Lecture Transcript
Insertion Sort

Hello, I would like to welcome you. In today's class, we'll discuss 'Insertion Sort, which is basically sorting, using a sorted list as an abstract data type or a sorted list implementation of the priority queue. So the idea here is, given an input sequence S, let's say [4 5 1 6 2]. You create an auxiliary data structure priority queue P which is initially empty and the idea is to scan the list S and insert elements in the order in which you scan into P. So, this makes P=[4] and you add the next element 5, you ensure that P has an increasing order of elements. So which means you're going to append 5 to the list, we'll get P=4 and then 5 and then for the next element, obviously you will need to prepend P with 1 that's 1 4 5 and so on. So as you can see, the scanning of the list will be order n but the insertion into P can in the worst case take n squared time sum across all the insertions. So for the first insertion, you will have incurred a complexity of 1, for the second one which is 5 you need to scan two. In fact, you will have to incur 1 2 3 up to n irrespective of how well arranged the elements in S are. So this is the order n squared insertion sort. As far as deletion is concerned, all you need to do is pick elements from P, starting with the first element and pick the minimum element and insert them into S. Now, what will this mean? Well, this will mean just a linear scan because P is already sorted.

So, you have 1 2 4 5 6 and all you need to do is push them in the same order into S. So this order n whereas insertion was an order n squared. Overall the complexity is order n squared because it's dictated by the more expensive insert. Now, this naive implementation of insertion sort should ring some bells. You should probably consider getting the sorted content of P straight away into S without having to populate a separate P. Why? Because you knew, note this that in the process of obtaining a sorted S, you've actually first created a sorted P. So that gets us to be in-place version of insertion sort and let's understand how this functions. So, as explained earlier we'll still have S which is [4 5 1 2 3]. The output is to get the same elements in S but sorted in increasing order.

So, what you do is as follows. You maintain the key that's need to be inserted and of course you get these keys in increasing order of the indices so you start with key equals 4, i is 1. The goal is to insert 4 into a P. What is this P? For us the P will correspond to a prefix of S. So this prefix will basically be S, elements from S ranging from indices 1 to i - 1. So your initial index was i. i was initially here at 4, i was 1. So initially the prefix P which corresponds to S[1...i - 1] will be empty. But going forward for i equals 2. This will be just 4. So for the time being we focus on the key at 4 at 5. So, P=4 and k=5 this is again when i=2. So, what you do is set the index j = i-1. The idea is to scan P starting from the rightmost element. So, as long as you've not hit the left most index of S or P basically. So while j is greater than 0 and till up a point where you

find that the jth element of S turns out to be greater than the key, you basically keep shifting the elements of S of b. Why are we doing this?

So let's understand with respect to the example. This is saying shift the elements of P to the right to make space for k until, so you do this until. You find that S[j] is already less than equal to k. So what happens in this case? Again, let's see. So for j equals 1, i equals 2, you're only interested in the prefix ending at 4 and you find that 5 is already greater than 4, greater than equal to 4. So in this case what happens is as follows. You'll need no shift, so the array S after this iteration will remain 4 5 1 2 3. But for next iteration, you will shift your focus on the i to be 1. Now when you again try an insert i into the prefix P preceding i. You find that i is less than 5, i is less than 4. So, what you will have to do is accommodate this i by right shifting the prefix members of S. So this we'll need a following transformation. This will mean S we'll need to make space for 1 4 and 5 shifted and then place 1 will be occupied by S and that is exactly what happens when you set S[j + 1]= k. 2 and 3 are unaffected.

So this specific insertion is happening here S[j+1]= k. You stop when you have no element to the left that is greater than the element k that is being inserted and in this case, there is no such element at all so the only place you can insert 1 here. So, I hope you understand the functioning of this specific algorithm. Is this correct? What is its complexity? Let's discuss that next. So correctness first and here is a claim for correctness. I will discuss in terms of a loop invariant. At start of each iteration of ith repeat loop, the subarray S[1.... i-1] consists of elements of the old S2, S[1.... i-1] but sorted in ascending order. Right. This is what a mimics a priority queue. So, what are we doing here?

So, let's understand quickly what the correctness on this invariant at initialization. So initialization, yes, indeed this holds because you initialize with i=1, which means there is a empty list, a prefix, prefixing up to i-1. So initialization holds maintenance. So, if this invariant holds before the ith iteration rather than i+1th iteration, will it hold at the i+1th iteration? The answer is yes. Please note that the i+1iteration is only going to compare the key at the i+1th position with all elements preceding it. And we are already assuming that the preceding elements are sorted. So you scan from the right to the left until you find an element in S[1... i-1], that is actually less than the element, less than equal to the element or the key that is being inserted. Right. And because you stop at that position, and because you know from the previous iteration that elements preceding the element that was found to be less than equal to the key are already sorted.

You can stop and in fact this also ensures that the loop invariant is maintained. So the maintenance holds since S from 1to j is sorted, this is referring to the algorithm shown in the previous slide. And we already found that S[j] was actually less than or equal to k, and since S[j+1] is going to be containing the key k, and the elements after S[j+1] that is S[j+2... i] are already sorted and shifted. And they were all found to be greater than k. So, this ensures that maintenance holds. Third property is termination, again this is very easy to verify, typically it is a maintenance property that requires some work. Analysis, we are going to look at the number of times each step of this algorithm is going to be invoked. So we have an initialization, order one complexity. We are going to iterate over all keys k except the last one actually. So, the complexity can be treated as n-1, the iterations are, require n-1, computations of k=S[i]. Thereafter, for each i you might have a variable number of shifts. So what you see here is, a variable number of shifts, by I

am specifically by shifts we mean right shifts. Until, I repeat until you find an S[j] which was less than equal to k.

So, this variable number of right shifts, we refer to by w(sub)i. Of course, there is one more additional step which is inserting k into S(j + 1). So, this gives us w(sub)i - 1 repeated n times and finally, once you have actually substituted S(j + 1) = k doing this n - 1 times. We can go ahead and sum up all these individual counts, so complexity given by this expression involving various constants. The only thing that needs to be elaborated a bit upon is what happens to this summation in the worst case? What happens to this summation in the best case? Everything else seems to be order n. So best case running time, what will be the best possible scenario for w(sub)i is? Which is, is there away that you can avoid shifting all together can the w(sub)i is basically 1 and it turns out that if your S was already sorted. So let us consider S = [1 2 3 4]. So it turns out that when you try to insert 2 or 3 or 4 you end with just one comparison. So, as a result, the best case running time is when the w(sub)is are 1 for all values of i. As a result these terms don't contribute anything at all, there is contribution from this term but it's again w(sub)i, so it just n times c(sub)6. So, you will have n times c(sub)6 the rest is as was discussed earlier these all turns out to be just order O(n). How about worst case running time?

So consider a list sorted in descending values. So, [5 4 2 3 1]. When I inserting a 4 you will need to go all the way to the left, in fact, hit the left most wall and put a 4 there. So, for any of the elements whether it's 4 or whether it's 2 you will need to go all the way to the left because you know you are dealing with this new smallest element. So in this cases, for w(sub)i for i = 2 will be 2 then w, i for i = 3 will be 3 and so on. So, w(sub)i will basically be i, so making the required substitution you find that will have a n into n-1/2 because you are dealing with summation i - 1, i = 2 , likewise for the factor involving c(sub)5 and so on. So overall this is O(n) squared. A questio is, is there something in between what about the average case? To discuss the average case, we will need to understand, when exactly does insertion sort do a swap? So, the most critical component of the insertion sort algorithm is a swap, and what is a swap, it is basically saying S[j + 1] =S[j] followed by S[j + 1] = k. So what are we doing? We are saying that 4 3 2 , so if I had to insert 3. Let's say the list was 1 2 3 and I was trying to insert a 3 here. How far will I need to go?

So, for every distance every element I cross for example, I cross 4, I am basically shifting 4 to the right. So I need to shift or rather do a swap corresponding to every inversion, what inversion is basically an i index less than j such that S[i] is greater than S[j]. So 4 happened to be greater than 3. This inversion let to a swap but the swap is very indirect you attain the swap by shifting and then substituting. A series of shifts and a substitution. Ok. So it's important to note that swap is realized through a series of shifts and a substitution. And, this occurs for very inversion, for every pair of indices i j, i¡ j, such that S[i]¿S[j]. We will see that there are other algorithms which are based on comparison, which don't have to do a swap for very inversion. But, unfortunately that's the case for Insertion sort. So, the number of operations is therefore the time required to scan each element plus the number of inversions. What is the average number of inversions? So let's say, 'i' is the number of inversions, then the time complexity is order n plus i. We'll try and come up with an expression for 'i' in the average case. You already know that for worst case, the number of inversions I is order n squared. For best case, number of inversions is zero.

The claim is that the average number of inversions in a list of n distinct elements is n into n -1 divided by 4. How did we come up with this number? So, here is some analysis. So, we will try and factor out redundancies in counting by clubbing together an array and it's reverse. So, factoring out redundancies in counting by clubbing together S and it's reverse. Okay. Let's see how that helps us. So, pair of elements e(sub)1 and e(sub)2 will be inverted in exactly one of S and S(sub)r. This is obvious. So, if e(sub)1 followed by e(sub)2 is the correct order, desired order it will reflect in S and the reverse will reflect in S(sub)r or vice versa. So, the total number of such pairs inverted in either one of S or S(sub)r. That's the advantage of clubbing we already see is n into n-1 by 2. Basically, this is n choose 2, because there are n choose 2 such pairs. So they have to be inverted in one of them. So, what we do is we'll consider the total number of such pairs inverted in one of them. And, therefore the total number of inversions across all permutations is this n choose 2 factor multiplied by the number of such sets of sequences.

So, this is number of permutations, but factoring out the reverse of each permutation. We don't want to count the permutation and it's reverse separately. That's why an n factorial by 2. So, what will be the average case? The average will just be the ratio. So, take the total number. This is the total, total number of inversions. And consider the total number of permutations. So, this ratio turns out to be n into n-1 by 4. So effectively, all we are saying is the number of pairs inverted in one of S of S(sub)r is n into n-1 by 2 and you know that this contributes to half to S(sub)r and half to S. So, therefore half of this should actually be the average number of inversions. That's an easier way of rationalizing. So, the average time, therefore, the average number of swaps therefore is exactly the average number of inversions. This happens to be order n squared. And this is to avoid any swap based algorithm that basically removes one inversion in a single step. So, here is the summary, the best case running time for insertion sort is order n, worst case i s order n squared. Our average again turns out to be order n squared.

We are not too happy about this. Can we develop an alternative that gives better average and worst case running times. In other words, can we avoid so many swaps or rather swap corresponding to every inversion. So, note that we, we used very naive property here, that we wanted the prefix to be always sorted. Can we relax that condition and avoid swaps corresponding to every inversion. And in fact, implicitly attain restructuring of the array through something like a min element, preceding all the other elements. Because that's what we really want. We want that a min or a max element precedes other elements and the ability to deal with all these minor or max elements at a later stage, So, that's exactly what we will discuss next. It is a Heap Sort algorithm. And, that is based on the heap data structure.

Thank you very much.