Lecture Transcript
Heap Sort

Hi and welcome to this lecture on 'Data Structures and Algorithms'. In this session, we will discuss Heap Sort. What is a motivation for heap sort? Well so far, we have seen that the insertion sort algorithm and the selection sort algorithm, both give you order n squared performance. Question is could we do better? And, what is better? Well that's exactly, what we want to understand by looking at a possible lower for sorting especially based on comparison. Insertion and selection sort both invoked comparison of elements. And in fact, we discussed the analysis of insertion sort based on number of inversions which correspond to number of swaps. Now, let us basically comparison.

So, the key observation will be as follows: we will try and view each of these algorithms that you have discussed so far or any other algorithm that we might come up in the future based on comparison as a run of an algorithm on a large binary decision tree. So, here is what you are saying, you are saying that, you have some large binary decision tree and this is all for a given array. And each of these algorithms or each of the runs of these algorithms would correspond to a path in this decision tree. So, this might be the path of insertion sort whereas for selection sort you might have a different path. So, let's try and understand what this tree is? Each node in this tree is basically a comparison between pair of objects, is x ¿ y or y¿x? So, we will use the index i for a permutation and i(sub)1 will be an element of the ith permutation, x(sub)i(sub)1¡x(sub)i(sub)2 would be a kind of comparison you would make. So, result of each comparison is a yes or no and that corresponds to the branching.

So, the sorting algorithm is basically a series of comparisons to decide which permutation of the sequence S is a sorted permutation. So, what sits at the leaves of at each leaf of the permutation tree, what you have the permutation. The number of permutations is n factorial therefore we have n factorial leaves. So number of leaves is n factorial. Because a number of permutation happens to be n factorial. Now, what is the minimum height of such a tree? Well, we know that since this tree is binary actually it is balanced, it's a complete tree because every pair of elements when they are compared you will have two outcomes and there will be no comparison that leads to a dead child. So, you will have a complete balanced tree and the minimum height of such a tree is log n factorial. And, in case you haven't seen what log n factorial is, you could expand. So, what is n factorial? So, this can be viewed as summation over i equals 1 to n of log(i) and this we know is less than equal to summation over i of log(n) because log(i) is less than equal to n as long as i is less than equal to n and this is nothing but n log(n). So, this gives us order(n log n).

This is of course an upper bound, but we also can find a similarly a lower bound. What we could do is substitute one half of these numbers with n/2. So, we know that the upper half of

these are all going to be greater than equal to n/2. So I know that this summation is greater than equal to i equals n/2 to n of log of n/2. And, well if you expand this, this is nothing but Omega(n log n), the n/2 is log(n)-log(2) and that just gives you, n/2 log n - n/2 log(2). So that is Omega(n log n). So, this is what the tree of permutations looks like. Here sigma i stands for the ith permutation. So, each of these leaf notes is a permutation and altogether there are n factorials as a unique permutation. We have actually made an assumption that we are dealing with unique elements. It's possible that some of the numbers get repeated, the analysis is not very different in that case. As we discussed, the number of leaves is n factorial and this implies that min height of such a tree is Omega(n log n). Ok.

So, what we are saying is for any algorithm that leads to make this comparisons and reach one of the permutations for a given sequence S. To reach from S to the current permutation, let's say this is a correct permutation, sorted permutation. So, you cannot but avoid traversing a tree from the root to the leaf which means you will have to travel at least n log n. Now is there an algorithm that helps us achieve this lower bound? So, recall that a Min-heap Abstract Data Type stores the index and the value of the smallest element in the root and the smallest element is defined as that which has a lowest value. This must hold not only for the root but for every substructure. So, in general the value of the parent of i must be less than equal to the value of i which is all the other descendants. We see that 40 is less than 70 and 90 and so on. This is actually a Min-heap. And this Min-heap can be structured, can be constructed for a sequence S. This can be done in at most order(n log n) time and this invokes us sequence of heapification operations on every sub tree. We will recap this when we discuss sorting using the Min-heap.

So the idea behind Min-heap sort, it should be the following given the sequence S convert it into P by calling the heapify function on S. Once we have that, you could use P to recover S as follows. You can retrieve the smallest element of P and keep adding them to S. For all e = min(P) add e to S, so this you would do of an appending manner and then goal back to P and delete e from P. So that's what we have shown here. First you insert elements from S into P while maintaining the Min-heap nature of list P. So recap that this is nothing but invoking P the Min-heap as a priority queue abstract data type. Now for every insertion, you are going to traverse the height of the existing tree. So the initially, the tree is very small, so you traverse a tree of size one log one and then you have two elements log two and so on up to log n and this is order (n log n). Similarly, you delete element e from P, this you do by removing that min element and pushing it into the sequence S.

This is order n when you remove the first element and its log n - 1 and so on, again eventually its order (n log (n)). So, order (n log (n)) overall. Is it possible to avoid constructing an external data structure P and instead do in-place heap construction? So, before we do that we'll see what the array corresponding to a Min-heap looks like. So, the element 10 which is the smallest element is going to be stored at position one. The immediate children are stored at position 2 and 3. And thereafter we have the leaves stored at position 4 to 7. But, what exactly is a structure here. The leaves of a node with index i are at position 2i and 2i+1. This is a property that holds for every node and if you can easily verify this in this simple example. So the basic idea is as follows. The root of the entire tree will be at the first position. Position 1, its children will be at positions 2 and 3 and in general for an element with index i which is basically root of some sub tree.

Its immediate children will be at positions 2i and 2i+1. Of course for i to have children this will mean that all these roots of sub trees should lie within the first n/2 elements of the array, n/2 floor and that we can see here for seven elements the intermediate non leave nodes cover the first three elements of the array. So, to do in-place Min-heap sort here is what we will do. We realize that since the top of the list is always going to be the minimum element amongst all the elements at follow it. We will, we will grow the Min-heap as follows. So, we initially build a Min-heap convert the entire array into a Min-heap. So, what we'll have is the smallest element here and this will be the smallest. However, it is not necessary that the next element is the next smallest and so on. So, what we do is invoke min-heapify on a next node.

So, the next minimum node must be between positions 2 and 3, you do a comparison and thereafter push thus next smallest element to the second position. Now you may have to do little bit more work. Because, the tree that has resulted might not completely satisfy the Min-heap structure if you just treat the third node as a root node. In fact it may not be even a completely balanced tree. In general, the Min-heap tree would have a bunch of nodes and its complete except for the last layer. So except for the last layer the tree will be complete. So, therefore, you may have to restructure the node to make it complete. So that Min-Heapify should take care of. So let's discuss this in-place Min-Heap sorting in some more details. All this is happening within a single array, so formally a Min-Heap of n elements is a complete binary tree with all levels except the last one being full. The last level is filed from left to right as I pointed out. The value of an item at parent is less than equal to the values of children and the minimum element will be at the root. So the array set up, the child of node k will one be at 2k and the other child will get 2k+1. Of course, this is provided the later two are less than equal to n, if not then one of them may be empty and that's why it's possible that the last level is not necessarily completely filled up.

So the idea of the heap sort is use the left portion of 'S' up to index i -1 to contain element sorted so far. So this is the element sorted so far and the right portion stores remaining elements in a heapified form. And here is the algorithm more formally given an input sequence S, the first step is to convert S into a Min-Heap and you would like to do this in-place, of course, you don't want to use an auxiliary data structure, this can be done. So the idea is to iterate from the left to the right. We already expect after the Build-Min-Heap operation we expect the first element of this array to be the minimum element. So, what you need to do subsequently is mean Min-Heapify, the remaining array. We are not building an Min-Heap from scratch because we know that to a large extend the heap structure is satisfied, it's only about where to push this element, new element at the top and retain the heap structure at each of its children and correct at the child at which some violation might take place. So this is the Min-Heapify which needs to be invoked on this remaining sub-array. This you keep doing till you hit the length of the array S. What is a Min-Heapify subroutine? So if you are at a position "i' consider its children who are at position l = 2i and r = 2i+1. So if the array element at 'l' happens to be less than the value of the array limit at 'i' then you know that the min is 'l'. In which case you'll need to do some re-ordering or re-structuring.

So you basically do some book-keeping and keep track of which of these is the min. If 'i' happens to be the min there is not much to do, in fact, you could stop. Likewise, for 'r' you make a comparison with 'i'. You don't need to compare between 'l' and 'r' because each of them only needs to be ensured as a Min-Heap structure. So once you have found the minimum element,

check if that min element is 'i' if it is not 'i' then what you need to do is replace the 'l' within 'i' or 'r' within 'i' corresponding to which of them turned out to be minimum and that we are keeping track of through min. So, S(i) swap with the S(min) which are of these is and thereafter you again call Min-Heapify S(min). There is no need to Min-Heapify other sub-tree, the sub-tree that did not correspond to the min element and the reason for this is we already had a Min-Heap before 'i' got introduced. So the whole purpose here is to ensure that insertion of a new element will honor the heap structure. The Build-Min-Heap Subroutine builds this entire initial heap from the array 'S' and again this happens in please manner. So you begin with the first non-leaf node from the right-hand side. So, S =[1 to [n/2] to n]. So we know that this is the first non-leaf node or internal node from the right-hand side. So you scan this array from n/2 floor from this index and till you find the left-hand wall end of the array, you invoke Min-Heapify on the sub-array rooted at that element.

I am spanning until the end of the array. So for an arbitrary index 'i', what this will mean is invoking min-heapify with respect to this sub-array. What are we doing in this process? Well, first of all, we don't really care for the last n/2 nodes because they are the leaves and all we want to make sure is every sub-tree has optimal min-heap kind of structure, the desired min-heap structure and moving left is basically like having a new element added and you need to make sure that, that is element is compatible with the existing min-heap structure. So, you can just invoke min-heap and have 'i' trickle down into that branch which gets affected because of 'i'. Now, this complexity as follows, the min-heapify needs to traverse the height of the tree. So the actual expression for this is summation i = 1/n log i, where log i is the cos invoked for the i'th invocation. And we know that this is at most n log n. Question is, can we get a better bound? Is there a better upper bound? The answer is that log i being an upper bounded by log n is too loose. We could actually tighten it and we do it as follows. We change the summation from summation over the nodes i = 1 to n to summation on height.

Now height can go from 1 to log of n, and what we find is this height can repeat for multiple examples. In fact, for tree of height n, for a tree of height h, at most n/2 raised to h + 1 nodes exist. Right, This we know from the fact that an n element heap has height log n. So, h = log n work backwards and find the number of nodes, the maximum number of nodes in a tree of height h. So, what we can do is, add this scaling factor n/ 2 raised to h + 1. But this is now applied to a tree of height h. So you will, of course, have the original linear time complexity in the height. So we just rewritten the expression here which I am marking in yellow slightly differently in terms of height. And it turns out that with some amount of manipulation, this is nothing but an order of (n). So, in particular, we have made use of a very important equality which is summation over k = 0 to infinity of k times x raised to k is x divided by 1 by x squared and in particular, we have set x to the value half.

So you can convince yourself that by setting x to the value half I am using n here while, n is total number of nodes, you can actually get the same expression, so this is order n. So what is interesting is we have found a more efficient way of building a Min-heap. So that is order n, however, our continuous invocation of min-heapify in heap sort will still constrain our upper bound to be n log n. So recall, that our min-heapify subroutine was log n and this will be invoked n times and that makes a complexity n log n. So we have found one algorithm which runs in O(n log n) and that is a heap sort. Question is, are there other algorithms? Is this unique? Where

it turns out there are a couple of other algorithms and very important framework for discussing some such algorithms is the merge sort framework that will be in the next session.

Thank you.