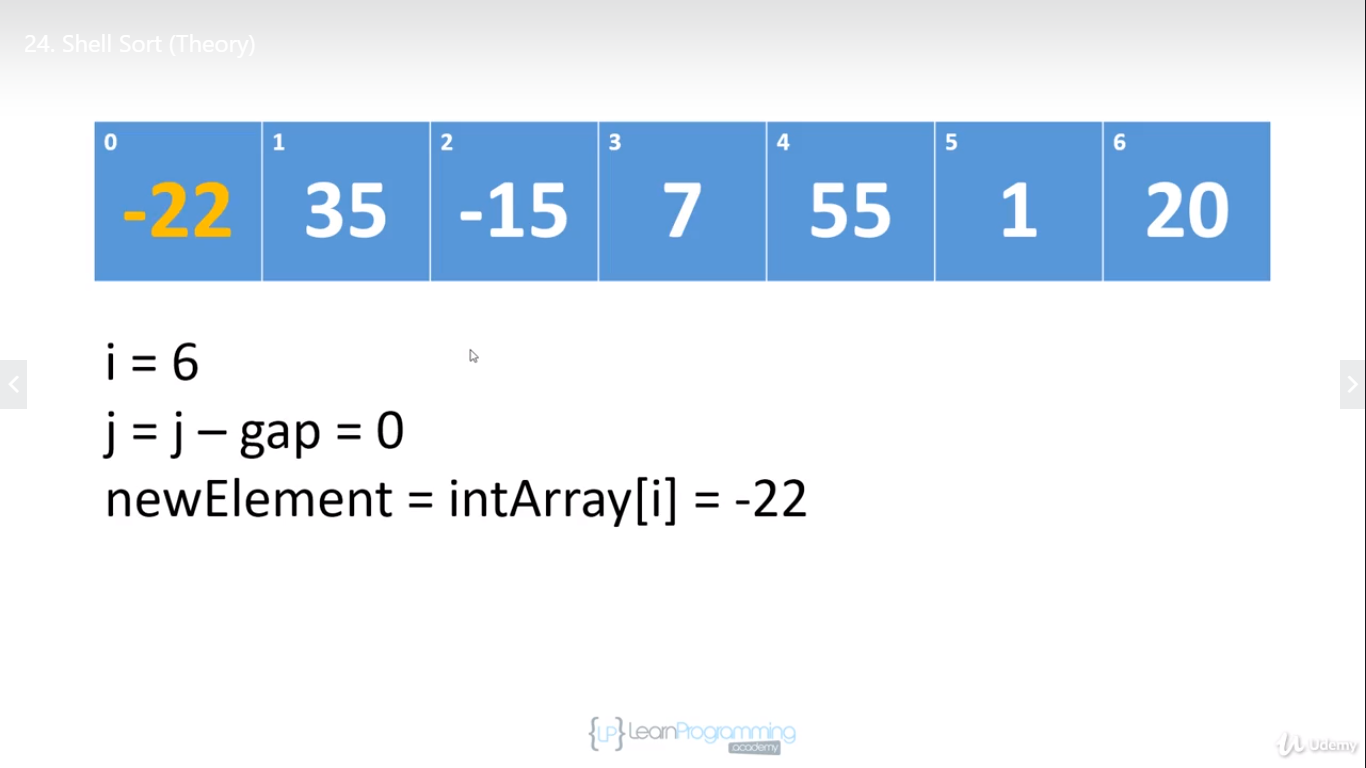
And so now we're gonna move to -22 and we're going to assign that to newElement and we're gonna compare it to the element that's 3 positions away from it and -22 is less than 20 so we're going to assign 20 to position 6.

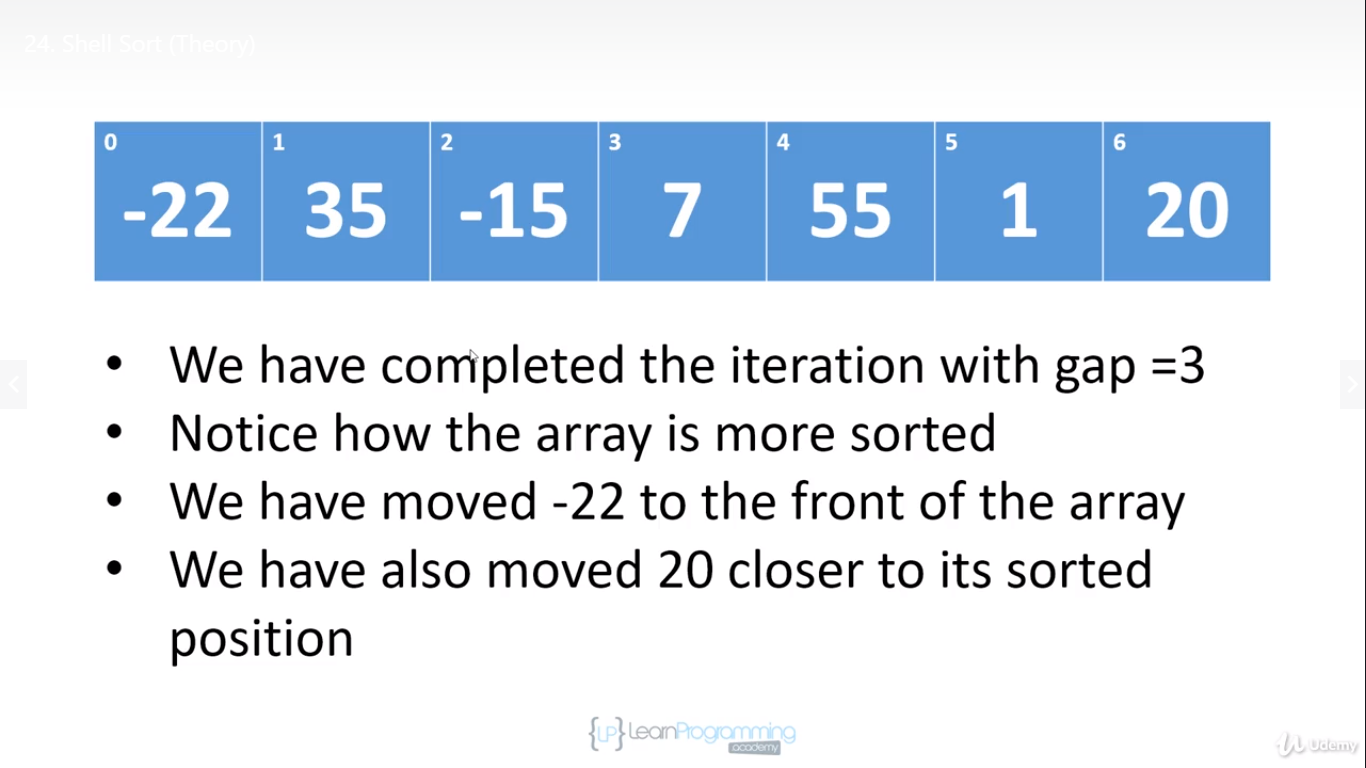


Now at this point, we're gonna subtract the gap from j and then we're going to compare -22 against what's at position 0 because we wanna go 3 elements away again. -22 is less than 7 so we're going to shift 7 into position 3.

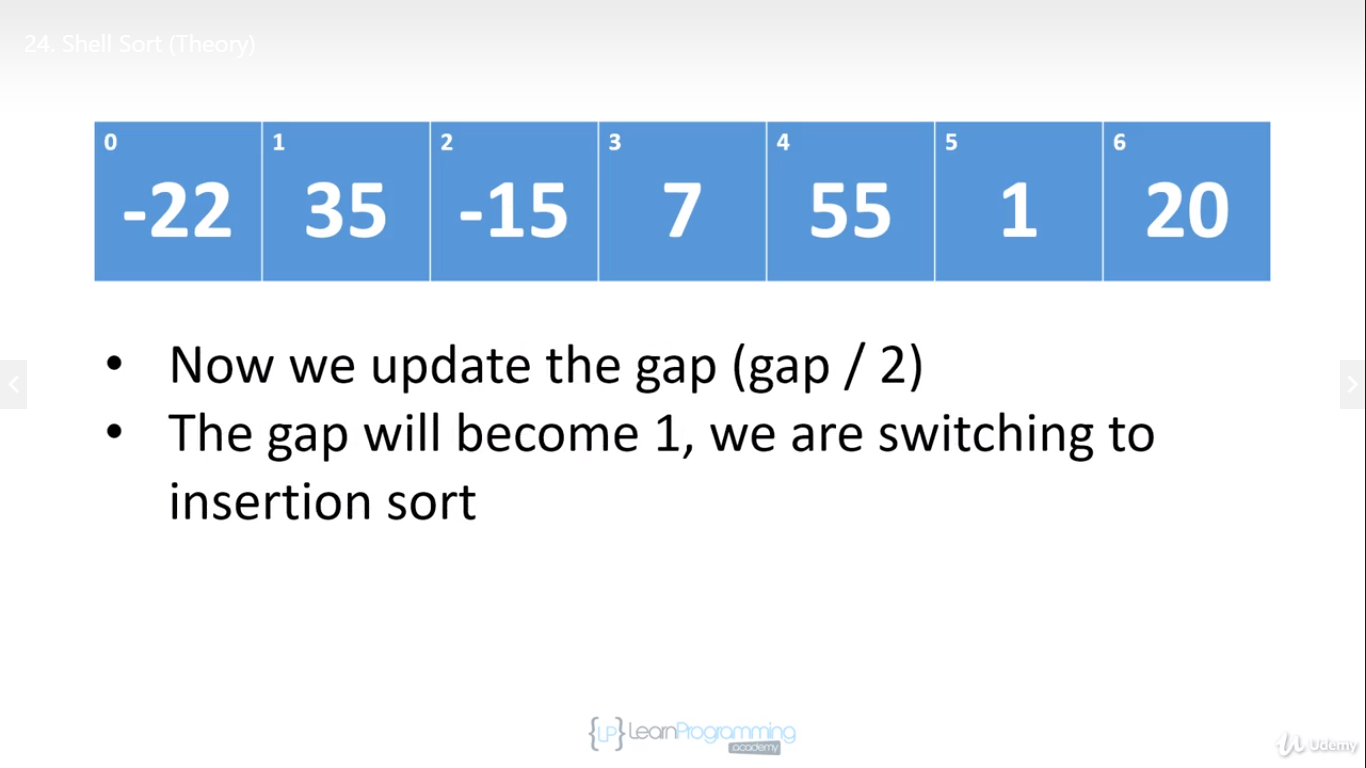


And at this point, we've hit the front of the array. There are no more elements to compare -22 against and so we assign -22 to position 0. And at that this point, we've hit the end of the array





* so we have finished our first iteration with gap equal to 3.
* Now notice that the array is more sorted now than it was before
* and we've moved -22 all the way to the front of the array and we did that with one assignment. And so in insertion sort, pure insertion sort, at some point we would have had to have shifted -22 down or rather in the implementation I showed you every single element would have to have been shifted up to make room for -22.But in this sort of pre-sorting phase, when we're using a gap of 3,-22 is moved very quickly down to the front of the array.
* We've also moved 20 closer to its sorted position. 20 started at the front of the array and now it's only two positions away from where it usually ends up in the sorted array.x`

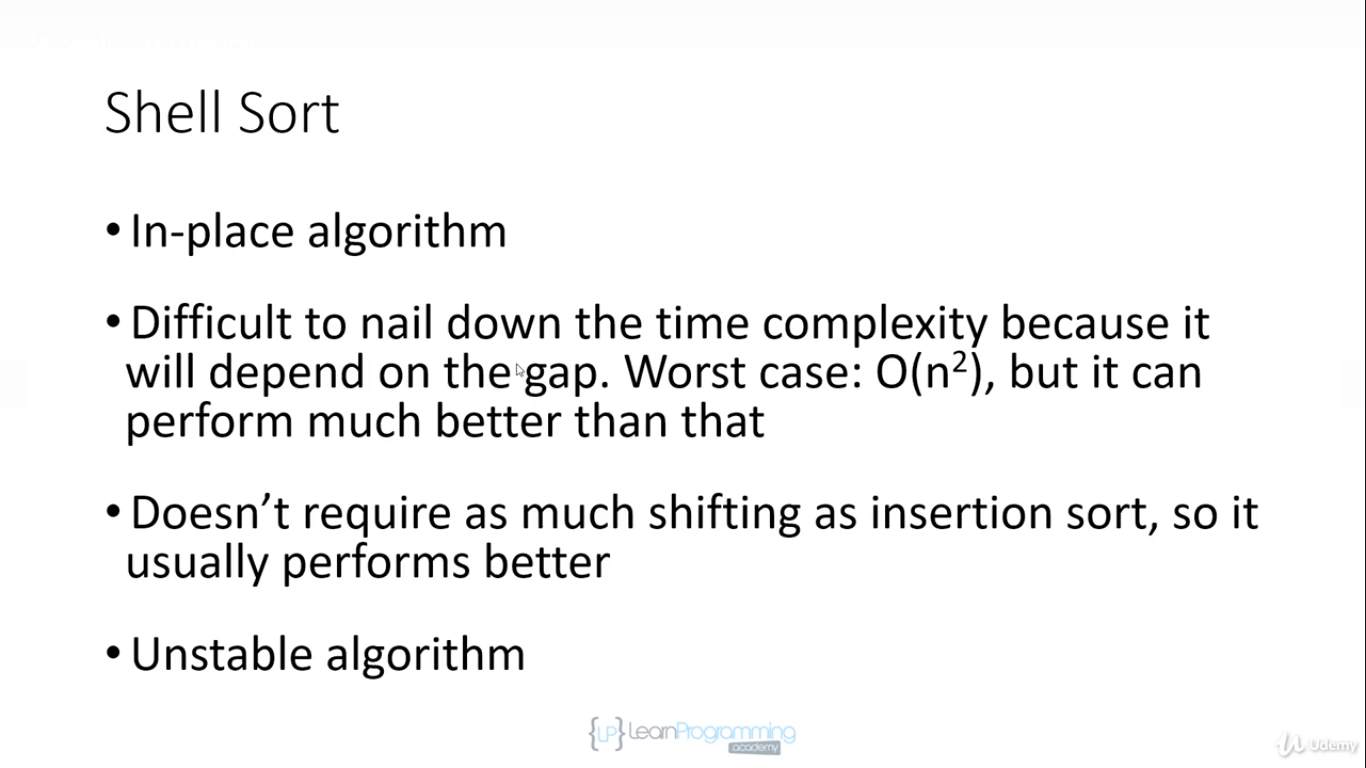


So you can see how doing this preliminary work

before we get to insertion sort will cut down on how much shifting we have to do.

So at this point, we're gonna update the gap. We're gonna divide 3/2 and we'll get 1

and so at this point,we will actually do an insertion sort because the gap is going to be one so we're gonna be comparing everything to its neighbours and when we shift we're gonna be shifting up by one. And so at this point, we'll do an insertion sort, but we're doing an insertion sort on an array that has had some preliminary work done on it and so there's gonna be a lot less shifting and that's what Shell sort is all about.



* So Shell sort is an in-place algorithm just like insertion sort. We're working within the original array.
* Now as you saw, it's really difficult to nail down the time complexity because it's going to depend on the gap.It's gonna depend on the method that you're using to choose the gap. In the worse case, it can be quadratic. We saw that when we went out to the table in Wikipedia, but it can also perform much better than that.
* It doesn't require as much shifting as insertion sort. So as I said, it usually performs better than insertion sort.
* However, it's an unstable algorithm. Insertion sort is stable, but Shell sort is unstable and you can see why. It's because in the pre-insertion sort phase when we're comparing elements that are far away from each other, it's very possible that if we have a duplicate element, the rightmost element will be shifted in front of the leftmost element. So the fact that we're examining elements that are further away from each other can lead to the relative positions of duplicate items not being preserved.

**You can also improve bubble sort using Shell sort and it would be the same type of idea. You would use a gap interval. Remember in bubble sort, we're always comparing elements to their neighbours and then we're swapping and bubbling elements up. Well, it's the same sort of idea. So in bubble sort, if we use a gap of 1, that means we compare it to the neighbours and everything gets swapped up 1 position. Things get bubbled up. If we do some preliminary work and in this case rather than just shifting elements, we'd actually swap them, we can improve bubble sort. So you can apply Shell sort to insertion sort and bubble sort and improve both algorithms that way. With insertion sort,you're cutting down on the number of shifting. And with bubble sort, you'd be cutting down on the number of swaps that you have to do.**

**Implementation**

