Module 14 Sorting

The Sorting Problem 1.2

In this Module, we're going to talk about sorting algorithms. Let's start first by stating the problem, kind of obvious by the name you can probably guess what this problem is trying to solve, but let's be formal here. So given an array, AR of numbers, let's change their order. So at the end they would show in an increasing order. So for example if our original array contains some values: five, eight, twelve, seven, and so on. After sorting, the array elements would change their order. So, they are now there would be: five, seven, eight, eight, ten, twelve, again in increasing order.

Sorting Algorithms 1.3

There are a lot of sorting techniques; one is selection-sort, insertion-sort, bubble-sort, merge-sort, quick-sort, heap-sort, and many, many more. In this model we're going to focus on only two of them: the selection sort and merge sort. Selection-sort is a very basic naive sorting algorithm. Merge-sort is a more sophisticated one but a very nice one as well. Let's go ahead and start with a selection-sort.

Selection Sort 2.1

Let's take a look at selection sort. As I said earlier it's very basic our sorting algorithm, maybe the most intuitive or naïve algorithm we can think of; sort of how you would explain your kid how to sort using this approach. So, let me demonstrate the idea here. So, assuming we have a collection of elements, obviously not sorted: five, fourteen, ten, eight, thirteen and so on. It would be an iterative process each time placing the minimum element at the next position; so it's going to work something like that. So let's take a look at the entire range we're sorting and first iteration, we are looking to place at the zero position, in the first position, the minimum element. So we'll search for the minimum element, we'll get this one, index-five, and then we'll just swap them. So one would be at the beginning and five would be somewhere at the rest of the elements. So our range would now be only the elements from the second on. This iteration, second iteration, will place the second minimum at the second position; we'll search for the minimum in that varied range: that would be two and we'll swap the two in fourteen placing two at that position. The range is shorter now; we'll search for the minimum that would be three and we'll swap them, three and a ten, so three would be in the right position. So now we have one, two, and three already ordered, and next we find the minimum in the valid range. That would be three in the index-eleven and we'll swap them. So, three would be at the right position and so on and so on. Yeah, let's try to implement this idea.

Selection Sort Implementation 2.2

Let's implement a selection-sort algorithm. So we'll write a function, selection-sort, that is given an array, basically the starting address int ARR brackets and logical size ARR size, and this function should reorder the elements so they would be in an increasing order. No return value here, so it would, it's a void function basically.

As we said it's an iterative algorithm, so we need to iterate over the elements one after the other, over the indexes basically or the positions, one after the other each iteration placing the correct element, i-th order element, in the i-th position. So we'll use a for loop to iterate over the indexes, and we'll use an index i ranging from one up to ARR size minus one; so i goes from zero less than there ARR size plus plus. Each iteration will search for the index where the minimum element is so we'll have min index equals to

the location where the minimum element is. For that we would define a function: find index of min, right. So each iteration min index would get whatever find index of min function returns and after we do that we need to swap them. But let's first define the prototype of find index of min. So find index of min would get basically the array but it also should get the range where we are looking for the minimum index. So initially it would probably be the entire array, then it would be with one item less, and another item less and so on and so on through. So let's assume that find index of min would get the low index and the high index to represent the range where we're searching for the minimum index.
So find index of min is given the position where our array starts, low and high indexes to indicate the range we're searching the minimum in, and it should return an int, basically the index of the minimum element in the positions between low and high. So, assuming we have this function, in a minute or two we'll implement that as well, but assuming we have this function each iteration we need to find the index of min in the current valid range. So first iteration it should be from zero to the end, then it should be from one to the end, then from two to the end, and then from three to the end, and so on. So basically we are searching for the index of min on the array ARR starting at index i, right; first iteration starting at index zero, second iteration the starting index is one, the next iteration the starting in the two, and so on. And every iteration we stop at the end of the array; so first edition it goes zero to the end, then one to the end, then two to the end, and three to the end. Each time we're searching for the minimum index in that range. After we get that min ind for each iteration, we're just swapping the current element ARR i with the element at the min index, ARR min Ind. That's basically it; that's the selection-sort algorithm.

Selection Sort Implementation 2.3

Let's implement the find index of min so we would have that as well. So once again find index of min is given the starting index, the starting address of the entire array, but in order to indicate the range we are searching for the minimum index we have low and high, two indexes that basically indicate that range. We would need to accumulate the index of the minimum element; actually, let's accumulate two things, not only the index of the minimum value, but also the minimum value itself. So that have two variables min and min ind. And let's accumulate these two values over the path of the elements between low and high. So initially, let's set min to be the first element: ARR at the low position, the min ind would be low obviously. And then, we'll iterate over the rest of the elements using a for loop basically iterating from low plus one to high, accumulating the minimum and the minimum index. Eventually after doing all of that, we would just return the index of the minimum: the min ind. So let's just iterate and accumulate that value. So we'll use an index i ranging from low plus one, low is already taken care of before the loop, so we'll start at low plus one, up too high so for i goes from low plus one less or equal to high incrementing by one, i plus plus. Each iteration let's compare ARR i, the current element, to the minimum we've seen so far, right. So if ARR i is less than the minimum we've seen so far that is the new minimum. Basically we'll assign min to be ARR i and also save min index as our current location as i. So if we first set min and min index to be the first element, then we're starting to iterate from the second and on up to i, each time checking whether we need to update the minimum index. We are accumulating both min index and min itself, eventually we can just return the min index. So all together, we have a function that finds the index of min in a range from low to high and by calling it over and over on range that changes each time and the swap, we are basically sorting the entire array.

Runtime Analysis 2.4

Let's analyze the running time of selection-sort and see how efficient this algorithm is. Before we analyze selection sort, let's start by analyzing find index of min, this helper function, so we would know what's the cost of each call there and then we'll just use that when we are analyzing a selection sort

algorithm. So let's start with a find index of min; let's find the running time and the running time of that function. Like all algorithms that we are analyzing, we need to give the running time as a function of the size of the input. So, in this case the size of the input N would be the number of elements in the range were searching for the index of min; so let's have N as the number of elements between low and high, arithmetically we'll just have high minus low plus one, that the number of elements between low and i so this is our N. And let's analyze the running time as a function of N. So once again, before we start the for loop, we are doing a constant amount of operations and after the for loop we are having a constant amount of operations. So min equals ARR low, minInd equals low, that's theta of one, and return min Ind that's also theta of one. And we're mostly interested at the cost of this for loop. Once again the body here, each iteration is constant; it's only two or three primitive operations so that's theta of one. Which means that the number of iterations, that's basically the cost of this for loop. Here the number of iterations is N, so the entire cost is theta of N.

Runtime Analysis 2.5

Let's go to the selection-sort algorithm now that we know that find index of min is theta of N, in other words is linear, let's use that when we're analyzing the selection-sort algorithm. Once again let's define N, as the size of the input ARR size in this case, and let's try to figure out what T of N is. So, we have a for loop that we need to figure out what's the cost of this for loop. But not like the other algorithm we had so far, each iteration the body costs differently. And let's try to see what the cost of each iteration. So basically the most expensive, I would say operation here, is the call of the function find index of min. And then since the range changes from iteration to iteration, basically the linear cost or the number of elements that we are going over, changes from iteration to iteration. I would say that the cost of the body could be described as theta of N minus i, right. So when i equals zero, when we're going over the entire range, we are basically paying N, right. When i equals one, when we're going over one item less we're basically doing N minus one. And when i equals two, we're doing N minus two and so on. So basically when we are summing up all the cost of calling find index of min, we'll pay N for the first call, and we'll pay N minus one for the second call, and N minus two for the third call and so on and so on, till the last call would be two and one. So that's basically the cheapest score. So the entire set of calls to find the index of min would be the sum of that, of these numbers: N plus N minus one plus N minus two and so on. You know that this here, actually we've seen it in one of our previous models, that we can think of it as a kind of a triangle inside a square. And we know that it is theta of N squared. You can look at the exact formula and figure out where it's theta of N squared but we've already seen that this kind of a sum is a quadratic running time. Yeah, so basically meaning that selection-sort is a quadratic running time algorithm for sorting.

Merge Sort 3.1

The second sorting algorithm we're going to talk about is called merge-sort; it is a recursive algorithm, a recursive sorting algorithm. We can think of, again a few approaches to recursively sort a sequence of numbers, but let me show you how merge-sort works. So for example we have this array of numbers: fourteen, five, eight, ten, thirteen, one, eighteen, and three, and we want to sort it.

One way of doing it is to have a three step process. First step, would be, would say let's store, let's sort, recursively the first half of the numbers: fourteen, five, eight, and ten. Again it's recursively, so our assumption is that when we are trying to sort it, it would just do the job basically sort the elements so it would reorder the elements in a sorted order. So, first step would reorder the first half of the numbers to be sorted; kind of a magic, but okay, that's one part of the job. Step two, you can probably guess would say sorts recursively the second half of the numbers, in this case: thirteen, one, eighteen, and

three, should be sorted in an increasing order. That would reorder them to be: one, three, thirteen, eighteen. Now that we have two halves that each one is sorted on its own, we would need to merge these two halves into one sorted sequence. So step three would say you merge these two halves together into one sorted sequence. So after doing step three, we would have the numbers all sorted together: one, three, five, eight, ten, thirteen, fourteen, eighteen.

Let's do it one more time. Step one: sort the first half of the numbers. Step two: sort the second half of the numbers. And step three: combine these two sorted halves into one sorted sequence. Cool; not very intuitive. But yeah, actually that that's a recursive algorithm. We'll implement it first and after that we'll try to trace the execution and see how the recursion here unfolds into sorting the entire array. Let's go ahead and implement.

Implementation 3.2

Let's implement this sorting algorithm, and it's a recursive algorithm, so let's do it correctly. So we'll write a function merge-sort, it's supposed to rearrange the elements so is doesn't need to return any value since it's a void function. And the parameters are obviously the array and basically something to tell the range of elements. So we can do it in two ways, either pass the starting address of our current part of the array we're trying to sort and the logical size, or we can always pass the starting location of the entire collection. And in order to indicate the range we're trying to sort, we'll have two indexes, low and high, that kind of indicate the range. Something similar to what we've done in the find index of min at selection-sort, where we had the starting position sticked and the low and high indicated where we we're searching the index of min, same thing here. We want to sort only in the range from low to high but it's located in a more bigger sequence starting at address ARR, so merge sort would get the starting position of the entire sequence and low and high to indicate the current range we're trying to sort.

Okay, so that the prototype of the merge-sort algorithm. And then since it's a recursive algorithm, we need to start with the base case. So if we're trying to sort the smallest sequence possible, basically the sequence of a single element or in other words if low and high are equal to one another. In this case, it is already sorted so we don't need to do anything, just return an empty return statement with no value because it's void we don't need to return any value, just break out of the function. Otherwise, if it is not a single element, a collection, we have a real range of elements we want to sort. Let's do it with the three step algorithm we've just described.

So we have our array, right. We have low and the high, indicating the range of elements we want to we want to sort. And then first step is basically for the first half of the elements. So we would need to figure out where this first half starts, yeah we know what starts at low, but where it ends. So we would need some kind of an index mid; let's define this mid. And just as we had in the binary search algorithm in our previous model, mid would be the average between the low and the high, basically low plus high div two. Now that we have mid, let's sort the first half, right. Let's make it in increasing order, so let's call merge-sort; it's a recursive algorithm so merge-sort would do the job, basically sort the first half of the numbers if we call it correctly. So in order to indicate the range of the first half, the starting address would be the same array, but the low and the high, in this case would be low and mid. We want to sort the elements starting at low ending at mid. After we've done that, so this line here sorts the entire first half of the elements. Then we need to sort the second half of the elements. Once again, let's call merge-sort in order to recursively sort. And then, let's pass the second half as a parameter.

So that, in order to create that half we need to know where it starts; so it starts one after where the first act ended at mid plus one and where it ends, obviously at high. So let's call merge sort with the same starting address ARR but the range of elements we want to sort now is starting at mid plus one ending at high. So after these two calls, we have the first half of the number sorted and the second half of the number sorted, we now just need to merge them together into one whole sequence.

For that we better create unique function, a designated function, to do that job. We'll call it merge; and merge should take these two halves and combine them together. So the parameters for merge would be the starting ARR, and some values to indicate where the first half starts, where the first have ends, where the second half starts and where the second half ends. So we would have the low left and the high left, to indicate where the first or the low left index is and where the index of the high element of the left side. And instead of having low right and high right to indicate where the right side starts and ends, we don't need both of them because the right side starts right after the left side ends, right. It should start one after high left. So, we don't need to pass that as a parameter; let's only have the high right. So we have where the left side starts where the left side ends, and where the right side ends, obviously we know where the right start starts.

So the prototype would be: ARR low left, high left, high right. And it's a void function, it rearranges the elements, it doesn't need to return any value. So assuming we have that, we'll make and implement this function in a few minutes, assuming we have the merge after calling the two recursive calls; after having the first to have sorted, we just need to merge them by calling the merge function that just passed the right values for the parameters of merge. So, let's call merge with our ARR low and mid for the bounds of the left side, low is the low left, mid is the high left, and mid plus one would be the low right, but obviously high that's the high right. So, this call basically merges these two halves together and after sorting the first half, sorting the second half, merging them, we have one long sequence from low to high that is sorted. That's basically the merge-sort algorithm.

Merge Sort 3.3

What we need to do now is basically implement the merge algorithm. Let me demonstrate how we should merge two sorted sequence into one big sorted sequence, and after we do that let's go to the computer and just implement it all. But let me first show you the idea so it would be easier for us.
So, assuming we have two sorted sequences: one, three, six, ten, and four, seven, eight, thirteen, fifteen, twenty, so two sorted sequences. We want to merge them together into one big sequence, to a new big sequence. We can do it in a lot of ways but in order to take advantage of the fact that these two sequences are already ordered in an increasing order, already sorted, we can do something kind of cool.

If you think about it, obviously the first element can't be, I don't know six or ten or thirteen or twenty one, it could either be the first element in the first one or the first element in the second one. So basically we should choose one of these two to come first in our resulted emerged sequence. So we have two candidates for the first element, in this case when we are comparing one and four we would obviously want take one to be the first element and then we can pick another candidate to be the next element. It would either be three or four. Now that we have taken care of the one, we have to pick between three and four to be the next element. In this case it would obviously be three. Now we have two candidates for the next element, either six or four, right. Obviously it can't be ones that are greater than six and ones that are greater than four before we put the six and four in their sorted merged output.

So then we'll obviously pick four and then we'll have to pick between six and seven; we'll choose six. And then we'd have to pick between seven and ten, and we pick seven. And then we'll have to pick between eight and ten, and we'll pick eight. And then we need to pick between thirteen and ten, obviously we'll take ten. And then basically we're done, adding all the elements from the first array but then the second array still have a tail of elements. So let's just take all this tail and copy it to the remaining positions in our result merged array. And that basically completes the merging of these two sorted sequences into one big sorted sequence. So let's take the merge-sort algorithm, the merge idea, implement them all on our computer and make sure it works. After that, we'll also trace the execution of these runs.

Merge Sort Implementation 3.4

So let's execute and implement the merge-sort algorithm. So I have a min here that we have an array of eight elements, obviously not sorted: fourteen, five, eight, and so on. I've declared ARR size to be eight. I did all hardcoded but you can see how we can make it in a general form. And then we're calling merge-sort, in order to sort this array. Initially, it will pass ARR in the entire range of elements as our low and high; basically low is zero and high would be the last index ARR minus one and then we're calling print array in order to print the array after merge-sort basically sorted the elements. Merge-sort is a very simple function that just iterates over the elements, prints them with a space and a new line at the end; so just prints the array one by one, element by element. And merge sort as we have seen before is a recursive algorithm it knows i don't equal anything otherwise we set the min index to be low plus high div two, we call him merge-sort with the first half from low to min index. And then we call merge-sort just for the second half from an index plus one to high and then we merge these two halves together, basically having the first half from low to mid-ind and the second half ending at high. We now only have to implement the merge algorithm. We've described before but let's implement it in C++.

So we have the prototype; let's take it over here and implement it. Okay. So first thing we have the low left, high left, and only right, so let's declare low right variable, just so we can set it to be high left plus one, right. That's where the right side starts: one element after the left side ends, right. So now we have low left, high left, low right, and high right. Another thing, let's also figure out how many elements we're trying to merge together so let's have a size variable. Let's set size to be the high right minus low left; that's basically the entire range we're sorting, we're merging plus one, so that's the size here. And then we need to create an array for the result of the merge sequence; so let's have pointer int star merged array. And let's allocate memory for that array: merge ARR equals new int of size elements. Again, we cannot do the merge in place because we'd be stepping on values that we're not necessarily done working on. You can try thinking why we can't implement this merge in place, basically on the same array we are reading, we have the sequence on, so we're using an additional array. And again, since we don't know how many elements we're going to have in this array; it would kind of be a static array, has to be in a dynamic array, that's why we're using the memory allocation here to allocate a new array of size elements.

Now that we have this array and we have the input arrays, the two halves, we can start merging them into the merged array. For that as we've seen we would need three indexes: one for the current element we're looking in the first half, then for the first element we're looking in the second half, and then the index where we are currently writing our result. So, let's declare these three indexes here: we'll have the index of the right side, the index of the left side, and let's also have the index of the results. So I'll have iRight, iLeft and iRes. Let's start with iLeft; let's set iLeft to be the low left, right; that's where the left side starts. iRight would be where the right side, starts basically high left plus one or we have low right

that we created. So we have iLeft and iRight set to the beginning of the left and right side. iRes would be set to zero, right, because that's where we want to start writing the result into the merged array. And then we should over and over repeatedly choose the element, the smaller element, between the one at the iLeft position and the one in the iRight position; so we'll use a while loop here, again, let's keep the Boolean conditions for a later stage. And then each iteration, let's pick one from iLeft and iRight, so let's ask: if the element at the iLeft position is less than the element at the iRight position that means that this element should be taken, right, there the iLeft one is smaller so it should become first. So, let's put at merged array at the current position, at iRes, let's put there the element from iLeft, this element here, right. So we are writing to the result array the element that is currently smaller between iLeft and iRight, the left one in this case.

Then we need to increment iLeft and also obviously increment iRes, since we've written to that position. Otherwise, if basically ARR iLeft is not less than the iRight one, we should take one from iRight. So let's put it at the merged array at the iRes position, in this case the array element from iRight, and now we should increment iRight in addition to incrementing iRes, right. So each iteration we are choosing one from iLeft or iRight to write to the merged array, depending on which one is smaller; either rewriting iLeft incrementing iLeft by one, or writing the iRight elements to the array and incrementing iRight. We do that over and over and over, as long as we have two candidates to choose from. Eventually one of them would go over the entire range of the array and we would have to take the tail of the other array.; so we can keep on going while they are still validly inside the range of their halves. So basically let's keep on going while the current index in the left is still less or equal to the high left, right; that means that the letter iLeft is inside the left half. And we also need the same for the right side, so the iRight is less or equal to the higher end of the right side, high right, right.

So while they are both inside their arrays, we want to choose one of them. Eventually one of the halves, we would end our work on it and then we would need to take the tail from the other parts. So we could either use an IF statement here to see which one ended and copy the rest, or we can just use a while, actually two while loops here that… So we'll have a while here and a while loop just after that. This one would take care of the tail we have in the first array, and this one would take care of the tail we have in the second array, we are certain that after the first while loop, after this loop, there would be only one tail so currently only one of these two whiles would be executed. If there is a tail here it would be copied and then this while loop won't even execute once or if there is no tail here, so this while won't be executed and this while would copy the data from the second half. So, let's take care of the first array first half. So while, again, iLeft it is less or equal to high left; that's basically saying there is a tail in the left side that should be copied. In this case, we should do what we've done here: basically taking two of the merged array at the iRes position, the element at the iLeft position, right, incrementing iLeft and iRes. So, that takes care of the tail left in the first half in the left half of our array. And if there is no left half then this while won't be executed at all right. If iLeft is not less or equal to high left, this body won't be executed. And same thing for the right side, so if iRight is less or equal to high right then again the merged array at our current position would just be the element from iRight and then let's increment this value and iRes. So now that's basically it; now we've picked the right candidates between iLeft and iRight over and over and over, when one of them ended we just copied the remaining tail to the merger array. After these three whiles, we are definite that all the elements are at the merged array.

Since we said that the merge function originally should reorder of the elements in the array from lower left to high right, it's not good enough for us to have the merge sequence in a totally new array, right. So we would need to copy the elements from the merged array back to our array. So for that we would have a for statement and let's do something like, we would need to copy from the merged array to the

original array and the merged array basically ranges from zero to the size, right size minus one, where the array we want to put the elements not from the zero index but from the low left index. So we would need to set two set of indexes here; we would need i for the merged array and let's do ARR that would be low left for the second array. Okay, something like that. Let's just declare i and iARR, so we have these two indexes. And in the second you'll help me figure out what we want to do here, but basically the body would be copy to ARR at iARR position, the element from the merged array at i index.
So we'll have both indexes increasing or getting bigger simultaneously, so we need i plus plus and i ARR plus plus. But they start at different positions, so we want i to be an index in the merged array starting at zero and want iARR to be an index if they are are a starting at low left. So we just set them initially with different values, each time we'll increment them, we just need to do it the right amount of times; we can either use our iARR to control it or i to control it. Let's just do less than size; size, if you recall, that's number of elements we have all together that we want to merge or in other words the size of the merged array. So we can go from zero to size, each time incrementing by one copying this element to the corresponding position in ARR, basically the iARR position. After we've done doing that copy, basically we're done, right.

So we have the merged sequence in our ARR array, what we need to do now is just delete the memory, the location we had for the merged array. So for that let's use the deleted operator, which is an erase so I'm using empty brackets, that seems to be this function here.

And now that we have merge-sort and everything, let's just try to execute it; see that it really sorts.
We have the sequence sorted increasing order: one, three, five, eight, ten, thirteen, fourteen, and eighteen, which is exactly the sorted sequence in the array. So it seems to be working fine.

So merged is a fun function; we had a few details when we worked on it. It's a cool function but I think more interesting, more surprising here, is the merge-sort algorithm, where we surprisingly called merge-sort two times assuming it sorts the first two halves, and apparently it does. All we have left to do is just merge them together and we've implemented the merge function, so that it is kind of surprising, you can think. But let's go ahead and trace the execution and convince yourself how this basically works.

Tracing Merge Sort Execution 3.5

Let's trace the execution of the merge-sort algorithm. So at first I want to make sure that all of you get inductive idea here, where we have three steps first sorting the first and then sort of the second half and then merging them together and logically understand why these three steps, in that order, basically soars the entire sequence. But I think it would be even more convincing if in addition to that we would also trace one execution from start to end. My suggestion is don't do it too many times because understanding the inductive idea for recursive algorithm, that's the important thing. But let's do it for merge-sort.

So, we have eight elements sequence: fourteen, five, eight, ten, thirteen, and so on, which we want to sort. So we are basically sorting the first half and the second half and then, since it's a recursive call each one of them is going to sort its first half and second half. So we have calls for two element array and each one of them also would call two sorting halves of one element and that would be the base condition. But that's not the order that these calls are called; let's try tracing the merge-sort algorithm now in order. Okay.

So we're first calling the original eight element array. It, as step one, calls to sort the first half of the elements; the first four elements, right. And then when this function executes, it's also a recursive call. That's not the base condition so we also called to sort the first half of the elements; basically sorting fourteen and five, and fourteen and five is also not the base case so it also calls to sort the first half of the elements, which is fourteen. Fourteen: that's a single element array that is already sorted, so nothing to do here just return. And then fourteen five calls to sort the second half, which is five, so now fourteen and five after having two halves that are sorted it should merge them together to a sorted sequence. Basically reordering them to be five fourteen; that's the merge. And then the four elements: fourteen, five, eight, ten, now need to call to sort the second half, basically eight and ten. Eight and ten, that's not the base, so it calls to sort the first half of eight which is sorted. and then it calls to store the second half of ten, which is sorted and then it merges them together. In this case no reordering is needed the assorted sequence is eight and ten.

Now the four element array has two halves that are sorted by fourteen and a ten. We need to merge them together in order to get a four element sorted sequence: five, eight, ten, fourteen. So that's basically the end of the first step in the original call; merge sort of the first half it would recursively merge-sort that half. And eventually would have the elements, five, eight, ten, fourteen, in the sorted order.

And now would come the second step, basically sorting the second half of: thirteen, one, eighteen, and three. That's not the base case so we would start by sorting the first half of that, which is thirteen and one. That's not the base case so it starts by sorting the first half of that, thirteen which is sorted, and then sorting the second half of that, one which is also sorted. And then we need to merge them into: one, thirteen. And now we need to sort the second half of the four element, which is eighteen and three. Not base case, so let's recursively sort the first half: eighteen which is sorted. Let’s recursively sort the second half: three which is sorted. Let's merge them together to three, eighteen. Now we have two sorted halves of the four element array.

We need to merge them together into one sequence of four elements that are sorted: one, three, thirteen, eighteen. That’s the end of the second step of the original sort of eight elements. We've sorted the first half recursively; we've sorted the second half recursively. And now comes step three, where we have the two halves sorted that are needed to be merged together. And that’s step three of the merge function, basically merging them into one sorted sequence: one, three, five, eight, ten and so on. It's a bit crazy way to trace this execution, but that's how basically it was. I think it's easier and more understandable to think of it inductively. Basically, thinking step one would do the job basically sort the first half, step two would do the job basically sort the second half, and then we just need to merge them together to get the entire job done, basically sort the entire sequence.

Runtime Analysis 3.6

Now that we know how merge-sort works; now that we've implemented merge-sort, we've even traced the execution of merge-sort. Let’s try to analyze the running time of merge-sort and try to compare it to the selection-sort, or the most basic sorting algorithm, that by the way executes in a quadratic time. I hope merge-sort will give us a better result than quadratic; otherwise it wasn’t worth all the effort we gave to that. It's not as easy as it sounds to analyze this kind of an algorithm because it's not as like we had an iterative process where we just summed up the number of operations we're doing each iteration so it was all like in front of our eyes and we just needed to figure out what's the body's cost and just add more and more values to figure out the total cost of the iterative process. In the case of a recursive call,

a lot of operations kind of sneak in between the calls and we need to better model the number of operations that are done during the process of executing a recursive algorithm; very common model to do that is called a recursive tree model and it works something like that. We'll try to basically show all the recursive calls and their costs, all together, so it starts like that.

We first have a call of an array of size N. This call has two or calls two times to a function of size N by two, each one of them calls two times to a function of size N by four, each one of them calls to N by 8 and so on; they kind of keep on splitting, here we have a lot of calls to a single element array of size one. So this is the entire set of recursive calls and their sizes, but it's not that each call here is constant, there are calls that cost us constant amount of time but there are calls that are much more expensive. So let's try at the side of each call, each circle here, write the cost of that call: how much operations do we need to pay for that call.

So the first one in addition to the recursive calls, we need to merge to N by two element arrays into one N element array. The merge function, I don't think we've mentioned it explicitly but it's a linear function; its cost is basically the sum of the sizes of the arrays we're trying to merge. So if we are merging two N by two arrays, the cost of merge would be N, right, the number of elements we are writing basically to the merge ARR, basically copying to our array. So the cost here of this call, in addition to the recursive calls, is another N. And each one of the N by two calls costs the merge basically N be two, so the first one costs N by two and the second course costs N by two. And each one of the N by four costs N by four, right? Basically the cost is the cost of merge and that's basically the size of the elements that we're trying to sort, that's the number of elements who are basically also trying to merge. And so on and so on til the base case; each one costs one. So in order to figure out the total cost of this execution, we need to sum up all the reds here, all the extra costs. And if we know how much all of the merge function cost together, that would be the total cost of merge-sort. It's kind of difficult to sum up all the red here but then a nice observation says that it would be easier for us to sum the reds in each level first and then if we know how much a level cost, we would be able to some all the levels sums up and that would be the total cost. It works something like that.

So let's try to relate the level number to the cost, the total cost of that level. In order to do that we would need two additional values that we want to relate here: that the number of calls in the level, number of circles basically, and the cost of each call in the level, a single red cost. Let's start here; so in level number, the first level, I prefer to name it level to start numbering here at zero. So at level number zero, the number of calls is one, right. There is one node at the first level. The cost of this call is N, right, and then the total cost of the first level is basically N times one which is basically N. So the first level costs N, we can see that the first level costs N. The second level, level number one has two calls, right. There are two nodes here; each one costs N by two, therefore the total cost of the level is two times N by two which is also N. Interesting. Third level, level number two, four calls each one costs N by four, once again not surprising four times N by four that would also be N. Next level would have eight nodes, each one would cost N by eight, right, and that would also sum up to N. I have a feeling that all, each one of the levels here with sum up, would total up to N; let's try to justify it.

So for example level number K: how many calls are we expecting there? You can see the relation here between the level number and the number of calls; it's two raised to the power of the lever number, right. Eight is two to the power of three. Four is two to the power of two. Two is two to the power of one. One is two to the power of zero. Therefore, the number of calls in level number K would be two to the power of K and the cost of each call is N over that. So it would be N over two to the power of K and then the total cost would be two to the power of K times N over two to the power of K; you can see that

that is also N. So I'm kind of convinced now that not only the first four levels sum up to N, but all levels with this behaving like that also sums up to N. The big question is how many levels we have here because if we know how many levels, let's say we have thirteen levels then it's going to be thirteen times N obviously the number of levels depends on N, it's related to N, so can be just thirteen. So the big question mark is: what's the level number of the last level. Once we figure that out, we know how many times we need to multiply by N, right. Let's do that.

So the most obvious thing we know about the last level is that the cost of each one is one, right, because size is one and we know that the cost here should be one. If you look back at the calculations we've done in Binary search, we've already seen that the value of K where N over two to the K equals one is when K is Log of N. That means that the level number that gives us cost of one should be Log two of N which means that the number of calls in that level is two to the power of Log two of N. If you're familiar with the logarithmic rules, you know that two raised to the power of Log two of N basically equals N, so we have N calls, each one costs one that also adds up to N. Not surprisingly we've seen that; so basically we know that we have N in each level, Log N times. So the total, T of N here, is N times Log N or Log N times N. So the running time is paid theta of N Log N, which is better than N squared, N times N. We know that N times Log of N is significantly better than N times N, so N Log N merge-sort is a much preferred algorithm, sorting algorithm than for example selection-sort.