Data Structures & Algorithms in Python: Sorting Algorithms

In this 11-video course, learners explore workings of some of the most widely used sorting algorithms in Python and examine how their performance affects various measures. Key concepts covered here include various properties of sorting algorithms when selecting the right one for your data; the operations involved when sorting a list of values by using the selection sort algorithm; and the process of a bubble sort when applied to a list of values. Next, you will learn about the performance of the bubble sort on various measures such as time, space, and number of swaps; how to describe the steps involved in performing an insertion sort and compare it to the bubble sort; and learn the workings of the shell sort and the performance metrics of this divide and conquer algorithm. Finally, you will learn the process of sorting a list of elements using merge sort and list the complexity of this algorithm on various measures; and learn how the quicksort algorithm partitions and sorts a list of elements.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview*. [Video description ends]

Hi, and welcome to this course on sorting algorithms.

[Video description begins] *Your host for this session is Kishan Iyer. He is a software engineer and a big data expert*. [Video description ends]

My name is Kishan Iyer, and I will be your instructor for this course. A little about myself first. I have a Master's degree in Computer Science from Colombia University and have previously worked in companies such as Deutsche Bank, and WebMD in New York. I presently work for Loonycorn, a studio for high-quality video content. Data structures and algorithms are the bread and butter of coding. Any programmer will use these every single day.

And a good programmer will have an in-depth understanding of the various data structures and algorithm which can be applied in different situations. Data structures and algorithms go hand in hand. And which ones you pick determine not only the kinds of operations which you can perform, but also the speed with which they are executed and the compute resources which they consume. One of the most fundamental tasks in programming involve sorting a list of elements in ascending or descending order. This course lays the theoretical foundation for various sorting algorithms which are widely adopted in programming and are also the basis for questions in many coding interviews.

These include basic algorithms such as selection sort, bubble sort, and insertion sort. And also divide and conquer algorithms such as shell sort, merge sort, and quick sort. We visualize all the individual steps involved in these algorithms and also examine the time and space complexity for each of them when sorting a list of elements. Once you're done with this course, you will have a firm grasp of the workings of each of these sorting algorithms. You can then take up a course where you implement each of these in code.

An Introduction to Sorting

[Video description begins] *Topic title: An Introduction to Sorting. The presenter is Kishan Iyer.* [Video description ends]

In this course, we will be taking a look at many different techniques which are used in order to sort a list of items. That is, to arrange them in either ascending or descending order. Just to give you an overview of sorting, there are several algorithms which can be used for this purpose, and sorting algorithms come up in many different interviews. The different types of sorting algorithms all have various complexities both in time and space. And one of the reasons why many interviewers like to ask questions related to sorting is to gauge whether the candidate has a firm grasp of the concept of complexity.

One thing to note, though, is that sorting alone will not be enough for many different interview questions because this may very often turn out to be the foundation on which further questions are built. This does not change the fact, however, that it is crucial for you to have a firm grasp of the different sorting techniques and what the complexities, both in time and space, are for each of the sorting algorithms. So what are the types of questions which may be asked which are related to sorting, and are meant to test your understanding of sorting techniques?

[Video description begins] *Trade-Offs in Sorting*. [Video description ends]

Well, a lot of these are to do with the trade-off of the various sorting algorithms. The most common question is for you to be asked what the complexity of a particular sorting algorithm is, both in terms of time complexity and also in terms of the space required in order to execute that algorithm. You may also be posed questions such as whether the sort is stable. And another important one is the number of comparisons and the number of element swaps which are required in order to execute that algorithm. You may also be asked whether this particular sorting technique is one which is adaptive to the input array. I can imagine now that many of these terms will be entirely new to you, so I'll just rephrase these questions in a manner so that it becomes clear exactly what these terms mean.

We have already seen that the complexity of the algorithm is a measure of how the algorithm scales as the size of the input increases. Do the number of operations scale linearly with the size of the input, or is there an exponential relationship between the two? In terms of the space complexity, this is a measure of how much extra space is required in order to hold information while the sorting is being performed. Once again, given that sorting involves an input array, which will then produce an output array in a sorted order, there may be many different operations which are involved which require data to be stored in memory while the input is being sorted. When it comes to stable sorting algorithms, this is one where elements which are equal in value in the input array retain their order after the sorting has been performed.

As an example, consider that the data which you're sorting happen to represent the names of individuals. And specifically, you're performing the sort based on the last name. In your input, there may be several individuals who have the same last name but different first names. In such a case, will the ordering of the individuals in the input array be maintained after the sorting has been performed? And finally, when it comes to adaptive sorting techniques, these are algorithms which usually work better when the input array is already sorted to some degree. Now, there are some sorting techniques which are non-adaptive. That is, they will perform the same number of operations irrespective of the ordering in the input.

Adaptive sorts, on the other hand, are able to adapt themselves to the input, and will often break early if the input is already sorted. You can see here that it does help to have an in-depth understanding of the specific sorting technique which you are implementing. And it is important for you to know the various trade-off in the different sorting algorithms. These details are useful to know, not just from the point of view of programming interviews, but also when you're building any kind of application. For example, if you know that the input which you're getting already has some kind of ordering involved, then you know that you should make use of some adaptive sort in order to most efficiently sort that data. So now that you have a general idea of sorting algorithms, it is time to get specific. From the next video, we will begin taking a detailed look at different sorting algorithms.

The Selection Sort

[Video description begins] *Topic title: The Selection Sort. The presenter is Kishan Iyer.* [Video description ends]

It is now time for us to dive deep into the most commonly used sorting techniques. We will begin with the selection sort. Like any sorting algorithm, we will expect that there will be an input array containing either numbers or text, or even objects, and there this needs to be sorted in some order. Now in the case of a selection sort, we will be going through multiple iterations on the input data in order to sort it. Add at each of these iterations, we will select one element and then compare it with every other element in the list in order to find the smallest one. Once we find the smallest element, this becomes the first element in a sorted array. And at this point, we move on to finding the second smallest element in the input array and place it in the second position in the output. We then repeat this process for the third smallest element, the fourth and so on until the entire array is sorted.

So, in each iteration, we are selecting one element for a output array. And this is why this is called a selection sort. Let us now visualize exactly how this will work. We have here an input array which consists of nine elements, all of which are integers. So how do we sort all of these integers using selection sort? As I had mentioned earlier, we will be doing a number of iterations, and in each of them, we will find the smallest element within this array. But rather than place the elements into a brand new array, we will try to be a little space efficient. That is, we will simply rearrange the elements in this existing array so that we don't have to create a brand new array in order to perform the sorting. So how do we perform the selection sort in this case? Well, for the first element, we will perform a comparison with every other element within this array. So we compare the integer value of 15 with each of the others.

And then among all of the other elements, we pick the one which happens to be the smallest. In a specific example, this is the integer 11. And now that we have the smallest element, we need to place it at the front of the array. And the easiest way to do this is to swap the elements 15 and 11. So when we perform the swap, what we have is that the smallest element is now at the front of the array. We can now effectively reduce a sorting problem to sort a collection of the eight remaining elements in the array. So moving on now to the second iteration, where for the second position, we perform a comparison of all of the elements. So we compare 32 with each of the other elements in the array. And we find that the smallest element now remaining is the one containing the integer, 14. We select this particular value and then swap it into the second position in our array. So now 32 has moved to the end and we now have an array where the first two elements have been sorted and are in their rightful place.

For the next iteration, we simply need to repeat the process, but for position number three. So we compare 26 with each of the other elements. We find that among the remaining values, 15 happens to be the one which is the smallest. So we swap 26 and 15 so that we now have three elements which are in their rightful place. Moving on now to the fourth iteration, this time the number 19 happens to be the smallest, so we swap 19 into position number four. And now this process goes on. Among the remaining five elements in the array, number 26 is the smallest, so that is the one which now goes to the front of the unsorted array. And now this becomes the end of the sorted array. So we just repeat this process once more. It is time for number 32 to move to position number six. And now for the seven position, we only need to compare three values. So you'll observe that for each of these iterations, the number of comparisons which need to be performed keeps decreasing.

So we now swap element 36 into position number eight. And now we just have two remaining elements in the array. These also will need to be swapped because they are not already in order. So with this final swap, what we end up with is a fully sorted array. In order to execute this sorting algorithm, we did not need to allocate any additional space in order to create a sorted array. And we simply rearrange the elements within the input array. When we do something like this, we say that the sorting was performed in place. Though this may not seem to matter in the case of a specific sorting example where we only had nine integers to sort, in real life settings where sorting will need to be applied, you may often find that you are working with millions of elements. In which case, memory utilization does become a concern. And it is best to have in-place sorting algorithms rather than ones where the entire array will need to be recreated in additional space in memory.

So just to summarize how the selection sort algorithm works, we saw that the algorithm selects one element at a time and then compares it to every other element in the list. Out of the unsorted elements in the array, the smallest element is located in each iteration. And this is placed into its correct position in the sorted part of the array. So now that we have an idea of how the selection sort algorithm works, what exactly is the complexity of this algorithm, both in terms of time and space? We saw that in each iteration pretty much the entire list is checked in order to find the smallest element. The number of comparisons which need to be performed in each iteration is directly proportional to the length of the array. So at each iteration, we can say that there are about N comparisons which need to be performed. This, of course, is a bit of an approximation, as we have seen that the number of comparisons which need to be done drops by one with each successive iteration. So we start off with N comparisons in the first iteration and -1 comparisons in the second, and -2 in the third and so on.

So if you remember the formula for summing up such a sequence, it is N times N plus one over two. This though, is directly proportional to the square of n, which is why we say that the complexity of the selection sort algorithm, that is, the time complexity, is $O(n^2)$. You will recall that even if the number of operations which need to be performed contain both terms of n-square and N, we assume that the value of N is very, very large, in which case, N will be very small compared to n-square. So the complexity of such an algorithm will need to be $O(n^2)$. So what are some of the properties of the selection sort? For one, this is not a stable sort. This is because entities which are equal in value in the input array may in fact be rearranged in the sorted array. This is because when we perform the swap operation at each iteration, some elements may move from the beginning of the array over to the end.

So it is unpredictable exactly how equal value elements will be sorted. One more thing which we saw was that when we perform the sorting in place, then there is no extra space which is required. So we can see that the space complexity for this algorithm is O (1) since the extra space required is a constant value of zero. And as far as the number of comparisons and swaps are concerned, we have already seen that the number of comparisons can be represented as $O(n^2)$. whereas there will be a single swap operation which is performed at each iteration. As a result, the number of swaps can be represented as O(n). So this covers the selection sort algorithm. In the next video, we will take a look at another basic sorting algorithm, which is the bubble sort.

The Bubble Sort - Part 1

[Video description begins] *Topic title: The Bubble Sort - Part 1. The presenter is Kishan Iyer.* [Video description ends]

We will now take a close look at the bubble sort algorithm. This is one which is another basic sort. And its name refers to the fact that at each iteration, the largest element in the input array will effectively bubble to the end of the array. Let us now visualize exactly how the bubble sort works. We start off with the same input array of nine integers as we did previously in the selection sort video. And while in selection sort, the smallest element left in the unsorted array moves it to its rightful place, if you will. Something similar happens with bubble sort, except that the mechanism is just a little different. So we can now start off with the first iteration of a bubble sort. Here we compare the elements in the first two locations of our input array. And we check which one of these is larger. What we want is for the largest element to be moved to the right of the array. And in this case because 32 happens to be larger than 15, there is no movement necessary.

But what we do instead in this iteration is to move on to the next two elements. Specifically the ones in the second and third locations in our input. Here we compare 32 and 26, and because 32 happens to be larger than

26, we need to move it over to the right. The way we do this is to effectively swap the positions of these two elements. So in the first step of this iteration, where we compared 15 and 32, no swap was necessary because the larger element was already on the right. But in the second step, we did perform a swap in order to move 32 further to the right. We now continue with the comparisons. So the elements in locations 3 and 4 are compared, and once again, 32 is the larger element here. So we just swap the positions of these elements and now we move on to the next pair. Once again, the larger element is to the left, so we need to move it further to the right. And we once again perform a comparison between the next two elements. Since this pair is already sorted, there is no swap necessary, so we move along to the next step. And again, this pair is also sorted, so we move along now to the final step in this iteration. Where we compare the last two elements in our input array.

Here, the smaller element is the one on the right, so we need to perform a swap. So with this we end one iteration of the bubble sort. You'll observe that what has happened is that the largest element in the input array, specifically 44. Has now effectively bubbled its way over to the right. There are other implementations of bubble sort where you start off at the end and bubble the smallest element over to the left. And both of these are acceptable variations of this algorithm. However, the procedure remains pretty much the same except that the directions are reversed. Now we need to move on to the next iteration because we just have one element which is in its rightful place. We now still need to sort the remaining eight elements in the array. And for that we will once again perform a comparison for each pair of elements starting with the first two. So in comparing 15 and 26, there is no swap necessary. When we move on to the next pair of elements, we need to perform a swap in order to move the larger value of 26 over to the right. For the next pairwise comparison, we find that no swap is required.

So we move along and compare 32 and 19 in which case we will swap these two elements. So the number 32 is moving closer and closer to its rightful position. For the next pair of elements, again, no swap is required. So we proceed to the next step, no swap required here either. So we move along to the final pair in this yet unsorted array. So we will exchange the positions of 42 and 14. And what we now have are the largest two elements in their rightful place. At this point you will notice the similarities to the selection sort. But rather than finding the smallest value in every single iteration, what we do here is perform a series of pairwise comparisons. The effect though, is pretty much exactly the same. We proceed now to the third iteration in order to move the third largest element over to the right. And taking a quick look at what remains unsorted in this array, we know that it is the element 36 which will find its way to its rightful spot. But let us now see exactly what the remaining array will look like after all of the exchanges have taken place. So we perform a swap of the first two elements in order to move 11 over to the left.

There is no swap necessary for these two elements. So we head over to the next pair, 19 and 26 do need to be swapped, so we perform that. 26 and 32 are already sorted, so we just move along. This pair does not require any sorting either. But for the final pair we will need to swap them over so that 36 makes its way to the sorted part of the array. So we have now completed three full iterations. And you will observe at this point that just like with the selection sort. The number of comparisons which need to be performed at each iteration decreases by one. So in our first iteration for this input of nine elements, there were eight different comparisons which we needed to perform. And we are now left with six elements. So for our next iteration, there will be five comparisons of pairs which will be performed. So we start off with the first two elements and no swap is necessary. And the same also applies for the next pair. In fact, there is no swap needed for the third pair of elements either and as we move further along the array we find that no swap is required, once again. So with each iteration you are likely to find that the number of swaps which are required does go down.

This is because over the previous iterations, larger elements have moved just a little bit to the right and smaller elements to the left. And while this definitely applies in the worst case, where you have an array which is sorted in the reverse order. A swap will be required at every single step. We proceed with the comparisons now for the fourth iteration, and the final two elements will need to be swapped. So that we now have four sorted elements and five elements which are yet unsorted. We proceed over to the next iteration, once again, no swap is required here. The next step also doesn't require a swap. And this also applies for the third step in this iteration, but we need to move 14 closer to the left now. So we will exchange these two values and our fifth iteration is complete.

Heading over now to iteration number seven, there is no swap here. No swap here either and we will exchange the positions of 19 and 14.

All right, we are almost there, so we now have a rather small iteration. No swap is required for the first two and when we compare the last two elements, we will just exchange the values for 14 and 15. So at this point the element is actually sorted, but we still need to go through the last iteration where we just compare the final pair of elements. And we know that there is no swap required. In which case, all of the elements are in their rightful place and the smallest element is also at the left. What we have at this point is a fully sorted list. So now that we have an idea of how the bubble sort algorithm works. In the next video, we will take a look at a lot of its properties including the time and space complexity of this algorithm.

The Bubble Sort - Part 2

[Video description begins] *Topic title: The Bubble Sort - Part 2. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we visualized exactly how the bubble sort algorithm works. So let us now quickly summarize what we have learned about bubble sort. In this particular example, we saw that the largest element bubbles over to its correct position at each iteration. This is accomplished by means of pairwise comparisons of adjacent elements and then swapping them over if they're not in the correct order. We could also vary the bubble sort so that it is the smallest element which bubbles to the left part of the array, but the logic of these pairwise comparisons remains intact. Now, in the worst case, that is, if the list is in the opposite order in which it needs to be sorted, then there will need to be swaps at every single step.

Though the number of swaps increases in such a case, the number of comparisons does not. Now, the number of comparisons or operations for each iteration is again proportional to the length of the list. And in this way, the bubble sort is similar to selection sort, which is why its complexity in terms of time is also $O(n^2)$. Bubble sort, however, happens to be a stable sort, because we won't be performing a swap if the two elements are of the same value.

So the elements of the same value in the input array will retain their ordering in the output as well. Now, the implementation of bubble sort, which we saw, did not provision extra space for the output array. So it doesn't really require much extra space. What you may require, though, is a variable in order to perform the swapping of elements at every step. Whatever the case though, when you perform an in-place bubble sort, the amount of extra space which you require is constant, no matter what the size of the input array. Which is why we say that the space complexity for the bubble sort is O(1). Also, when it comes to the number of comparisons and number of swaps, in a simplistic implementation of the bubble sort, you may perform the same number of comparisons, no matter what the state of the array.

In such a case, the number of comparisons is always $O(n^2)$. And this sort is not really adaptive. However, to make the sort adaptive, and then exit if the array already happens to be sorted before all the iterations, you can keep track of the number of swaps performed at every iteration. If one iteration goes through and there were no swaps necessary, it means that the array is already sorted and you can just break away from the loop. On the other hand, in the worst-case scenario, we saw that a swap will be required at every single step of every single iteration. In which case, the number of swaps can also be represented as $O(n^2)$.

The Insertion Sort

[Video description begins] *Topic title: The Insertion Sort. The presenter is Kishan Iyer.* [Video description ends]

We will now explore the insertion sort algorithm. This is one which may be familiar to many card players, where once you get a set of shuffled cards, you will go through each of the cards one by one and insert each card into its correct position in an orderly sorted set. We will now visualize what the insertion sort algorithm looks like by once again starting with this array of nine integer values. So at first, we assume that there is nothing in this array which is sorted. And we begin the first iteration by comparing the first two values. We notice that they are already in order, so a swap need not be performed. In this case, we assume that we have a sorted list containing one element, which at the moment only contains the integer, 15. And then we go on to take a look at the third element. Each element which we examine needs to be compared with the previous one. So in this case, we compare 26 with 32, and since 26 is smaller, we will need to perform a swap. The swapping is not quite complete though, as we will now compare 26 with the previous element in the array.

So here, 15 and 26 are already in the correct order. So at the left, we now have two elements which are in the correct order. So this is what our sorted list is, whereas the remaining seven elements to the right are unsorted. One important thing to keep in mind, though, is that what I'm referring to as the sorted list is not quite static. It only means that at this point in time the two elements to the left are in a sorted order, but the elements which occupy those positions can move around. We now proceed with the comparisons. So we take a look at the fourth element in our list, which is 11. And since this is smaller than the previous value, we will perform a swap and move it further to the left. We will compare with the previous element once more, and since 11 is smaller, there is one more swap which needs to be performed.

And a third swap is required for this element to move further to the left of the list. Now the sorted part of the array contains three elements. 11, 15 and 26 are in a sorted order, whereas the remaining six elements are not. At this point, you may recognize that the insertion sort resembles the technique used by many card players when arranging the cards which they have been dealt. We will now proceed with the comparisons. So the element 36 is compared to 32, but no swap is necessary now. So we assume that the sorted part of the array now contains four elements. The element 19 now needs to make its way to its correct position in the sorted part of the list so we swap this with 36. We then compare with 32 and then swap once more, and then compare with 26. And this is the final swap which is required.

When we perform this comparison with 15, we see that 15 happens to be smaller, and we can be confident that in this part of the array, all elements further to the left of 15 will be smaller. Which means that we can now move ahead to the next element in our array, since the first five elements have now been sorted. So here, we start off by comparing elements at positions 6 and 7. No swapping is necessary here. And we assume that the sorted part of the list is now six elements long. For the next two elements, no sorting is required either. So once again, a sorted part of the list just grows a little larger. For the final pair of elements, we notice that 14 is smaller, so we now need to move it to the correct location in the sorted list. Comparing with the previous element, we realize a swap is necessary, and we'll need to keep performing swaps until 14 makes its way to its correct location. So we exchange once more.

14 is smaller than 36, and smaller than 32 as well, and the element needs to go all the way to the second position in the array. So we will continue swapping until the comparison with the previous element of 11 shows that there is no further this element can go in this list. And at this point, the sorted part of the list is equal to the entire array. So what we have now is a fully sorted array. So we can now move ahead and take a look at the properties of the insertion sort. We saw that at each iteration, we come up with a sorted subarray, or a sublist. With each iteration, the size of the sublist continues to grow until at the final step, this sublist becomes equal to the entire list. And it is at this point where we consider that the array is fully sorted. We begin the insertion sort by assuming a sorted list with the size of 1 and then keep inserting additional elements into the right position within the sorted list. And the size of the sorted list keeps growing. In the worst-case with the insertion sort algorithm, when the elements are in descending order, we will effectively need to perform swaps at every single step in order to move an element over to the left of the list.

This will also maximize the number of comparisons which are necessary, which means that at every iteration, we need to perform N element comparisons and swaps. The number of iterations itself will be proportional to the size of the array, which is why the time complexity of the insertion sort, definitely in the worst-case, can be considered to be $O(n^2)$. In the case of elements of equal value in the input array, there won't be any swapping performed between the elements. In which case, we can say that they will retain the order in the output, which is why the insertion sort is a stable sort. Just like with the previous sorting algorithms we have seen, we perform the sorting in place, which is why the space complexity for the insertion sort is O(1). And we have already covered the fact that in the worst-case, the number of comparisons and the number of swaps which would be required are proportional to the square of the size of the input array, which means that these are O(n^2) operations. So one thing to note is that this is very similar to the bubble sort, in that the insertion sort is an adaptive algorithm.

We saw in our example that when moving an element towards the left of the array, once we encounter an element to its left which happens to be smaller, we can come to the end of that particular iteration. As a result, if the input array happens to be sorted already, then we effectively just need to perform one comparison at each iteration rather than repeat all of the same stuff irrespective of the state of the array. In the worst case though, when the input array is in descending order, the number of sorts and comparisons will be very high. So we say that the insertion sort is an adaptive sorting algorithm. One important factor about the insertion sort is that the overhead for this is rather low, which is why it is often the algorithm of choice when sorting subarrays in other sorting algorithms which adopt a divide and conquer approach. We will be exploring these just a little later in this course.

Many of those algorithms will ensure that the insertion sort gets to work with sublists which are already sorted to a great degree. And the adaptive nature of this sort lends itself perfectly for this purpose. We now compare the insertion sort with the other adaptive sort algorithm, which is the bubble sort, and see why exactly insertion sort is a little better. In the case of a bubble sort, if at the end of an iteration, the array already happens to be sorted, we still need to do one pass over the entire array. And then only when we know that there have been no swaps performed in that iteration can we exit from the sorting function. Also with the bubble sort, we need to perform N comparisons at every single step. Whereas with an insertion sort, we can stop the comparisons once we find that an element can no longer move towards the left of the array.

Also, the bubble sort algorithm requires a number of writes and swap operations to be performed, which is why it performs rather poorly on modern hardware. The number of comparisons which are required and the number of swaps means that there will be several cache misses when running the bubble sort algorithm. And this is what leads to its additional overhead when compared to the insertion sort. The insertion sort performs well thanks to the fact that it can stop the comparison of elements when it finds that the element has found its rightful place within the sorted part of the array.

The Shell Sort

[Video description begins] *Topic title: The Shell Sort. The presenter is Kishan Iyer.* [Video description ends]

When we previously discussed the insertion sort algorithm, I had mentioned that there are many divide and conquer algorithms which make use of insertion sort in order to arrange the elements in the subarrays which those algorithms produce. One such divide and conquer algorithm is the shell sort. As always, we will visualize how exactly the shell sort algorithm works by taking a look at our array of nine integers. One feature of this list is that the second smallest element, which is 14, happens to be at the very end. In the previous sorting algorithms which we covered, we observed that there are a number of different operations, both in terms of comparisons as well as swaps, which need to be performed just in order to move that element over to the left. In a real life setting where the arrays will be much larger, such problems will only be magnified. What many divide and conquer algorithms are able to accomplish is to move a lot of these elements closer to their rightful place much quicker and thus avoid having to perform a lot of swaps and comparisons in the subsequent iterations.

How does this work in the case of shell sort? Well, the principle behind this sorting mechanism is that the list is divided into a number of different sublists. Each sublist will be sorted using insertion sort, and the sorting of each of those sublists comprises one iteration of the shell sort. In the next iteration, we will create fewer sublist than in the previous one and then use insertion sort to sort each of those. In the final iteration, we will have only one sublist which is equal to the entire list. At this point, the array will be nearly sorted and we can make use of the adaptive nature of the insertion sort to very efficiently sort that list. So in this particular example, we will only examine a case where we divide our array of nine elements into two sublists. The element shaded in red represent the first sublist, and those in blue represent the other. Here we say that, for each of the sublists, the gap between the elements in the original list is 2, or the increment = 2.

So we go ahead and sort the red sublist, and what we see is that the element, which is 14, has moved all the way from the right over to the left. You may also instinctively know that each of the elements is now much closer to its final positions in the sorted list. And this is precisely what the shell sort seeks to accomplish. We will now go ahead and sort the second subarray, which is currently represented in blue. And what we have now are two sorted sublists. In this particular example, we created two sublist out of the original input, so the initial gap, or increment, was 2. In a more realistic setting, your gap may be much larger than that. And in the demos of this learning path, you will see that we start off with a gap which is equal to half the length of the entire input array. Whatever the gap which is selected, what you will typically end up with is an array which is far closer to being sorted than it was previously.

And we can keep reducing the size of the gap or increment until, at the final stage, there is just one sublist which we need to sort, and this is equal to the entire list. So in this case, we say that a gap or increment is equal to 1. So once again, we will perform an insertion sort in order to get these elements in the correct order. After the first iteration, the element 11 moves over to the sorted part of the list. And since we're moving from an increment of 2 to an increment of 1, the most number of swaps we will require for each iteration of this insertion sort is 1. So 14 and 15 don't need to be swapped, so we move ahead to the next iteration. These numbers don't need to be swapped either, so we move ahead to the next iteration once more.

No swap necessary again, which is also the case with the next iteration. Again, we keep encountering the fact that the elements are already sorted and no swaps need to be performed. And this goes on until we come to the final two elements in a list, and this is the one occasion where a swap is necessary. We will now need to compare 42 with the previous element. But in the previous shell sort iteration, where we had two different sublists, both 42 and 36 were in the same sublist and they were already sorted. Which is why we can stop the comparisons right here, and what we end up with is a fully sorted list. Just to reiterate, this divide and conquer algorithm makes use of the fact that an insertion sort is an adaptive sort and that it works very well when the array elements are already sorted to a great degree.

Adaptive sorts tend to suffer a lot when the elements in the input list happen to be very far away from their rightful place. By dividing the entire list into evenly spaced subarrays, we can bring a lot of elements, which are far away from their rightful place, much closer to their final location, within a very few steps. So now that we have visualized how the shell sort works, Let us quickly summarize some of the properties of this algorithm. As we had discussed, at the very heart, shell sort uses insertion sort in order to sort each of the sublists. And the main contribution of the shell sort is to generate these sublists. Now, even though we could intuitively guess that a shell sort performs much more efficiently than a regular insertion sort, and this is also backed up by trials on a lot of real data, getting the exact complexity of the shell sort and quantifying its performance is very difficult.

This is because the exact performance of the shell sort very much depends on the values of the increments or gaps which are chosen. Which can then lead us to the question, what is the best value for this increment? And the answer is that it's not really clear. What we know, both instinctively and also empirically from tests, is that the complexity of shell sort is better than that of regular insertion sort. And this is because the final iteration of the shell sort involves an insertion sort on a nearly sorted array. The worst case complexity of an insertion sort is O(n^2), and the best case complexity is O(N). So what we can safely say is that the complexity of shell sort is somewhere between these two values. The reason we specify a range rather than a specific value for the complexity, is that the different values for the increments can produce different complexities.

Now there are various values which we can use for the increment. And once again, as I mentioned in the demos of this learning path, we will start off with an increment value equal to half of the length of the input array. But you can say that the average complexity would be O(n to the power 3/2). Because shell sort at its heart is an insertion sort, this is also very much an adaptive sorting algorithm. If the original input array happens to be sorted to a great degree, then the number of swaps and comparisons required in each of the iterations for each of the sublists will be that much lower. And finally, when it comes to the space complexity for the shell sort, just like the insertion sort, the amount of space required is constant. And it's not dependent on the size of the input array, which is why it is O(1).

The Merge Sort

[Video description begins] *Topic title: The Merge Sort. The presenter is Kishan Iyer.* [Video description ends]

We previously took a look at shell sort, which is one of the divide and conquer approaches towards sorting a list of elements. We will now take a look at another one of those divide and conquer algorithms, which is the merge sort. So we begin by taking a look at some of the properties of this algorithm. As mentioned earlier, is to adopt the approach of divide and conquer. What exactly does that mean? Well, consider that the original problem which we have is a rather large one. To be more precise, it's a very large list which we need to sort in ascending order. The way to do that is to break down this large problem into a number of more manageable sub-problems. Specifically with merge sort, the original list is broken down into a number of smaller parts in a recursive manner. So we start off with the original large list, we break it down into few smaller lists, we break down both smaller lists into even smaller ones and so on. When this process continues, at some point we will be left with a list containing exactly one element. And we consider this one element list to be a sorted one.

So this is the divide part of the divide and conquer. And what exactly takes place in the conquer phase? Well, this is where the name merge sort comes from. We start off with a bunch of single element lists, and then we merge these lists together into a number of sublists. And during this merge process, we ensure that these sublists are sorted. We continue merging each of these sublists until at the end, what we end up with is a single list which is entirely sorted. So, the merge sort algorithm is a classic case of a recursion-based algorithm. That is, you continually divide the problem until each unit is so small, that solving that particular sub-problem is virtually trivial. Once you solve the trivial case, then solving the bigger problem as you merge the sublists together becomes that much easier. Until at the very end, you get the solution to your problem.

Now one thing to be careful of when using the merge sort algorithm is that the few lines of code which we require in order to implement recursion does almost seem like magic. However, as with any recursive algorithm, you need to be very careful in accounting for the boundary case. So now that we have a high-level understanding of the merge sort algorithm, we can visualize exactly how this will work on an array of elements. So here we have an array of eight integers, so one fewer element than what we have been working with so far. This divide and conquer algorithm starts off by dividing this input array into two subarrays. Each of these subarrays in turn will be broken up into two subarrays. And the idea behind this is that sorting small subarrays and then combining them, is a lot easier than sorting the large arrays directly. So we're not quite done with our dividing though, where we will break down each of these sublists of two elements into, quote unquote, subarrays of one element each.

Each of these one element subarrays, can be considered to be sorted in their own right. And this is where the divide part of a divide and conquer ends, and the conquer phase begins. So now that we have eight sorted subarrays, the idea now is to reverse the division process, and begin a merge process. While merging each of these one element subarrays into subarrays of two elements though, we will make sure that the sublists of two elements are in sorted order. For example, the elements 26 and 11, which in the previous step were part of the same subarray, will be merged together. However, while this is happening, we will need to exchange their position so that they are in sorted order. Something similar happens with the elements 19 and 36 as well.

The other two pairs of elements are already in a sorted order, so no swapping will be necessary there. So we will now perform the combine operation or the merge operation, and we end up with four sublists of two elements each. Each of these sublists are sorted. We continue with the merging process, where we will now combine two pairs of the sublists, to produce two sublists of four elements each. Each of these sublists are now sorted. And in the final phase, we will just combine these two sublists in order to produce a single array. And this of course is the fully sorted array. So this gives an idea of how the merge sort algorithm works, and we can now go ahead and summarize some of its properties. We have seen that merge sort adopts the process of divide and conquer, in order to create smaller and smaller problems until it gets to the point where we end up with single element sublists.

Now, as for the complexity of the merge sort, we will need to consider the fact that at each recursive step, the length of the array which needs to be sorted is effectively halved. Now, the exact derivation of the overall complexity is a little complicated, and is not really relevant for programming interviews. Though it may come up if your job happens to be heavily math focused. It is enough for you to know that the complexity of merge sort is O(nlogn). The logarithm over here has a base of 2, because we are subdividing our list into two equal parts at each iteration. So what about the space complexity of the merge sort?

Well, when we create all of the sublists which we require for this, it takes up O(N) extra space, since we're effectively allocating space for an entire copy of the original list. This sort, however, is one which is stable, so elements of equal value will retain their original order in the sorted list. And finally, this is a sorting algorithm which is not adaptive. So no matter what the original ordering of the elements, merge sort will execute in exactly the same time, and will perform the same set of operations.

The Quicksort - Part 1

[Video description begins] *Topic title: The Selection Sort. The presenter is Kishan Iyer.* [Video description ends]

It is now time for us to examine another divide and conquer approach towards sorting an array of elements, and this is the quick sort algorithm. In many programming languages, if you call the native sort function in order to sort an array of elements, the algorithm which is actually implemented under the hood, is in fact a quick sort. Let us now begin by taking a look at some of the properties of this algorithm. Once again, just like merge sort or shell sort, this is a divide and conquer approach towards solving the sorting problem. In this case, the list which needs to be sorted is partitioned into smaller units. However, unlike the merge sort algorithm, this partitioning is not really based on the length of the original array or some artificial index value. But it is based on something which is called a pivot. We will see exactly what this means just a little later, but for now it is enough for you to know that a pivot is one of the elements which is picked from a list. The list is partitioned in such a manner that all the elements smaller than this pivot element goes to one side of the list.

And those elements which are larger than the pivot move over to the other side. So at each iteration, we have two different partitions for the list, one containing elements smaller than the pivot, and the other containing elements which are larger. Each of the partitions now represents a sublist, which in turn are partitioned based on a pivot element. So this partitioning of the list will keep going on until all of the sublists created by the partitioning happen to be sorted. As always, it is now time to visualize exactly how the quick sort algorithm works. We go back to our array of nine integers here. And the first step for us is to choose one of these elements to become the pivot based on which we will partition the list. Now, there is no specific element which has to be picked as the pivot. But common practices include taking either the leftmost element in the list, the rightmost element, or the one in the middle of the list. The pivot element will need to be selected for each of the sublist which will be partitioned later on.

But for the first iteration, the sublist we will be working with happens to be the entire list. And in this particular example, we will be selecting the rightmost element of the list as a pivot element. So when we make this choice of pivot, what it means is that this particular element, which is 14, will make its way to its correct position in the list during this iteration. In the course of doing that, we must also ensure that all the elements to the left of the pivot happen to be smaller than it, and all of the elements to its right are larger. So we can now iterate through each of the elements to the left of our pivot element, starting with the element in the first position, which is 15. Since this is greater than the pivot element, we won't really do anything with it. The same also applies to the next two elements, which are 32 and 26.

However, when it comes to the element 11, this happens to be smaller than our pivot element. This means that we need to move 11 right over to the left of the list. The way to do this is to swap this element with the element which is in the leftmost point of the list, so we will just exchange the positions of 11 and 15. And we have now carried out our first swap in our sorting operation. We'll now iterate through each of the other elements in the list. And we realize that there are no more elements smaller than our pivot, so we don't do any more swaps in order to move elements over to the left. This means that our pivot element belongs to the second position in the list, which is currently occupied by the element 32. So to end this iteration, we will swap the positions of these two elements, so that what was a pivot element of 14 is now in its rightful place in the array.

Now all of the elements to the left of 14 are smaller, and those elements to its right happened to be larger. We now have two different sublists, one containing just the single element of 11, and the other containing the elements 26 through 32. Since any single element list can be considered sorted, we will ignore that and we'll instead go ahead and sort the elements to the right of 14. We apply the quick sort algorithm for this once more. And on this occasion, the pivot element will once again be chosen as the one to the extreme right, which in this case is 32. So we start off by comparing 32 with each of the other elements within the sublist starting with 26. And in each case, we will be examining the position as a potential position for this pivot element to move to. So 32 happens to be larger than 26, so its rightful place in this array is going to be to its right. So the potential place for the pivot element is now the fourth position in this list.

And this is currently occupied by 15, which is smaller than 32, so the rightful place for our current pivot element is further to the right. And when we examine the fifth location in this list, this is potentially the rightful place for our current pivot. So we put a marker over there, and then examine the elements to its right. Any elements which are smaller means that the rightful place for our current pivot element of 32 will need to move further to the right. And also, we need to move the smaller element to the left of that position. So in this case, we need to swap the elements so that the first five elements now are less than 32. And the potential rightful spot for 32 is now the sixth position in this array. We then examine the contents to the right of this potential rightful spot to see if any of those are smaller. And given 42 and 44 are in fact larger than 32, we now know that the rightful place of the element 32 is the sixth position in our array.

That position is currently occupied by the element 36. So we'll simply swap these two out, and with that, the element 32 is now in the rightful place in our array. So at this point, the elements 11, 14 and 32 are in their correct places. However, we will now apply the quick sort algorithm to the unsorted parts of the array, which are to the left and right of 32. That is, the arrays which are on either side of what was previously our pivot element. So we will now apply quick sort on this subarray of three elements, and once again, we will pick the rightmost element as our pivot. We once again iterate through the elements until we find that 15 happens to be less than 19. So the potential final rightful place for the element 19 is where 15 is right now. So we move 15 over to the left of that spot by swapping it with 26, and then we move this pivot over to its correct location by swapping positions with 26.

So with that, 19 is now in their rightful place in the array. And the unsorted arrays to its left and right contain exactly one element, 15 and 26. So we can consider those elements to be in the rightful place as well. So after we used 32 as a pivot element and moved it to its correct place, we then went ahead and sorted the partition to its left. And it's now time to sort the other partition, the elements to its right. By virtue of being in the rightmost position, it is the element 36 which becomes the pivot. And since the other two elements in the subarray happened to be greater than 36, the rightful position of this particular element happens to be at the left of this subarray. So we just swap out the positions of 36 and 42.

So now, 36 makes its way to its rightful place in the array. There are still two more elements which makes up the subarray to the right of 36 which need to be sorted. 42 becomes the pivot this time, and its rightful position is towards the left of the subarray, so we make the swap in order to get it there. And with that done, there are no more partitions for us to sort, so what we end up with is a fully sorted array.

The Quicksort - Part 2

[Video description begins] *Topic title: The Quicksort - Part 2. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we visualized exactly how the quick sort algorithm works. Now, when it comes to the implementation of the quick sort, there is a partition function which needs to be defined. This is given a subarray as its input. And what it does is pick one of the elements in that subarray as the pivot and then make sure that the pivot element moves to its rightful place in that subarray. So the element to the left of the pivot will be smaller and those to its right will be larger. This will usually return the index at which the pivot element was moved to in the subarray. In addition to the partition function, there is also a quicksort function which needs to be implemented. And this is the one which makes all of the recursive calls to the sublists.

It is the quicksort function which invokes the calls to the partition function and ensures that each of the partitions are sorted. We will see exactly how this is implemented in the labs of this learning path. So just to summarize the quick sort algorithm, this, once again, uses the divide and conquer approach in order to create smaller problems to solve. The smaller problems in this case refers to the sublists which need to be sorted. Now like with the other divide and conquer methods which we have seen so far, the complexity for the quick sort is something which needs to be derived. And the exact derivation is a little too complicated and is beyond the scope for this course. In fact, this is not even required for programming interviews. However, what you do need to know is that the average case time complexity for this algorithm is nlog(n).

Once again, the base for the logarithm is two. Moving along now to some of the other details for this algorithm. In terms of the space complexity, this is a factor of O(log(n)), at least in the average case. The worst case space complexity for quick sort is O(n). The mathematical derivation of this complexity is something which is beyond the scope for this course. As far as the stability of quick sort is concerned, this is generally considered unstable, though there are certain implementations which make it a stable sort. Another feature of quick sort is that it is not adaptive. So it doesn't matter what the original ordering of the elements is, the number of operations which quick sort performs will be the same.

Exercise: Sorting Algorithms

[Video description begins] *Topic title: Exercise: Sorting Algorithms. The presenter is Kishan Iyer.* [Video description ends]

In this exercise, you will start off by describing some of the considerations when picking a sorting algorithm. There are many different metrics for the performance of sorting algorithms. And these are definitely things which you should keep in mind when making your choice. You will need to recall what these metrics are and which properties of the algorithm they measure. Next, you will list down the properties of the insertion sort algorithm. During this course, we took a look at an example of how the insertion sort algorithm works.

You will describe at a high level what exactly it does and then also list various properties which include the complexity of its operations. Next, you'll summarize the shell sort algorithm. This is a divide and conquer approach towards sorting a list of elements. And you will need to recollect the fundamental sorting algorithm used by shell sort and what exactly its divide and conquer approach implies. Each of these topics were covered during this course, so please pause this video and spend a few minutes to do this exercise on your own. For the first task, you needed to identify some of the considerations when picking a sorting algorithm.

[Video description begins] *Solution. Considerations in Sorting Algorithms.* [Video description ends]

Now, there are various factors which can affect the type of algorithm which you pick, but one of the crucial ones is always its time complexity. This is a measure of how the number of operations performed by the sorting algorithm varies with the size of the input. So does the algorithm sort in O(n^2) times, or does it take O(n) in the best case? These are all factors which you need to consider. Another crucial element is the space complexity of the algorithm. And what is usually meant here is how much extra space is required in order to perform the sort. Does the sorting occur in place, in which case, the elements are simply moved around in the input list. Or does an entirely new list equal to the size of the original array need to be recreated? The space complexity of an algorithm can affect the amount of memory which is used by it, and this is why this is something you really need to consider.

Another thing you will keep in mind is the number of swaps and the number of comparisons which are performed by your algorithm. Whichever metric you adopt, though, whether time, space, swaps, or comparisons, you should always consider not only the average case, but also the worst-case scenarios. Now, whether there is a vast difference between the average and worst case depends a lot on whether the sort is adaptive. An adaptive sort is one which works more efficiently if the list it is working on is already sorted to a great degree.

And one more factor to keep in mind when picking a sorting algorithm is whether the sort is stable. A stable sort is one where elements of equal value in the input list which occur in a specific order will retain that order in the sorted list. Once you have all of this information in hand, then picking the correct sorting algorithm for your specific situation becomes a much simpler task. Next, you needed to identify some of the properties of the insertion sort algorithm.

[Video description begins] *Insertion Sort Properties*. [Video description ends]

We saw that when using the insertion sort algorithm, you iterate over each element in your list and then insert it into its correct position in a sorted sublist. You start off with the first two elements in your list and sort those. You then move to the third element and make sure the first three elements are sorted. And then you ensure that the first four elements are sorted and so on. Now, when performing the insertion sort, the worst case time complexity is $O(n^2)$. This is the situation when the input list happens to be in descending order. When that happens, the number of comparisons and the number of swaps which you need to perform is $O(n^2)$. Essentially, you will be performing comparisons and swaps at every single step.

However, the insertion sort is also an adaptive algorithm, which means that it works much better when the list is already sorted to a great degree. In fact, the best case time complexity is O(n). When it comes to the space complexity of the insertion sort, if the implementation involves an in-place sorting, and the extra space required will be constant, or O(1). Insertion sort is also a stable sort, which means elements of equal value in the input list will maintain their original ordering in the output list. And finally, insertion sort, due to its very low overhead, also happens to be the base algorithm for many divide and conquer sorts, such as shell sort, for example. And speaking of shell sort, we now move to the last task in the exercise where you needed to describe that divide and conquer algorithm.

So here is a short description of this algorithm. So this is where you divide the input array into a number of sublists and then sort each of those sublists using insertion sort. You typically start off with a large number of sublists and then at each iteration, you reduce the number of sublists. Given that insertion sort is an adaptive sorting algorithm, the goal is to start off with a very trivial sort involving very small sublists. And as the size of the sublist grows in each iteration, we will be working with data which is already sorted to a great degree. And by the time we get to the final iteration, we are left with just a single sublist which is equal to the entire input array. At this point, however, most of the sorting has already taken place, in which case this iteration will be very quick and there will be far fewer comparisons and swaps than had you used insertion sort directly on the input list.

Moving on now to the properties of the shell sort, its time complexity is in fact rather hard to determine. And, in fact, it varies with the size of the increment or gap which you use to form the sublists. However, it has been calculated that the time complexity lies somewhere in the range between O(n) and $O(n^2)$. This algorithm performs an in-place sort, which is why the extra space requirements is O(1). It is also an adaptive sort because shell sort is based on insertion sort, which in turn is an adaptive algorithm.