Function programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

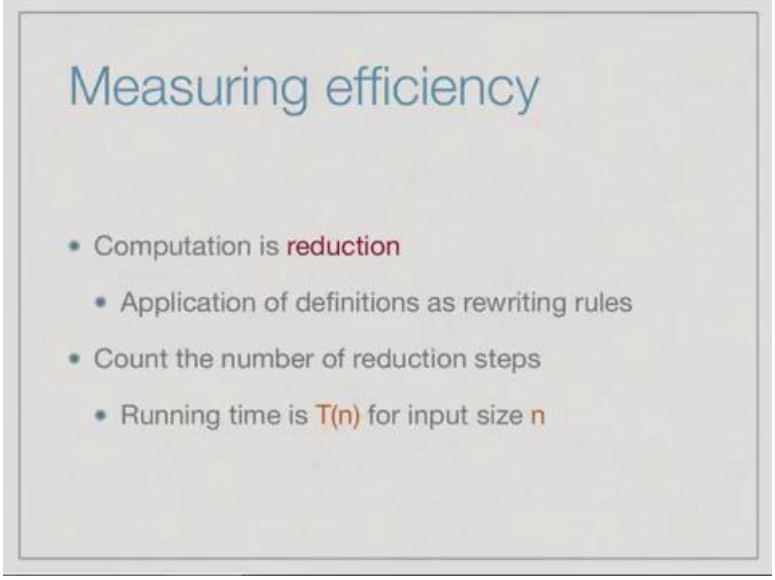
Module # 04

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 and how to compute the amount of time required by a Haskell program.

(Refer Slide Time: 00:17)

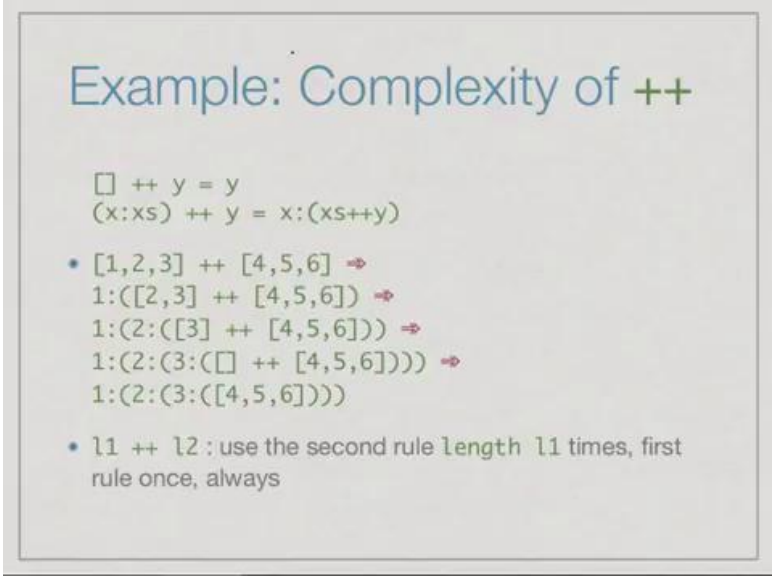


Remember that the notion of computation in Haskell is rewriting or reduction, in other words we take function definitions and when we use them to simplify expressions by replacing the left hand side of defining by it is right hand side. In this way we keep applying these rewriting rules and till no for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

Now, normally the running time of programs 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 is described, it depends of the time on the input size.

(Refer Slide Time: 00:12)



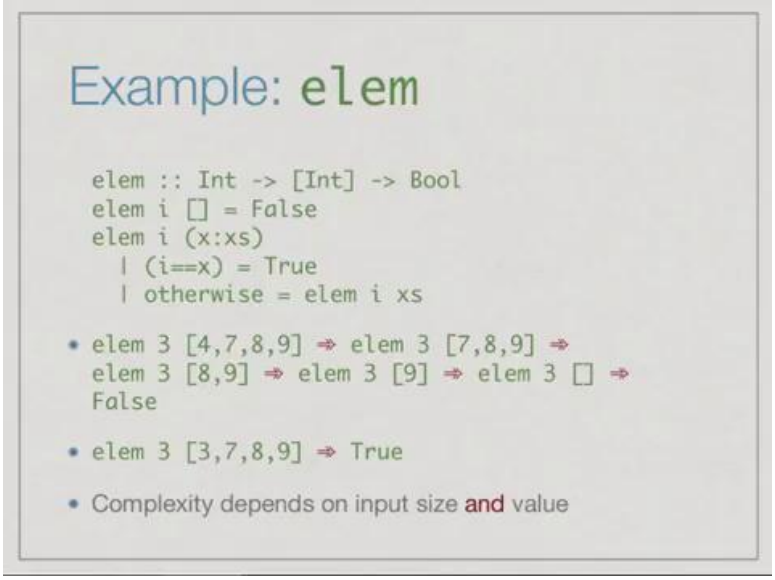
So, let us start with an example, here is the definition of the built in function plus, plus that combines to list into a single list, so the function is defined by induction on the first argument. So, if we combine the empty list with the list y, then we just get y itself, if we combine the non empty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

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

Now the base case applies, so the empty list plus, plus 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. So, from this it is clear that we execute plus, plus for each element in the first list we have to apply the second rule once. So, the second rule is used length of l 1 times and finally, when the length of l 1 becomes zero, we will apply the first rule once and this behavior is independent of the actual values of l 1 and l 2. We will always use the first rule length of

l 1 times followed by one application, always uses the second rules length of l 1 times followed by one application of the first rule.

(Refer Slide Time: 03:02)



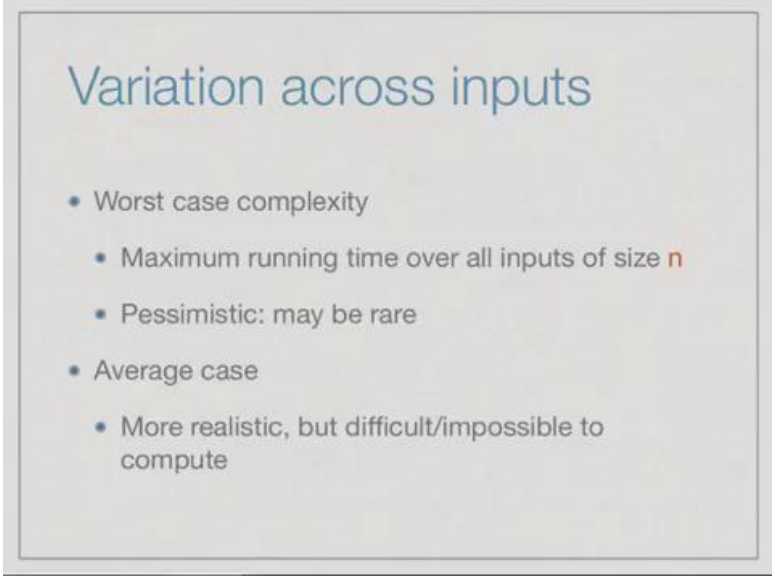
On the other hand, let us look at this function elem, which stands for element of, we checks if a given integer belongs to a list of integers. So, the base case is that the element i never belongs to the empty list and otherwise, we check whether it is the first element of the non empty list, if so it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

So, we start with l of 3 of 4, 7, 8, 9 then in turn we strop of the 4, the 7th, 8th, the 9th until we get to left three of the empty list and then say false. On the other hand if you 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 looking for and we would turn through. So, in general the actual execute usual times of function on an input depends both on the input sides and the actual value of the input.

When we executed plus, plus, 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 is the

length of the first list and then execute the first definition once. But, it most functions depending on what we past to the function the execution pay type less or no time.

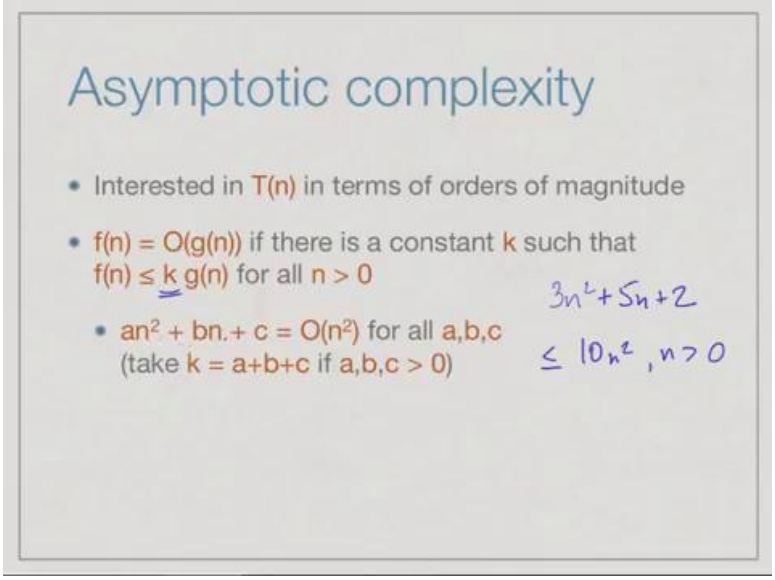
(Refer Slide Time: 04:40)



So, to account for variation across the values of the inputs, the standard idea is to look at to the worst possible input, this is called the worst case capability. Now, this basically takes the maximum running time over all inputs are size n and defines this to be the worst case complexity of the function, this is perhaps the bid pessimistic, because the worst case made a clerked very rarely. But, on the other hand this is the only congregate 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, it many cases 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, which is either difficult or impossible to compute in general. So, we must unfortunately settle the worst case complexity.

(Refer Slide Time: 05:42)

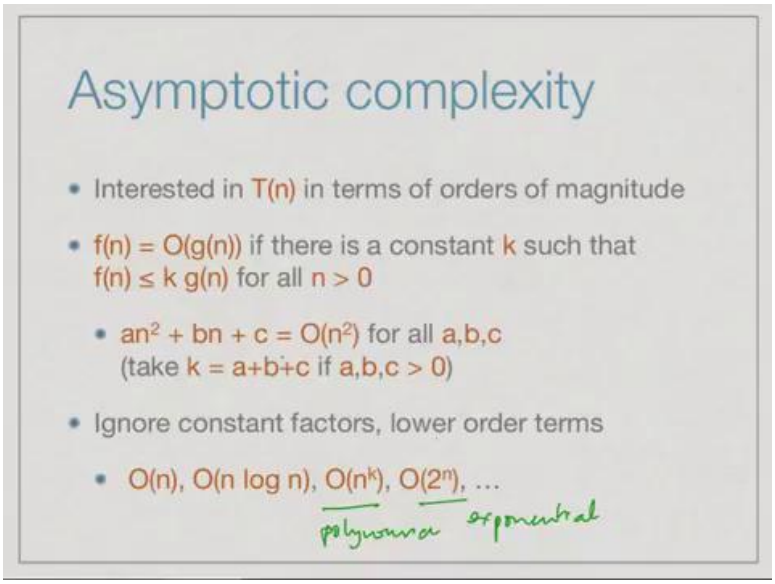


The other feature that is usually use then analyzing algorithms is to use what is called a asymptotic complexity. So, we are interested in how T of n close as a function of n, but we are interested only in orders of magnitude, we are not really interested in exact details of the constants more. So, the standard way to express this, this to use this so called deep over notation. So, big over notation says that f n is no bigger than g n, in other words f n is dominated by some constant time g n for every n greater than 0.

As an example, suppose we are the congregate function f of n as a quadric a and squared plus b n plus c. We claim that this is actually big over n squared for instance, supposing we take a concrete values such as 3 n squared plus 5 n plus 2 then we could take 3 plus 5 plus 2 n say and this is always less than or equal to 10 n squared for all n greater than 0. So, if a and b and c or all positive then we can just add up the coefficients to come up with this value k.

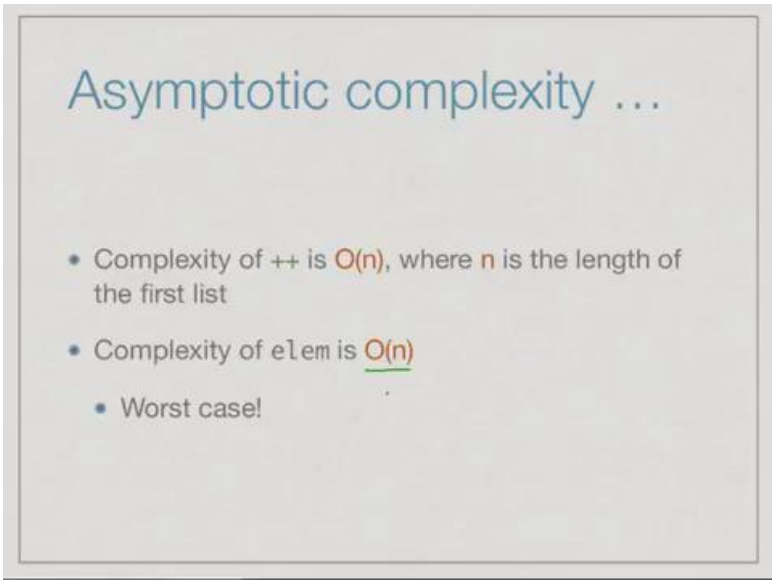
You can check that the when you of the cap coefficient is negative you can just stop it. So, you can just add up this sum of the positive coefficients that should work. So, usually it will ((Refer Time: 07:14)) which is just take the highest path.

(Refer Slide Time: 07:16)



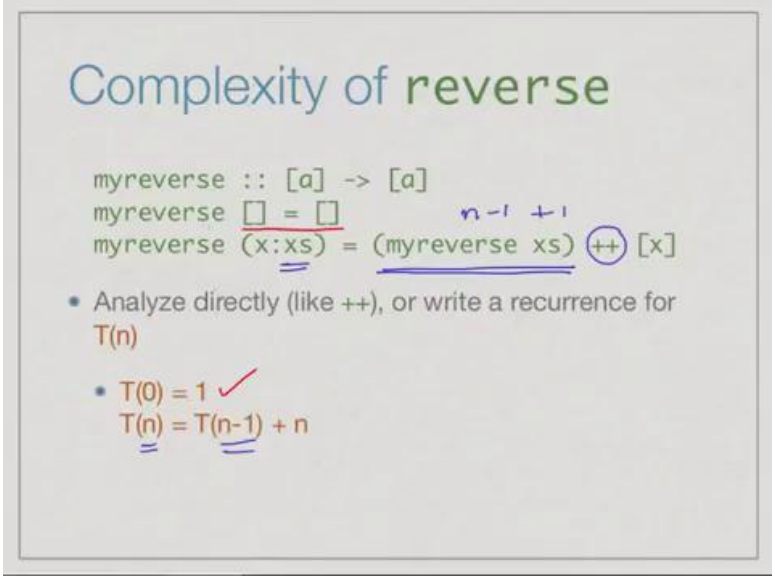
So, usually ignore the constant factor, so we ignore constants like a, b and c and we take them among the terms a contributes to the complexly the highest power and we say that this function over n squared. So, given this we typically express the complexity of the function terms of functions like n log n for n to the power k for some k. So, these are the so called polynomial functions are 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 complicity of plus, plus is O of n, when n is the length of the first list, the length of the second list is immaterial. Therefore, it really is the irrelevant as for as the input gives. On the other hand, for the function l n we again for a linear dependence O of n, but this is not true for all inputs, we saw that it could actually terminate in one step if the first element match is 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)

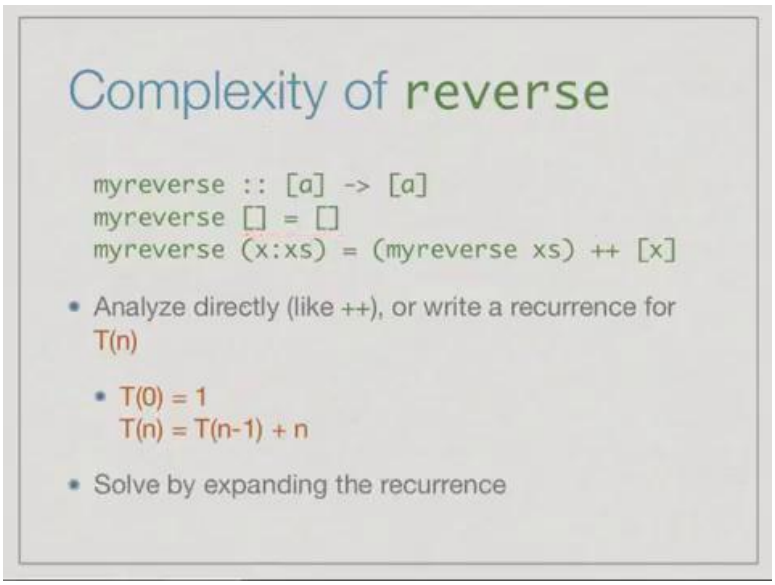


So, let us try and analyze the complexity of function that we wrote earlier. So, this is our inductive definition of reverse, we set that we can reverse the empty list by just return in the empty list. On the other hand, if we have an non empty list then we pick the tail of the list, reverse if inductively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

So, we could write to analyze is directly whole we could use the fact that we knows something about plus, plus to right what is called a recurrence, recurrence express is T of 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 of the length of the list to be reverse. 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.

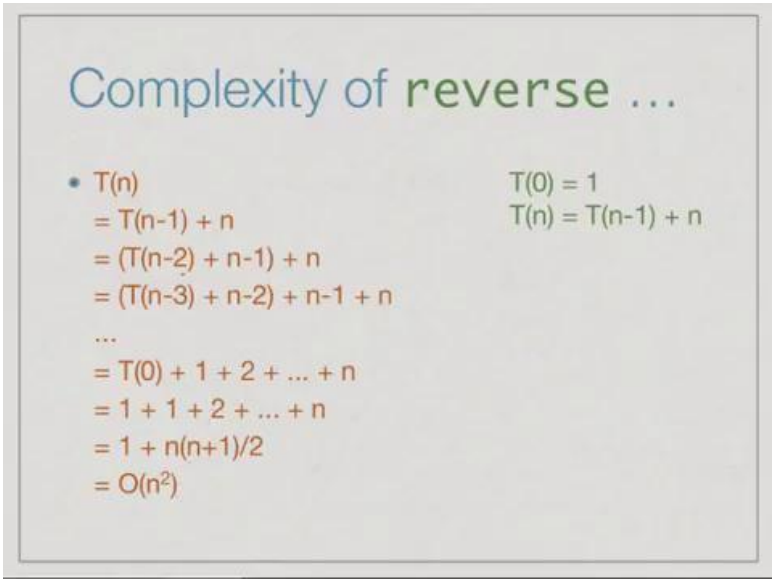
So, this means that in order to reverse the length of list of length n we have to first reverse the list of length n minus 1 and then what we say in our first analysis was that this function plus, plus will take time propositional to n minus 1 at iterations of the second definition plus 1 iteration of the first definite and therefore, it will take n steps. So, this gives as 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 a an expression for T of n, well the easiest way is just to expand the recurrence.

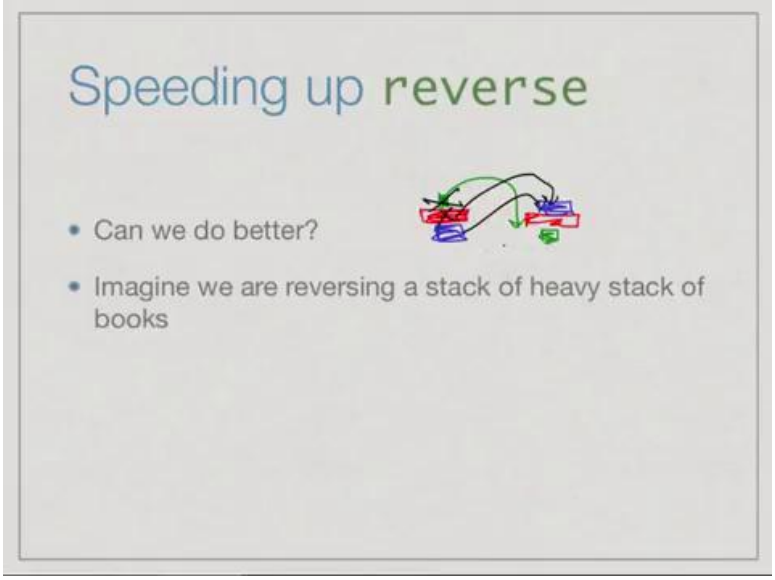
(Refer Slide Time: 10:17)



So, here is a recurrence now says T of 0 as 1 and T of n is T of n minus 1 plus n. So, now, we start with T of n and using the second item of the recurrence we expanded does T of n minus 1 plus n. Now, intern we can apply the same definition T n minus 1 and get it as T of n minus 2 plus n minus 1. So, this is just expansion of this recurrence, where n is substitute we will firmly the n minus 1 in this way we keep expanding.

And so we are building up this term over here n minus 2 plus n minus 1 plus n and we have what remains, eventually we come down to the point where n minus n which is 0 comes to us and we have on the right if you check this will be n minus n minus 1 so 1, 2, 3 and all that. So, this is the summation of i is equal to 1 to n of i and this is well known is n into n plus 1 by 2 and hence going by a earlier way of calculating will be go the highest terms of this is n squared by 2 plus n by 2. So, the highest term is n n squared and if we ignore all constant this turns out to be order n squared. In other words we are actually spending n square time in order to reverse the list of n elements with seems rather inefficient.

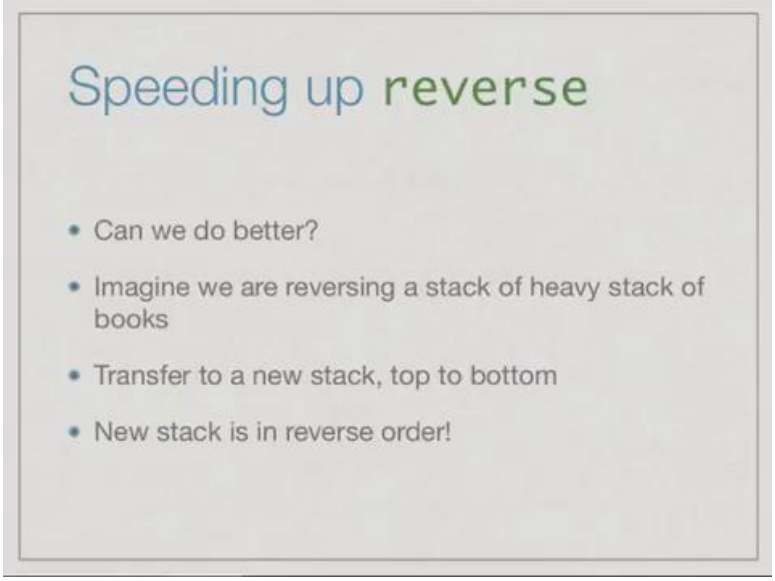
(Refer Slide Time: 11:44)



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 stack of this, 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 on to this and now

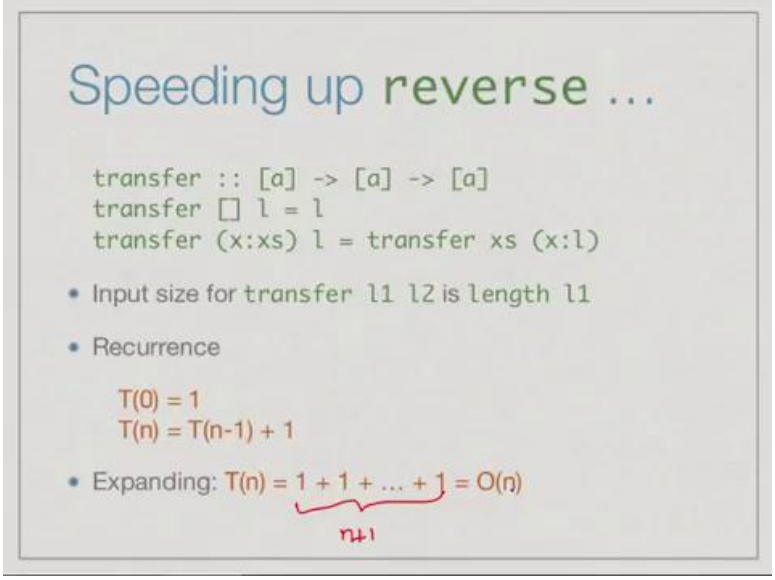
we have a red book here and no red book. Finally, we move the blue book from new stack, now we have a blue book on top and now notices in this second stack is the reverse are the first stack.

(Refer Slide Time: 12:33)



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

(Refer Slide Time: 12:46)

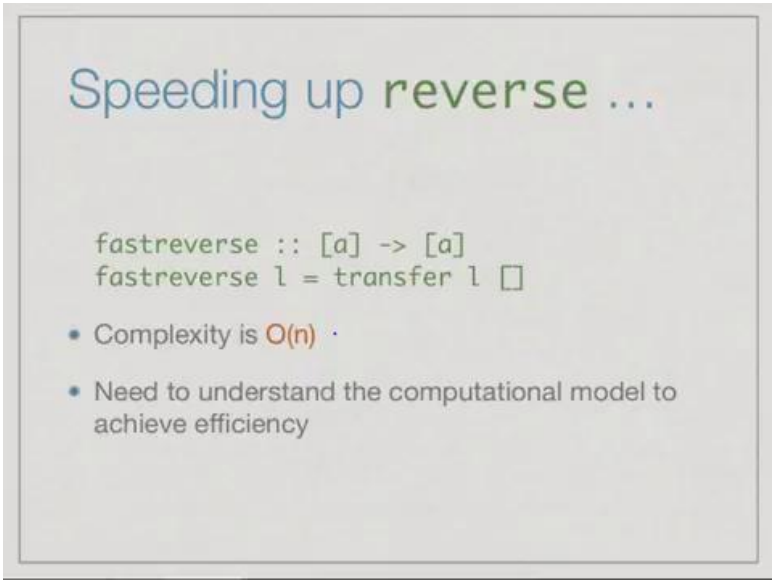


So, here is the idea of moving a list from one side to another side and reverse. So, what we do is we transferred the contents of our first list to the second list. So, the first list is empty then there is nothing to transfer and just leave the second list as list, if the first list is non empty then 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 colon x s to l then we keep the x is 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 at this pass. So, this is the bit like plus plus. So, the input size is actually the length of l 1 and it is clear that we have recurrence of this form with 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 steps to produce then instance of transfer of size n minus 1.

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

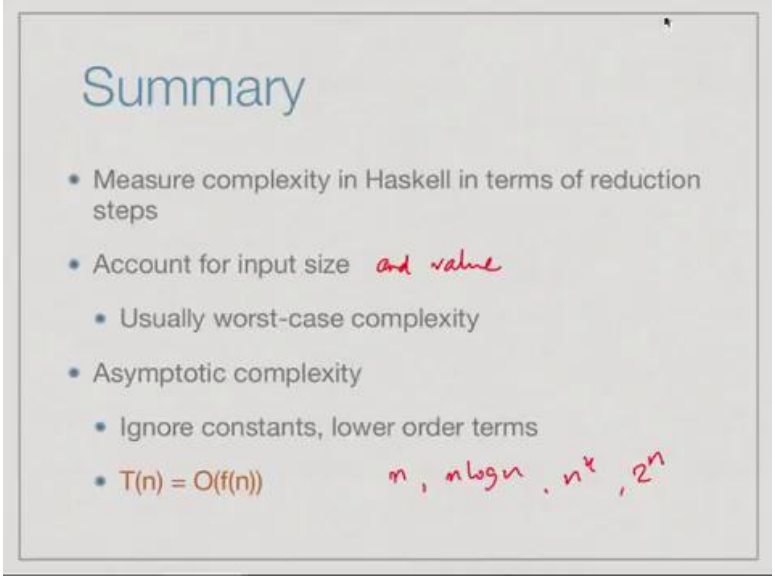
(Refer Slide Time: 14:24)



And now we are done, because we can just start with the empty second stack as we said before and transfer everything in the first stack to the second stack. So, we have a fast reverse which is recommended in terms of this linear function transfer and fast reverse of

n is just transfer the Constance of n to the empty stack. The complexity of this function is linear, so notices that we have to take a look at how list are treated in ((Refer Time: 14:53)) and how computation was in order to come up with the slightly non obvious definition of the reverse, which matches the intuitive complexity that we have for the function. So, the shows that you cannot blindly applied needs 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 summaries we measure the complexity of the Haskell function in terms the number of reduction steps you take to arrive at the answer. So, reduction consist of applying the definition in the function and rewriting the left hand side by the right land side. Now, when we complexity of the function we have to account for the input size, but also for the input value, we say the function l m could return quickly or take a long time depending on whether not the value I 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 it 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 Constance ignore lower order terms and write T of n in terms of below of f of n, where f of n will typically the function like n or n log n or n to the power k or 2 to the power n.