Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

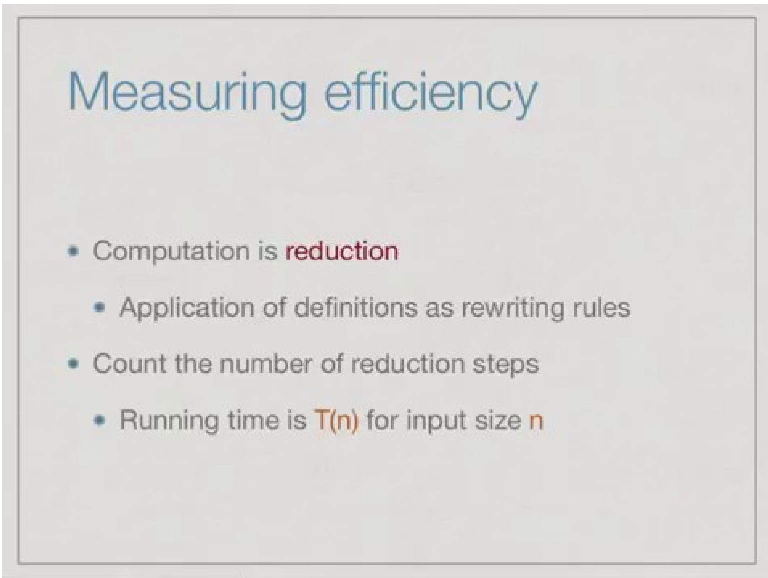
Module # 04

Lecture – 01

Measuring efficiency

Whenever we write a program to solve a given task, we need to know how much resources it requires, how much space and how much time. Here we will focus on how to compute the amount of time required by a Haskell program.

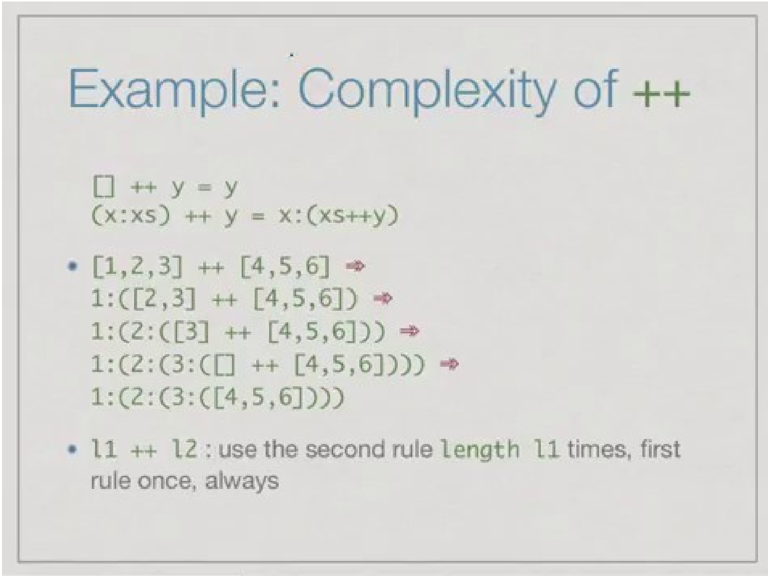
(Refer Slide Time: 00:17)



Remember that the notion of computation in Haskell is rewriting or reduction, in other words we take function definitions and when we use them to simplify expressions by replacing the left hand side of a define by its right hand side. In this way we keep applying these, rewriting rules until no further simplification is possible for the given expression. Hence it makes sense to count the number of reduction steps and use this as a measure of the running time for Haskell program.

Now normally, the running time of a program depends on the size of the input, it obviously takes more time to sort a large list than a small list. So, we typically express the running time as a function of the input size, if the input size is n, let us write T of n is the function which describes the dependence of the time on the input size.

(Refer Slide Time: 00:12)

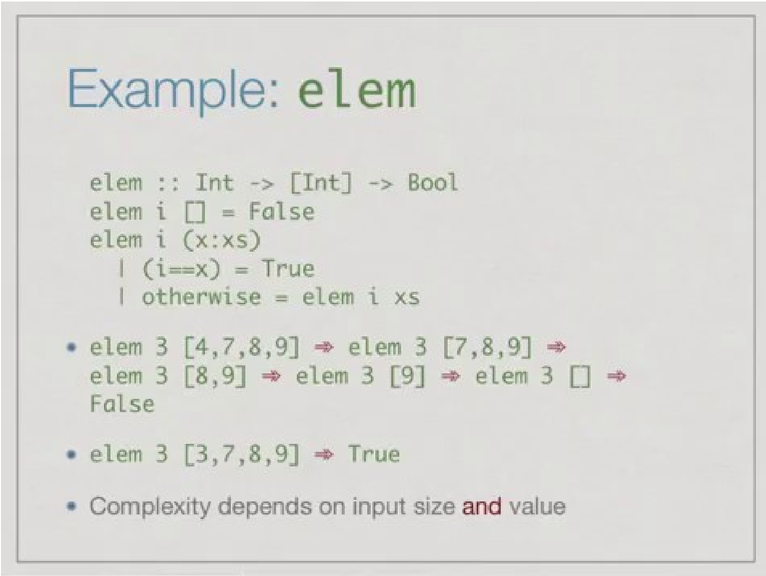


So, let us start with an example, here is the definition of the built in function ++ that combines two lists into a single list, so that function is defined by induction on the first argument. So, if we combine the empty list with the list y, then we just get y itself, if we combine the non empty list with y, then we take the first element of the first list x and move it to be the first element of the inductively combined list xs++y.

So, let us execute this on an input such as [1,2,3] ++ [4,5,6]. Since, the first list is non empty, the second definition applies and so we now have to append 1 to the result of [2,3] ++ [4,5,6]. Once again we apply the second definition, so inside the bracket now we have 2 appended to the list [3] ++ [4,5,6]. Once again we apply the second definition. So, we get the list 3 appended to the [] ++ [4,5,6]

Now the base case applies, so [] ++ [4,5,6] is just [4, 5, 6]. So, we get the final answer which is 1 appended to 2 appended to 3 appended to the list, 4, 5, 6 written as 1: (2: (3: ([4,5,6]))). So, from this it is clear that when we execute ++ for each element in the first list we have to apply the second rule once. So, the second rule is used length of l1 times and finally, when the length of l1 becomes zero, we will apply the first rule once and this behavior is independent of the actual values of l1 and l2. We will always use the second rule length of l1 times followed by one application of the first rule.

(Refer Slide Time: 03:02)

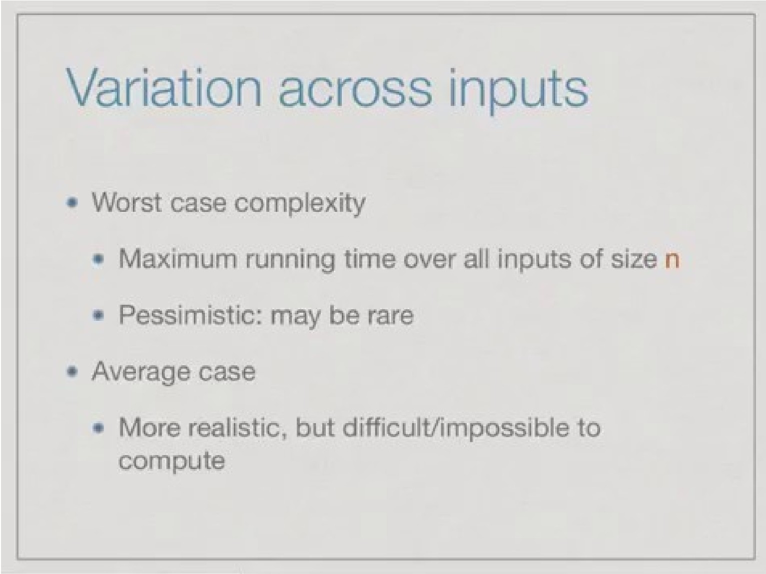


On the other hand, let us look at this function elem, which stands for element of. It checks if a given integer belongs to a list of integers. So, the base case says that the element i never belongs to the empty list and otherwise, we check whether it is the first element of the non empty list, if so we return True, if it is not so, we continue to search for the element in the rest of the xs. Now, if we apply the element function to a list which does not contain the value, then the second clause gets executed as many times as the length of the input until we reach the empty list and gets false.

So, we start with elem 3 of [4, 7, 8, 9] then in turn we strip off the 4, the 7, the 8, the 9 until we get to elem 3 of the empty list and then say False. On the other hand if we are lucky, we might find the element right away. For instance, if the first element of this list was not a 4, but a 3, then in one step we would find that the first element matches the pattern we are looking for and we would return True. So, in general the actual execution times of function on an input depends both on the input size and the actual value of the input.

When we executed ++, we saw that the value of the input did not play any role, we would always execute the command, the second definition as many times as the length of the first list and then execute the first definition once. But, in most functions depending on what we passed to the function the execution may take time less or more time.

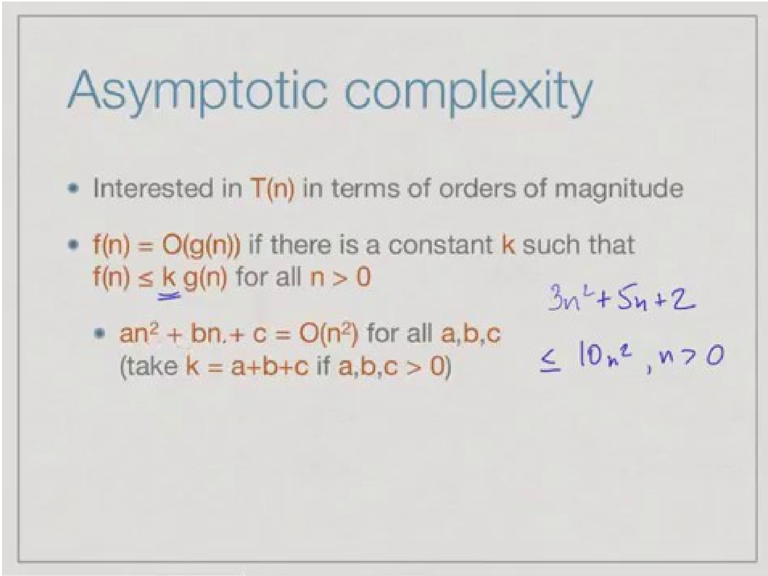
(Refer Slide Time: 04:40)



So, to account for variation across the values of the inputs, the standard idea is to look at the worst possible input, this is called the worst case complexity. Now, this basically takes the maximum running time over all inputs of size n and defines this to be the worst case complexity of the function, this is perhaps a bit pessimistic, because the worst case might occur very rarely. But, on the other hand this is the only concrete case that we can typically analyze.

It would often be nice if we could actually make some kind of statistical average and compute the average case. But,in many cases it is difficult to define a standard distribution of probability across all inputs and compute a meaningful average. So, though the average case complexity is a more realistic measure of how long the function takes, it is either difficult or impossible to compute in general. So, we must unfortunately settle for the worst case complexity.

(Refer Slide Time: 05:42)

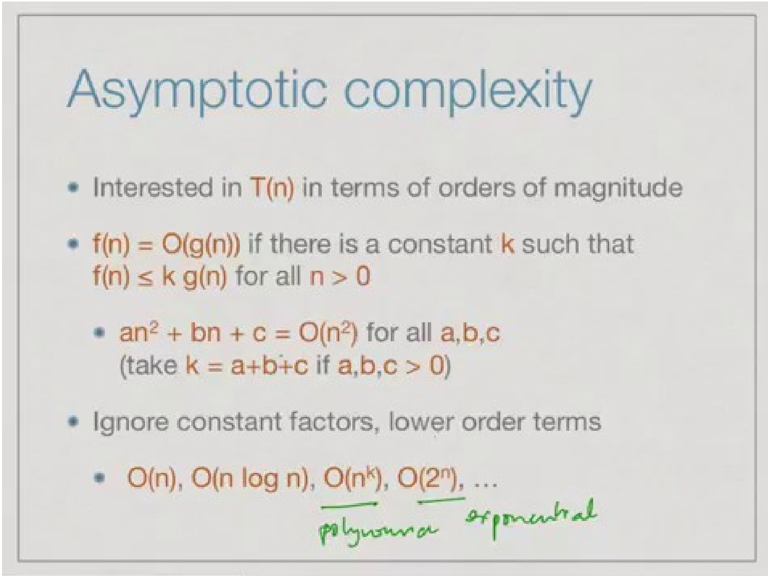


The other feature that is usually used when analyzing algorithms is to use what is called asymptotic complexity. So, we are interested in how T(n) grows as a function of n, but we are interested only in orders of magnitude, we are not really interested in exact details of the constants involved. So, the standard way to express this is to use this so called Big O notation. So, big O notation says that f(n) is no bigger than g(n), in other words f(n) is dominated by some constant times g(n) for every n>0.

As an example, suppose we have the concrete function f(n) as the quadratic an2 + bn + c. We claim that this is actually Big O of n2. For instance, supposing we take a concrete value such as 3n2 + 5n +2 then we could take 3+5+2 and say that this is always <=10n2 for all n>0. So, if a and b and c are all positive then we can just add up the coefficients to come up with this value k.

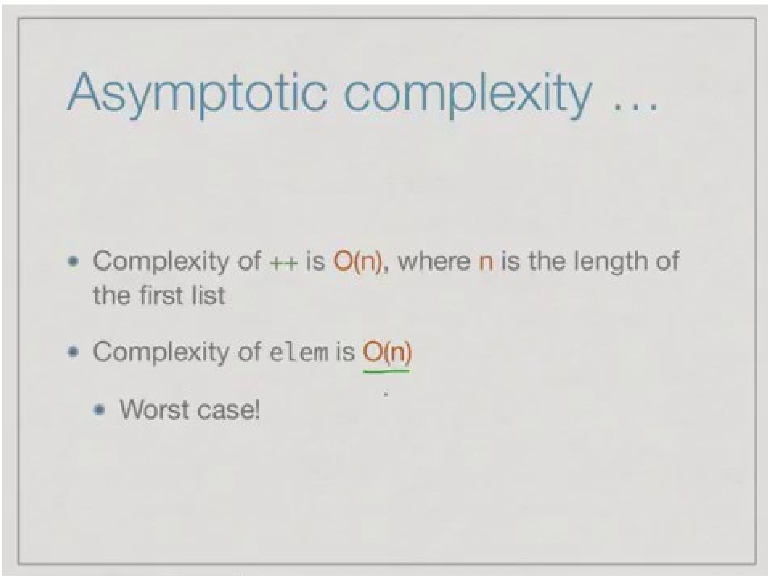
You can check that if any of the coefficients is negative you can just drop it. So, you can just add up this sum of the positive coefficients and that should work. So, usually it turns out to be a simple problem which is you just take the highest power.

(Refer Slide Time: 07:16)



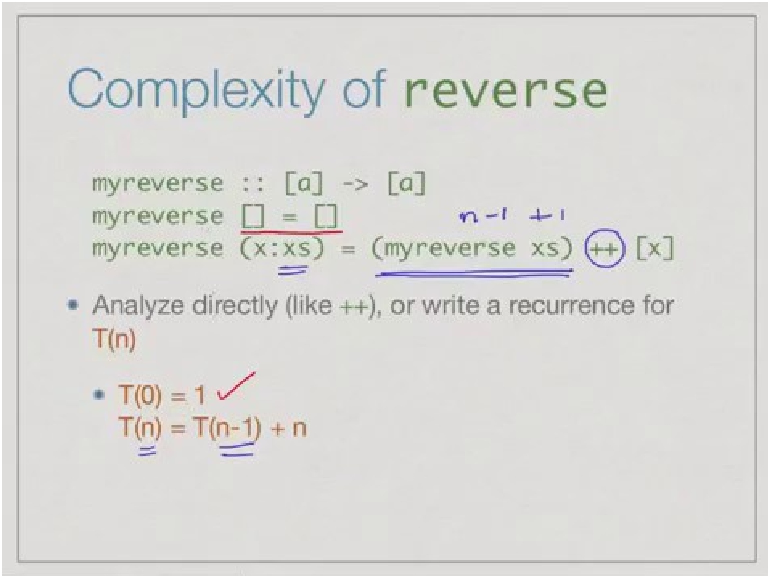
So, usually we ignore the constant factor, so we ignore constants like a, b and c and we take then among the terms that contributes to the complexity the highest power and we say that this function is O(n2). So, given this we typically express the complexity of the function terms of functions like nlog n or nk for some k. So, these are the so called polynomial functions or we have exponential and so on. So, this is the typical notation that we will use to describe the complexity of our functions.

(Refer Slide Time: 07:53)



So, in this notation we saw that the complexity of ++ is O(n), when n is the length of the first list, the length of the second list is immaterial. Therefore, it really is irrelevant as for as the input goes. On the other hand, for the function elem we again got a linear dependence O(n), but this is not true for all inputs, we saw that it could actually terminate in one step if the first element matches the length element we are looking for. So, this is really a case where we are applying this worst case definition in order to determine the complexity of the function.

(Refer Slide Time: 08:32)

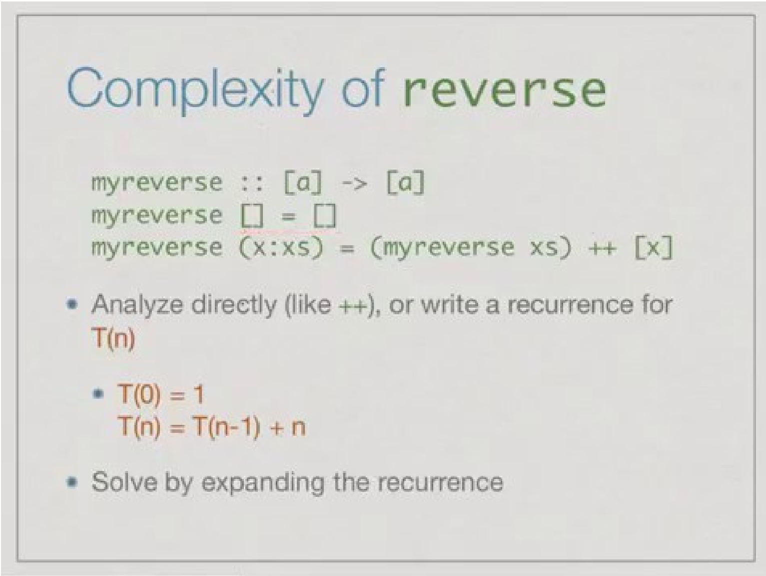


So, let us try and analyze the complexity of function that we wrote earlier. So, this is our inductive definition of reverse, we said that we can reverse the empty list by just returning the empty list. On the other hand, if we have a non empty list then we pick the tail of the list, reverse it inductively and then append the first element afterwards. Now, unfortunately this append we know depends on the length of the tail.

So, we could try to analyze this directly or we could use the fact that we know something about ++ to write what is called a recurrence. Recurrence expresses T(n) in terms of smaller values of T. So, in this case if we have an empty list, if the list length is 0, so here this input size is the length of the list to be reversed. So, if the length is 0 then clearly we can reverse it in one step. Now, if the length is not zero then we have to first reverse the tail.

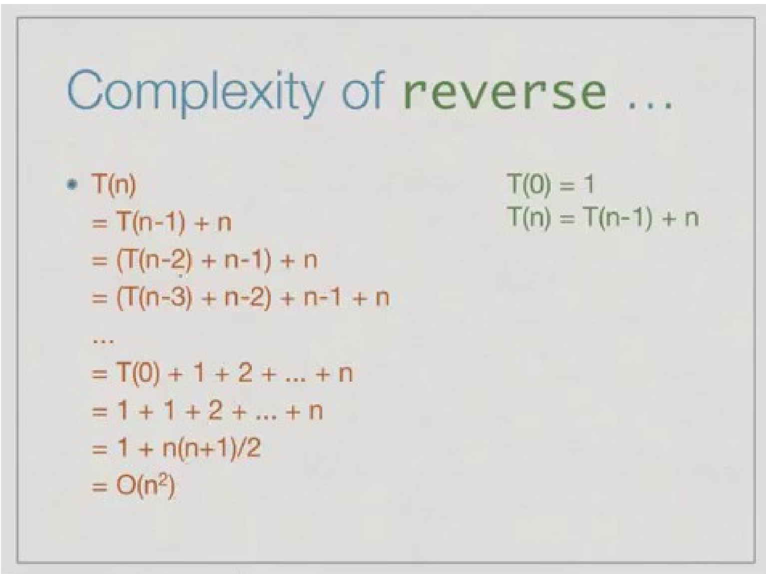
So, this means that in order to reverse the length of list of length n we have to first reverse the list of length n-1 and then what we saw in our first analysis was that this function ++ will take time proportional to n-1 iterations of the second definition plus 1 iteration of the first definition and therefore, it will take n steps. So, this gives us the behavior of the time complexity of reverse in a recursive form.

(Refer Slide Time: 10:08)



So, how do we solve this kind of thing to get an expression for T(n), well the easiest way is just to expand the recurrence.

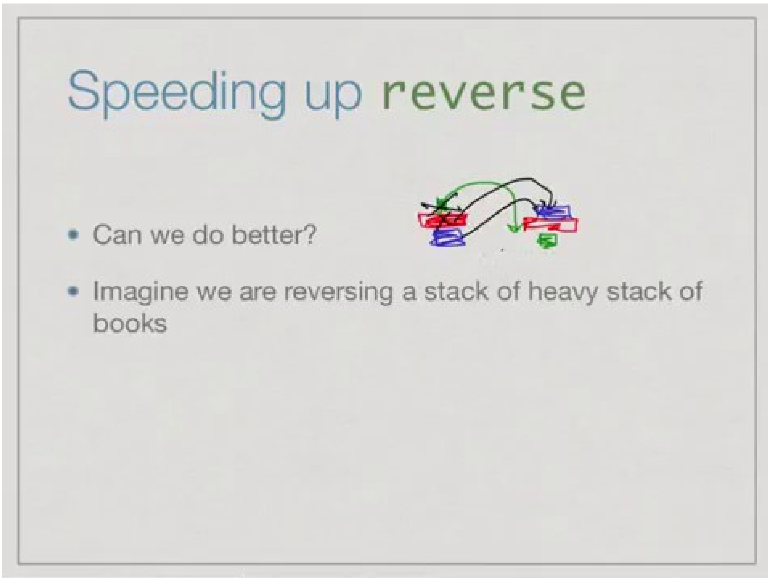
(Refer Slide Time: 10:17)



So, here is our recurrence now. It says T(0) is 1 and T(n) is T(n-1) + n. So, now, we start with T(n) and using the second item of the recurrence we expand it as T(n-1) + n. Now,in turn we can apply the same definition to T(n-1) and get it as T(n-2) + n-1. So, this is just an expansion of this recurrence, where n is substituted uniformly by n-1. In this way we keep expanding.

And so we are building up this term over here n-2 + n-1+ n and we have what remains, eventually we come down to the point where n-n which is 0 comes to us and we have on the right if you check this will be n-(n-1) so 1, 2, 3 and all that. So, this is the summation of i= 1 to n and this is well known is n*(n+1)/2 and hence going by our earlier way of calculating the Big O the highest terms of this is n2/2 + n/2. So, the highest term is n2 and if we ignore all constants this turns out to be order n2. In other words we are actually spending n2 time in order to reverse the list of n elements which seems rather inefficient.

(Refer Slide Time: 11:44)

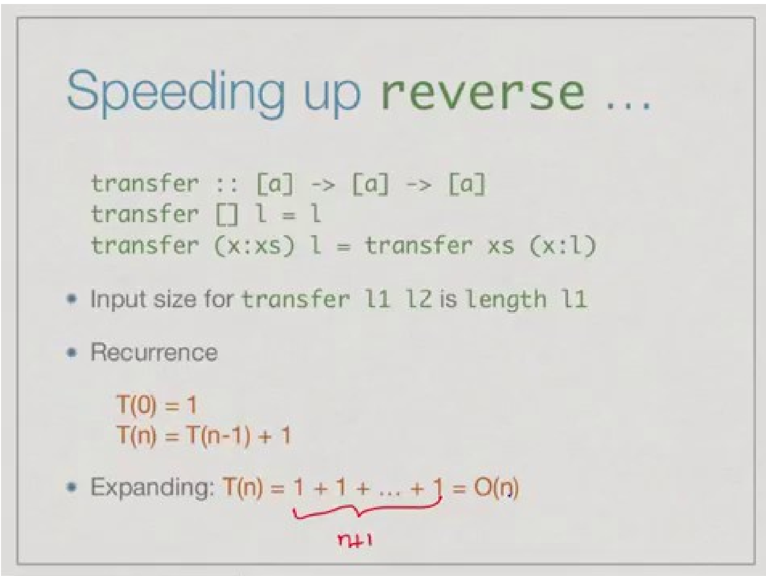


So, how do we improve on this? So, the idea is that we do not reverse the list in place as we are trying to do, but build up a second list. So, imagine that we have a stack of books, so maybe we have a red book and blue book then a green book ((Refer Time: 12:06)). So, now, what we do is we move this book to a new stack. So, we now have a green book here and we have move green book here, then we move the red book onto this and now we have a red book here and no red book. Finally, we move the blue book to new stack, now we have a blue book on top and now notice that this second stack is the reverse of the first stack.

(Refer Slide Time: 12:33)

So, in other words we transfer to the new stack from top to bottom, and the new stack is the old stack in reverse order. So, we can use this idea to write a more efficient version of reverse.

(Refer Slide Time: 12:46)

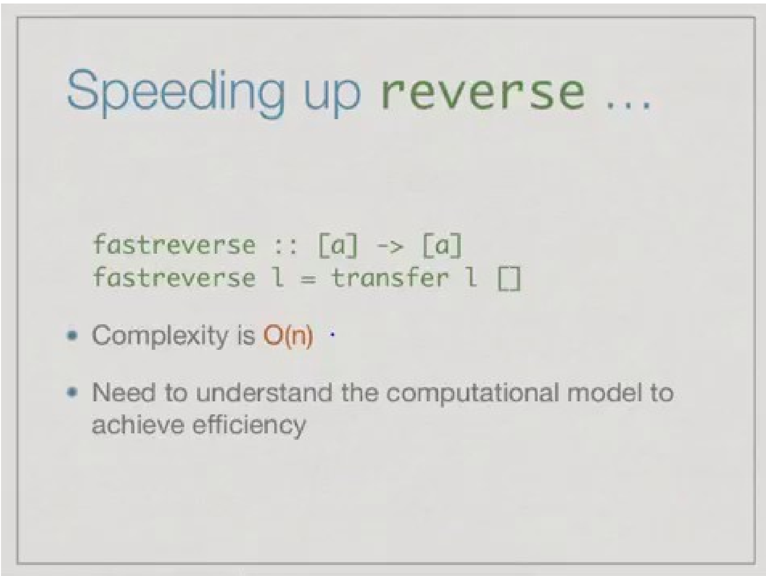


So, here is the idea of moving a list from one side to another side and reverse. So, what we do is we transfer the contents of our first list to the second list. So, the first list is empty then there is nothing to transfer and just leave the second list as it is, if the first list is non empty then this x is the book on top of the stack, so we move it to the top of the second stack. So, if we want to transfer x:xs to l then we keep the xs in the first stack and move this x from the first stack to the second stack.

So, now, it is clear that this function does not depend on the second list that is passed. So, this is the bit like ++. So, the input size is actually the length of l1 and it is clear that we have a recurrence of this form which says that if I have an empty list, then I do it in one step as a first argument, if I have a non empty list then it takes me one step to produce an instance of transfer of size n-1.

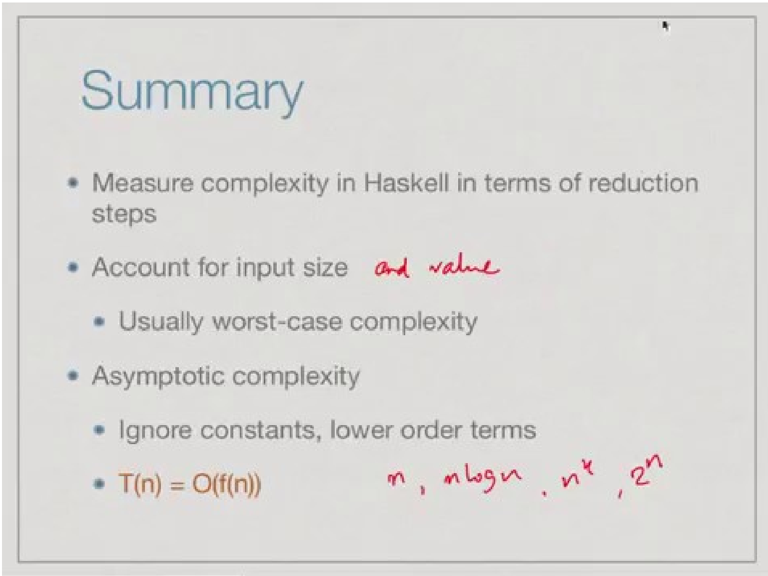
So, T(n) is T(n-1) + 1 and now using our expansion if we expand this n times we come down to T(0). So, we get 1+1+1…. n+1 times and n+1 is just order of n, in other words as we clearly know from the way we describe the process manually transferring a list in reverse from one stack to another stack takes times proportional to the length of the list.

(Refer Slide Time: 14:24)



And now we are done, because we can just start with an empty second stack as we said before and transfer everything from the first stack to the second stack. So, we have a fastreverse which is written in terms of this linear function transfer and fastreverse of l is just transfer the contents of l to the empty stack. The complexity of this function is linear, so notice that we had to take a look at how lists are treated in Haskell and how computation works in order to come up with the slightly non obvious definition of reverse, which matches the intuitive complexity that we have for the function. So, this shows that you cannot blindly apply the techniques from one programming language to another without understanding the computation model, if you want to achieve the efficiency that you like.

(Refer Slide Time: 15:16)



To summarize, we measure the complexity of a Haskell function in terms of the number of reduction steps we take to arrive at the answer. So, reduction consists of applying a definition in a function and rewriting the left hand side by the right hand side. Now, when we compute the function we have to account for the input size, but also for the input value, we saw the function elem could return quickly or take a long time depending on whether not the value we are looking for belongs to the list.

So, to account for the input size and the value, we usually use worst case complexity, because the desirable goal of computing average case complexity is very hard. And finally, we said that we will use traditional algorithmic ideas and express the efficiency in terms of asymptotic complexity. So, we will ignore the constants, ignore lower order terms and write T(n) in terms of Big O of f(n), where f(n) will typically be a function like n or nlog n or nk or 2n.