Lecture Transcript
Running Time of Program: Average and Worst Case Complexity, Asymptotic Analysis

Hi and welcome to this next session on Data Structures and Algorithms. In this session, we will continue our discussion on running time of a program. And in particular, we will look at average and worst case complexity. We will look at something called Asymptotic analysis, which is basically all pen and paper work. Unlike empirical evaluations that we talked about in our last discussion. So, going back to our search algorithm A and you might want to recall. That our search algorithm A was basically a linear scan algorithm meant to find an element e in a sequence S. We will now start looking at analysis as function identification. Now, we will look at two different notions of analysis. One is average, the average basically across all instances of the input.

Now, all instances of the input subject to some constraint is also a very frequently occurring scenario. So, we will look at all instances. For example of S, but with a fixed length of S. The other is maximum a worst case. This is again across all instances of S with the fixed length, but we are only interested in the worst case performance. For the first of the two, which is average. You will of course need to know how likely is each instance. So, to calculate average , you need to compute probability distribution over inputs. Now, a probability distribution needs a specification of what is a random variable. So, first of all we need to determine the probability of S having successful search for the element e. So, this is the probability at e is found in S at all. The second is the distribution over the location of e, the probability that e is found in S at position i. So, let's take the average case for search algorithm A. We will assume that the probability of success is say p and given that there is success, conditional probability of element e being at index i is 1 by n and this is basically for all i. So, what exactly is the probability, well we can first of all assume some success probability, let's say p is half.

Now, what the specific value of p is, will also dependent on how likely your element e is given the kind of sequences that get generated. But p equals half, is a reasonable assumption. So, what is the average case, well you need to sum over every possible location in the sequence. And the time required for search if the element e was at position i and this is the probability of having that element in that position. Plus one minus p that there is actually an unsuccessful search for sequence of length n. So, we recall that the time required, times required for successful and unsuccessful search, were respectively determined to be 4i+5, where i was the location of the successful search. And in case of unsuccessful search, we had found it to be 4N +2. So, substituting these values, we basically get the average time to be 3N + 2.5. So, it takes time that is linear in the size of the input on an average. How about worst case? The worst case is exactly when the element is not found, you need to scan the entire list. And that is something we already computed, this 4N + 2. So, the average is 3N + 2.5, in worst case it is 4N +2. The difference isn't really significant with the linear scan. How about an alternative search algorithm which shows some reasonable difference between the average and the worst case. And remember that we have discussed such an algorithm, an algorithm that we expect to be efficient, the binary search algorithm which basically starts with the extreme ends of the sequence S, the begin and end. It terminates when S is empty, which is begin ¿ end. Otherwise, it checks if the midpoint of S, is the number if it is found, it exits. If not it is going to check for the presence of or the possibility of finding the element e or the number to the left so this is the mid is going to prop if the number is likely in this section. If yes, then it launches binary search on the left half.

Similarly, for the right half, the query is, if, n is greater than equal to s of [mid] and that's highlighted here binary search a recursive column right-hand side, obviously exactly one of this to need to be invoked. So, what's the analysis? So a time taken in one function call is as follows. So if you count the number of math operations that get invoked. So, there is this mid, there are couple of operations where you look for equalities or inequalities that comprise comparisons. There also some assignments. The operation, math operation and assignment could be counted separately. So, found for example is that assignment. So that three, exactly three assignments/ math operations the remaining amount to comparisons. In fact, two of these comparisons will invoke cost of two each that's because they also require you to compute the element at a particular location as a sequence. So, that's basically array access or sequence access. So, overall this comparison of 5 of course, for the final call you don't need to invoke any of these and as a result for the final call you just return a false and skip the expensive computations sorry, this is the found as true, this is a final call , i am just going back. For the final call, you can avoid doing the second access which is check for if num is less than S[mid] and exit with found = true. So, therefore there is just 3 comparisons for the final call. 2 for this (S[mid]= = num) and 1 for begin ¿ end. The final point we note here is that the typical function call or the recursive call we make here to binary search, does invoke some additional cost.

A cost that needs to be factored in when you have lots of recursions and the reason is that you have to store the state of the current program before you invoke the new program and when you come back to the original program you need to retrieve the old state. So, all of this gets stored in a stack and that leads to more expensive function call, we have assigned an arbitrary additional cost for function call C. A general note is that recursive calls can involve more overheads as I already mentioned this is because you need to save and also retrieving parent program state. This is done through a stack. As an example, the factorial program implementations can be done in two different ways. So, you can recall the factorial program one was a recursive call fact(n) is n times fact of n-1 this is a recursive call you can also have an iterative version of the program which is equivalent where you iterate over indices, i = 1 to n and basically assign to factorial the product of the previous or the factorial so far with the current element iterator point. This is the same but now you don't need to invoke a stack to maintain states and so on. So, wherever possible it is advised to invoke recursive iterative programs but often recursive programs are compact and for brevity you might write recursive programs whereas while implementing you might go for the iterative versions. So, let us look at the worst case for Algorithm B and that is when element e is not present in the array which basically amounts to scanning every position in the array. So, this time required is basically C + 8 for each call except for the last call, the last call you basically find that the array is empty and you exit. So, how many such calls? Well, first we can try and represent this whole search in the form of a tree. So, the first call is of the range 0 to N - 1. But, in every recursive call, you reduce a search range by half of the factor. So, you look at N/2 and then N/4. Now, you might invoke the left or the right branch of this binary tree. But, what you'll be effectively doing is traversing a path in this tree.

The termination is when you hit a leaf node in the tree. Termination is equivalent to the case when begin exceeds end in that recursive function. So, we know that the depth of such a tree, a balanced tree is basically log to the base 2 of N, or you could also reason out based on shrinking sizes of the sub-arrays, that at log to the base 2 of N, you will basically have an array of size 1. So, the time required is C + 8 into log to the base 2 of N. This is for all the calls except the last one.The last case, you'll have a cost of 6 that really corresponds to the leaf. So, our recurrence relation is T(N) = T(N / 2)+ C / 8. How do you solve this? In the following lecture, we will talk of the Master's theorem and you can solve this using the Master's theorem. Basically, the Master's theorem gives you a template for solving recurrence relations in the Asymptotic case. And when I say Asymptotic case, I mean for reasonably large values of N.

So, how do these two algorithms compare? We'll assume that Algorithm B, the solution to the recurrence relation is exactly what we found by analyzing the decision tree and that happens to be (C + 8) log to the base 2 of N + 6. This is the worst case for Binary Search. The linear search has a worst case of 4N + 3. Which of these is faster? One might jump and conclude that log to the base 2 of 10 is smaller than N. But, please remember that we have whole bunch of other factors sitting here. So, assuming that C = 10 and let's take small values of N, such as N = 2. You'll find that for N = 2, 4N + 2, 11 is actually less than 18 times log to the base 2 of 2, which is 1, 18 + 6. So, for 2 and in fact likewise for 3 and for 4, Algorithm A actually is faster. In fact, this goes on for N up to 20. But, for N ¿= 21, Algorithm B suddenly becomes faster. Now, the most important point to highlight here is that Algorithm B becomes faster and remains so thereafter, which means for all values of N exceeding 21 or 20 basically, you don't have to bother about Algorithm A. So, there is a kind of monotonicity in terms of improved performance and this actually is consistent with experimental observations at for different values of N. In fact, this is the kind of analysis that is desirable. You get your insights and set your expectations before you even perform the experiment.

Now, can we leverage this notion of monotonicity and define the notion of complexity. So, yes indeed we can. So, there is something called Asymptotic analysis. Asymptotic analysis is really about this exploiting or leveraging this monotonicity, which is about analyzing and comparing running times only when input size is large, that is we focus on analysis for N, more specifically we use N when N is very large. For small inputs, even a bad algorithm such as linear scan will perform better and we did find that for N up to 20, a bad algorithm like linear scan did perform better. But, believe me, nobody in practice uses linear scan for searching, especially if the array is sorted. The other thing is we also want to get out of unnecessary details and we do that by focusing on order of growth.

Now, what does this mean? This means that we dont want to make any serious distinction between 40N + 400 and 2N + 1. This might be surprising for beginners. But the point is for large values of N, We have other more serious concerns. The way 40N grows with N is not the way N squared grows with N. So N squared grows much faster than 40N. So we want to make a big distinction between 2N+1 on the one hand and 400 log N + 1000 on the other hand or as I pointed out N squared on the other hand. So we do want to make this distinction. So the order of growth is really based on what we want to distinguish from what. So we want to be able to say that 2N+1 is slower than 400log N+1000 and it's faster than N squared. How do you say that? The intuition is as follows. The intuition is that function N grows faster than log N. So we are talking about rate of growth for those who are familiar with calculus, you could look at the derivative. The derivative of N, dN dN is 1, where as the derivative of log N with respect to N is 1/N.

So we have some intuition for continued valued numbers, for real numbers example that N does grow faster than log N. Now, can we articulate this algorithmically and we'd retry and build up on this intuition. Intuition is that, irrespective of the constants involved in the analysis, we know that after some large value of N, algorithm A will become slower than algorithm B. So here slower is where we talk about the rate of growth. So recall that T(N) for algorithm A was basically proportional to N and we will give it a name called Linear for B T(N) was proportional to log and we give it a name Logarithmic. What have we done in this process is we have ignored constants, constants don't matter, what matters is that one was linear the other was logarithmic but again we need something more formal. So before we look at the formal definition, let's just plot these different functions and convince ourselves, log N indeed grows much slower than N. Well N we expect to be linear, what we have also shown here are other functions such as N log N which grow faster than log N but grow slower than N squared. N cube and so on grow really fast 2 power N is something we would never like to have. So, the fundamental running time of an algorithm is called it's time complexity. we have already motivated that we would like to look at time complexity in terms of the dominating terms, we call the orders, that basically brings us to the order of complexity of an algorithm.

Let's discuss the order of complexity in some specific details. So one of the very well known and popular notions of order of complexity is this Big-Oh it's call the Big- Oh and denoted by this calligraphic O here. We say that the time complexity of an algorithm with input size N is Big-Oh f(N), where F(N) is the function. If there are positive constants C and n(sub)0, says that T(N) is less than equal to c times f(N) whenever N greater than equal to n(sub)0, that means, well you might have had T(N) exceeding a particular function f(N) for some point of time but if you are guaranteed that beyond a value of N, T(N) is consistently less than f(N) then you would say that T(N) is Big-Oh f(N). So this is being stated in some words, you can find a point n(sub)0 after which T(N) smaller than the linearly scaled version of f(N). So we have allowed some scaling factor c, so it is possible that T(N), if not scaled well or f is not scaled well might have T(N) going above f(N). But we are allowed to suitably scale f(N) so that the order of growth is still respected. You still don't want that for an arbitrary value of N, T(N) overshoots f(N). So the key here is that the c must be independent of N, once having determined c you need to stick to it. Constant c helps ignore multiple additive constants as I just pointed out, it ignores multiple additive constants. n(sub)0 helps you ignore any additive constants. The whole idea is to focus on dominating 'N' term.

Thank you.