Lecture Transcript
Order of Running Time of an Algorithm (Big-oh, Small-oh, Omega, and Theta)

Welcome to this next session on 'Data Structures and Algorithms'. In this session, we'll discuss the notions of Big-Oh. Basically continue on Big-Oh from the previous lecture introduce new concepts such as that of Omega, theta and 'o'. In addition, we will also provide an useful framework for analyzing recurrence equations. So recall our formalization of the first notion of rate of growth. We tried and bounded the rate of growth of a function based on something called the Big-Oh and it says that the upper bound for T(N) is c times f(N), f(N) for all values of N ¿= n(sub)0. So, asymptotically the function is bounded, T(N) is bounded by some other function f(N). We also explained the benefit of the c. c makes us, c makes it possible to ignore the multiplicative constants, that may be associated with f(N). On the other hand, n(sub)0 helps ignore additive constants that may be a part of T(N). So the idea is to focus only on the dominating term which is N. So let's understand this growth with an example. So, we claim that a particular function f(N) defined here us 5 into n + 6 is order of g(n), where g(n) is defined to be n. So, what are we saying that using f(N) 5n + 6 is order (n) which means for all n ¿= n(sub)0 f(n), is upper bounded by some constant c times g(n). So let's understand with this illustration. So, we find that the pink line f(n) is indeed bounded not by g(n) itself but a scaled version of g(n) which is seven times g(n). So you could think of c as 7 and therefore conclude that f(n) is order(n), 5 n+1 is order (n). This happens for some value of n(sub)0 which is some where between 2 and 4 . So, certainly for all n ¿= 4 you have this inequality. How about some other interesting functions? How about g(n) itself?

Now, we find that g(n) is always below f(n). Can we define something more interesting between f(n) and g(n)? It appears that f(n) is sandwiched between g(n) under scaled version of g(n). So, this is basically seven times g(n) and the curve here is basically g(n) and here is f(n). It turns out that one can formally define something called theta and Omega. So Omega is basically stating a lower bound just like Big-Oh gives you an upper bound. One can say that f(n) = Omega g(n) i.e., for all grade n ¿= n(sub)0, f(n) is ¡= some c prime times g(n). Here the c prime happens to be 1. So let me rewrite this. This is indeed c prime equals 1 and this corresponds to 1 times g(n). A small mistake it make inequality. f(n) is lower bounded by c prime times g(n) and the fact that f(n) is O(g(n)) and Omega(g(n)) sandwiched between 2 versions of g(n) prompts another definition called theta. So, f(n) is theta (g(n)). If, f(n) is O(g(n)) and f(n) is also Omega(g(n)). One could also draw some very other interesting conclusions from this graph.

The fact that f(n) has been lower bounded by g(n) with the scaled value of seven should also give you a hint that g(n) appears to be lower bounded by f(n) for some value of c prime. So in fact,

one can say that g(n) equals Omega(f(n)) with c prime value of 1/7 exactly the reciprocal of what was used here. On similar terms, one could talk of g(n) being theta (f(n)) this is based on the fact that g(n) is Omega (f(n)) and you could also conclude that g(n) is O(f(n)) and what implies at g(n) is O(f(n)), g(n) is O(f(n)) with c prime equals 1 or c = 1 if you want to be consistent with O. So, g(n) is O(f(n)) g(n) is Omega f(n) that gives us g(n) as theta f(n). So, let us formalize the other definitions. So, T(N) is Omega (f(N)) if there are positive constants c and n(sub)0 such that T(N) greater than equal to c of f(N) whenever N greater than equal to n(sub)0. So, it is just that now you've found a lower bound beyond an n(sub)0. So is an asymptotic lower bound or the growth rate of T(N) is asymptotically more than that of g(N). T(N) is theta (f(N)) if they have similar growth rates that is growth rate of T(N) and f(N) are the same.

Now we can also define a o(f(N)). So, T(N) is o(f(N)) if for all positive constants c there is an n(sub)0 such that T(N) is strictly less than c times f(N) when N is greater than n. That is a growth rate of T(N) is strictly less than that of f(N). So, you could think of specific instance of O. So if, T(N) = o(f(N)) you can conclude the T(N) is also O(f(N)). Now, what is really the difference between o and O? Well, if T(N) is O(f(N)) it may still be theta (f(N)) if T(N) also happens to be Omega (f(N)), but this is not true with o, if T(N) is o(f(N)), it will not be theta (f(N)) there is no way that (f(N)) gives you a tight bound for T(N), because even if T(N) had Omega (f(N)) you don't get in fact, you get a contradiction. So, let us give some examples to illustrate what we just discussed. 2N + 3 can be showed to be order N. Now it's possible to choose a value of n(sub)0 and a corresponding value of 6 as c. Such that T(N) is less than c f(N) that is 2N + 3 is less than 6n for all n greater than equal to 1. Now there is no unique combination of c and n(sub)0, but all you are interested is existence.

Now, one might be wonder is there a unique function f(N) which gives you O for a given function T. No. So, it turns out that T(N) + 3 is also O(N) squared it is also O(N) cube. In fact, its O n raised to k for k greater than equal to 1 actually, but what we are interested in general is the tightest or lowest order. So, note that O was basically an upper bound. So, while you can get arbitrary upper bounds, in fact, an upper bounds for the function that is basically O for a function T(N) are also upper bounds. So, if T(N) is O(f(N)) and f(N) in turn is O(g(n)) then you can show that T(N) will also be O(g(n)) and so on. But what we want is the tightest upper bound. We want to go as much close to T(N) as possible. So, generally by O(f(N)) for T(N), we mean the tightest upper bound. So, going back to the example you actually can show that you can't get better than order N for 2N + 3. So T(N) is actually order N and in fact, T(N) theta f(N) where f(N) is given by N. So in which case, the other higher order terms actually become meaningless. The other point to note is you might have lots of training terms, so 4N squared + N + 5.

Now you can show that this is nothing but order N squared, it is also order N squared + N. Again, order N squared is order N squared + N. So N squared is order N squared + N and therefore all these higher orders will also be order of a 4N squared + N + 5. But as before we are interested in the tightest upper bound and one more hint towards getting to this is to ignore lower order terms. So you typically ignore these lower order terms. Well, of course, you interested in the lower order terms for the original function but not while stating the order. So thus 5N + 3 log N is order N, because a log N is also order N. So we don't formally prove finding of c and n(sub)0 that's the general convention, you just write the order intuitively based on the dominating term and again point out that for this particular example 4N squared was a dominating term or 5N was

a dominating term. So here is an exercise for you, try and build an alternative search algorithm and we call it interpolation search. Now imagine that you were trying to find an element in a sequence 'S', but a lot sequence 's' and imagine that you had the sequence 'S' written across hard disks. So let's say this part of the sequence 'S' was written on hard disk one, another part was written in disk two.

So remember that binary search required you to find a mid element. Now how do you find the mid element? Even an array for an array that get sorted across a disk, you might have that the high element is here and the low element is in the sub-array or the other disk. So the idea is instead of looking for mid all the time, you can sometimes look for the next element. The call for the next element, this is should be remind you of linear scan. So what we are doing here is interpolating between a linear scan search, which was an order N and binary search which was order of log N. Now, again when is this practical? Well, it makes sense only if every access is very expensive. So retrieving this array from disk one and then retrieving the array from disk two and then going back to disk one and so on is expensive, holding it in memory can be expensive. So, therefore, each comparison might require disk access. The other requirement is that data must not only be sorted, it must be fairly uniformly distributed. What you don't want is a sequence like 1, 2, 4, 8, 16 the sequence handle,' binary search may not be productive. So a phone book is fairly uniformly distributed. So for example, if the input items are 1, 2, 4, 8 the distribution is not uniform. So you can try out the exercise of analyzing such an interpolation search. To begin with, you might want to formulate the interpolation search algorithm, then analyze it.

We are going to next discuss a very important theorem that we could make use of to prove self-recurrence relations, as we'll see several algorithms such as divide and conquer have recursions, they are recursive in nature and therefore recurrence relations become very natural in expressing the time complexity. What does it mean? This means that T(n) is T (n/a) + T(1-1/a) times N + something, something and so on. How do you solve such general recurrence relations? It's enough to give some asymptotic characterization for associating the cost of an algorithm. So you might not be interested in T(n) for very small values of n, as long as we are able to characterize both recurrence relation and its solution for N greater than equal to n (sub)0 for all N greater than equal to n(sub)0 we are fine. So the master theorem gives such a tool for solving recurrence relation in the asymptotic case. More specifically if T(n) is of the form 'a' times T(n/b) + f(n), where 'a' and 'b' are constants greater than equal to one and f(n) is a function. You could invoke that the master theorem to solve. For more details of the master theorem, you could look at section 4.5 of the third edition of CLRS. So what's the master theorem? So let's say T(n) is 'a' times T(n/b)+f(n). Now this n/b here could be replaced with the either the floor or its ceiling. and master theorem provides a follow asymptotic bounds for T(n).

So it's all based on cases and there are three cases. If f(n) is Big-Oh, I repeat we are looking at an upper bound for f(n) which looks like n raised to log to the base b of a minus epsilon. This has something to do with the structure of the problem that is invoked to prove the master theorem but many cases this just turns out to be something like square or cube so n raise to two, there is very classic example. How are you need a small epsilon which is nonzero? In such a case you can say that T(n) is theta n raised to log to the base b of a. Another example or another case is when f(n) is Big-Oh of n raised to log to the base b of a but there's no epsilon, the epsilon is missing. It turns out that T(n) continues to have a theta component with n raised to the log base b of a

but there is also an additional log n. So, this is a kind of price that you are paying for not being able to prove strict better lower bound that we had in case one. A third case is when you actually have a lower bound for f(n) with a plus epsilon you are not even able to prove lower bound with n raised to log to the base b of a if you meet the epsilon shift additionally if f (n/b) times a is ¡= c (f(n)) for some constant c and for all sufficiently large n then T(n) is theta of f(n). So now we, what will be find? We're basically saying that f(n) started dominating the third case. In the second case, and in fact even in the first case, we somehow were able to get f(n) on par with T(n). But in the third case, the f(n) actually just dominates, the recurrence relation yields to large value of f(n).

So, the intuition here is basically to compare function f(n) with n raised to log to the base b of a. Larger of the two determines the solution. The case three, is where the f(n) was really very large. So case one is larger than f(n) case two is f(n) and n log to the base b of a, are of similar size, case three is when the function f(n) is dominating. And here are some examples, so T(n) is 9 times T of n/3 here a is 9, b is 3, there is also f(n) which is n. So for suitable values of a and b for these values of a and b and for a suitable epsilon, we can show that n raised to log to the base b(a) is actually theta n square. So log to the base b of a, is nothing but square is two. So, the n square minus epsilon and this is basically theta n square. So since f(n) is Big-Oh of n raised to the log to the base of 3 of 9 where epsilon is one, we can apply case one to find the T(n) theta n squared. So, what's we have here is n raised to log to the base b of a actually dominates. We can also consider another example a different case here of this involves case two. Previous was case one. So, case two is about f(n) and T f(n)b and n raise to the log based b of a being comparable and log to the base b of a is log to the base 5 by 3 of 1.

Now, you can find out f(n) and f(n) turns out to be theta of one. This is basically can shown to be theta constant one. In this case you can actually apply the second case T(n) is theta log n. So, remember we have this additional log n component and will have the n raised to log to the base b of a, which turns out to be just theta one. A third example is when we have an Omega, lower bound for f(n). So, f(n) actually starts dominating. So, a is 3, b is 4, so b the base for the log is actually larger than a. So, you can show that n raised to log to the base b of a is order of n raised to 0.793 and this basically shows that f(n) is Omega times n raised to 0.793 plus epsilon, now epsilon is whatever needs to be computed the remaining component. This is 1 minus 0.793 and you can easily show that this leads to the case three of which basically gives you a theta and log n tight bound. There are several cases where you can't even invoke masters theorem. So for example if T(n) is cos(n) and not very clear, how we could invoke masters theorem because the fundamental problem is T(n) is not monotone. You could also look at other exponentiated forms of f(n), so f(n) could be 3 raised to n. It's not even polynomial because remember that we were comparing f(n) with n raised to log to the base b of a, therefore a and b are expected to be constants so you are not able to deal with cases where n sits in the exponent.

So we are in general interested in polynomial f(n)s as far as master theorem is concerned. Now well you might have a polynomial, T(n) is under root n squared plus 3 but then b is not a constant. Again you could have a case when a is not a constant. If it all boils down to writing down these expressions here and identifying a possible exponents for n but in neither case you will find n of the form n raise to log base to b of a for constant a or constant b. And finally, here is an interesting case where f(n) is not even positive and the well that isn't the case that master theorem can't

handle.

Thank you.