Advanced Data Structures & Algorithms in Java: Sorting & Searching Algorithms

Data structures and algorithms are vital tools in writing robust and performant code. Being trained in the proper use of these tools is the responsibility of all Software Engineers. Using these tools correctly involves recognizing which algorithms to use for which scenarios.

In this course, you'll identify and implement several algorithms to sort data stored in a list. You'll explore the various trade-offs made with sorting algorithms in terms of time and space complexity. You'll recognize the nitty-gritty details of sorting algorithms such as selection sort, bubble sort, insertion sort, and divide and conquer algorithms like merge sort and quick sort.

Finally, you'll learn to use searching algorithms that allow you to quickly look elements up in a sorted list such as binary search, jump search, and interpolation search.

Table of Contents

Course Overview

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

Hi, and welcome to this course, implementing sorting and searching algorithms in Java. My name is Janani Ravi, and I will be your instructor for this course.

[Video description begins] *Your host for this session is Janani Ravi. She is a software engineer and big data expert.* [Video description ends]

A little about myself first, I did my master's from Stanford University and have worked at various companies including Google and Microsoft. I presently work for Loonycorn, a studio for high quality video content. The study of data structures and algorithms involves the study of concepts that are the foundation of writing correct, clean and robust programs. The knowledge of basic data structures, their characteristics and the algorithms that go hand in hand with these data structures are important tools in a software engineer's toolkit.

Having the right tools for the job can make the job a lot easier. Being trained in the proper use of the tools is essential to be effective. Data structures and algorithms allow you to write performant code, which minimizes the use of scarce resources and helps ensure that your code is not just correct, but also fast. In this course, we'll understand and implement a number of algorithms to sort data stored in a list.

We'll explore the various trade-offs that we make with sorting algorithms in terms of time and space complexity. And we'll understand the nitty gritty details of sorting algorithms such as selection sort, bubble sort, insertion sort and shell sort. We'll also explore divide and conquer algorithm such as merge sort and quick sort.

We'll also study searching algorithms that allow us to quickly look up elements in a sorted list, algorithm such as binary search, jump search, and interpolation search. Once you're done with this course, you'll be able to visualize how exactly sorting and searching algorithms work, and implement even the most complex algorithms in an efficient and performant manner.

Sorting Algorithms and Trade-offs

[Video description begins] *Topic title: Sorting Algorithms and Trade-offs. Your host for this session is Janani Ravi.* [Video description ends]

Any study of algorithms is incomplete if we don't study sorting algorithms. Sorting algorithms are what we use to sort a list of comparable values into ascending or descending order. All of the sorting algorithms that we'll discuss here in this course apply equally well to sort elements into ascending or descending order. But for the most part, we'll only work with sorted items in the ascending order just for the sake of simplicity. Any student of computer science needs to have a strong understanding of sorting algorithms.

Because you need to know the right trade-offs that each kind of sorting algorithm makes. Sorting algorithms are also an interview favorite. So if you plan to interview for a job, these will be extremely useful to know and implement. Sorting algorithms are an entire category of study by themselves. There are many varieties and flavors of sorting algorithms, which have different time complexities and different space complexities.

So the sorting algorithms tend to be very different and make different trade-offs in terms of time and space. It's quite possible that you will run into sorting algorithms often. Whether it's in a programming interview for a software development job or in the real world when you're coding up a project. Many difficult problems, whether they are interview problems or other kinds of problems are built on top of the ability to sort entities quickly.

So no matter what you're working on, sorting algorithms form a foundational building block for any student. So it's important that you understand how sorting works, and understand the time complexities of the various algorithms involved. If you are to choose between sorting algorithms in the real world, your decisions will be based on the trade-offs that are involved. So every sorting algorithm makes trade-offs.

So you need to ask yourself the right questions. The first question that you will ask yourself is, what is the time complexity of the algorithm used in the worst case? While answering this question, what you're trying to figure out is how does the algorithm scale as the input size increases? So if the input to an algorithm is n, that is, you have to sort n elements, how does the algorithm scale based on n? If you need your sorting algorithm to run very, very fast, you'll choose an algorithm with a lower time complexity. The next question that you might ask

yourself is, how much space does the sorting algorithm occupy?

Now if you're sorting n elements, there is of course, the space that the n elements already occupy. But does your sorting algorithm need additional space? This is referred to as the spatial complexity of the algorithm. For every sorting algorithm, you need to know if it requires additional space to hold information during the sorting process. If the answer is yes, additional space is needed, how much extra space does this need? The spatial complexity of a sorting algorithm is particularly important when you are in a space constrained device such as on mobile devices.

You might want to trade-off spatial complexity versus time complexity. That is, you'll choose a slower algorithm which performs sorting in place rather than use additional space. Time and space are both resources available to you. And this trade-off is the most important trade-off that you'll make for sorting algorithms or for any other algorithm and any other purpose. Time and space can be traded off so that you use more of the resource that you have plenty of, and less of the resource that is constrained.

There are other details about sorting algorithms that might be important to you, such as, is the sort stable? A sorting algorithm is said to be stable if equal elements maintain their original order after sorting. Let's say you have two elements with exactly the same value. In the original unsorted list, they were in a certain order if they continue to remain in the same order. But in the right sorted positions, after sort, that sort is stable. Another detail that might be important while making your choice of sorting algorithms is how many comparisons need to be performed? And how many element swaps need to be performed in the actual sorting process?

This question is relevant because there might be certain algorithms that work better with nearly sorted lists. So if lists are almost sorted, there are certain algorithms which can get it into the completely sorted form with very few swaps and comparisons. And this detail brings us to the next question which is relevant for a sorting algorithm. Is the sort adaptive?

Adaptive sorts tend to be far more efficient than non-adaptive sorts because adaptive sorts have the ability to break early. Without going through all of the iterations required in a sort if the algorithm finds that the list is already sorted, it'll simply stop and will not perform the remaining iterations. Because it knows that the iterations are not needed. If the sorting algorithm is not adaptive, even if the list is completely sorted, the sorting process will continue till the very end.

Selection Sort

[Video description begins] *Topic title: Selection Sort. Your host for this session is Janani Ravi.* [Video description ends]

The first sorting algorithm that we'll visualize and then implement in just a little bit is the selection sort. The selection sort can be thought of as the introduction to sorting. It's the simplest of all sorts to understand but it's not very efficient. So if you're choosing a sorting algorithm in the real world, you won't really choose the selection sort. So how does the selection sort process work on a list of elements.

In every iteration of the sorting algorithm you will choose one element and compare that element with all other elements that exist within your list. You'll then find the smallest of all elements. So one iteration comparing one element with all other elements will give you the smallest element and this is the chosen element. And this chosen element you will move to the very beginning of your list. You'll make this smallest element the first element.

At this point, sorting in ascending order, the first element which is the smallest is now in the right position. And once you've done that, you will repeat this process with the next element to find the second smallest element. Once you've found the second smallest element, you will move it to the second position in your list. You will continue this process over and over again till all of the elements in your list are in their final sorted form.

This is selection sort, a simple idea, also simple to implement. I've explained how the sorting algorithm works. But you really need to visualize what happens with sorting in order to get an understanding of every sorting algorithms. So let's start with this unsorted list that you see here on screen. I'll now apply selection sort to sort this list of elements in the ascending order.

Let's start with the first element, element 15. I'll now compare element 15 with every other element in this list. My first comparison is going to be with element 32, which is located right after 15. Now, 15 is clearly smaller than 32. So in this comparison, we don't need to change the order of elements. 15 is still the smallest that we've seen so far. I'll now compare the element 15 with the next element in the list that is 26. 15 is still smaller than 26.

We don't need to change anything. I'll now move on to comparing the element 15 with the element 11, which is the next element in the list. When we compare 15 with 11, we clearly see that 11 is smaller than 15. We need to move 11 to the beginning of this list. So we'll take 15 and 11 and perform a swap operation. So 11 now comes to the very first position in this list. And 15 moves to where 11 originally was. The smallest element found so far is 11 and we continue comparing 11 with the remaining elements in the list. We first compare 11 with 36. No change here, 11 is still smaller. We'll then compare 11 with 19.

Again, no change at all. 11 is smaller than 19. We then compare 11 with 42. 11 is still smaller. We continue, we compare 11 with 44 and 14. 11 is still the smallest element. This completes the first iteration of selection sort. We've compared one element with all others and in this process found the smallest element in this list and moved the smallest element to the first position. The list is not sorted yet.

We've only got one element into position. We repeat this process with the next element in the unsorted list, that is element 32 and we compare it with every other element. The first comparison here, 32 with 26, will show us that 26 is smaller that means a swap operation is in order. We move 26 to the second position, swap it with 32. We then continue comparing 26 with every other element. Immediately you'll see that the first comparison with element 15 shows us that 15 is smaller than 26. A swap operation is performed once again. 15 has now moved to the second position. We'll now compare 15 with the remaining elements that we haven't looked at yet in this iteration, 36, 19, 42, 44.

All of these elements are larger than 15, there's nothing to be done there. But the element 14 is smaller than 15. A swap operation is in order, we move 14 to the second position in this list. And we effectively have now two elements in the right sorted order. We have two elements in our sorted list, it's slowly growing.

We start with the third element here, element 32. We compare it with every other element. And we immediately find that 26 is smaller than 32. Once we've made this comparison, we need to perform the swap. We'll move 26 to the third position in our list and compare 26 with the remaining elements. We'll see that 19 is smaller than 26. The comparisons will continue till we finally encounter element 15. 15 is smaller than 19.

One more swap is in order. Now we have 11, 14 and 15, which are the first three sorted elements in our list. We move on to the next element that is element 32 and compare it with every other element in this list. We find that 26 is smaller than 32 which means we need to perform a swap operation. And 26 comes to the beginning of the unsorted portion of our list.

We'll perform additional comparisons and find 19 is smaller than 26. Another swap is performed, 19 now comes to the beginning of this list. The first four elements are in sorted order. The first element of the unsorted portion of the list is 36. We compare this with every other element and we find that 32 is smaller than 36, which means 32 and 36 need to be swapped.

Once a swap operation is performed, we'll compare 32 with the remaining elements. We'll find that 26 is smaller than 32. We'll perform another swap here, and 26 now becomes part of our sorted list. By this time, you're getting a hang of what's happening, but we'll see this process through. 36 compared with every other element, 32 is smaller than 36. We perform a swap operation, get 36 to the beginning of the list. It's now part of the sorted

list.

We then compare 42 with every other element. 36 is smaller than 42. We perform a swap operation where we move 36 to the end of the sorted list and it becomes a part of the sorted list. Only 2 elements left now. We compare 44 with the one remaining element, 42 is smaller than 44. That means one last swap left to be performed. We performed the swap and we now have a fully sorted list.

All of our elements are in the right positions. And this is how selection sort works. It gets its name from the fact that we select one element at a time and compare that element to all other elements in the list. Once we've found the smallest element and moved it to the right position, we move on to the remaining elements in the list. Now it's pretty clear that selection sort does not require any additional space. But how do we calculate the running time of this algorithm?

Let's perform a simple calculation here. In the very first iteration, the number of comparisons that we have to make is equal to N- 1, where N is the number of elements in the list. In the second iteration, we make N- 2 comparisons, in the third iteration, N- 3 comparisons. Till finally in the Nth iteration, we make exactly 1 comparison. So we know how many comparisons are needed for each iteration. So the number of comparisons is equal to (N -1) + (N -2) + (N -3), all the way down to 1. The sum of the sequence is basically N multiplied by N -1, the whole thing divided by 2. That is a standard result, that is almost equal to N square.

So the time complexity of selection sort is N square. Another way to think about this is that for each element, the entire list needs to be checked to find the smallest element. So in the worst case, N elements need to be checked every selected element. This gives the complexity of selection sort as big O of N square. One of the biggest advantages of selection sort is the fact that it's a very simple algorithm and it doesn't require extra space.

It takes an O(1) extra space, it sorts in place. It also makes O of N square comparisons and performs O of N square swaps to get the elements in sorted order. And finally, it's important that you know that selection sort is not a stable sort. Entities which are equal might be rearranged, their order will not be preserved. And selection sort is not an adaptive sort. There is no way within the sorting algorithm to figure out if the list has already been sorted and break early. We have to iterate through till the very end.

Implementing Selection Sort

[Video description begins] *Topic title: Implementing Selection Sort. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll implement the simplest of all sorting algorithms, though it doesn't have a great time complexity, and that is selection sort.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window. At the bottom, there is a Console window. In the left pane, there are various packages.* [Video description ends]

Start off in main.java, we won't be using any additional files, we'll write all of our code here within this Main class. Selection Sort, as we've seen, is an order of N squared sort and it performs sorting in place, so we don't need any additional storage. We'll first set up a utility method that we'll use in performing selection sort and that is to swap two elements within a particular array. For the sake of simplicity, we'll perform all of our sort operations on integer arrays, though you can perform sort with any kind of data that is comparable. This swap utility takes in three input arguments and returns void. The first is the integer array which we are in the process of sorting. Then we have an iIndex and jIndex.

[Video description begins] *Line 5 reads as: public static void swap(int[] list, int iIndex, int jIndex) {.* [Video description ends]

This method will swap the element present at iIndex with the element at jIndex.

[Video description begins] *Line 7 reads as: int temp = list[iIndex];.* [Video description ends]

We'll first initialize a temp variable that will hold the element at iIndex, then we set jIndex to the element at jIndex.

[Video description begins] *Line 9 reads as: list[iIndex] = list[jIndex];.* [Video description ends]

And then set jIndex to be the temp variable, this performs the swap.

[Video description begins] *Line 10 reads as: list[jIndex] = temp;.* [Video description ends]

Before we write the code for selection sort, I'm going to set up an import for java.util.Arrays. This is a helper method that will allow us to display arrays in an easy to read format.

[Video description begins] *Line 3 reads as: import java.util.Arrays;.* [Video description ends]

We're now ready to write the code for selection sort. A method that performs selection sort is called selection sort, and the only input that it takes in is the integer array, which contains the list of elements that we want sorted.

[Video description begins] *Line 15 reads as: public static void selectionSort(int[] listToSort) {.* [Video description ends]

Selection sort requires the use of two for loops, one nested within another. Observe the outer for loop on line 17 which goes from i = 0 up to the length of the entire list, i is less than listToSort.length. And we increment i by 1 each time.

[Video description begins] *Line 17 reads as: for (inti=0; i < listToSort.length; i++) { .* [Video description ends]

Now, for every iteration of this outer loop, we'll print out the value of i to screen.

[Video description begins] *Line 19 reads as: System.out.println("\ni = " + i);.* [Video description ends]

The nested inner for loop with the variable j starts at the position after the current value of i. j = i + 1, j goes on till the end of this array listToSort.length; j++, it's incremented by 1.

[Video description begins] *Line 21 reads as: for (int j = i+1; j < listTo Sort.length; j++) { .* [Video description ends]

For each iteration of the i loop, that is the outer loop, we'll run through all of the elements in the list, starting at i + 1. And compare the element at index j with the element at index i. If listToSort[i] is greater than listToSort[j], we need to perform a swap.

[Video description begins] *Line 23 reads as: if (listToSort [i] > listToSort [j]) {.* [Video description ends]

We'll invoke the swap utility and swap the values of elements present at index i and j.

[Video description begins] *Line 25 reads as: swap(listToSort, i, j);.* [Video description ends]

Once we've performed the swap on line 25, we'll print out the elements that we've swapped, using a system out print statement. And also print out the current elements in the array using Arrays.toString.

[Video description begins] *Line 27 reads as: System.out.print ("Swapping: " + i + " and " + j + " ");. Line 29 reads as: System.out.println(Arrays.toString(listToSort));.* [Video description ends]

At the end of the first iteration of the outer for loop, the smallest element in the array will be at index 0. We'll then process the remaining elements in the array starting at index 1 in exactly the same way, and this will continue till the entire array has been sorted. Now that we've written the code for selection sort, we're ready to test this out.

Here is my unsortedList, it's a list of integers and you can see that the values are in any order. And the values are all numbers between 0 and 100.

[Video description begins] *Line 38 reads as: int unsortedList[ ] = new int [] {40, 50, 60, 20, 10, 70, 100, 30, 80, 90};.* [Video description ends]

I'll first print out the original array using Arrays.toString and then call selectionSort on this unsorted list.

Video description begins] *Line 40 reads as: System.out.println(Arrays.toString(unsortedList));. Line 42 reads as: selectionSort(unsortedList);.* [Video description ends]

Thanks to the print statements that we have within the selection sort algorithm, you'll be able to see every step of the sort process. Go ahead and run this code and let's take a look at the result here. At the very top, we have the original unsorted array. At i = 0, we perform 2 swap operations. Going through the original unsorted array, you can see that the first element is 40. At index 3, we have the element 20.

We need to swap these two elements and we do so, so we now have the array with 20 at index 0. 20 is moved to the index position 0 after we've swap the values at index 0 and 3. 20 is at this point, the smallest value that we found so far, and we continue looking through the remaining elements in the array.

At index 4, you can see that we have element 10, which is smaller than 20. So we then swap the elements at index 0 and 4, so 10 is now at the beginning of the array. At the end of the first iteration of the outer for loop, the smallest element in our array 10 is at index 0. That one element is in its sorted position, so we move on to index position i = 1. Originally, the value 50 was present at i = 1. We swap index 1 and index 3 and we get 40 at i = 1. We continue iterating through the remaining elements.

We swap 1 and 4 and we get 20 at i = 1. The first two elements are now sorted. The process that we've just seen continues for every iteration of the outer i loop. When i = 2, at the end of the three swap operations that you see, you can see that the value 30 is in index position 2. 30 is in the right sorted order at this point in time. For i = 3 at the end of 2 swap operations, we get the value 40 in the right sorted order.

This process continues, we continue swapping elements till all elements of this array are in the right sorted order. And when i = 8 after the final swap, you can see the final sorted values in the array. After the final swap here, when we swap 8 and 9, we have the final sorted array. 10 is the smallest element in this array and 100 is the largest. One disadvantage of selection sort is the fact that if your array is sorted early on, it's not possible to stop early.

Let's start with an unsorted list where the elements are almost in sorted order. If you look at our unsorted list now, you'll see that there are just a few elements out of place. The elements 40 and 30 are out of place, and the elements 70 and 60 are also out of place, but the rest of this list is in sorted order. I'll print out the original list and then invoke the selection sort function.

And if you look at the print statements within the console window, you'll see that we perform only as many swaps as are needed. Starting with the original unsorted list, we swap the elements at index positions 2 and 3 which are out of order so that 30 and 40 are now in the right sorted order. At i = 5, we perform another swap for

elements at 5 and 6, so that 60 and 70 are now in the right sorted order. But notice that there is no way to stop this process early. We'll never know whether the entire list has been sorted without iterating all the way through to 9 in the outer for loop.

Bubble Sort

[Video description begins] *Topic title: Bubble Sort. Your host for this session is Janani Ravi.* [Video description ends]

Another sorting algorithm that is fairly simple and intuitive to understand is the bubble sort. The bubble sort, like the selection sort, is used in simple cases. So how exactly does the bubble sort work? We'll first understand using words and then visually. For every iteration of the bubble sort, we'll compare every element in our unsorted list with a neighboring element, this is the adjacent neighbor. Now, in this comparison, we'll find which one is the smaller element. We'll swap the element so that the smaller one is earlier in the list.

At every step, we compare adjacent elements, find the smaller element, and perform a swap to get the smaller element earlier in the list. Now, this will bubble up the largest element that exists in our unsorted list to the very end, to the last index position. As we move along the list comparing adjacent elements, the largest one bubbles to the end. That's what gives the sort the name bubble sort.

Using words to understand sorting algorithms, well, that's not always very helpful. Let's see how this works in practice using the same unsorted list that we saw earlier. We'll first perform a comparison of the first two elements. Observe that these elements are adjacent to one another. Comparing 15 with 32, it's clear that 15 is the smaller element.

These two elements have the right relative positions. So we move on to the next two adjacent elements. Now, when we compare 32 with 26, these are not in the right sorted order. We'll perform a swap operation so that 26 comes to the second position and 32 to the third position. We'll also move our window forward so that the element 32 is now compared with the next adjacent element, that is 11.

These two elements are not in the right relative positions. Another swap is clearly needed. We'll perform the swap and move the window further along the list. 32 is now compared with 36. 32 is smaller, nothing to be done here. We'll simply move the window further along. 36 is now compared with 19. A swap needs to be performed and once the swap is performed, we can move the window further. The element 36 is now compared with 42.

They are in the right relative order, nothing to be done here, we'll move the window further along. Now we compare 42 with 44, they are in the right relative order. No swap operation required, we'll move the window further. Now we compare 44 with 14. 14 is clearly smaller than 44. So we'll perform a swap operation that'll bring 44 to the very end of this list.

Observe how our comparisons of adjacent elements bubbled up the largest element in the list to be at the last position. We'll now assume that 44 is no longer in the list. We've shrunk our unsorted list by one. 44 is in its right, final position. We'll go back to the beginning of the list and compare adjacent nodes 15 with 26. They are in the right relative order. So we'll move on to comparing 26 with 11. The element 11 is clearly smaller than 26, so a swap is in order. We'll perform the swap and compare 26 with 32.

No change here, we'll move the window further and compare 32 with 19. A swap is called for here, we'll perform the swap and carry on. The next comparison of adjacent nodes is 32 with 36, nothing to be done. 36 with 42, once again, nothing to be done here. Let's compare 42 with 14. 14 is clearly smaller, we'll perform the swap operation. 42, the next highest element in our unsorted list has now been moved to the very end. Observe for each iteration, the highest element in the unsorted portion of the list bubbles up to the end.

Let's continue our comparison. 15 with 11, a swap is clearly in order. Perform the swap, compare 15 with 26.

Nothing to be done here, we can slide the window further. 26 with 19 will tell us 19 is smaller, perform the swap and carry on. Compare the element 26 with 32. Once again, in the right relative order. Compare 32 with 36, again, in the right relative order. Compare 36 with 14. 14 is clearly smaller, perform the swap operation.

And after the swap, 36 has bubbled to the end to the list. Go back to the beginning and let's start comparing adjacent elements. 11 and 15, nothing to be done. Move on to comparing 15 and 19, nothing to be done here. We'll move on and compare 19 and 26. Once again, nothing to be done. Let's continue, we'll compare 26 to 32. Again, nothing to be done. But finally, when we compare 32 to 14, a swap is an order.

Perform the swap operation. Now, 32 has been bubbled up to the end of the list. As we perform bubble sort, observe that the number of swap operations that we need to perform keeps falling. As elements get into the right relative order, we'll have to perform fewer and fewer swaps. Let's do this again, starting with 11 and 15. Nothing to be done here, no swap operation. 15 and 19, nothing to be done here as well, we can move on. 19 and 26, nothing to be done here. We have to move on and compare 26 and 14.

This is the only operation that requires a swap. Perform the swap, 26 is at the end. We go back once again to the beginning. 11 and 15, nothing to be done. 15 and 19, nothing to be done. The only swap comes when we compare 19 and 14. 14 is clearly in the wrong position. Perform the swap and we can go back again and start comparing. Once again, we start at 11. We compare 11 with 15, again, in the right relative order.

We compare 15 with 14, there is a swap required. Performing the swap will give us our finally sorted list. But we won't stop yet, we'll compare 11 and 14, they are in the right relative order. At this point, we have a fully sorted list, nothing more needs to be done. Now, bubble sort has a huge advantage over selection sort. Though you'll find that their time complexities are mostly the same. The fact is that in bubble sort, we can break out of the loop early if we find that the list is already sorted. How do we know if a list has already been sorted?

In every iteration, when we compare adjacent nodes, if we find that we haven't performed any swaps, that is an indication that the list is completely sorted. All of the elements are in their final positions. Even though we might have just bubbled a few of the largest elements to the very end of the list. If we perform no swap operations, that means the remaining elements are also completely sorted. And in that case, we'll break out of the sort early.

Now that we've visualized the sorting algorithm, here is a quick summary of what exactly we did at each step. In every iteration, the largest element is bubbled up to the correct position by comparing adjacent elements in our list. Bubble sort has the ability to stop early, if the list has already been sorted. But in the absolute worst case is when we start off with a list in the descending order. This gives us the worst case time complexity for this algorithm. For each of N elements, we make N comparisons and N possible swaps.

We saw that in bubble sort, we didn't need extra space to hold the elements that we were sorting. It's an in place sorting algorithm, it takes constant extra space. That is, O(1) extra space. The entire sorting algorithm makes O of N square comparisons and can perform O of N square swaps. This is, of course, in the worst case. Thus, the time complexity of bubble sort is the same as that of selection sort, order of N square.

Remember, we always consider the worst case while computing the time complexity. Even though bubble sort has the same time complexity and spatial complexity as selection sort, there are two advantages. Bubble sort is a stable sort, entities that are equal are not rearranged in the final sorted list. Also bubble sort is far more efficient than selection sort while working with almost sorted lists. It's an adaptive sort. When the list is sorted, we break out of the sort early.

Implementing Bubble Sort

[Video description begins] *Topic title: Implementing Bubble Sort. Your host for this session is Janani Ravi.* [Video description ends]

The next sorting algorithm that we'll study here in this demo is Bubble Sort.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

Now, Bubble Sort is also an O of N square sorting technique. So the worst case time complexity is the same as that of Selection Sort that we saw earlier. One advantage that Bubble Sort has over Selection Sort is the fact that you can stop the sort early if you find that the list has already been sorted.

Just like the Selection Sort, Bubble Sort also performs its sorting in place. It doesn't use any additional storage. Let's take a look at the Bubble Sort method that we have here. It takes as an input argument the list that we want to sort. For the sake of simplicity, we'll just work with a simple integer array. This of course works for any list which contains comparable elements.

[Video description begins] *Line 15 reads as: public static void bubbleSort (int[] listToSort) {.* [Video description ends]

Once again, Bubble Sort uses two for loops, one for loop nested within another. So we have two fors here. There is an outer for loop which uses the variable i. And then we have an inner nested for loop which uses the variable j. The outer for loop iterates over the elements of a list in the reverse order. Starting from the last element at list length - 1 going down to the first element. If you take a look at the conditions of the outer for loop, you'll see that this for loop runs so long as i > 0, and for every iteration, we decrement i by 1, i--.

[Video description begins] *Line 17 reads as: for (int i = listToSort.length - 1; i > 0; i--) {.* [Video description ends]

For each iteration of this outer for loop, we'll move the largest element that we have in the unsorted portion of the list to the very last position in the unsorted portion. And this way, we'll then decrement i by 1, indicating that the unsorted portion of the list is now smaller by 1 element. In order to see what's going on in the sorting process, for every iteration of i, we'll print out to screen the value of i.

[Video description begins] *Line 19 reads as: System.out.println("\ni = " + i);.* [Video description ends]

That is on line 19. Let's take a look at the inner for loop. Observe that the inner for loop starts at the first element at index 0, j = 0 is the initial value. And this for loop runs so long as j < i. So this for loop runs over all of the elements in the unsorted portion of the array. And we increment j by 1 for each iteration of this inner for loop.

[Video description begins] *Line 21 reads as: for (int j = 0; j < 1; j++) { .* [Video description ends]

For each iteration of the for loop with j, we compare the current element that is at index j with the element just after it, that is, the element at index j + 1.

[Video description begins] *Line 23 reads as: if (listToSort [j] > listToSort [j + 1]) {.* [Video description ends]

So if the element at j is greater than the element at j + 1, we swap these two. We swap the element at j with the next element.

[Video description begins] *Line 25 reads as: swap(listToSort, j, j + 1);.* [Video description ends]

We'll print out a swap statement to screen. And we'll also print out the current status of the entire unsorted array so that we can see how the unsorted array slowly gets into sorted form.

[Video description begins] *Line 27 reads as: System.out.print("Swapping: " + j + " and " + (j + 1) + " ");.* [Video description ends]

The only logic within this if block is our swap. We swap j with j + 1 so that the largest valued element bubbles through to the very end of the array. One thing to note here is that in this Bubble Sort algorithm, we haven't yet implemented early stopping. So if a list has already been sorted, we won't perform additional iterations, that logic is not in here yet. Let's go ahead and first test out this simple Bubble Sort. We start with an unsortedList.

[Video description begins] *Line 38 reads as: int unsortedList[] = new int[] {40, 50, 60, 20, 10, 70, 100, 30, 80, 90};.* [Video description ends]

I'm going to print out the contents of the unsortedList, and then invoke bubbleSort to sort this unsortedList, this is on line 42.

[Video description begins] *Line 40 reads as: System.out.println(Arrays.toString(unsortedlist)};.* [Video description ends]

Time to run our Bubble Sort algorithm.

[Video description begins] *Line 42 reads as: bubbleSort(unsortedlist);.* [Video description ends]

Once you run this code, at the very bottom here, you can see the fully sorted list. But what's really interesting is the intermediate steps, the swapping operations that led to this fully sorted list, and that's what we'll look at first. Observe that we start off with the unsortedList which has 40 as the first element and 90 as the last element. When i = 9, that is the entire length of our list, our list has a total of 10 elements.

This means that the entire list is unsorted, there is no sorted portion of this list. We then compare each adjacent pair of elements in the list. Let's look at the unsortedList at the very top of the Console window. We compare 40 and 50, 40 is less than 50, they are in the sorted order. We compare 50 and 60, once again, in the sorted order. We then compare 60 and 20. Clearly, they're not in the right sorted order.

We wanted ascending sort order, so we perform a swap, that's what you see. Swapping elements at index 2 and 3 is the first swapping operation when i = 9. In exactly the same way, we perform other swapping operations when i = 9. We swap elements at index 3 and 4, 6 and 7, 7 and 8, and 8 and 9. The state of our unsorted array at the end of the iteration when i = 9 is what you see here at the very top of the Console window. 40 is the first element, 100 is the last element. The largest element 100 here has been moved to the very end of the list. So you can imagine that the list is now unsorted only from index 0 to index 8.

Index 9 has the right sorted element within it, which is why in the next iteration i = 8, we won't consider the element at index 9 at all. That is already in the right sorted position. When i = 8, we start comparing adjacent elements once again. When i = 8, at the top of the console window, you can see that the elements 50 and 20 are not in the right sorted order. So we swap the elements at index 1 and 2 so they are in the sorted order. And we continue this process till we have element 90 at the very end of this array.

When i = 8, after the very last swap of elements at index 5 and 6, you can see that the value 90 is in the second to last position, just before 100. So the last two elements are in the right sorted order. Now, this process continues for i = 7, and then i = 6, and then i = 5. Note that for each iteration of the outer for loop, i, the number of swap operations that you need to perform to get the list in sorted order keeps falling.

For i = 5 here, we perform just a single swap operation for elements 2 and 3, and we get the list in the final sorted order. For i = 4, i = 3, and all other values of i, no swap operations are performed at all. You can see that we are now working with a fully sorted list. We've sorted the list early, we don't need to go through all iterations of i. This becomes much more obvious when we start off with an almost sorted list. I've changed the unsortedList here.

And this is a list that is almost sorted, there are only two elements that are out of place. In this unsortedList, the

elements 40 and 30 are not in the right order. And the elements 70 and 60 are also not in the right order. But the rest of the elements have their final positions. Invoke bubbleSort, and you can see that the only time that we perform swaps to get elements in the correct position is in the first iteration when i = 9.

Once we've swapped the elements at index values 5 and 6, you can see that the list is in the completely sorted order, starting with 10 ending at 100. But we still iterate through i = 8, 7, 6, 5, 4, all the way through to 1. It's pretty clear here that all of the other iterations and comparisons that we end up performing are wasteful.

Implementing Bubble Sort With Early Stopping

[Video description begins] *Topic title: Implementing Bubble Sort With Early Stopping. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll make a fairly significant improvement to our bubble sort algorithm, allowing us to stop the sorting process early if we find that our list is already completely sorted.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

This can improve the running time of your bubble sort in many cases. If you look at this bubbleSort method that we have here, you can see that the basic structure of the sort remains exactly the same. We have two for loops, one nested within another. We have the outer for loop, going from list length - 1, so long as i > 0, and we have an inner for loop that goes from j = 0, so long as j < i. The new detail that we've added here is that we've introduced an additional variable, which we initialize to false for every iteration of the outer i loop.

On line 19, observe that we have a boolean swapped that we set to false. Now, this will keep track of whether any swapping operation was performed to rearrange the elements in our unsorted list for every i iteration. Initially, we'll assume that there were no swaps performed.

[Video description begins] *Line 19 reads as: boolean swapped = false;.* [Video description ends]

But within the inner for loop, when we compare the elements j and j + 1. This is on line 25. If we find that these adjacent elements are not in the right sorted order, we perform a swap. This is on line 27.

[Video description begins] *Line 27 reads as: swap(listToSort, j, j + 1);.* [Video description ends]

Each time you perform a swapping operation, set the swapped variable to be = true. This is the additional change that we have introduced. Every time we perform a swap, we know that there are some elements still not in their final positions. We set swapped = true. The rest of the code here remains exactly the same including all of the print statements that we've used to track the intermediate values during the swapping algorithm. There is another important change. As soon as we break out of the j loop and get into the outer i loop, we check the swapped variable. This is the code on lines 37, 38 and 39.

[Video description begins] *Line 37 reads as: if (!swapped) {.* [Video description ends]

If no swap was performed and this iteration, we know that we're dealing with a completely sorted list. If we've performed even one swap operation, we'll continue sorting and performing bubble sort. But if no swap was performed for this iteration of i, we know that we can break out safely. We have a completely sorted list. I'm going to test this bubbleSort with early stopping on our almost sorted list. This is the list where there are just two elements out of place.

Elements 40 and 70 are not in their correct final positions. I'm going to print out the unsortedList first and then invoke bubbleSort. Let's run this code and you'll see something interesting. Observe when i = 9, we perform two

swapping operations to sort the list. And at the end of these two swapping operations, we get a completely sorted list.

For i = 9, we performed at least one swapping operation, which is why we get into the next iteration of the i loop, i = 8. But when i is equal to 8, we don't perform a single swap operation, the swap variable remains false and we break out of the i loop as well. Observe how many iterations and how many comparison actions that we've saved on using this early stopping.

Insertion Sort

[Video description begins] *Topic title: Insertion Sort. Your host for this session is Janani Ravi.* [Video description ends]

We are now ready to move on to yet another sorting algorithm. In this video, we'll see a visualization of how insertion sort works. We'll work with the same unsorted list that we've worked with so far. Here is the unsorted list. Insertion sort starts off by considering the first element in the list. This is a single element list, we can consider this simple list to be already sorted. The basic idea behind insertion sort is that the other elements in this unsorted list, we'll try and insert into the sorted portion of the list. At this point in time, to start off with, the sorted portion of the list comprises of just the first element, 15.

The next step is to check the next element after the sorted portion of the list. This happens to be element 32. So where exactly does 32 fit in the sorted portion of the list? Well, at the very end. At this point in time, we now have two elements in our sorted portion of the list. We've effectively inserted 32 into the right sorted position.

Let's look at the next element that we want to insert into the sorted portion. This is the element 26. Where does the element 26 fit in in the sorted portion of this list? Well, we compare 26 with the last element in the sorted portion, that is 32. Well, that's not the right position, so we perform a swap operation. Next we compare the element 26 with the element 15. Are they in the right relative positions?

Yes they are. At this point we have three elements in the correctly sorted order, 15, 26, and 32. The next step is to now insert the element 11 into the right position within this sorted list, which currently contains three elements. We'll compare the element 11 with the last element in the sorted portion of the list, that is 32. 11 is smaller than 32, we'll perform a swap operation. Next we'll compare 11 with 26. Is 11 in the right position within the sorted list? No, we need to perform a swap operation. And 11 now is compared with 15. Is it in the right position?

No, it isn't, we'll perform a swap operation and that will give us four elements in the sorted order. With four elements in our sorted portion of this list, the next step is to compare the element 36 and insert it into the right position within this sorted list. 36 is already in the right position. When you compare it with the last element of this list, we can simply leave everything as is. Time to get the next element, that is 19, into the right sorted position within this sorted subportion. We compare 19 with 36, we'll perform a swap operation. We'll compare 19 with 32, this also requires a swap operation. We'll compare 19 with 26.

One more swap is in order here. And now we'll compare 19 with 15. 19 is in the right position within the sorted list. Our sorted list has grown by one. We can move on to the next element here, 42. We'll compare 42 with 36 and find that 42 is already larger than the largest element in this sorted portion of our list. We leave everything as is. We'll now move on to the element 44. Comparing 44 with 42, the last element in our sorted list shows us that 44 is in the right position.

We leave everything as is and our sorted list has now grown to encompass all elements except one. We now need to insert element 14 into the right position within this sorted subportion. We compare 14 with 44, perform a swap operation. We'll compare 14 with 42 and perform another swap operation. 14 is compared with 36, one more swap here is in order. 14 is then compared with 32. Clearly another swap is needed. 14 with 26 shows us

14 is smaller. A swap is needed here to get 14 into the right position.

We then compare 14 with 19, another swap is needed. We can move on to comparing 14 with 15, this mandates another swap. And finally, when we compare 14 with 11, we can tell that 14 is in the right position within this sorted list. All of the elements in this list are sorted now, we have a fully sorted list. Insertion sort tries to grow the number of elements in a sorted subportion of the entire list. By inserting into a sorted sub-list at every step, the sub-list soon grows to be the entire list. At each iteration, the sorted sub-list grows by one till we have a fully sorted list. We started the process of insertion sort by making an implicit assumption.

We consider any list that has just one element to be a list that is already sorted. We start with this assumption and then insert additional elements into the sorted list. Now, the worst case for insertion sort occurs when you have an already sorted list and that list is sorted in the descending order and you're trying to sort in the ascending order. N elements are checked and swapped for every selected element to get to the right sorted order. Checking N elements for each of N elements gives us a time complexity for insertion sort as order of N square.

Thus, the time complexity for insertion sort is exactly same as the time complexity for bubble sort and selection sort. It's not an improvement in terms of time complexity. Insertion sort is a stable sort, though. As entities bubble to the correct position, the original order of the entities is maintained for equal elements. Just like all of the other sorting algorithms we worked with so far, insertion sort does not take up additional space. It takes O(1) extra space, it's an in place sorting algorithm. If you want to sort an entire list of N elements, insertion sort requires order of N square comparisons and order of N square swaps.

This is what gives us the time complexity of N square. Another advantage of insertion sort is the fact that it is an adaptive algorithm. It's similar to bubble sort in that when you work with lists that are almost sorted, the sorting process completes very, very quickly. Fewer comparisons and swaps are performed. Insertion sort also has the advantage that it has a very low overhead and traditionally, this is the sorting algorithm of choice when you're okay with O of N square time complexity.

So if you have to choose between the selection sort, bubble sort, and insertion sort algorithms, you'll typically pick insertion sort because it tends to be more efficient in the long run. The advantage of insertion sort over selection sort is pretty obvious, because insertion sort is a stable sort and it allows for early stopping. But let's see how bubble sort compares with insertion sort.

With bubble sort, you require an additional pass over all elements to ensure that the list is fully sorted. With insertion sort, if the elements have already been sorted, you require just one comparison with the last element in the sorted portion of the list to know that sort is complete. At every iteration, both bubble sort and insertion sort have to perform O(N) comparisons. The difference is that with insertion sort, we can stop comparing elements early once the right position for an element has been found in the sorted sub-list.

Modern hardware that we work on today tries to optimize read write operations, that is, input output operations. Bubble sort performs poorly here because of the number of writes and swaps that it performs. Even though the time complexity of insertion sort and bubble sort are the same, O of N square, the actual number of operations performed tends to be lower with insertion sort.

Implementing Insertion Sort

[Video description begins] *Topic title: Implementing Insertion Sort. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll see code for how Insertion Sort works.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

Insertion Sort has a worst case running time of order of N square. But in the best case, when you're working with an already sorted list. The running time of this algorithm can be O(N). Here is our insertionSort method which takes as an input argument, the integer array that is to be sorted. Once again, insertionSort contains two for loops. We have an outer for loop with the integer i and an inner nested for loop.

The outer for loop starts with index 0, i = 0 and goes up till the length of the array, that is one less than the length of the array. i < list.length - 1. We increment i by 1 each time. While performing insertionSort, we try and grow the sorted array starting at index 0. Initially the entire array is unsorted. At the end of one iteration of the outer loop i, the element at index 0 will contain the right element in the right sorted order. We then focus our attention on the unsorted portions of the array starting at index 1.

The inner j loop focuses on the first element in the unsorted portion in the array and gets it to the right position in the sorted portion of the array. Remember, the array till element i is considered sorted. So this inner for loop j starts at i + 1, and it goes backwards, we decrement j down to 0. The objective of this inner for loop is to find the right position for the element which is at index j + 1. So for each iteration of a j, we'll check to see whether the element at index j is smaller than the element at j- 1. If that is the case, the element at index j is not in the correct sorted position. Which means we have to perform a swap operation. So we swap j with j- 1.

And we'll write out print statements to screen so that we see what's going on in the intermediate steps of our sorting. We'll also print out the list at this point in time. If we enter the code in the else block at that point in time. We know that the element j is in the right position in the sorted portion of the list starting at index 0. And in that case, we simply break out of the inner for loop for variable j and we go back to the outer for loop once again. Let's go ahead and initialize an unsorted list which contains ten elements, all integers, and use insertionSort to get this list in sorted order.

[Video description begins] *Line 44 reads as: insertionSort(unsortedList);.* [Video description ends]

I'm now going to run this code and as has been the case previously, the intermediate steps will tell us how exactly the sort has been performed. You can see here that at iteration i = 8, we have the list in a fully sorted order. Let's take a look at the intermediate steps. They are of course more interesting. We have the unsorted list printed to screen at the very top. When i = 0, we only consider the element at the zeroth position in the unsorted list. That element 40, well, it's fully sorted.

A list of one element is always sorted, let's consider i = 1. So we consider the elements 40 and 50. They are in sorted order, we do nothing. When i = 2, we know that elements 40, 50 and 60 are in sorted order. The inner loop j which starts at i +1 will examine the element at index 3. The element in the unsorted list at index 3 is 20. And 20 is clearly not in the right position. As far as the first four elements in this array are concerned. Using swap operations, we'll bring the element 20 into the right sorted order. The first swap operation we perform is when we swap elements at index 3 and 2, 20 is now at index position 2.

Then we swap 2 and 1, 20 is now at index position 1. And finally, we swap 1 and 0, and 20 comes to the zeroth position. The first four elements, 20, 40, 50 and 60 are in the right sorted order at this point in time. In the next iteration of the outer for loop, when i = 3, we know that the first four elements starting at index 0 up till index 3 are completely sorted. j is now equal to 4 and we look at the element at index 4 and that element is 10.

You can see this in the last array printed out for the iteration i = 2. Now, in the next iteration when i = 3, we have to find the right position of this element 10 in our already sorted array. We swap 10 with 60, so 10 is now at index position 3, then we swap 10 with 50. It's now at index position 2, we swap 10 with 40. It's now at index position 1. And finally we swap 10 with 20, and 10 comes to the zeroth index position. And now at this point our sorted list encompasses the first five elements, 10, 20, 40, 50, and 60.

Once you've understood this, the remaining steps do exactly the same operation over and over again. We try to increase the size of the sorted list by 1 for every iteration of the outer i loop till we get a completely sorted list.

Let's test our insertionSort once again with a slightly different array. We'll see how we can work with an array that is almost sorted. And you'll be able to see here that the number of swap operations that we perform is very small. At i = 2, we perform a single swap operation to get the element 30 to the right position in the sorted list.

After this there are no more swap operations to be performed till we get to i = 5. Here, we perform a single swap operation to get the element 60 to the right position in the sorted list. And with that we have a fully sorted list. And we don't need to perform any additional swap operations. But we can't stop early though with Insertion Sort. Because we won't know whether a list is fully sorted till we have checked all of the elements at least once. This is the outer for loop with variable i.

Shell Sort

[Video description begins] *Topic title: Shell Sort. Your host for this session is Janani Ravi.* [Video description ends]

The next sorting algorithm that we'll study and understand is the shell sort. Because we've already studied the insertion sort algorithm, you'll find it very easy to work with the shell sort. Because the shell sort is simply a modified version of the insertion sort. The actual sorting will be performed using insertion sort. We'll make a few tweaks to get the shell sort algorithm. The main difference between shell sort and insertion sort is that rather than comparing adjacent elements. Elements a certain interval apart are compared. We start off with a fairly largish interval and then slowly decrement this interval.

Or you can start off with a small interval and increment this interval in shell sort. There are different variations of shell sort based on how you choose the interval which you use to compare elements. If you start with the interval N/2, what you'll do in the first iteration is to ensure all elements which are N/2 apart are completely sorted. At this point, all elements N/2 apart are sorted. You will then reduce the interval that you're working with to N/4. You half the interval. You'll then ensure that all elements N/4 apart are sorted. At each step, you'll half the interval from N/4, you'll go to N/8, N/16 and so on till the last interval is equal to 1.

In this final iteration, you'll ensure that all elements that are 1 apart, that is, elements which are adjacent to one another, are sorted. The intervals that I've picked here are shells intervals, the original intervals used with shell sort. There are other variations that you can use as well, such as notice intervals and so on. We'll see an implementation of that when we take a look at the code. But first, let's visualize how shell sort works. Here we are with a list that is unsorted. This is the list that we've been working with so far.

We'll now perform shell sort on this list. In the first pass for our shell sort, we need to pick an interval across which we'll compare elements. Now here N=10. That is the total number of elements that we're going to sort is equal to 10. The interval that we'll pick first is N/2 = 5. This means that in the first pass of our shell sort algorithm, the elements that we'll compare will be element 0 will be compared with element 5. 1 will be compared with 6, 2 will be compared with 7, 3 with 8, and so on.

We can now kickstart the comparison process. I'm going to compare the element at index 0 with the element which is five away from index 0, that is index 5. When we compare 15 and 19, they are in the right sorted order, no change here.

Next we'll compare the element at index 1 with the element at index 6, 42 with 32. They are clearly not in the right sorted order. We'll perform a swap operation so that 32 is now earlier in the list as compared with 42. We'll move on to the next pair of elements in the sequence. We'll compare 26 with 47. These elements are in the right sorted order, so let's move on in the sequence. We compare the element 11 with the element 28. They are in the right sorted order. Moving on, the next element in the sequence is 36, which we compare with 24.

Clearly not in sorted order, so perform a swap operation. At this point in time we've reached the end of pass one in this shell sort algorithm. All elements that are five apart in this list are now completely in the sorted order.

Where do you perform insertion sort? That's a question that you might be asking yourself right now. Don't worry, it'll become a little clearer once you see the rest of this algorithm. When we get elements in sorted order, that is essentially insertion sort. Now the next step here is to reduce the interval for shell sort.

We'll reduce the interval to N/4, which is equal to 2 in our case. The compare sequence will be the element at index 0 will be compared with the element 2, 2 will be compared with 4, 4 with 6, 6 with 8 and so on. Also the element at index 1 will be compared with the element at index 3, 3 will be compared with 5, 5 with 7, and so on. With the increment equal to 2, we make the second pass in the shell sort algorithm. Our comparisons will be for alternate elements, 15 with 26, then 24 then 42 then 28, 32 with 11, 19, 47 and 36.

Let's get started. We'll first compare 15 and 26. They're in the right sorted order, no swap here. We'll then compare elements 32 and 11. These are not in the right order, we'll perform a swap operation. 11 is now before 32. We'll move one step further. And we'll compare 26 with 24. They're not in the right sorted order, we'll perform a swap operation. So 24 is before 26. But we're not done yet. We now need to get 24 in the right position for elements that are two apart.

This means that we need to compare the element 24 with the element 15 and see whether they are in the right sorted order. They are, 24 is in the right sorted order for elements that are one apart. Actually what we've done here is use insertion sort to get 24 in the right position in the list of elements that are one apart. We can now continue with this algorithm. We'll compare the element 32 with 19. They're not in the right relative order, we'll perform the swap operation. We'll now compare 19 with the element 11.

They're in the right sorted order, there's nothing to be done here. 19 has been inserted in the right position in this list of elements that are one apart. Moving on, now compare the element 26 with 42, they're in the right order. We then compare element 32 with 47. Once again in the right order. We continue and compare 42 with 28. There is a swap required here, we perform the swap operation and percolate this backwards. We compare 28 with 26. 28 is in the right relative position in this list of elements one apart.

This means we can move on with our comparisons of elements one apart. 47 with 36, perform a swap operation here so 36 moves backwards through this list. We'll compare 36 with 32. 36 is in the right position relative to 32. And now you can see after pass two, all elements that are one apart in this list are in the right sorted order. The elements in purple are 15, 24, 26, 28, 42, sorted order. The elements in green are 11, 19, 32, 36 and 47, once again in the sorted order. It's now time to reduce the interval further.

Remember, number of elements is equal to ten. Our last interval was N/4. So we now reduce it to N/8 that is equal to 1. This means that in pass three we'll be comparing adjacent elements and getting them in the right sorted order. The first pair of adjacent elements that we compare are 15 and 11. There is clearly a swap needed here so we swap them. 11 is now in the right position as is 15. We now move on to comparing 15 and 24. They are in the right relative order. We move on to comparing 24 and 19.

Clearly not in the right order, perform a swap operation to get 19 before 24. We'll need to compare 19 with 15 to see whether they are in the right order as well. They are, we can move on with our comparison, we'll compare 24 with 26. No change required here. We'll compare 26 with 32. Again, no change. We continue, compare 32 with 28. 28 is smaller than 32, perform the swap operation. Now we need to compare 28 with 26.

The adjacent element on the left side, they're in the right sorted order, so we can carry on with our comparison. 32 with 36, in the right sorted order. 36 with 42, once again in the right sorted order. And finally, 42 with 47, again in the right sorted order. At this point in time, you can see that we have a fully sorted list. Let's quickly summarize what we understood about shell sort. Shell sort uses insertion sort.

The entire list is divided based on increments and those sub-lists are sorted. If you were to ask me to calculate the time complexity of shell sort, I would say it's quite hard. That's because it depends on the increment values that you've chosen. It has been found that the best increment sequence is N/2, N/4, down to 1. This is shell's sequence and this is what we picked when we demonstrated shell sort. Empirically shell sort has been shown to

perform far better than insertion sort. As the final iteration when increment = 1 has to work with a nearly sorted list.

Very few swap operations have to be performed. We can estimate the complexity of shell sort as somewhere between O(N) that is linear time complexity and O of N square. Just to reiterate here, based on the increment values that you've chosen, you will get different time complexities for your shell sort algorithms. For example, you might have increment values based on k, where k is the number of passes.

No matter what interval sequence you pick, the worst case time complexity is always less than or equal to order of N square. This is better than what any of the earlier sorting algorithms gave us. This algorithm is adaptive since it's based on insertion sort, which is also adaptive. And finally, this algorithm works in place. It doesn't take up any additional place for the data that you're sorting, spatial complexity O(1).

Implementing Shell Sort

[Video description begins] *Topic title: Implementing Shell Sort. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll see code for how shellSort works.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

Now, shellSort is a more generalized version of insertion sort. Since we just studied insertion sort, it's a perfect time to study shellSort. shellSort is another sorting algorithm with a complexity that is worst case complexity, which is order of N square.

But the base case and average case complexity for shellSort tends to be order of N log N, which is why you might prefer to use shellSort over bubble sort and selectionSort, both of which have average case of N square as well. The basic concept behind shellSort is that it first sorts elements, which are far away from one another, separated by a certain interval. And it then reduces the interval bit by bit till you get a completely sorted list. There are different techniques that you can use to pick the original interval, and then reduce the interval gradually.

Observe that for the shellSort algorithm, I have two methods, a method call shellSort which will perform shellSort. And then another method call insertionSort. That's because shellSort uses insertionSort to sort elements that are separated by a certain interval. Let's first look at the outer shellSort code. And then we look at insertionSort. This is a slightly modified version of insertionSort. shellSort takes as an input argument the list that is the integer array that you want sorted.

The first thing you need to do is specify the increment or the interval between elements that you want to compare and sort. The initial value of increment that we've chosen here is the length of the list divided by 2. This increment and it's successive values are known as shell's increments, and that's where shellSort gets its name.

[Video description begins] *Line 17 reads as: int increment = listToSort.length / 2;.* [Video description ends]

Having initialized the increment to half the length of the original list, we run a while loop. So long as this increment is greater than or equal to 1, we'll invoke insertion sort and sort all elements that are separated by this increment or interval.

This insertionSort will ensure that all elements separated by increment are now completely sorted.

[Video description begins] *Line 21 reads as: insertionSort(listToSort, increment);.* [Video description ends]

You then reduce this increment using increment is equal to increment divided by 2, we halve the increment value or the interval.

[Video description begins] *Line 23 reads as: increment = increment / 2;.* [Video description ends]

This is how shell's increment go. If N is the length of the list, the first interval or increment is N/2. Then it becomes N/4, N/8, N/16, and so on. The insertionSort code that we'll use with shellSort is the same insertionSort that we've seen before where we steadily grow the length of the sorted list by inserting each additional element in the right position within the sorted list. The one little tweak here is that insertionSort takes in an additional input argument, that is the increment or the interval across which elements should be compared. The code for insertionSort remains the same.

So there are two for loops, there is an outer for loop with variable i, and an inner for loop with variable j. The tweak is in how we increment the counter for the for loop. We start with int i = 0, and we continue the outer i loop so long as i + increment is less than the length of the list. And after each iteration of this outer while loop, we increment the value of i by 1, i = i + 1. For every iteration of i, we'll print out the value of i and the current value of increment.

Next, let's take a look at the inner j for loop. The first element that we compare to insert into the sorted list is at i + increment. So we initialize j to be i + increment. And we iterate down the sorted portion of the list so long as j- increment is in valid range, j- increment should be greater than equal to 0. And for each iteration, we'll reduce the value of j by increment, that is the interval.

The comparison here within the inner for loop is to compare the element at index j with the element at index j- increment. If the element at index j is greater, we perform a swap. We swap the element at index j with j- increment. And we'll write out a few print statements. As usual, in the else block, we simply break out of the inner for loop and head over to the next iteration of the outer for loop.

The code for the insertion sort here is exactly the same, except that at every step, we consider only those elements that are a certain interval apart to be part of the list. We are now ready to test out our shellSort program. I'm going to set up an unsortedList. This is the same unsortedList that we worked with before. Invoke the shellSort function on this unsortedList, and let's take a look at the result.

[Video description begins] *Line 56 reads as: shellSort(unsortedList);.* [Video description ends]

At the very bottom here, you'll find the completely sortedList. The sort is in ascending order. We have element 10 as the first element in the list, and 100 is the last element.

As usual, the intermediate steps will be more interesting for us to examine and analyze. At the very top of the console window, we have the original list that is completely unsorted. Initially, i = 0 and increment is 5. This means that, we only compare the elements at index 0 and index 5, that is the elements 40 and 70. They're already in the right sorted order, we move on, no swap operations are performed.

Let's consider i = 1 and increment 5. Then we consider the elements 50 and 100. We are also in the right sorted order, nothing to be done. When i =2, and increment 5, we consider the elements 60 and 80. They're also in sorted order, nothing to be done. i = 3 is when things get interesting. The element at index 3 is 20, and 5 elements away, we have the element 10 at index 8. 20 and 10 are clearly not in the sorted order, 10 should come before 20. So we swap the elements at index 8 and 3 and we get the array that you see here at the bottom of the screen.

Elements 10 and 20 have been swapped, and relative to one another, they are in the right sorted order. We continue this operation till all elements that are separated by an interval of 5 are completely sorted. We then decrement the interval observe for i = 0, the increment is now equal to 2. We'll now compare elements that are separated by an interval of 2. When i = 0 and increment = 2, we are working with the array that you see here at the top of the console window.

We'll compare the elements 40 and 60, they are in the sorted order so we'll do nothing. When i = 1 and increment 2, we'll compare the elements 50 and 10, they're clearly not sorted, so we perform a swap, we'll swap 3 and 1. So 10 and 50 are now in the right sorted order, relative to one another. This process continues till all elements that are separated by an interval of 2 are completely sorted. The value of increment is now further reduced to be equal to 1. When increment = 1, we'll compare adjacent elements to get our final sortedList.

At the end of this comparison, when increment = 1, we'll have a fully sortedList, as you can see down here at the very bottom of the console window. What we just worked with was shellSort using shell's increments, and that works very well. But there have been other interval values proposed as well, which work with shellSort. Here is another example that we'll look back where we'll use Knuth's Increments.

At every step k, Knuth's Interval is given by 3 multiplied by k- 1, the whole thing divided by 2. So we start off with k = 1, where the interval is 1, then go to 2, 4, 5, 7, 8, and so on. The rest of the structure of the shellSort code and insertionSort remains exactly the same. Let's go ahead and run this and take a look at the result. Initially observe that the increment values are equal to 1, which means we'll compare every adjacent element to see whether they are in sorted order.

Once we know that adjacent elements are in sorted order, we'll increase the interval or the increment. Increment will now be equal to 2. We'll then compare elements that are at an interval 2 apart. We gradually increase the increment interval. From 2, we move on to 4, then to 5, and then to 7 and 8. At this point in time, we know our list is completely sorted.

Merge Sort

[Video description begins] *Topic title: Merge Sort. Your host for this session is Janani Ravi.* [Video description ends]

We are now ready to discuss our first sorting algorithm that follows the divide and conquer approach. Divide and conquer algorithms are far more performant as compared with all of the sorting algorithms that we've seen so far. We're going to discuss Merge Sort. So far in all of the sorting algorithms that we've understood and written code for, we try to sort the entire list at once.

We don't try to break the list down and then sort smaller lists. That's what we'll do in Merge Sort which follows the divide and conquer approach to create smaller problems. And these smaller problems tend to be more tractable and easier to solve, divide and conquer makes things easier. In the divide and conquer approach, when you see a large problem, you'll basically put up your hands and say, well, this is too hard to solve. How can I make it a smaller problem? Thus, in the case of sorting a list, that list is broken down into smaller and smaller parts recursively. Smaller lists are, of course, easier to sort intuitively.

This process continues till at some point you get a list with just a single element, and we've discussed this before. A single element list is a list that can be considered to be sorted. Consider this list as a sorted list if you had N elements in the original unsorted list, you now have N sorted list of length 1. You'll then start merging the sorted lists together to get the fully sorted list.

At each time you'll work with pairs of lists, they're sorted, you'll merge them together to a third sorted list. And this process continues till you have the entire list once again, but this time in sorted form. This divide and conquer approach seems to involve writing a lot of code, but really there is a secret weapon that you have at your disposal. That is recursion, recursion allows you to perform divide and conquer intuitively with compact code. We've had a lot of hands-on practice with recursive problems in this learning path, so hopefully you're comfortable with recursion.

The basic principle behind any recursive problem is that you solve for the trivial of all cases. That is the base case of the recursion, and then build up the complete solution as your recursion unwinds. When you follow the divide and conquer approach of Merge Sort, this is a classic recursion-based algorithm. You essentially divide

your list till it's so small that the problem of sorting becomes trivial. We'll now start off with an unsorted list of numbers. And see how we can use Merge Sort, the divide and conquer strategy of Merge Sort, to sort this list.

Now, when you see a list that's this long, you'll say, I can't sort this, let me divide this list so that we have smaller lists. At each step, you'll half the length of the list that you're working with, we started off with the list of eight numbers. I've now broken this down so that we have two lists roughly half the size of the original list, two lists of four numbers. The list that we have are smaller but they're still not simple enough to work with, so we'll further break down each individual list.

The first four element list is broken down to two lists of two elements each. The second four element list is broken down further into two lists of two elements each. We are in the divide portion of the Merge Sort algorithm and we are not done yet, these lists can be made even simpler if we tried. I'm going to further break down every list that you see here. Every sub-list that we have contains exactly two elements.

I'm going to break this down further so we now have a sub-list with exactly one element each. Each element in our original unsorted list is a separate list. If you consider each of these single element list in isolation, each list is essentially a sorted list. If it has just one element, that element is in the right position.

Let's just consider two sorted lists at a time. Working with two sorted lists, it's easy for us to simply merge together two sorted lists to get a third list that's also sorted. And that's exactly what we'll do here, I'm going to consider the list containing elements 32 and 15, and put them together in a sorted order. So we'll get a new list with 15 first and then 32.

Next, we look at the list containing the elements 11 and 26 and merge them together in the sorted order. This will give us a single list with 11 first and then 26 after that. We'll then move on to the list containing 19 and 36 and merge them together in the sorted order. This will give us a two-element list with 19 first and then 36. Let's look at 44 and 42, merge them together in the sorted order, that will give us a two-element list with 42 first and then 44.

Now, if you look at all of our two element lists, each individual list is in the sorted order. So when we perform further merge operations, we're always working with sorted lists, which we merge together to get third sorted list. Let's combine the two element lists which contain 15 and 32, 11 and 26, so that we get a sorted list with four elements, 11, 15, 26, and 32. In exactly the same way, we'll merge together two element lists containing 19, 36, and 42, 44. So we get a single list in the sorted order, 19, 36, 42, and then 44.

We are now almost done with the merge process. All we have to do is to combine this four element sorted list to get a single sorted list with eight elements. And this sorted list with eight elements is basically our original list. Initially, it was unsorted, but now thanks to Merge Sort's divide and conquer algorithm, it's fully sorted. The merge sort algorithm is a divide and conquer strategy, as you've seen. We create smaller problems that are easier to tackle and work with the most trivial of all problems and then build up the solution. Now in order to calculate the complexity of merge sort, we need to consider the recursive step where the problem is further divided into two.

And we also need to consider the step where we merge two list together. The divide portion of the problem is where we go from a list of length N to two lists of lengths N/2. The merge portion of the problem is where we go from two sorted lists of length N/2 to one list of length N.

Now, this process is done at every step, so it's a rather complicated derivation to get the running time complexity of merge sort. The exact derivation is not really relevant, somebody has already performed this derivation. If you need to know though what the time complexity is. The running time of merge sort is much better than any of the sorting algorithms that we worked with so far, it's order of N logN. Before we move on to writing code for Merge Sort, let's understand some of the characteristics of the Merge Sort algorithm.

The Merge Sort algorithm is not adaptive, there is no way to stop early if we find along the intermediate steps that we have an already sorted list. We have to run the algorithm through to completion. Merge sort is a stable

sort. If equal entities maintain their positions relative to one another, they are not rearranged during the sorting process.

And finally, Merge Sort is a sorting algorithm that requires additional space. Each time we perform the divide step, we need additional space to store the sub-list that we have created. The extra space that is required in Merge Sort is roughly order of N.

Implementing Split and Merge

[Video description begins] *Topic title: Implementing Split and Merge. Your host for this session is Janani Ravi.* [Video description ends]

We are now ready to perform the first of our divide and conquer algorithms for sorting. This is Merge Sort with a worst case running time of order of N log N.

Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

This is a divide and conquer algorithm because at each step we'll split the input array into two parts, try and sort those smaller parts. And then merge the two sorted arrays together. So the first method that we'll implement here in this merge sort algorithm is the helper method that'll allow us to split our input array into two sub-arrays.

Now, observe something different here, we've chosen to work with a list of strings rather than a list of integers. This is just for a little variety and to show you that sorting can be performed with any kind of element, so long as the elements are comparable. This split function takes in three input arguments. The first is the original list that we want to sort, that is, the listToSort. And then two additional input arguments, which are also string arrays, that is lists, listFirstHalf and listSecondHalf. While coming into this split function, both of these arrays, first half and second half, will be empty.

We are now going to iterate over the listToSort and populate listFirstHalf and listSecondHalf. As the name suggests, listFirstHalf will contain the first half of listToSort. And listSecondHalf will contain the second portion or the second half of listToSort. Let's start the split process. The listSecondHalf will hold all elements which are not in the first half.

So the start index for the second half is equal to the length of the first half, and this is what we assign here on line 9.

[Video description begins] *Line 9 reads as: int secondHalfStartIndex = listFirstHalf.length;.* [Video description ends]

We then run a for loop through the entire length of the original list, that is, listToStart. We start at index = 0, and go till the end of the list, so long as index is less than the length of the list. And for each iteration, we increment this index by 1, index++. So long as the index is less than the secondHalfStart index, which we know is equal to the length of the first half list. We assign the element at this index to the first half. listFirstHalf[index] = listToSort[index]. So the entire first half of the original list will be assigned to listFirstHalf.

[Video description begins] *Line 15 reads as: listFirstHalf[index] = listToSort[index];.* [Video description ends]

We enter into the else block if the current index is greater than or equal to the secondHalfStartIndex. In that case, we assign all elements from listToSort in the second half of this list to listSecondHalf. The one thing to note here in the split operation is that listSecondHalf also starts at index 0. This is an entirely new list, so to get the actual index within listSecondHalf, you have to do index-secondHalfStartIndex.

[Video description begins] *Line 18 reads as: listSecondHalf[index - secondHalfStartIndex] = listToSort[index];.*

[Video description ends]

Split here takes care of the divide in our divide and conquer strategy. The next helper method that we'll set up here is the merge method. This is the method that'll take care of conquer within divide and conquer. merge will try to merge two list together in the sorted order, and the original two list will already be sorted. That is the assumption that merge makes.

The merge input method takes in three input arguments once again, listFirstHalf and listSecondHalf. These are already sorted portions of the original list. We want to merge these together into the third list that is passed in here, listToSort. At the input to the merge method, you can assume that listToSort is empty. Whereas listFirstHalf and listSecondHalf contains elements which have already been sorted in the right order. We start merging the two lists together starting at index 0.

[Video description begins] *Line 25 reads as: int mergeIndex = 0;.* [Video description ends]

mergeIndex is initialized to 0, firstHalfIndex and secondHalfIndex are both 0 as well.

[Video description begins] *Line 27 reads as: int firstHalfIndex = 0;. Line 28 reads as: int secondHalfIndex = 0;.* [Video description ends]

Next, we'll run a while loop that'll check the elements at the head of the two lists that we want to merge together. We'll pick the smaller element and add that element to the final list, that is, listToSort. We run this while loop so long as the two indexes into the first half and second half lists are not out of range.

[Video description begins] *Line 30 reads as: while (firstHalfIndex < listFirstHalf.length &&.* [Video description ends]

So long as firstHalfIndex is less than listFirstHalf.length and secondHalfIndex is less than listSecondHalf.length.

[Video description begins] *Line 31 reads as: while (secondHalfIndex < listSecondHalf.length) {.* [Video description ends]

We then use an if condition to check which element is smaller, the one at the head of the first list or the second list. For this we use the compareTo operation.

[Video description begins] *Line 33 reads as: if (listFirstHalf[firstHalfIndex].compareTo(listSecondHalf[secondHalfIndex]) < 0) {.* [Video description ends]

If list of FirstHalf[firstHalfIndex] that is, the head of the first half is smaller than the head of the second half, then we merge in the element in the first half. This is what we do on line 35 and 36. At line 35, we know that the element at the head of the first list is the smaller of the two elements, and that's what we merge into the final list.

We also increment the index of the first list, firstHalfIndex++. We move into the else block if we know that the head element in the second list is the one that's smaller. So on line 40, we merge the element at the head of the second list into our final sorted list, the merged list. We then increment secondHalfIndex by 1.

When execution reaches line 44, we know that we've merged in the smallest element from the head of both lists into the final list. So we increment mergeIndex by 1, mergeIndex++. Now, at some point, we will break out of this while loop. And the condition under which we'll break out is when we've reach the end of either the first list or the second list. So one of these lists have no more items to merge.

So when execution reaches the line of code, line 47, outside of the while loop, we have to see which list still has elements left. If at this point in time, firstHalfIndex is less than the length of the first half list, there are elements here that need to be merged into the final list. And we run a while loop so long as mergeIndex is less than the

length of the final list. We basically assign all elements from listFirstHalf to the final merged list, this we do on line 51.

[Video description begins] *Line 51 reads as: listToSort[mergeIndex++] = listFirstHalf[firstHalfIndex++];.* [Video description ends]

For every iteration of this while loop, make sure that you increment both the mergeIndex as well as the firstHalfIndex. Now it's possible that we broke out of the while loop while there were still elements in the second list that we hadn't looked at.

If secondHalfIndex is less than the length of the second half list, there are still elements here that we need to merge into the final list. If that's the case, we'll run a while loop so long as mergeIndex is less than the length of the final sorted list. And we'll merge all the remaining elements in the second half list to our final list. Make sure you increment both mergeIndex as well as secondHalfIndex for every iteration of this while loop.

Implementing Merge Sort

[Video description begins] *Topic title: Implementing Merge Sort. Your host for this session is Janani Ravi.* [Video description ends]

Now it's time for us to look at the code for the final mergeSort algorithm. Which is a recursive algorithm that'll use the two helper methods that we set up earlier, split and merge.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

Here is the mergeSort algorithm which takes a single input argument, the list we want sorted, this is a list of Strings. We first specify the base case of the recursion. If the length of the list that we want sorted is = 1, that is it has just one element.

Well, a list with just one element is an already sorted list. We can assume that this list is sorted, and simply return. If it has more than one element, we try to find the midpoint of the list and store that in midIndex. So midIndex is calculated by dividing the length of the original list by 2 and adding either 0 or 1. Based on whether the original list have an even number of elements, or an odd number of elements.

[Video description begins] *Line 69 reads as: int midIndex = listToSort.length / 2 + listToSort.length % 2;.* [Video description ends]

Now it's time for the divide portion of our strategy. We'll create two lists here, listFirstHalf and listSecondHalf using this midpoint.

[Video description begins] *Line 71 reads as: String[] listFirstHalf = new String[midIndex];. Line 72 reads as: String[] listSecondHalf = new String[listToSort.length - midIndex];.* [Video description ends]

These two lists will be originally empty and we'll invoke the split method to fill these up with the elements in the first half of the list to sort. And the second half of the list to sort.

[Video description begins] *Line 74 reads as: split(listToSort, listFirstHalf, listSecondHalf);.* [Video description ends]

This is what the split method does on line 74. Once we've divided the original list into two halves, we'll invoke the mergeSort function on the first half of the list. And the second half of the list, this is our recursive call.

[Video description begins] *Line 78 reads as: mergeSort(listFirstHalf);. Line 79 reads as:*

mergeSort(listSecondHalf);. [Video description ends]

MergeSort will ensure that the two halves of the original list will be sorted. After the recursive calls on the two halves of the original list are complete. On line 81 we'll be left with two sorted lists, listFirstHalf will be sorted, listSecondHalf will be sorted as well.

[Video description begins] *Line 81 reads as: merge(listToSort, listFirstHalf, listSecondHalf);.* [Video description ends]

We can now use our merge helper function in order to merge these two list together to get one list that is completely sorted. And we'll print out this merge then sorted list to screen. I'll now create an unsorted list of strings which I'll then sort using mergeSort. You can see that this string contains names, Fiona and Dora are the first two elements and the list ends with Carl. I'll print out the unsorted list call mergeSort, and at the very bottom of the console window here, you can see the final sorted list.

[Video description begins] *Line 97 reads as: mergeSort(unsortedList);.* [Video description ends]

Alex is at the first position, then we have Ben and Carl. Jeff is the final element in this list. But as usual, it'll be more interesting for us to see the intermediate steps in the mergeSort. The first part of this algorithm will involve splitting the list into two halves till we're left with two lists with just one element each. Here is our unsorted list at the very top of the console window. The first split gives us two lists, the one starting with Fiona, the second starting with Irene.

We then recursively split the first half of this list into smaller and smaller lists. Fiona, Dora, Alex in the first half, Jeff and Elise in the second, Fiona, Dora in the first half, Alex in the second, till we're left with just Fiona and Dora. Single element lists can be considered to be sorted, which means we start the merge process.

Dora and Fiona are merged, the elements are sorted with Dora first. We merge in Alex as well, so we now have Alex, Dora, and Fiona. Elise and Jeff are then merged together, and sorted with Elise first then Jeff. And then we have five elements of the firstHalf of the list, which are now merged and in the sorted order, starting with Alex ending with Jeff.

The same split and merge operation is performed on the second half of the original list. I'll leave you to work this out but you can follow step by step how we split till we have just one element in each individual list. And then start the merge operation till we get a final sortedList.

Quick Sort

[Video description begins] *Topic title: Quick Sort. Your host for this session is Janani Ravi.* [Video description ends]

We are now ready to tackle the last of the sorting algorithms that we'll study here in this course. This is quick sort. Quick sort is a highly efficient in-place sorting algorithm. And is often used in real world use cases to get great performance with unsorted elements.

Exactly like merge sort, quick sort is a divide and conquer sorting algorithm. At every step, you partition the list so that you're working with smaller lists. You don't try to tackle the entire list at one go. The basic driving principle for quick sort and merge sort is exactly the same, but the implementation is completely different.

Quick sort, in fact, does not use the additional space that merge sort does. With quick sort, the partition is not based on the length of the list or on an artificial middle index that you found for a particular list. The partition is based on a pivot, and this pivot is an actual element within the unsorted list. Looking at the list, you'll select one of the elements from the list completely at random to be your pivot.

And once you figured out the pivot for one iteration, you'll then perform a number of swap operations. So that your list is partitioned with all elements smaller than the pivot on one side of the pivot. And elements which are larger than the pivot are moved to the other side of the pivot.

The elements which are to the left of the pivot are smaller than the pivot. They need not be in sorted order. Elements which are to the right of the pivot are larger than the pivot. They need not be in sorted order either. At the end of one iteration, for any selected pivot, we've found the right position for the pivot in our sorted list.

And once we have the right position for the pivot, we can then work with the sub-lists to the left of the pivot and right of the pivot. With each sub-list, we apply the pivot partition once again. And this process continues till the entire list is sorted. Every iteration with the pivot will move the pivot element to the right position in the sorted list.

We are now ready to see how quick sort works in practice. It's always easier to visualize the sorting algorithm. Here is the unsorted list that we are going to be working with. Now from this list, you can pick any element at random to be the pivot.

Now consistently, I'm always going to pick the left-most element in any sub-list to be the pivot. Once we've selected the pivot, it's now time to perform our swapping operations. All elements which are less than the pivot are moved to the left of the pivot. We'll just go through the remaining portion of the unsorted list and move all elements smaller than the pivot to the left of the pivot. And all elements greater than the pivot are moved to the right side of the pivot element.

Now it's of course possible to have two elements with equal values within your unsorted list. In that case, you'll simply move all elements less than or equal to the pivot to the left of the list. Alternatively, you could move all elements greater than or equal to the pivot to the right of the pivot. You'll simply include the greater than sign in one of the comparisons that you perform.

Now, we'll perform this operation in a very specific manner. We'll start from the right-most end of our unsorted list. We'll assume those elements are to the right of the pivot, and then find the first element smaller than the pivot. We'll then perform a swap operation with the pivot. This will get the smaller element to the left end of the list and the pivot has moved somewhere to the right. We'll then start looking through the list, starting from the left end.

This is where all elements smaller than the pivot will live. We'll find the first element there from the left, which is larger than the pivot, and we'll then swap that element with the pivot. This will once again get the pivot in a position so that smaller elements are to its left, larger elements are to its right. Now if this seemed confusing, don't worry, we'll see this in action. The process that you see outlined here will be repeated till the entire list is sorted.

Let's see how. At this point in time, we're working with the entire list. We haven't partitioned the list yet. We've chosen our pivot, that is the left-most element in this list. We'll now start searching from the right end of the list for an element that is smaller than the pivot.

So we look at 21. Yes, 21 is smaller than the pivot. We'll now perform a swap operation of the pivot element with the smallest element that we've found so far. The pivot has now moved to the right end of our list. We'll now start searching from the left end of our list to find an element that is larger than the pivot.

We've found the element here, that is 78. We'll perform a swap operation with the pivot. So now the pivot 48 is where 78 used to be. We'll now switch back to searching from the right end of the list. We're looking for an element that is smaller than the pivot. We've found 11.

We'll perform a swap operation to swap 48 and 11 and start searching from the left end of the list for an element that is larger than the pivot. We found our element, 65 its to the left of the pivot and larger than the pivot, for a swap operation. 48 has now moved to the position where 65 used to be.

Back to searching from the right end of this list, we found 13, that is smaller than the pivot. This requires a swap to be performed. 48 has now moved to where 13 used to be. Now when we start at the right end of the list and search up to the pivot, we find that there is no element smaller than the pivot element. At this point, we have found the right position for the pivot element in our final sorted list. All elements smaller than the pivot are to its left. Elements larger than the pivot are to its right.

Now the pivot element 48 is in the right position. Pivot has effectively partitioned our list into sub-lists that we can work with. We'll now repeat this process with the sub-list that is to the right of the pivot and the sub-list that is to the left of the pivot. I choose 65 as my new pivot value. And I'll hunt for an element smaller than 65, starting from the right of the list. There is a no such element, and 65 has no element to its left as well, so we found the final position of the element 65 in our sorted list.

Now 65 has further partitioned our list, we have a list with only the element 78. A single element list is anyway sorted. It's the pivot of this one element list and it's in the right final position. We can now turn our attention to the remaining partition, that is Partition 1. We'll select a pivot element in Partition 1, which is the left-most element. Element 21 is the pivot. And then we start our search from the right end of this partition for an element that is smaller than the pivot. There is such an element here, and that is the element 13.

So we need to perform a swap operation 21 and 13. Once we do that, we start searching from the left end of the partition for an element larger than the pivot. We've found the element 30. We'll perform a swap operation and start searching from the right end of the list for an element smaller than the pivot. Find such an element, that is 11. We'll search from the left end of the partition for an element larger than the pivot, that element is 32.

We'll swap the position of the elements 32 and 21. And at this point in time, the pivot is in the right final position. We found the right final position for element 21. And this has partitioned the unsorted portion of our list. And we can work with now Partition 1 and Partition 2. Work with Partition 2, P is the pivot element.

When we search from the right end of the list we find 30, that is smaller than 32. We'll perform a swap operation. At this point in time, the pivot 32 is in the right final position. And the smaller partition list contains just a single element. As we've discussed before, the single element 30 is already sorted. That is the pivot and is in the right position. We're still left with Partition 1 that we need to sort.

Let's now find a pivot element for this, that is the left-most element, element 13. Starting from the right end of the list, we'll find an element smaller than the pivot. The element 11 is smaller, we'll perform a swap operation. Starting from the left of the list, we'll find an element larger than the pivot, that is 14. We'll swap 13 with 14. Again, moving from the left end of the list, we'll find the element smaller than the pivot, that is 7. We swap 13 and 7, and now finally 13 is in the right final position.

We have fixed the position of the pivot 13 and the pivot has partitioned the unsorted portion of our list. To the right of the pivot, we have a single element list with just the element 14. That can be considered to be sorted in its right final position. We can turn our attention to this two element list here and choose a pivot. The pivot is 11. We'll see that 7 is smaller than 11, starting from the right end of the partition.

A swap operation is in order here. We'll perform a swap that'll get 7 to the beginning of the list. The pivot 11 is now in the right final position. 7 is a single element list and can be considered to be sorted. Our entire list has now been sorted using quick sort. This will become clearer when you write the code for quick sort. There are two distinct phases. The partition method finds a pivot and moves elements to before or after the pivot. And the quicksort method does a recursive call to sort the sub-lists.

Quick sort uses the divide and conquer approach to create smaller problems that are easier to tackle. As is the case of merge sort, the time complexity of the quick sort algorithm has to be derived and this derivation is not really relevant. With quick sort it's possible that if your list order is very skewed to start off with, the time complexity for your algorithm is O of N square. The average case time complexity is O(N log N). And overall, quick sort tends to be more efficient than merge sort.

Implementing Quick Sort

[Video description begins] *Topic title: Implementing Quick Sort. Your host for this session is Janani Ravi.* [Video description ends]

We are now ready to look at the last divide and conquer sorting algorithm that we'll work with today.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

This is quick sort and it often is the preferred sorting technique when your sorting lists. The worst case time complexity of the quick sort algorithm is order of N log N. It's divide and conquer because it partitions the input array into sub arrays which are then sorted.

The advantage of quick sort over merge sort is that it uses no additional space. It is an in place sorting algorithm. Now, quick sort will make use of the helper method swap which we've seen before, which swaps the elements at the jIndex and iIndex. This is exactly the same swap method that we've used in previous algorithms such as selectionSort and bubbleSort.

In this divide and conquer algorithm, the partition helper method is what helps subdivide the input array into sub arrays that we'll work with. The partition method takes in two input arguments, the list that we are in the process of sorting, and a low and a high value that gives us a sub range within this list within which we'll operate. In any call to the partition method, we won't consider elements that have index values lower than low and index values higher than the high value passed in.

The first step in the partition process is to figure out the pivot. Now the pivot can be picked at random. It can be any index in the range low to high. I've chosen the pivot to be the element at index low. I assigned that to the variable pivot.

[Video description begins] *Line 17 reads as: String pivot = listToSort[low];.* [Video description ends]

I then have temporarily variables l and h that will hold the low and high values to start off with. The next step is to run a while loop that will position the pivot at the right index value within this sub-range low to high.

At the end of this partition call, all values smaller than the pivot should be before the pivot in this sub range. And all values greater than the pivot should be after the pivot in this sub range low to high. So long as l is less than h, we run a while loop to perform this operation. If the element at l compared to the pivot is smaller, and l is still less than h, we simply let that element remain as is within the sub range and increment l by 1. This nested while loop that we have on line 26 will compare every element with the pivot.

And if it's less than or equal to the pivot, we'll leave that element as is and increment l. Now, at the end of this while loop on line 29, the index l will reference an element that has a value greater than the pivot. We then enter the while loop on line 30. The while loop on line 30 will compare the element at index h with the pivot. So long as this element is greater than the pivot, we'll decrement h, h--. So when execution reaches line 33, l points to a value greater than the pivot and h references a value smaller than the pivot.

So long as the index l is less than h, we perform a swap operation and swap the elements at l and h. Once the swap has been performed, we can go back to comparing the elements at index l with the pivot, make sure that they are smaller, and comparing the elements at index h with the pivot and make sure that they are larger.

This process will continue till the pivot is in the right position within this sub range, with elements smaller than it before it, elements larger than it after it. When we break out of the while loop, we'll perform one final swap operation where we'll swap the pivot to the right position. On line 42, we'll swap the element at index low with the element at index h.

The pivot is now at position h and the return value from this partition method is the index h. This is on line 51. It's now time for us to write the code for the quickSort method that will make use of this partition helper method. Here is quickSort which takes in the list that we want sorted and a range low to high. This range low to high gives us the sub-range of the list that we want sorted.

Now if at any point in time, the low index is greater or equal to the high index that means, this list is now completely sorted and we return. This is the base case of the recursion. The next step is to partition this summary in the range low to high around a pivot. We invoke the partition function, pass in the list to sort and the low and high range. This will return to us the index of the pivot value that we had chosen. At the end of this partition operation, the pivotIndex will be in the right position of the list.

[Video description begins] *Line 59 reads as: int pivotIndex = partition(listToSort, low, high);.* [Video description ends]

Then we can recursively call quickSort, but this time we'll specify a different range. We'll go from low to pivot index -1, and then quickSort from pivotIndex +1 to high.

[Video description begins] *Line 62 reads as: quickSort(listToSort, pivotIndex + 1, high);.* [Video description ends]

These recursive calls to quickSort will sort all elements other than the pivot, to the left of the pivot and to the right of the pivot. We are now ready to test out our quickSort functionality. We start with an unsorted list of names.

Fiona and Dora are the first two elements, we end with Harry and Carl. We print out the unsorted list, use quickSort to sort this list. Observe that the first invocation of quickSort considers the entire list, starting from index 0, all the way through to index length of list -1.

[Video description begins] *Line 75 reads as: quickSort(unsortedlist, 0, unsortedlist.length - 1);.* [Video description ends]

And once the quickSort process is complete, we'll print out the final sorted list. Run this code, and at the very bottom of your console window, you will see the list in its final sorted order, Alex and Ben, Carl and ending with Jeff. What will be more interesting is to look at the intermediate print statements that we have to see how the process of quickSort is carried out.

We have the original unsorted list here starting with Fiona and ending with Carl at the top of the console window. The first pivot that we've chosen is the name Fiona. You can then see that the partition operation performs a number of swaps of elements. The objective here is to get Fiona to the correct position in this list. After three swaps, when the partition is complete, you can see the result in partitioned. Observe that Fiona is somewhere in the middle of the list. Ben, Dora, Alex, Carl, and Elise, which come before Fiona and the elements which come after Fiona are Gerald, Irene, Harry, and Jeff.

So Fiona is in the right position. Next, we'll perform partition on the first half of the list from Ben to Elise, that is before, the pivot Fiona. The new pivot is Ben, we perform a bunch of swap operations and Ben gets to the right position, which is the second position in the list. That's what you see in the partitioned output here. This process continues till one by one in every sub list, the pivot element is moved to the right position. And once this is done, we have a fully sorted list, as you see here at the bottom of your screen.

Let's try quickSort once again. This time, we'll start with a list that is almost in sorted order, you can see that Dora is in the wrong position. Carl and Ben are also not in the right order, but the remaining elements seem to be mostly positioned correctly. I'm going to print out the unsortedList called quickSort and then print out the final sorted list.

Go ahead and run this code and observe that each time we partition the list around a pivot, there are very few swaps performed. This is because the list is almost in the sorted order. And finally, at the very end of all of the partition operations, you'll get a fully sorted list. Now I don't know if you noticed, there is one unnecessary swap we perform each time.

Once we found the final position of the pivot, we swap the pivot with itself in the same position. Observe here on screen when the pivot is Irene, we swap the element at index 8 with the element at index 8. Now we can improve this by performing a simple check. Head over to our partition operation where we perform the swap into the final position. Get rid of this code that we have there and paste the same code here within an If check.

Now we'll perform the swap operation only if the two indices are different. If low is not equal to h, then perform the swap. Just this little change here will get rid of one unnecessary swap operation for us. When you run this code, you'll take a look at the result and you'll see that the swap operation won't be performed if the pivot is in the right position. This is a small improvement that can really help especially if the lists that you're working with are very large.

Binary Search

[Video description begins] *Topic title: Binary Search. Your host for this session is Janani Ravi.* [Video description ends]

From sorting algorithms, it's only natural to move on to a discussion of searching algorithms. Searching algorithms typically work with sorted list. There is only one way to search an unsorted list, and that is use linear search. A popular and performant searching algorithm for use with a sorted list is binary search, and that's what we'll study in this video. Binary search tries to be smart about how you search for elements within a list. A naive way to search for an element is to check every element till we find the right one.

Now, this will work for both sorted as well as unsorted lists, and the time complexity of this search, called linear search, is O of N. With unsorted lists, we don't really have an option but to perform linear search. But with sorted lists, we ought to be able to do better. We should be smarter about how we search for an element in a sorted list.

Binary search follows a divide and conquer approach to search for an element within a sorted list. We first choose an element somewhere at the midpoint of the sorted list and check to see if that element is equal to the element that we are looking for. If it is, we've found the element in our sorted list. If not, we'll check to see whether the midpoint is smaller than or greater than the element that you're looking for.

One of these conditions have to be true, either the midpoint element is smaller than the element that you're looking for, or it's larger than the element that you're looking for. Let's say that the element at the midpoint is larger than the element that you're searching for. This bit of information immediately allows you to halve your search space. You halve the portion of the list you need to search by only considering those elements that are before the midpoint. Your element, if it exists in the list, lies only before the midpoint. Binary search thus can only be performed with a sorted list.

Here is a list, it contains unsorted values, if you want to be able to apply binary search, you need to sort this list of element. You can use any of the sort techniques that we've encountered before. Once you have a sorted list, let's say that you're looking for the element 42 within this sorted list. Let's start off by calculating a midpoint for this sorted list. For binary search, you're always working with an index range in your list. You get the min-index in your list, the Max-index. The min-index and the Max-index encompass the range of elements within which you're performing your search.

Let's now compute the midpoint. This is usually done by computing Max-index plus min-index divided by 2. The element found at the midpoint in our case is the element 26. Now, this 26, that is the middle value, is clearly

less than 42, the element for which we're hunting. 42 clearly cannot lie in that portion of the list that comes before 26.

So we can eliminate that portion of the list entirely. Instead, we'll focus on the portion of the list that comes after 26. We have a Min-index, Max-index, and we'll compute a midpoint. The computation of the midpoint here will give us the element 36, that is the middle element. We'll compare 36 with our lookup value, 42. 42 is clearly greater than 36.

So we can ignore that portion of the list which comes before 36. We'll focus our attention on the remaining portion of the list. We'll have a min-index for this range, a Max-index, and we'll compute the midpoint. Once we compute the midpoint and see the element that's stored at this middle index, you'll find that the element is 42 itself, there is a match. We've found the element that we're looking for, our search has ended successfully.

If at some point our Max-index becomes less than the min-index and we haven't found the element, that's when we know our binary search is unsuccessful, the element does not exist in our list. It must be pretty obvious to you that binary search works much faster than linear search. By halving the search area at each step, binary search has a much better running time. The complexity of binary search is order of log N.

Implementing Linear Search

[Video description begins] *Topic title: Implementing Linear Search. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll implement the simplest of all possible searching algorithms. This is linear search.

[Video description begins] *The screen displays an Eclipse IDE window, which shows various lines of code in a code editor window.* [Video description ends]

Now, linear search is the only searching algorithm that works with both sorted as well as unsorted lists. Now linear search doesn't really have a fast running time. Its time complexity is O of N. Because we need to iterate and look through all elements in the list before we can determine whether the element that we're looking for has been found.

Now linear search is so simple and straightforward that it's kind of weird to call it an algorithm, but it is indeed. Here's the code for the linear search algorithm that will look through every element in the list before it finds the one that we are looking for. Linear search takes in two input arguments. The first is the list within which we'll perform our search. The list here is just a string array. The second input argument is the element that we're looking for within this list.

We'll print out a message to screen indicating what element we're looking for. We'll then run a simple for loop that iterates over each element in this list starting at index i = 0. So long as i is less than the length of the list, we increment i by 1. For every iteration of this for loop, we'll print out the index value and check to see whether the element at this index is equal to the element that we're looking for.

[Video description begins] *Line 13 reads as: if (list[i].equals(element)) {.* [Video description ends]

If yes, we return the index position of the element. This is on line 14. Now if we run through all of the elements in the list and we haven't returned, that means this element is not present in the list. We simply return -1, indicating the element was not found. For all other search algorithms, we'll work with sorted list. But for this particular algorithm, we'll use an unsorted list.

Here's an unsortedList, which is an array of names. So first, I'm going to perform linearSearch to find the name Harry within this list. Harry does exist and if you run this code, you can see that it is at index 8. As soon as the element Harry is found, the for loop exits. And we return the index of the element Harry.

Let's try linearSearch once again, this time we'll search for elements Jeff and Nora within our unsortedList. Run this code, you can see that Jeff is present at index 11. And the element Nora has been found at index 15. Now let's search for an element that is a name that we know does not exist in this list. I'm going to search for the name Zoe.

Now this time when we run this code, you'll find that linearSearch searches all the way through the last element at index 15. Nothing was found so it returns -1. Often the lists that you work with in the real world are sorted. The disadvantage of Linear Search is that it offers no improvement in performance time when it works with a sortedList. Here is a sortedList starting with the name Alex going all the way through to Peter.

Let's perform linearSearch for the name Harry within this list. Even though our list is sorted, linearSearch is not able to take advantage of this fact. linearSearch will start searching from the element at index 0. It goes till index 7, Harry is found. And that is the index that we return.

We'll now search for the elements Jeff and Nora. Once again, a linearSearch will iterate through each element till the element that we are looking for is found. Jeff is found at index 9. And Nora is found at index 13. Now, let's look for a name that does not exist in our array, that is Zoe. If you run this code, you will see that all elements have to be checked before we know that a particular element is not present. There is no way to short circuit this operation in spite of the fact that the list is in sorted order.

Implementing Binary Search

[Video description begins] *Topic title: Implementing Binary Search. Your host for this session is Janani Ravi.* [Video description ends]

A very popular algorithm used to search for elements in a sorted list is binarySearch.

[Video description begins] *The screen displays an Eclipse IDE window with various lines of code.* [Video description ends]

The complexity of the binarySearch algorithm is order of log N. It's a divide and conquer algorithm that takes advantage of the fact that a list is in sorted order to narrow the search range. The search space at each iteration is halved. Our binarySearch method here takes as an input argument the list within which we want to search for an element. And the element that we're looking for, that is the string element. We print out the element that we are searching for, and we then assign values for the variables low and high.

Now we want to search the entire list to start off with. So we start with low = 0, and high is the last element in the list, list.length - 1. Now binarySearch can be performed iteratively or recursively. This first solution that we're going to look at will perform binarySearch in an iterative manner. Our initial search space is the entire list, and that's why low starts at the first index position 0 and high is the last index position.

So long as low is less than equal to high, we perform the search algorithm. That is our while loop. We find the midpoint of the range within which we are looking for the element, mid = (low + high) divided by 2. This midpoint is what we use to divide our search subspace into two. We then perform an if check on line 19.

To see whether the element at the center of the list at the midpoint is the element for which we've been searching. If list of mid equals element, then we simply return the midpoint index. The element has been found, we can stop the search process.

Now if the element at the center of the list is not equal to the element for which we are hunting, we'll now narrow our search. If the element at the midpoint is smaller than the element that we're looking for within this list. We then know that the element is to be found in that part of the list which lies after the midpoint. So if list of [mid].compareTo(element) is less than 0, then the low range within which we'll hunt for the element is mid + 1.

Otherwise, if the element at the midpoint is larger than the element for which we've been hunting, we set high to mid - 1. At any point we'll only look for the element in the index range low to high. At every iteration of this while loop, we've halved our search space. We'll only look for the element in one-half of the list. Once again, we find the midpoint of one-half of the list and continue this process till the element has been found. Now if we break out of this while loop and execution reaches line 31. That effectively means that we haven't found the element that we've been looking for, we simply return -1.

Now it's time for us to test out our binarySearch code. Remember, this only works with sorted list. So make sure you specify a sorted list before you invoke binarySearch. Here is a list of names, it's sorted. I'm going to search for the name Harry. Once you run this code, you'll immediately see something interesting. Our initial search space is the entire list starting at index 0 going up to index 15. The midpoint here is 7 and the element at this mid index is Harry. Harry is found right away and we return the index of the name Harry, that is 7.

So, we essentially found the element Harry with just one iteration of the while loop. Let's try this once again, we won't always be so lucky. Let's search for the element Jeff within our sorted array. Run this code and scroll down in the console window where we have our result. Observe that when we start our initial search, we consider all elements in the array. Low is 0, high is 15. The midpoint is 7, and the mid element is Harry.

Now we know that Jeff comes after Harry in this sorted list. This allows us to narrow our search space so in the next iteration of the while loop, we only look at elements starting at index 8 up to 15. Now the midpoint is 11. The element at the midpoint is Lewis. Once again, when we compare with the midpoint we know that Jeff should come before Lewis.

So we narrow our search space, once again, low is 8, high is 10. The midpoint now is 9 and the element at the midpoint is Jeff. We've found Jeff and we return the element index 9. The next element that we'll search for within the sorted list using binarySearch is Nora.

Go ahead and run this code, and if you scroll down here within the console window. You'll see how the search space narrows at every iteration of the while loop till Nora is found. Even though the element Nora is at index 13, that is towards the end of the list, we require just three iterations to find Nora.

Now let's look for an element that is not present within this list. This is the name Zoe. If you run this code, you will find that the search space is repeatedly halved at every iteration of the while loop. But each time we narrow the search space, Zoe is not found till at some point low will be greater than or equal to high. Notice in the last iteration, the low index is equal to 15, high index is also equal to 15, and the element Zoe has not been found. This is where we return -1, indicating search not successful. binarySearch can be implemented recursively as well and that's exactly what we'll look at here.

Here is the same binarySearch method, but this time the input arguments to this method are a little different. The input arguments include the list within which we perform the search, the element for which we are looking and also a low and high index value. This gives us the range of elements for this list within which we have to perform the search.

First is the base case of the recursion. If we find that the low index is greater than the high index, this means we have narrowed the search space and the element has not been found. We simply return -1. Otherwise, as is usually the case, we calculate the midpoint of the current search space, (low + high) divided by 2.

If the element for which we've been looking is found at the midpoint, that is, list mid equals element, we return the mid index. Else, we check to see whether the element at the midpoint of this range is smaller than the element for which we're looking. So list[mid] compareTo(element) less than 0, we invoke binarySearch recursively and we make the search space smaller. The search space goes from mid + 1 all the way through to high. Otherwise, if the element at the midpoint is larger than the element for which we're looking, we perform binarySearch on the first half of the range.

We go from low to mid- 1. This is on line 28. This is the same binarySearch as before, but we perform the search recursively rather than iteratively. Let's go ahead and set up a sorted list, and perform binarySearch. We need to specify a range for the binarySearch to start off with. We'll start off with the first element at index 0, and specify the last element at index length of list - 1.

Go ahead and run this code and see how we search for Gerald. In the range 0 to 15, the midpoint is 7, the element there is Harry, so we only search in the range 0 to 6. The midpoint here is Dora, we further narrow the search. So, we look for Gerald in the second half of the list after Dora. Now, this continues till finally we find Gerald at element index 6. The process here is exactly the same as before, except that we perform the implementation recursively. I'll leave it to you to test this recursive binarySearch out further with other elements.

Implementing Jump Search

[Video description begins] *Topic title: Implementing Jump Search. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll explore another search algorithm that works with sortedList. This search algorithm is called jump search, and it's an improvement over ordinary linear search which has O(N) complexity.

[Video description begins] *The screen displays an Eclipse IDE window with various lines of code.* [Video description ends]

Jump search has the best case time complexity which is order of square root of N, where n is the number of elements in your list. Jump search is so called because you jump ahead using a fixed step size while searching for elements within a sortedList. The fact that you jump ahead reduces the search space, where you'll have to perform a linear search to find the element that you're looking for. Because we jump ahead using a fixed step size or interval, the time complexity of jumpSearch depends on the step size that you pick.

It has been found that the best step size is square root of n. When you choose a step size of square root of N, the running time of this algorithm will be order of square root of N. We'll see that in just a bit, but first, we'll see how jump search is implemented. Here's my method jumpSearch which takes in 3 input arguments.

The list within which we want to search for the element, the element that we're looking for that is of type string, and the list is a string array, as you can see. And the third input argument here is the jumpLength. The jumpLength is the step size that we'll use when we jump over elements while performing the search.

We'll initialize the index i to be equal to 0, this we do on line 9. Now we'll run a while loop, which will jump over the list that we've passed in, trying to find the sub list within which we will perform linear search to search for an element. We'll check the element at index i. So long as the element at index i is less than the element that we're looking for, we'll jump ahead by jumpLength. This while loop that we have here on line 11 will run so long as the element at index i is smaller than the element which we're looking for.

Within the body of the while loop, we'll increment i to be i + jumpLength. As the increment i, we need to be sure that i does not go beyond the length of the list, does not exceed the range of our input list. If that happens, we simply break out of the while loop. When our code execution reaches line 22, at that point in time, the element at index i is the element that is just larger than the element that we are hunting for. So in order to search for the element in a sub range, we need to start the sub-range at i - jumpLength. So we set int startIndex to be i - jumpLength.

And the endIndex of the sub range within which we'll perform linear search is essentially i, or the length of the list. This is because if i is beyond the length of the list we'll only look for our element till the end of the list. In order to help us debug, I'll print out to screen that we are searching for the element between startIndex and endIndex.

Observe that now we'll perform linear search using the for loop. But we won't be looking through all elements in the list. We'll only be looking at a sub range of the list. And the length of the sub range over which we'll perform linear search is equal to n divided by the jumpLength that we've chosen. N here refers to the number of elements in the list as a whole. So let's take a look at the code which performs linear search. We initialize j to be equal to the startIndex, and j is incremented by 1 so long as j is less than equal to the endIndex. We then check whether the list element at index j is equal to the element that we are looking for.

If yes, we return the index j. Now, if we come out of the for loop, that is if execution reaches line 35, we haven't found the element in this sub range. This means that the element does not exist in the sortedList at all, we return -1. We are now ready to test out our code for jumpSearch. Here I have a sortedList of names with a total of 16 names. The initial jumpLength that I've chosen is equal to 6.

Remember, you can choose the jumpLength to be anything, but the best results will be obtained when jumplength is equal to square root of N. Here, jumpLength is 6, which is not square root of 16, instead, it's a little greater. But we'll run this code and see how it works. I'll then perform a jump search for the name Gerald within my sortedList and I'll pass in the jumpLength as well.

Go ahead and run this code. Observe that initially we start off at index i = 0. We see that the element that we're looking for is greater than or equal to Alex, so we jump forward 6 elements. And we look at the element Gerald, which is at index 6. We are looking for the element Gerald. So the element is greater than equal to Gerald, and the sub-range within which we look for the element is between index position 6 and 12.

Our linear search is performed between the indices 6 and 12. And the name Gerald is found at element index 6. That is what is returned. It's pretty clear from the simple example here that jump search works much faster than ordinary linear search.

Let's perform jumpSearch once again, and this time we'll look for the element Mary within our sortedList. Run this code, and you can see that we start searching for Mary by comparing Mary to Alex. Mary is clearly after Alex in a lexicographically sortedList . We then compare Mary with Gerald, we've jumped ahead six names. Mary is clearly after Gerald, we then jump ahead to Mary itself. Mary is clearly greater than equal to Mary, we start linear search between the elements 12 and 16. And we immediately find Mary at element index 12.

Let's try a few more searches. The next search we do is for the name Ophelia. Ophelia is towards the very end of the list. When we run this code, you see that we jump ahead from Alex to Gerald then to Mary. Then we search for Ophelia between the indices 12 and 16, and we find Ophelia at element index 14.

Let's do one more search, this time for the name Zoe, which we know is not present in the list. When you run this code, you'll see that you'll start our search at Alex, jump forward to Gerald and then to Mary. Zoe is still greater than equal to Mary. At this point, we've gone beyond the end of the list. So we search linearly from elements 12 to 16 to look for Zoe.

This element is not found, we return -1. For the best performing running time, the jumpLength for any list should be equal to the square root of the length of the list. So I'm going to use the Math.sqrt operation and find the square root of the sortedList . I'll convert it to an integer and assign that value to jumpLength.

Let's talk about how we compute square root of N be the best possible jumpLength. If M is equal to the jumpLength that we've chosen, and N is the number of elements in our list, the number of jumps that we make to search over all of the list elements is equal to N/M. In the worst case, we have to jump over the list N/M times to find the right sub range within which we want to perform linear search. Once we found the right sub range within which we'll search for our element, we'll need to perform linear search.

So if M is the jumpLength, N is the number of elements, the number of elements to check using linear search is equal to M - 1. This is the number of elements that we'll check within a sub range. Now with this information available, we can put this together with the number of jumps to get the total time complexity of jump search. So let's consider the time complexities that we've already calculated. The number of jump operations is N/M, so that

is the time complexity of the jumps. And the linear search is of the order of M - 1.

Thus, we get time complexity of jump search is order of (N/M + (M - 1)). The value of a jumpLength in the best case, which minimizes the running time of this algorithm, is square root of N. This is why we choose the jumpLength to be square root of N, to get the best case running time. When M is equal to square root of N, we get jump searches is big O(square root of N + square root of N). Two multiplied by square root of N is just big O of (square root of N). In our code here, we've set our jumpLength to be square root of N. Let's go ahead and run this code and you will see how jumpSearch performs to find each of these elements within our sortedList .

Implementing Interpolation Search

[Video description begins] *Topic title: Implementing Interpolation Search. Your host for this session is Janani Ravi.* [Video description ends]

Just like jump search is an improvement over basic linear search, interpolationSearch is an improvement over basic binary search. Let's see how we can perform interpolationSearch.

[Video description begins] *The screen displays an Eclipse IDE window with various lines of code.* [Video description ends]

interpolationSearch is an improvement over plain vanilla binary search. When the elements in the sorted list within which we perform our search are uniformly distributed. They are not skewed in any way and they are uniformly distributed within the range of the minimum and maximum element.

The main difference between interpolationSearch and binary search is where we look for a particular element. In binary search, we always split the list into two equal halves, and search for the element at the midpoint of the list. With interpolationSearch, we don't always look at the midpoint. The search location is different. And it depends on whether the element for which we're hunting is closest to the largest or the smallest element in our current range.

If it's closer to the largest element, then our search location will be closer to this largest segment. If it's closer to the smallest location, our search location will be towards the left, towards the smallest element. This will become clearer when you look at how interpolationSearch works. It takes us an input argument the list within which we'll search for an element and the element that we are looking for. The initial range of our search will be starting at index 0 and ending at the last element of the list, length of list - 1.

So long as low is less than or equal to high, we'll split the search subspace into two sub-parts except that they won't be too halves. Notice how mid is calculated, it's not a simple calculation of the middle of the list. This calculation here takes into account the actual value of the search element. element - list(low) will show us how far away the element is from the low end of this range.

We then multiply this value by the index range, that is, high - low. And divide the whole thing by the range of the elements stored within this list, list(high) - list(low). mid will be an index value closer to the lower end of the sub-range if element is close to the element at list index low. It'll be closer to the higher end of the sub-range if the element is closest to the largest element in the sub-range.

We'll print out Low, High, Mid and Mid element. And then the rest of the code remains the same. If the element at index mid is equal to the element for which we are searching, we simply return the mid index. If the element at the mid index is less than the element for which we are searching, low = mid + 1, otherwise high = mid -1. The rest of the code is exactly as plain vanilla binary search, only the way we calculate the mid index is different. If we break out of the while loop here and get to line 31. That means that the element has not been found in our list. We return index -1.

We are now ready to test out interpolationSearch. Remember, it works well when the elements in our sorted list are evenly distributed. Here is a sorted list with evenly distributed elements, fairly evenly.

Let's perform interpolationSearch to find the element 80 within this list. Go ahead and run this code, you can see low is 0, high is 13, the mid value calculated was 5. That's because the value 80 was closer to 10, which is the smallest element in this list. And further away from 180, which is the largest element in this list. The mid that was calculated was 5, and 80 was found at index 5.

Let's try this once again and this time we'll hunt for the element 90. Go ahead and run this code and 90 has been found at index 6. Low 0, high 13, 90 was found to be closer to 10 rather than 180. And so the mid picked was 6 and 90 was found at element index 6. Let's try this once again. This time we'll perform interpolationSearch to find 130. Run this code. And you can see in the first iteration while searching for the element 130, the mid that was calculated was 9. The number 130 is closer to 180. That is the largest element in this list, then 10, that is the smallest element in this list.

The midpoint calculated here is also close to the end of the list. The element was not found at index 9. We once again calculated the mid, which was at index 10, and the element was found at index 10. We'll try this one last time. We'll perform interpolation list and look for the element 122, we know this is not present in our sorted list. It takes just three iterations in order to figure out that 122 is not present and we return -1. I'm now going to make one simple change here.

I'm going to go to the code for interpolationSearch and change the way we calculate the midpoint to be the same as that of regular binary search. I'll just run the same code that I had earlier, searching for all elements 80, 90, 130 and 122. You can see that for each of these elements, we perform many more iterations before we find the element in our list. It's pretty clear here that interpolationSearch for uniformly distributed elements in a sorted list works much better than plain vanilla binary search.

Course Summary

[Video description begins] *Topic title: Course Summary* [Video description ends]

In this course, we explored in a lot of detail different algorithms that we could use to sort data stored in a list format and search for elements in a sorted list. We also saw how we could estimate the time complexities of our sorting and searching algorithms. And the trade offs that we make in terms of space and time during the sorting process.

We first understood and visualized how exactly basic sorting algorithms work. Such as selection sort, bubble sort, insertion sort, and shell sort. These are sorting algorithms with time complexity close to O of N square, and work well with smaller lists. For larger list, we saw that it's often better to use faster sorting algorithms which use divide and conquer techniques. Algorithms such as merge sort and quick sort.

We saw that both algorithms have time complexity close to O of N log N, but merge sort needs additional temporary space to perform sorting. We then moved on to searching algorithms. We discussed that the only search algorithm that works with unsorted list is linear search. But for sorted list, there exists several algorithms which are quite efficient.

In this context, we explored and implemented binary search, jump search, and interpolation search. In the next course in this learning path, we shall explore an interesting data structure which is based on binary trees. That is the binary heap.