Lecture Transcript
Quick Sort

Hello and welcome to this session on Data Structures and Algorithms. In this session, we will discuss another algorithm for sorting a very useful popular algorithm called Quick Sort. Quick Sort is another instance of the divide and conquer approach to solving a problem. In particular, given a sequences 'S', quick sort is hinged upon choosing a pivot element 'x' and based on this pivot element 'x' identifying the set of elements in 'S' which are less than 'x' and the set of elements in 'S' which are greater than 'x'. So, pick a random element 'x' and divide the n-element sequence to be sorted into two sub-sequences L and G, such that L has elements less than 'x', G has elements greater than 'x'. You know, optionally if there are multiple elements equal to 'x', you might identify a band here which comprises of more than one element and you call it E. For the rest of this discussion, we will assume that E has a single element because whatever we discuss for a single element also holds for multiple elements.

This comprises the divide part, what follows is conquer. So, in conquer you apply the same procedure of identifying a pivot element and identifying the subset of L that is less than x, L and the subset element that have greater than L and then applying conquer on each of those partitions again, ditto with G you will divide G into the three parts. Finally, you re-sort to combine after having then all the conquer you merge the result of the arrays L, E, N, G that are sorted. So let us look at an example, imagine that you have to sort 9, 7, 2, 6, 8, 1, 4. You randomly pick 6 as a pivot element, this will lead to a divide where you identified elements which are less than 6 that is 2, 1 and 4. And now you're invoking conquer through recursive divide on this subarray. With 4 as the choice of the pivot element you'll will get all the other elements of this subarray to be the sub subarray to the partitioned. There is nothing more to do here, just two elements.

So, we are not invoking conquer again, you could do the same thing on the right-hand side, combine happens at every level. So, we have combined at the top level now, merging L, G and E. This is of course after several more invocations of conquer and combine for the G part. So, this is the G that got combined. You already combined all the subarrays in this subtree to give you L. And finally, you invoke a combine of L, G and E at the top level where E is the singleton set 6. So, here is the quick sort algorithm we have discussed and in place implementation of quick sort. So given input sequence S, pick a random pivot position p, partition S into S1, S2. Now we would like to do this in place, what does this mean? How do you partition S based on p into S1 and S2? This will be the subject matter of an exercise, homework exercise. However, we will discuss a very simple implementation of partition(S, p) when p happens to be one of the extreme indices of S and your task is to generalise that to p, even it is not one of the extreme indices . So continuing, once you have identified S1 and S2 and as I pointed out we will do this in place. So S1 and S2 are basically indices into S. Begin an index for S1 and begin an index for S2. Rest is basically the conquer, you invoke quick sort on S1 and quick sort on S2 recursively, you can also come up with a non-recursive version of quick sort.

Now since we have S1 and S2 in place and they are already sorted, finally you just need to merge S1, S2 and p, this merge is the same as before. You could also come up with the three-way merge and exercise would be to come up with three-way merge. It is actually very trivial extension as you have seen. So merge is discussed already, let's see partition and consider simpler case of partition when p is S.length, that is p is an index of the last element of S. So let us discuss the case of partition (S, p), p is the position with respect to which you want to partition and you want to give as output the two sub sequences S1 and S2. We will assume that S begins at position 'l' and ends at position p. What we mean is, this is position 'l' and this is position 'p' and our goal is to have all S1 on the left and S2 on the right. But we'll assume that the pivot is with respect to this last element. Let's call it 'v'.

So the first thing we'll do is, set v to the value of S at the position p and assume that p is the last position. What we do next? We'll now do some book-keeping. We'll try to partition S in such a way that for a particular position 'i', let say sitting here. I am going to ensure that S1 is to the left of 'i' and S2 to the right of 'i'. And how do I achieve that. Well, initially my i = l-1. Basically, it is before the first element, it's basically the wall because initially, I have no clue of what S1 is. So everything is basically S2, which is also wrong, we don't want that to hold. So as we scan we'll correct S1 and S2. Our next step is as follows. We are going to keep track of elements iterate over the elements through an index j. So we'll use an index j as iterator over elements of S ranging from l to p-1. We are of course not interested in scanning p when we compare with p itself. So what we is for j = l, you need to start scanning from the first element to p -1. Now we are going to check if this j, jth element S[j] is indeed less than equal to v. So as I pointed out our goal is for all k which are less than equal to i, we want S[k] to be less than equal to v and for k greater than i, you would like S[k] to be greater than v.

Well, you can make it strict inequality. So S[j] is less than equal to v, then you want to update your i, now got hold a one element, one more element which is less than equal to v. So you'll say i = i +1 and immediately swap entry at position i with position j and move on. So you keep doing this till you reach the end. So you just adjusting for index i, updating an iterator j, index i only gets updated when you find an element which is less than equal to v, otherwise you are perfectly fine keeping that element where it was. And finally as promised before you will need to exchange the element at position i + 1 which is supposed to be greater than equal to p, but as such p is sitting at the right-hand, so you can peacefully exchange this with the element at p which is basically v. So your homework is to deal with the case when p is not necessarily the last index. So what about p, that is a random index. So let's try and understand the running time of quick sort. What is a worst case? Well, the worst case is when the split requires most elements to be moved. So when S[p] happens to be the unique minimum or the maximum element of S, you will basically need to move every other element and this case is basically illustrated below. So this case of unique minimum will mean lots of swaps. What do you mean by lots of swaps? Basically, for an array of size n you will have to do n-1 swaps. So let's try and understand this.

So let's say you split S and it turns out that one of S1 or S2 has size length (S) - 1 and the other has size 0. Which means you have to adjust length (S)-1 elements, and suppose this is case for every subsequence split, that one of them has every element except one the pivot, the pivot has always been chosen to be the least of the greatest and this basically leads to a completely skewed binary tree. So, let's start and understand what the cost will be. So for depth zero, the time will be n because you have to scan all the elements and do those swaps and then the next select depth what will have to do is again scan n - 1 elements and then n - 2 elements, at i'th depth n - i elements, up to n - 1 depth where you have 1 element. So, this is basically going to incur cost which is i equals 0 or depth equals 0 to n - 1 and the time required will be n-d. This is very similar to the worst case of insertion sort. So, run time is proportional to sum and you can show that this is actually theta n square. How about the best case? What we will consider is a nearly best case. The best case is an instance of that. The nearly best case is that there is a fixed proportion splitting at every level.

So at the first level, n gets split into subsets of size pn and (1-p)n where p is some fraction between 0 and 1. At the next level, again there will be a split p raise to 2 n and p into (1 - p)n and so on until you reach p raise to i n, p raise to i - 1 into (1-p)n. The best case would actually correspond to the merge sort algorithm, that p is 0.5 where at every level the array gets split into two subarrays of equal length. Now, what is the work done at every depth? Well, you have to anyways scan all the elements at every depth. You need to scan and merge elements. So merge cost or even the split cost put together at the first level will be n. The second level again you will need to scan all these elements while merging. So that will sum to pn + 1 - p times n. So you sum up at every level, this will again give you n. The sum at each depth actually gives you n. So this sequence is split based on a fixed proportion at each step. This will go on to a depth d such that p raise to d n is 1 or 1 - p raise to d n is 1. We'll only consider with the extreme cases.

The first case when p is less than 0.5. So we'd concern with the min of p and 1 - p. So, the termination is basically when min of p and 1 - p raise to d - 1 times n = 1. What does this mean? Well, this basically means if you take logs, you will find that d - 1 times log of either p or 1 - p, let's take p case + log of n = 0. So, you can easily determine that d must be of the order of log of n. So the amount of work done at each level is basically theta n and we need to do this for theta log n steps. So the total amount of time required is theta n log n. Alternatively, one could also solve the recurrence equation, the time required at n case is basically upper bounded by the time required for the two partitions t of p and t of 1 - pn + a constant merge times cn. And by the master theorem, you could show that this would been order of log n. This also holds if this split proportion is upper bounded by p. So, it may not happen that the split proportion is always p or 1 - p but as long as there is an upper bound to that proportion which is p or 1- p, this analysis holds. How would be the average case? Now what does this average case mean? Well it is not necessary that upper bound for that proportion exists. What if we have this case? So n gets split by a proportion pn and 1 - pn but exactly the next level the proportion is actually lost which is you basically have split pn into some constant size left array and the remaining to the right this k1, our special interest is k1 = 1. Similarly for the other side, let's take the special case that k2 = 1. This will mean that one element to the left and remaining elements to the right in both the left and right branches.

Now, it is possible that even this kind of splitting holds for some number of iterations till finally you go back to some fixed proportion again. So, the average case is basically interleaving of fixed proportion and worst case. It is a very loose way of stating what the average is. One can get a bit more regress and consider the partition algorithm. So remember that the partition algorithm we perform the swap and our swap was based on subsets of comparisons that were made. The actual cost incurred is through the number of comparisons of pairs (i, j) made across all calls to the partition subroutine. What we can show is that the average number of comparisons of pair is (i, j) made across all calls to the partition subroutine is order (n log n). And this comparisons is the most frequently invoked of all steps. While you can refer to the section 7.4.2 of the second edition of CLR, I am going to give this intuitive proof sketch. So, let's define a random variable x, x(sub)ij is 1, if S[i] got compared with S[j]. So, what are we talking about? Well a subarray [i j] where i and j get compared.

Now, the probability that i and j get compared is actually proportional, probability that x(sub)ij = 1, is actually proportional to the length of this subarray. What does it that mean? Well, i and j get compared if either i was chosen as a pivot or if j was chosen as a pivot. It's not necessary for i and j to be compared at all if something in between got chosen as a pivot. So, therefore the probability that x(sub)ij is 1 is that i is chosen as pivot plus the probability that j is chosen as pivot and when I say chosen as pivot I am choosing it from the subarray S(sub)ij. Now, what is the probability of having i or j as pivot? Well, this is basically 1 upon the length J-i +1 + 1 upon J-i+1, these are two mutually exclusive events. Now let's try and study, what the expected value of a random variable x is. So, this is basically first of all, the random variable x is defined as a sum over all the values of i and all the values of j, j can only be i+1 until n of x(sub)ij. So, we are interested in the expected value of x which is a sum of all possible pairs of comparisons and we want, what is the expected value with respect to all these random choices. This is nothing but summation over i, summation over j of expected value of the x(sub)ij. Now, what is the expected value of x(sub)ij? Well, it's basically one times the probability of x(sub)ij being one plus zero times the probability of it being zero which is not of interest.

So, we expected value will basically be 2/J-i+1. Now it turns out that with some amount of reordering and restructuring of a summation, we can simplify this expression. So, let's do that very quickly. So, the first thing I am going to do is substitute for J-i, let me call it k. This is going to be a summation now over i, I will retain i as ranging from i equals 1 to n-1, but now k will actually be allowed to range from all the way from 1 to n. So, my dependence of J on i has been eliminated through this and fortunately for us we had J-i in the denominator. So, this gives us 2/k+1. Now, I know that by decreasing the value of the denominator, I am only increasing this expression, so I can show this upper bound over i and k, summation over i and k, 2 / k. Now, this is a well-known expression, we know that this is basically upper bounded by (log n). Now, we know that the summation inside is actually upper bounded by (log of n). So, basically what we get here is summation over i of (log of n) and this summation is independent of i, so this just gives you (log n). So, I encourage you to look at a more regress proof of this in the CLR book.

Thank you.