Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

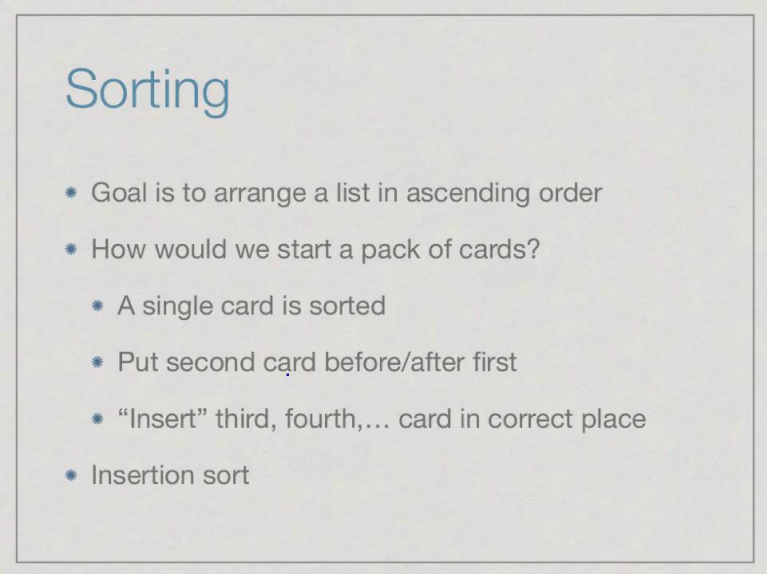
Module # 04

Lecture - 02

Sorting

Sorting a list is often an important prerequisite to doing other useful things sorted. For example, one way to search for whether a list has duplicates is to sort it and then, check if any adjacent values in the sorted list are equal to each other.

(Refer Slide Time: 00:19)

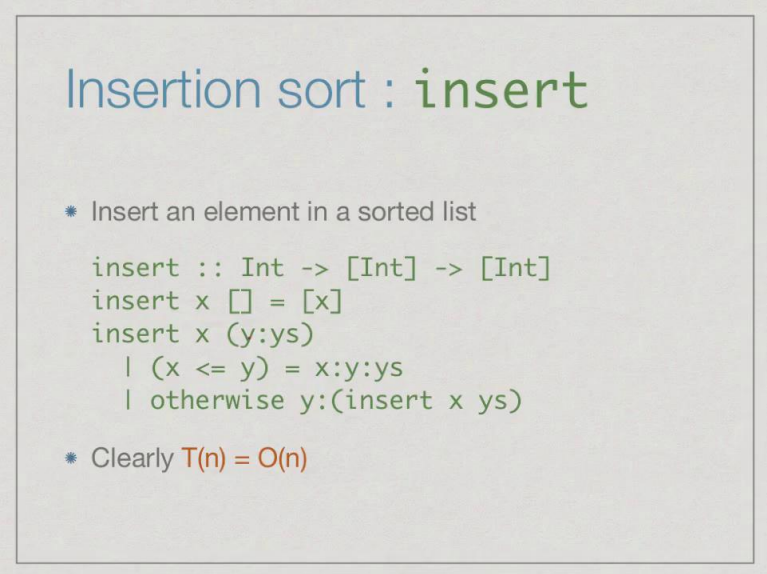


So, our goal is to arrange a list in ascending or a descending order of values. So, let us just focus on ascending order because; obviously, descending order everything will be symmetric using greater than instead of less than. So, understand a basic algorithm for sorting, let us try to sort a pack of cards. So, here is the simple way to sort a pack of cards and many of us are used to in practice, we start with the top card and we start forming a new pack of sorted cards.

So, since the top card is a single card by definition the new pack is currently sorted. Now, we take the second card and depending on it is value with respect to the first card we pick up, we either put it above or below. So, continuing in this way we take third card and put it in the appropriate position with respect to the first two cards and then insert the

fourth card in the appropriate position of the first three cards and so on, until the entire stack is sorted in place. So, since we keep inserting each new card into an already sorted list of the previous cards we have build up, this algorithm is quite naturally called insertion sort.

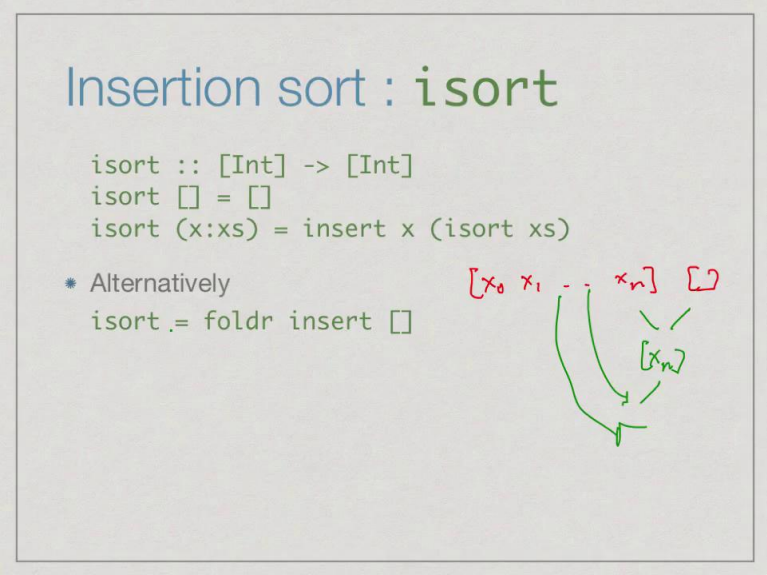
(Refer Slide Time: 01:37)



So, to describe insertion sort in Haskell, the first function we need to write is a function insert which puts an element into a sorted list. So, concretely let us assume we are sorting integers. So, insert takes an integer and a sorted list implicitly of integers and produces a new sorted list with the element we just add it, put in the correct place. So, the base case is to insert value into an empty list, we just produces one element list consisting of that type.

On the other hand, if you want to insert x into a non empty list of y s, we look at the first value and check whether or not x should come before it, if x is smaller than the smallest y, then we just take it up front, if x is not smaller than smallest y then overall between the x and the y s the first y is the smallest value. So, we pull that out in front and recursively insert the x into the remaining x. So, this in general put it square s to push x all the way to end of the list. So, if we take the input size of insert to be the size of the list into which we are inserting, it is clear that the worst case complexity T of n is big O of n.

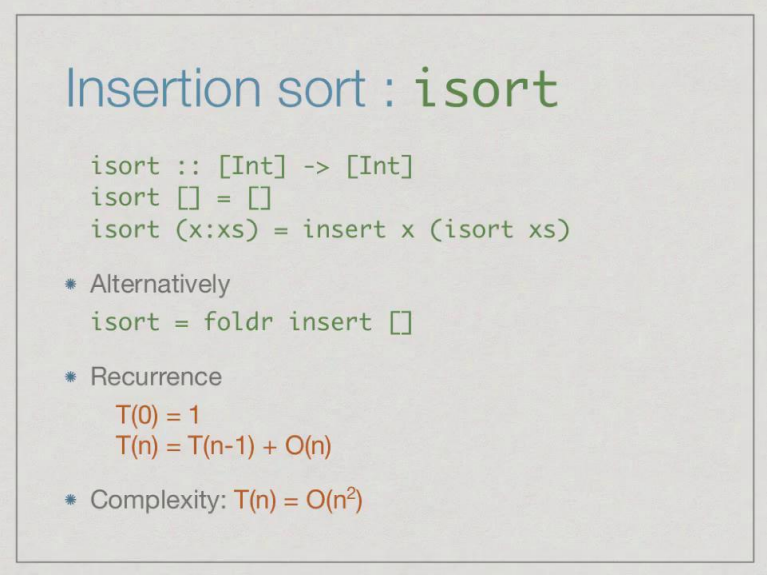
(Refer Slide Time: 03:07)



Now, we can express insertion sort in terms of the auxiliary function in certain that have been just used. So, we want to sort an empty list then we have to do nothing. So, we just get the empty list back, if we have to sort a non empty list then we first sort the tail and having sort of the tail we insert x into it in the appropriate position. So, the insert function does all the work and alternative way to write the same thing is to say that we fold the insert function from right to left.

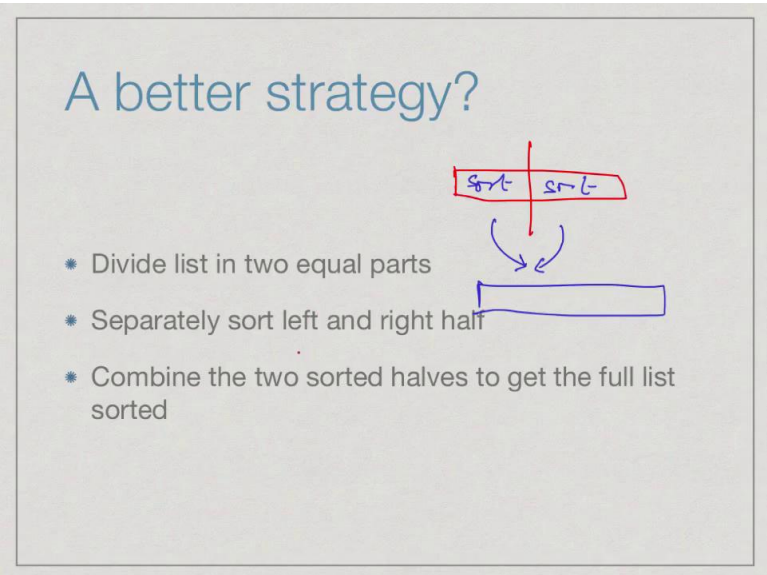
So, if we start with the list x 0, x 1 to x n minus 1 then we start with the empty list and then we insert x n into this and we get x n. And now we will take x n minus 1 and insert it into this, then take x n minus 2 and inserted into this and so on. So, I am just folding this insert function from right to left. So, concise definition of this recursive function is, just use our function foldr and say that isort is foldr of insert.

(Refer Slide Time: 04:12)



So, what is the complexity of insertion sort? Well, for the empty list T of 0 is 1 because in one step we get back the empty list and for the non empty case, we have to first sort n minus 1 elements and then having sorted n minus m elements, we have to insert the value into this list which takes order n time. Because, remember the complexity of insert is order n and therefore, the recurrence for insertion sort is set T of 0 is 1 and T of n is T n minus 1 plus O n. Now, we have seen this recurrence before and we know that we expand it out, you will get T of n is O n squared, because we are get something like 1 plus 2 plus 3 up to n.

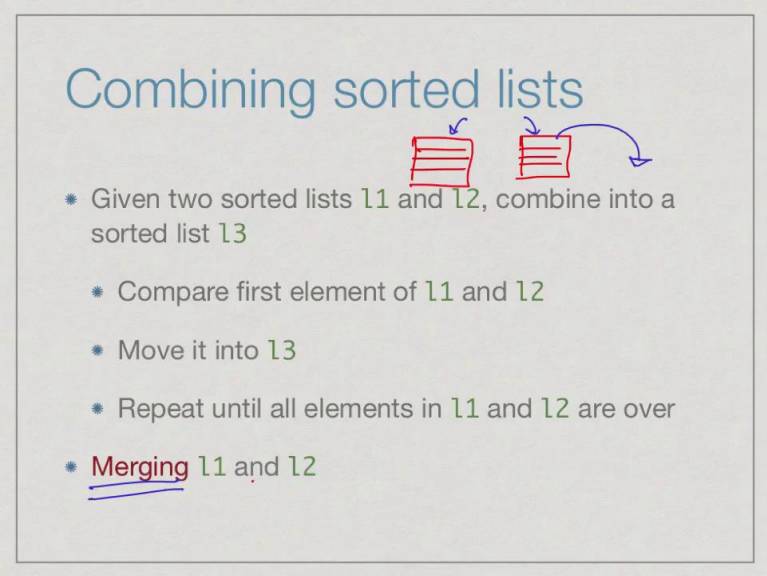
(Refer Slide Time: 05:00)



So, can we do better than O n squared for sorting? So, here is a better strategy which is

called divide and conquer. So, what we do is we take the given list and we divide it into two halves and then we separately sort this half, the left half and the right half and then we combine the two sorted list into a single over all sorted list.

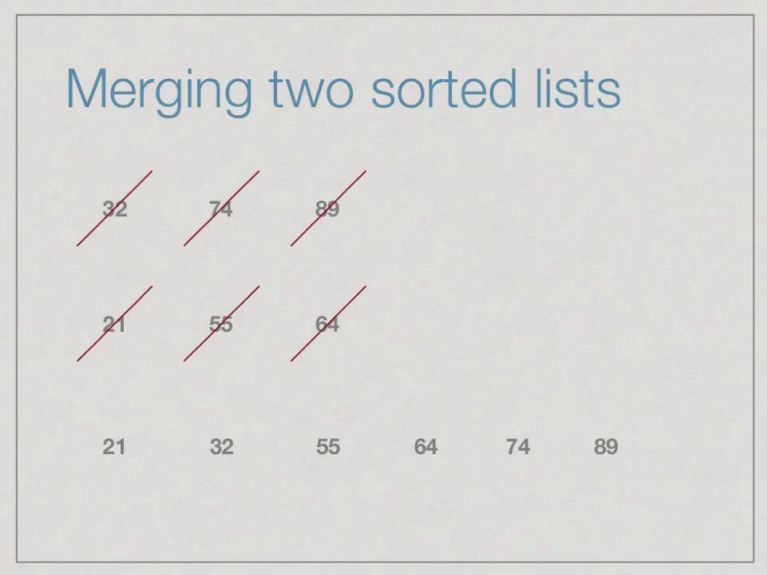
(Refer Slide Time: 05:29)



So, the final step requires us to combine two sorted lists l 1 and l 2 into a sorted list l 3. Now, this is easy to do, because the second imagine that you just had say two stacks of cards or papers whatever is sorted top to bottom, now you look at the top card in each thing and move the smaller of the two to the newest and each time keep looking at the top card of the two and move it to this smaller of the two.

So, in other words if you are looking at lists, we look at the first element of both list and move the smaller of the two to the new list and keep continuing, until we have exhausted both the lists, so this is called merging. So, we are merging two sorted lists into a single sorted list.

(Refer Slide Time: 06:14)



So, here is an example of how it will works and note that this is l 1, so this is sorted 32 smaller than 74 smaller than 89 and this is l 2, so this is also sorted. So, we start from the left where looking at the left most element. So, since 21 is the smaller of the 2, we remove it from the second list and move it to the new list. Now, since 32 now we are comparing 32 and 55 then we look at the smaller of the two which is 32 and move it to the list continuing with this we now move 55, because we are comparing these to every ones we get 55 and then 64. And now of course, we are nothing left in the second list. So, we can actually blindly copy the first list. So, we copy 74 and 89 and this is the merging procedure.

(Refer Slide Time: 06:59)

And now our earlier divide and conquer strategy was to sort the first half, sort the second half and merge, just remember that this notation means extract the ith element. So, it says sort from 0 to the midpoint, the midpoint to the end, so this should here round bracket. So, we start from 0 to n by 2 minus 1 from n by 2 to n minus 1, a merge sort is sorted halves into a new list l prime. And how do we sort the halves? Well, use the same strategy again we divide those into two and then we merge them and so on.

(Refer Slide Time: 07:41)

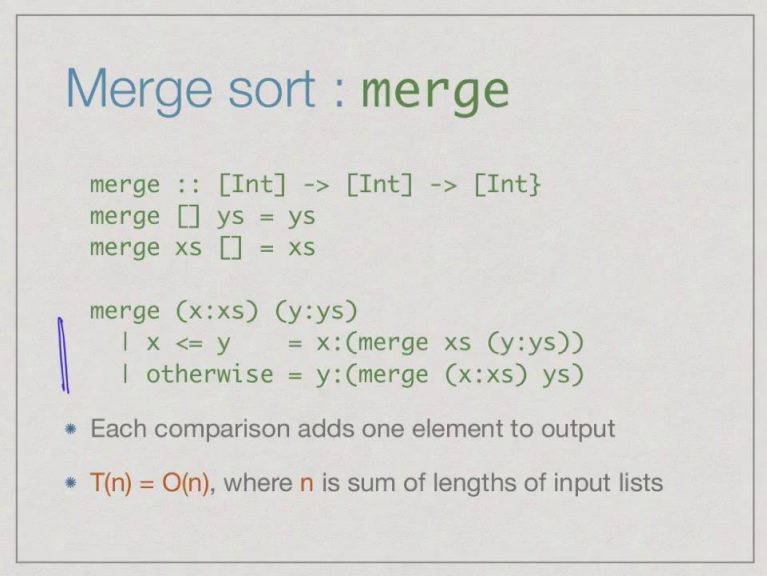
So, let us run through how merge sort would work on arbitrary list, so the first step you will divide at the midpoint. So, we will divided it is into two lists and try to sort them recursively. So, we will get the left list and the right list, now intern we will divide each of these into two, because we are obtain the same strategy. So, we get two list from the left and two list from the right, now we pretend that we stilled one how to sort. So, we will move it one more steps. So, we will split each of this list of length 2 into 2, so we will get individual list of length 1.

And now remember that a list of length 1 is by definitions sorted, because it is only one values it must be in ordered and now we can start merging. So, you want to merge these two, merge these two, merge these two, merge these two to get merge list which are sorted of length 2. So, it merge 43 and 32 we get 32 for our 43, the second pair gives us the same lists as before, the third pair again gets inverted, in the fourth pair gets inverted.

So, now, we have sorted lists of length 2 after merging lists of length 1, now you merge these sorted lists on length 2 to get two sorted list of less length 4. And finally, we merge

the two sorted links of length 4 list of length 4 to get sorted list of length 8. So, this is our merge sort works we divide, divide, divide and to gets singletons and then we work backwards merging, merging, merging until we get back of final sorted list.

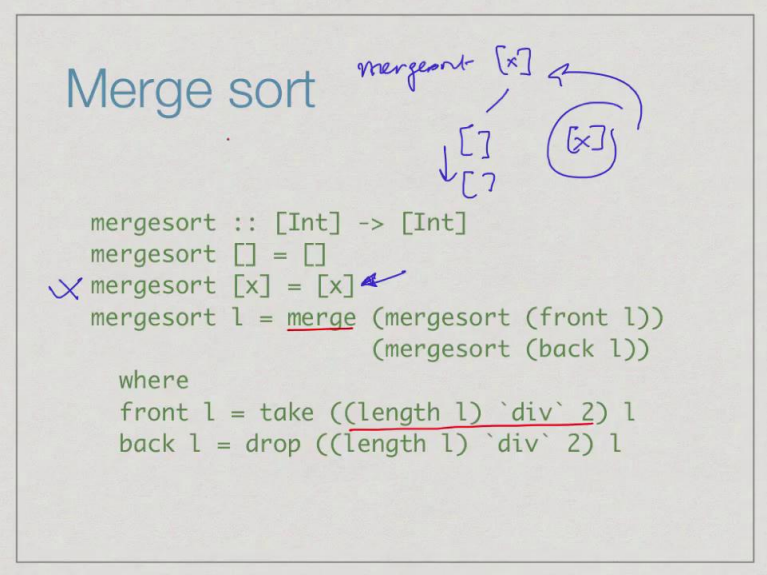
(Refer Slide Time: 09:15)



So, let us now write merge and merge sort in Haskell, so first we write the merge function, the merge function takes two lists which are implicitly assumed be sorted and produces a new output which is the combination of the two sorted list. So, we have base cases at the first list or the second list is the empty, we can just copy the other list without do anything. Because, it already sorted we just tackle it long, on the other hand if we have something to do can both lists are non empty, then we look at the first element and if the first element on x is smaller than y then we merge the tail which is that will x is with the remaining y s and then we stick x as the new first element of the overall x.

So, x is the smallest for all if x is not the smaller one then y is the smallest over all, so this is the very direct translation of an merging strategy that we saw before. So, to analyze merge one way to think about it is that every time we apply this second rule we are in principle adding one value to the final output and reducing the number of elements we merge by one. So, overall it will take as much time as the number of elements and the two list put together. So, we can say that the complexity of merge is big O of n, where n is the sum of the length if the two input lists.

(Refer Slide Time: 10:42)

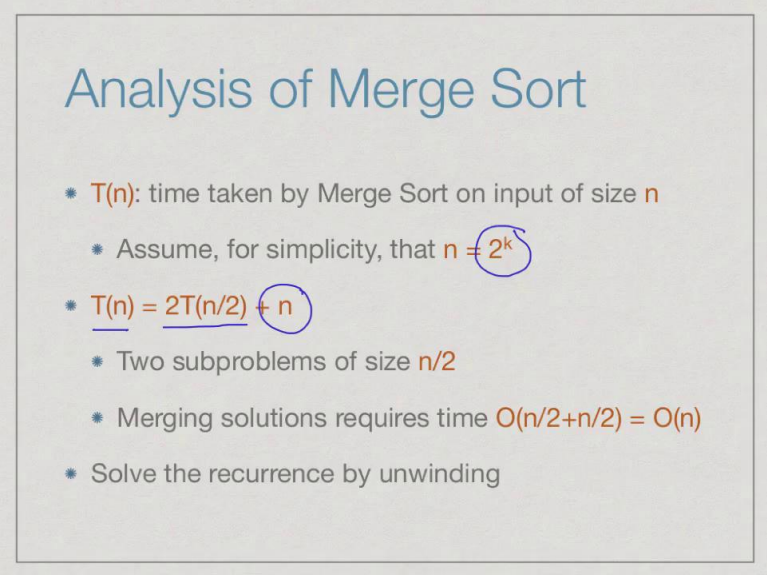


Once we are written merge then merge sort is immediate, so merge sort of the empty list is the empty list, merge sort of the one element list is the one element list and merge sort of the any element list with two or more elements consists of first recursively sorting the first half and the second half, which are defined in terms of taking half the elements. And remember that take and drop apply to this same argument exhaustively enumerate all the elements. So, take plus plus drop is always the list itself.

So, we can be sure that we are not losing an elements are missing out or duplicating elements we are taking the length of l integer divided by 2. And then we dropping the same number elements and using that is the back half of the list and having now constructed the sorted versions of the front and the back we have merging it is instructive to note that we need this case, we cannot just do with the base case of empty.

Because, otherwise if we take merge sort of x and we do not have this case, suppose we do not have this case then we will try to split into two. So, this would become merge sort of the empty list and merge sort x again now this would give us empty by the base case, but now this would again go back to the same case. So, we will end up with the loop, so we need a base case for the empty list, but we do in a base case for the singletons list as well; otherwise, we will end up with the an infinite loop and we come to the singleton and try to split into back and front.

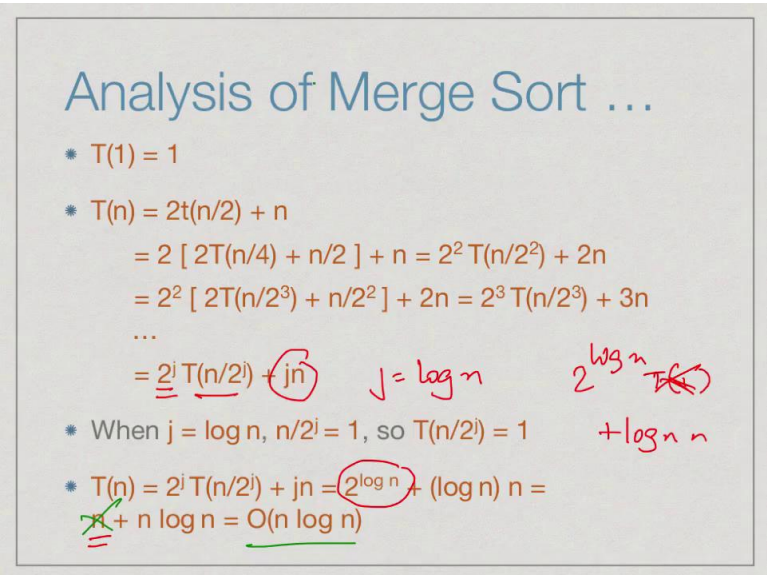
(Refer Slide Time: 12:15)



So, the analysis of merge sort a slightly more involved and what we have seen for the functions so for. So, let us since we keep dividing by 2 it is convenient to assume that original list of power of 2. So, let us assume for simplicity that the original list we want to sort is of length 2 to the k for some integer k, then the recurrence for merge sort says that to order to merge the list of length n we have to merge sort the front and the back.

So, we have to solve two sub problems of half the size and then merging takes linear time the some of the two input list. So, once again we can use unwinding to solve this recurrence is just the expression are little more complicated than we had seen for the earlier once.

(Refer Slide Time: 13:02)



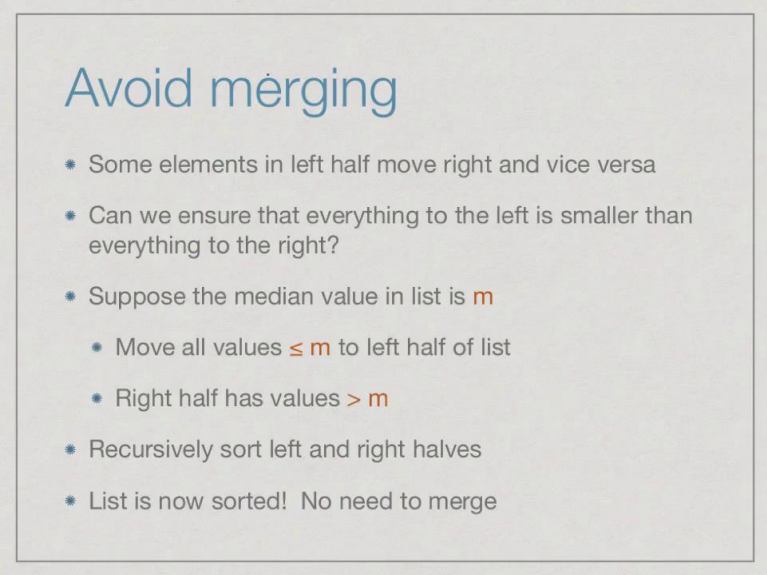
So, T of 1 is 1 and T of n is 2 t n by 2 plus n, so now we recursively expand 2 t n by 2 and we get 2 T n by 4 plus n by 2 and we will combine these twos and rewrite this fours. So, we will write this two times 2 as to 2 square we will write this 4 also is 2 square for a reason that will become clear and eliminate. So, now, we expand this n by 2 square and we will get another division by 2. So, we get 2 times T n by 2 cube plus n by 2 square.

Now, notice that this and this cancels so we get another end we already had to 2 n this n plus 2 n times n by 2. So, now, we will have 3 n and then 2 square into 2 this product will give us 2 cube start with three steps we have 2 to the power 3 T n by 2 to the power 3 plus 3 n. So, now, we see a pattern merging that this is three steps and we have a 3 here and 3 here and 3 here. So, it can check that if you do this j steps you get 2 to the j T n by 2 to the j plus j times n.

Now, this keeps going until this becomes 1 when does this becomes 1. So, n by 2 to the j is equal to 1; that means, 2 to the j is equal to n and other words j is the log of n to the base 2. So, when j is log of n then n by 2 to the j is 1 so we get T of 1, so this point we have 2 to the log n, because j is log n plus 1 times log n plus n times log n rather, because we have got j times n so now j is log n.

So, we have 2 to the log n from this term we have T of 1 plus log n times n and T of 1 is 1. So, this goes away, so we have 2 to the log n plus log n times n which is 2 to the log n by definition is n. So, n plus n log n, but then this is the smaller terms which process away and we get 4 of log n. So, merge sort takes times O of n log n you should remember that log n is much smaller function than n, so O n log n is actually a function which is much closer to O of n then 2 O of n square. So, merge sort is a significantly more efficient sorting algorithm then insertion sort or any other order n square sort it.

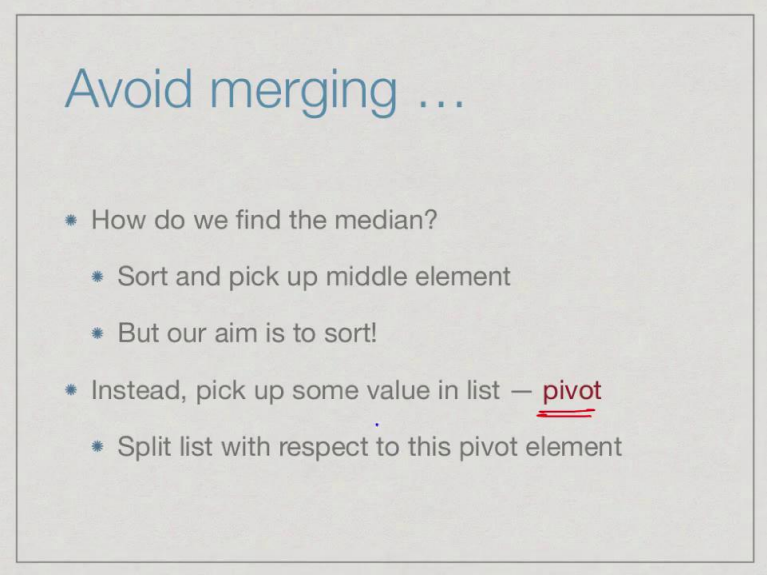
(Refer Slide Time: 15:42)



So, we can avoid this merging if we do not have to move elements between the left half and the right half after dividing the list in to two. So, convey ensure that everything on the left this already smaller than everything on the right, then if we sort the left and we sort the right we just have to stick them together. So, suppose a median value, the median value remembers is the value such that half the values are smaller and half the values are bigger.

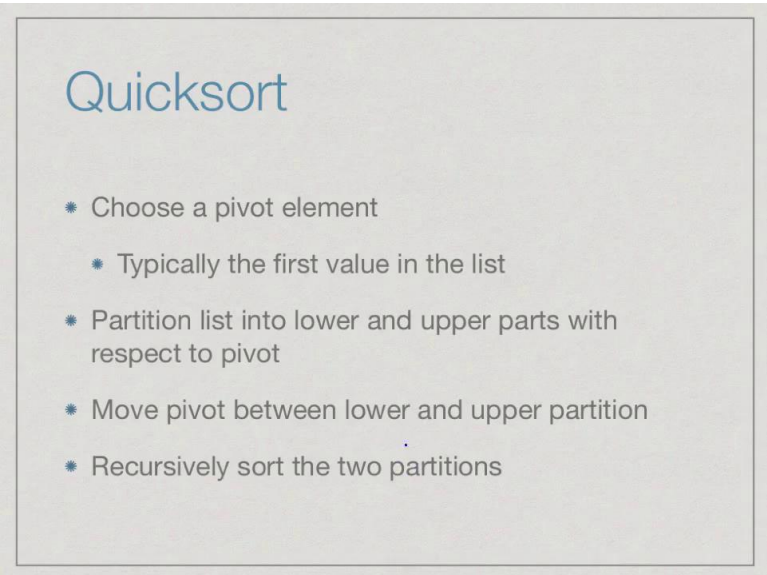
Suppose, the median values n, now if we take all the values less than or equal to m and to move it to the left and then we have half the values greater than m on the right and sort them then because m is the median value the right hand side lies strictly to the larger side then the left hand side. So, we can just use plus plus to combine these two from left and right, so I do not have to do any merge.

(Refer Slide Time: 16:49)



Now, the problem with this strategy is that it requires us to know the median and the standard way to find the median would be to sort the list and to pick up the middle element, but our aim is actually to sort the list. So, it is kind of circular to say that we have going to sort the list by finding the median. So, instance we will do this kind of splitting of the list with respect to some arbitrary line, we want to use the median exactly we will just pick up some value in the list and divided into value just smaller than this pivot and larger in this pivot. So, we pick up some value in the list call it pivot value and split the list of the split to this pivot element.

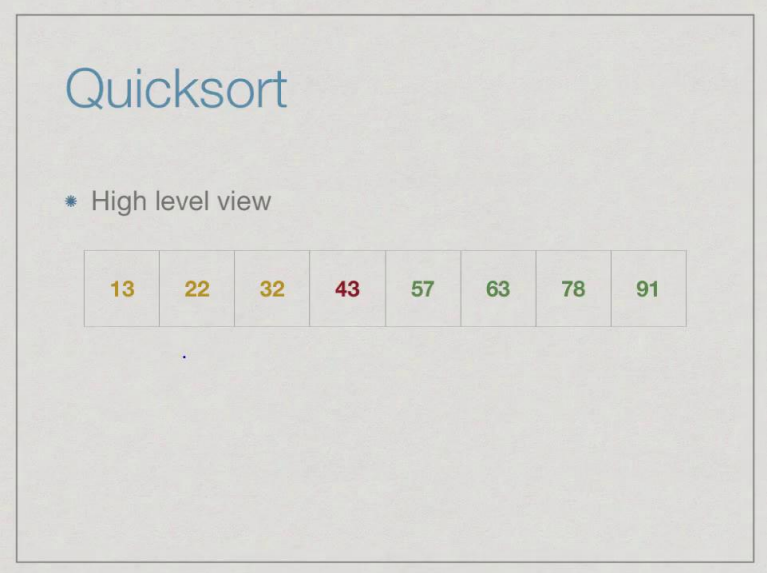
(Refer Slide Time: 17:28)



So, this algorithm is called quick sort and it is due to tony Hoare. So, you choose a pivot

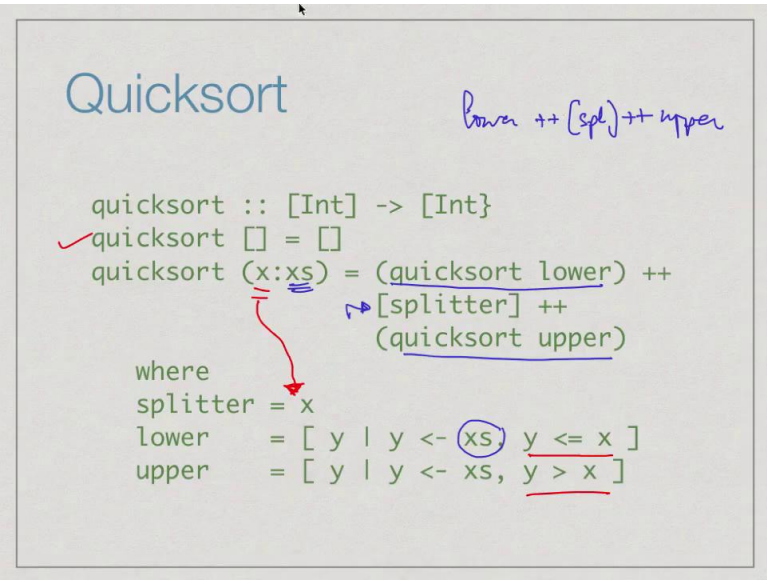
element, typically the first value in the list, we partition the list into lower and upper parts with respect to the pivot. So, the lower part is everything smaller than or equal to the pivot, the upper part is everything greater than the pivot. And now you sort of the two parts and move to the pivot in between and once you sort the two parts the pivot in between everything is automatically in order.

(Refer Slide Time: 18:02)



So, typically this is how quick sort would work, so you start with some arbitrary list and then you pick the first element say is that pivot. Now, you analyze the rest of the list and decide which one have to the left and which one has to the right. So, here princess the yellow values 32, 22 and 13 are smaller in the pivot and the green values are larger in the pivot. So, you re arrange the list, so that all the smaller values are to the left and all the larger values are to the right and now assuming that you can recursively sort those two, so you can sort the yellow and the green thing to get an overall sorted list.

(Refer Slide Time: 18:41)

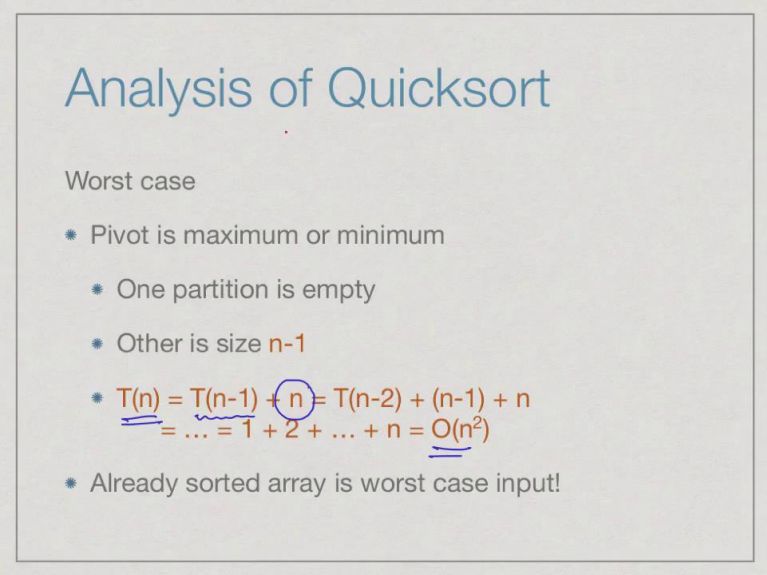


Quick sort in Haskell is extremely easy to write, so quick sort of the empty list is of course, the empty list, if I have a non empty list I pick this as the pivot. So, here is call this splitter, but it could also called the pivot and then I take this comprehension to take all the values which as smaller than or equal to the pivot and was splitter all those are greater than. Now, one important thing to note is that this comprehension is operating on the tail of this list.

So, the actual splitter thought it is less than or equal to itself is not be included lower, this is crucial; otherwise, we will end up or imagine into a situation where the list has a duplicate value, because we are going to put back this splitter here. So, this splitter is both in lower and is there on it is own then we have a problem. So, what we do is if we takes strictly those values excluding the splitter and we divide them into the lower and the upper, we recursively sort them using quick sort and then we put them into play.

So, we have the lower list and then we have the splitter, then we have the upper list and because the lower list is smaller than the splitter and the upper list bigger in the splitter no further merging is required.

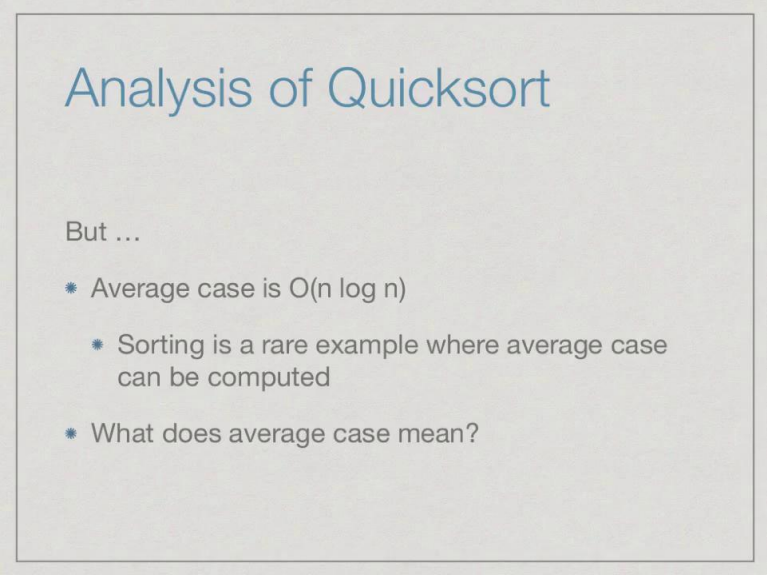
(Refer Slide Time: 19:56)



And the problem with this strategy is that we have no control over what value is at the beginning or wherever we choose the pivot. So, if the pivot happens to be the largest of the smallest value in the list then either the lower or the upper will become empty. So, then the recursive call to lower in sorting lower and upper is not n by 2, but it could be n minus 1. So, we may end up with this similar recurrence to insertion sort which says that in order to a sort list of n minus of n elements, we have to first split it.

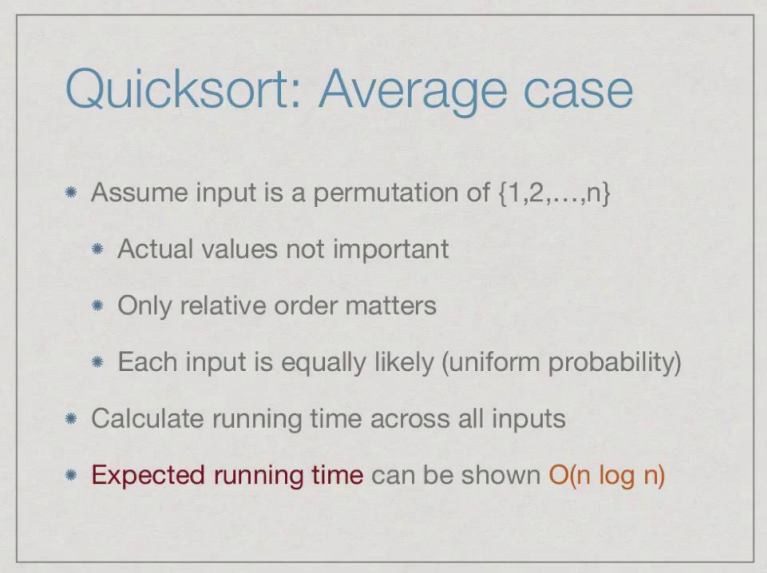
So, root is a splitting requires to walked on the list and move things lower and upper shows that splitting face requires order n times and then we may have recursively sort something, which is as large as n minus 1. And of course, we note that we expand this sub we get this 1 plus 2 plus summation of 2 n which is order n square. So, paradoxically is array which is already sorted for example, with the first element is that smallest value is a worst case input, because it is the smallest value and it to split the list as size 0 and size n minus 1. So, it appears the quick sort has not achieved anything, because we are got a worst case complexity which is as paired as insertion sort the first nice sorting algorithm that we discussed.

(Refer Slide Time: 21:11)



Now, turns out the quick sort or sorting in general is one of the rare situations, where we can actually compute the average case and the average case for quick sort turns out to be order n log n. So, let us just quickly look at what it means to compute the average case.

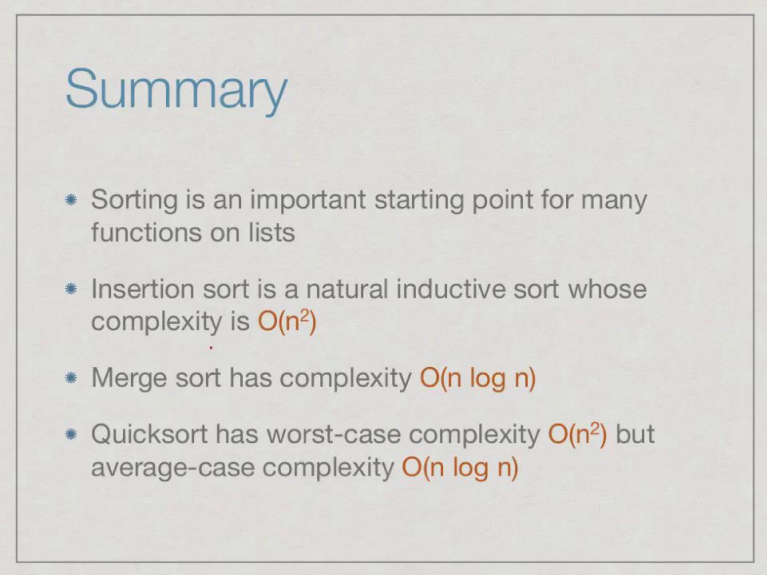
(Refer Slide Time: 21:30)



So, for sorting notice that the actual values that have been sorted or not so important as the relative order. So, we can always assume that if we are sorting the list of n elements that the elements are actually 1 to n and the actual list given to us is some permutation of 1 to n. So, therefore, the space of all n element inputs effectively becomes the set of all permutations of 1 to n and now we can assume that each of these equally likely. So, we can assign an sensible probability to each input saying it is 1 by n factorial.

And now we can calculate the run time across all these permutations and using standard probability theory. Although it involves a bit of calculation, you can actually show that the expected running time, which is the average that will weighting for is big O of n log n. So, this is why it is difficult to do in general, because for sorting we can kind of exhaustively characterize all the inputs of size n, but for more complicated functions it may not be so easy to do this and it may not be also so easy to assume a uniform distribution over all possible inputs and so on.

(Refer Slide Time: 22:41)



So, to summarize sorting is an important starting point for many functions on list. So, it is could to be able to sort a list efficiently, insertion sort is a natural inductive sort, but is complexity in the worst case is order n square. Merge sort on the other hand the uses divider concur and has a complexity of order n log n. Quick sort is a big simpler than merge sort, because we do not have to have a merge step we divide those things according to a pivot element and then we just paste the resulting list together, this has a worst case complexity of order n square. But, you can actually show that quick sort has an average case complexity of order n log n.