Lecture Transcript
Comparison Based Sorting

Hi! I am Ganesh Ramakrishnan and I would like to welcome you to this course on 'Data Structures and Algorithms'. Today we will begin with our first lecture on algorithms. We'll begin with sorting. We'll begin with a paradigm for sorting which leverages comparison between elements. So comparison in the sense, you look at ith element of an array and compare that with the jth element of an array and perform operations based on the outcome of this comparison. We'll consider some abstract data types for sorting that build upon this comparison for sorting. In particular, we'll look at Selection, Insertion and Heap Sort based on different choices of the abstract data type, we'll have these different versions. We'll also consider the in place variants of these 3 sorting algorithms and perform analysis. We'll then discuss a slightly different paradigm for sorting which is a divide and conquer based approach. We'll discuss the 2 instances, Merge Sort and Quick Sort and also discuss challenges involved in coming up with an in place version of merge sort in particular. We'll also perform analysis, runtime analysis.

So the basic idea of comparison based sorting is take pairs of objects and perform operations based on the outcome of comparing those 2 elements in the pair and the result of each comparison is basically a yes or a no. It turns out that for algorithms that are based on comparison there is a lower bound in the sense that there is no way an algorithm can perform better than a particular value and that turns out to be n log n, where n is the length of the array being sorted. What is also interesting is that there are several algorithms whose upper bound or worst case performance happens to match the lower bound for comparison based sorting. So in one sense the lower bound has been achieved even in the worst case by several comparison based sorting algorithms. We'll discuss different implementations of the abstract data type in particular priority queue and we'll discuss how you could do away with the use of an explicit external abstract data type and embed the functionality of that data type within the array itself that's what gives you in place variant. In particular if the auxiliary memory that you consume through the abstract data type is limited and doesn't grow with the size of the input you could come up with the in place equivalent variant.

We also discuss an important tool for proving the correctness of an algorithm called the loop invariant and we'll discuss three aspects of loop invariants. Finally, we'll discuss the complexity of the algorithms and also present lower bounds on running time that apply to comparison based sorting algorithms in general. A priority queue is a very natural choice of an abstract data type, the priority queue let's you specify the highest priority element and both insertion and selection from the data type, abstract data type could be based on priority. So different implementations of PQ lead to different sorting algorithms. So the basic idea of sorting using priority queue is that an input sequence S is iterated upon, and for every element that you iterate upon within S, every element e, place that into this auxiliary data structure P as I pointed out this is done

for each element e of S. Thereafter you pick elements from P in an intelligent manner. So, would intelligently enumerate elements from P and put them back into S. Of course you want the intelligent enumeration and the insertion to be both as efficient as possible.

So let's discuss some specific implementations. So in the priority queue is an unsorted list or an unsorted array. The algorithm for sorting is called selection sort. The choice of list as against the choice of an array doesn't make any significant difference. You could also use a sorted list based implementation of the priority queue in such an instance, the sorting algorithm is called insertion sort. You could replace the list with an array you'll have some slight efficiency improvement but the order of complexity doesn't improve as such. A more interesting implementation of the priority queue is as a heap and that gives you Heap Sort. Let's start with selection sort an unsorted list. What is the complexity? So insertion could happen either at the beginning or the end of the list P. So neither case the complexity is order one. It really depends on how the list is stored from the beginning to the end or the other way around. In either case, finding the index of min would require you to scan the list.

This is basically an order n time n is the size of the list as you're scanning. Delete requires just deleting that specific element, element at position m, in this process however you'll need to shift back elements and make the list compact. So the compaction time would depend on the size of the list. So initially the compaction time is just order 1 then it's order 2 and so on. So the overall time required is order n squared as I pointed out at every iteration scanning the list would require order n, where n is a size. Compaction in the worst case can be order n when you find the element to the other end of the list. Interestingly, the best and worst case running times for selection sort are both theta n squared. So recall that f(n) is order g(n), if for all n ¿= n(sub)0 f(n) ¡= M g(n) for some M and the relation between theta and O is as follows. f(n) is theta g(n) if g(n) serves as an upper bound for f(n) i.e. f(n) is order g(n) and in addition you also have that g(n) is order f(n). So which means f(n) for some constant m serves as an upper bound for g(n) and g(n) for some other constant serves an upper bound for f(n). So it turns out that theta is at the tight bound n square holds for selection sort. So, what is this indexOfMin? As I pointed out the indexOfMin requires to iterate over the element of P starting with particular index i to find the minimum element. Here we have defined indexOfMin more generally in terms of the starting offset i.

We will make uses of this particular subroutine in several cases. Now is there in place version of selection sort. It turns out that there does exist an array base in place selection sort. What is the idea here? The idea is once you find the minimum element of S starting at the particular position i which is the index of the min between position i and position n at the end of the array. You swap the elements at positions i and m respectively. The idea is to get the effect of finding the minimum element and storing back through a single swap operation. So you don't need to wait till you have emptied the array rather as you find the minimum element you know that position i needs to be updated and may consistent with sorting and that's what swap achieves. A new element that when into position m is not going to really affect the structure of the auxiliary data structure. As a result, this operation is not really going to hurt the unsorted list at all. An interesting point to note is the stopping criterion which is i = n - 1. So, it turns out that by the time you have exhausted finding minimum element between position i and the n. And a specially for i ranging from 1 to n-1.

So by the time i is n-1 it turns out that you already identified the nth element to be largest element anyways. So the answer to this question why does it suffice to execute until i = n - 1? The answer is that the nth element has already been unanimously found to be the largest element in the array. How do you prove the correctness of selection sort? So here is the claim at start of each iteration of the ith repeat loop the subarray S[1...s, i - 1] consists of the i -1 smallest elements of S in ascending order. So as I pointed out earlier instead of emptying the array all together all you doing is treating the first i - 1 elements of array S to be updated with respect to sorting and treating the rest of S as the abstract data type or the unsorted list that we deal with as we keep updating S. So the scan is over S[i ...n] whereas the insertion is on the array S[1....i - 1]. We need to discuss three properties of this loop invariant condition to be able to use this condition to prove correctness of the algorithm. The first is initialization. You need to prove that this condition of S[1 ..... i - 1] being sorted holds prior to the first iteration of the loop. You need to prove that after each iteration of finding the min and swapping this condition is maintained. And at termination invariant helps show that the algorithm is actually correct because at termination what you are referring to as the sorted subarray in the invariant conditions turns out to be the entire array itself.

So we will quickly show that all the three properties of initialization, maintenance and termination are satisfied for this loop invariant property. So how does it holds at the beginning? Well, it holds at the beginning because initially your i is 1 you are referring and therefore you are referring to an empty subarray. What happens at the next iteration? Is you find the minimum element of the remaining subarray and insert it that the minimum element at the beginning of this subarray through a swap. As a result, this loop invariant condition holds after the first iteration. In fact, what we have just proved is the maintenance property. You could use the same argument to prove that the loop invariant holds after the second iteration. Provided or assuming that the initialization has been already done which is the prefix subarray is already sorted. Simple argument is that every element of the prefix subarray must always be less than equal to every element of the remaining subarray. And therefore the new element that gets appended can only be greater than equal to elements in the existing subarray. And finally, termination what does this entail this loop invariant property entails that all the elements from 1 to n - 1 are sorted. And by virtue of the algorithm and swapping we know that the nth element must be greater than equal to every other element in S. So as a result, the termination condition which is S from 1to n - 1 is sorted, is sufficient to show the correctness of the selection sort algorithm. So, in our next session, we will discuss another implementation of the priority queue namely the sorted list. In fact, this gives us our next sorting algorithm called insertion sort.

Thank you!