Lecture Transcript
Merge Sort

Hello and welcome to this next lecture on 'Data Structures and Algorithms'. In this session we will introduce a new sorting algorithm, a sorting algorithm based on a paradigm of divide and conquer. In particular, we'll discuss one of the simplest algorithms for sorting based on divide and conquer called the merge sort. The idea of divide and conquer approach is to sorting or any other problem is very clearly spelled out in the name itself, the first phase is 'divide' which means divide the problem into a number of sub problems. In the case of sorting this will mean that you split the array S(sub)1 to S(sub)n into two parts. The next step is 'conquer', which means be able to solve these two sub parts and this is often done by solving them recursively. So the second step is conquer by solving them recursively, which means you will again split each of the parts into smaller parts and so on. Finally, having gone down all the way in your divide and conquer you need to go back all the way up and combine. Combine is the path upwards, combining the solutions to the sub problems and divide is a path downwards and conquer basically does division as well as combination. So in the case of divide and conquer or sorting is also called merge sort, the simplest implementation is the merge sort algorithm. You divide the n-element sequence into two sub-sequences of n/2 elements each.

Now, it isn't really necessary that the two parts be of the same size, but the simplest implementation says that we keep them of same size. Conquer is to solve each of the sub-problems recursively, so this will mean sort the two sub sequences recursively using merge sort. So merge sort, divide and conquer based sort has to be invoked on each part. Finally, combine is to go all way up by merging the two sorted sub-sequences to produce a sorted answer. So remember that combine will be invoked recursively in recursive calls to conquer. So here is an example, we have an array [7, 2, 9, 4] our first step is to divide. So we have divided into two parts of equal size. We want to determine the sorted list on the right-hand side, it's empty as of now, but we'll populate as we go down. So the first step is divide. Now, what you do with the left half is all left to conquer. So in fact, this left call is nothing but conquer. In fact you'll do two conquers. This is divide one; conquer 1 you'll also do conquer 2. The division is only one but there are two conquers.

Now within each of the conquers we'll have further invocations of divide and conquer. What you will do after all of these conquers, or within each iteration of divide and conquer is get together the outcomes of the conquers and combine. So the outcomes will be combined here. Let's see this further in action, so the conquer on the left-hand side has further invoked the divide. This is a conquer and I call it conquer 1 and conquer 1 has a divide 1,1 dividing 7, 2 into two array size 1 each, so there is divide. What follows next is combine. So, you're going to take the outcomes of each of these and combine them and that you see in this arrow ahead going up, this is a combine operation. What you do next is how I know that the combine operation requires you to place the

elements in the right order and you merge these elements from the two sorted lists. Similarly, on the right-hand side, we would have got another sorted list. Imagine right-hand side giving you a 4, 9, so 9 followed by 4 goes to 4, 9 again this needs to be communicated back and this happens through a combine. And, what combine will entail is interleaving elements between 2, 7 and 4, 9 which means you'll have 2 followed by 4 followed by 7 followed by 9. You aren't really bothered with comparisons within each group, but you are concerned about comparisons between selected elements in the left and right sub-groups respectively.

So, this is complete execution of merge sort for this example, if you get a sorted list. So you can visualize as any merge sort instance through a complete binary tree and the number of communications in terms of divide and combine are of the order of the height of this tree. So here is the merge sort algorithm. What we have done is we have invoked merge sort on each of the partitions. So, we have partitioned S into two, S(sub)1 and S(sub)2. Invoked merge sort on S(sub)1 and merge sort on S(sub)2. So the first step of partitioning is a divide step. The next two steps are the conquer steps. And finally, you have a merge step as a function itself says merge or combine. In fact, it is this merge that gives merge sort its name. We have called this an in-place merge sort, but I would like to highlight an important point. You could store this partition in an auxiliary array, a separate array and get them back into the original array. Merging will mean getting them back to the original array. It turns out that an in-place merge sort is somewhat expensive. We will avoid too much of discussion of complexity analysis, but an in-place merge sort is basically attained by plotting part of the array while using the rest as working area for merging.

So the idea is sort part of array, part of S and use rest as working area for merging. And, this actually leads to slightly bloated complexity which is order n squared. However, if you made use of an auxiliary data structure, merge sort using additional space. And now, how much is this additional space? Well, it turns out that you'll need at least order n-space. We will show that this leads to a better upper bound which is O(nlog n). So somehow in-place merge sort, we haven't accounted for all the time that is required to copy elements and bring them back to where they belong and so on. Whereas an additional space will always ensure that there's some place to push on these elements before the a brought back. The merge subroutine over two sequences S(sub)1 and S(sub)2 and as a pointed out we are making the assumption that additional space is made available. This additional space could be a part of the original array itself as was an in-place merge sort but that is generally not a good choice as far as time complexity is a concerned. But irrespective of that if we use two sorted sequences S(sub)1 and S(sub)2 with n/2 element each and want to merge them into an S what you need to do is look for the first element at the head of S(sub)1 compare that with the first element of S(sub)2 and append that first element which is smaller the S(sub)1.firstElement append that to S. So what are we doing here?

So if S(sub)1 is 1 2 4 and S(sub)2 is 3 6 8 so this first element is going to initially pick 1 append that to S. This is my S. Then compare the next two first elements so of course in this process you also remove the first element of S(sub)1. Compare the first two elements of S(sub)1 and S(sub)2 respectively and you find that amongst 2 and 3. 2 is the smallest again get rid of it. Compare 4 and 3 and you'll find, well 3 is the smallest. Get rid of it and then 4 and 6. 4 goes and it will be 6 and then 8. So basically, leveraging the fact that both S(sub)1 and S(sub)2 are sorted and avoding comparison within S(sub)1 or within S(sub)2 at distracting comparison to set elements between S(sub)1 and S(sub)2. A complexity of this merge is basically order n it is just

requires one scan of both S(sub)1and S(sub)2. The other remaining part is basically to take care of remaining elements. So it's possible that S(sub)1 has 1 2 4. S(sub)2 has 3 6 8 now once you have conveniently inserted 1 2 3 4 whatever remains in S(sub)2 needs to be also inserted. So you treat on this at the end and this is what we've shown here. Inserting remaining portion of S(sub)1 or S(sub)2 as a case maybe. So, let's analyze Merge Sort. The construction we are dealing with a complete binary tree of divide, conquer and combine operations.

We'll look at the number of operations and the number of sequences which need to undergo those operations at each level of this tree. So, we expect that initially at the first step, step 0, the number of sequences is just 1 and you are dealing with a sequence of size n. At the next level, you are looking at two sequences, each of size n/2. After all, all the elements get divided at each level. So, you need to make sure that number of sequences times the size must always equal n. So, going down at i'th level can easily fill in that you have 2 raise to 'i' sequences because that's the number of nodes at level or depth 'i' in a complete binary tree. The size will, therefore, be n divided by number of such nodes or number of sequences. So, what we see here is that each recursive call divides the task into two tasks of half the original size. Therefore, at any height 'i', the work done is 2 raised to 'i' (number of sequences) times n by 2 raised to 'i'. So, it's order 'n' at each level. Now, how many such levels exist? Well, the number of levels is the height of the tree, which is order log n. So, the total running time is, therefore, O(n log n). Again, we have assumed that we have auxiliary space to store elements, as we divide and merge.

So, this is assuming auxiliary or additional storage space. This is illustrated. In the next session, we will deal with an interesting variant of divide and conquer based sort called Quick Sort. Though it is divide and conquer, an advantage of Quick Sort over Merge Sort will be that an in-place implementation of Quick Sort will be very natural. Natural, in-place implementation. However, as we will see there is a disadvantage. And the disadvantage will be that Merge Sort which guarantees n log n irrespective of the input, you won't have the same advantage in Quick Sort. Complexity depends on input and in particular, the worst case is order n squared, the same that is offered by insertion of Selection Sort.

Thank You.