

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

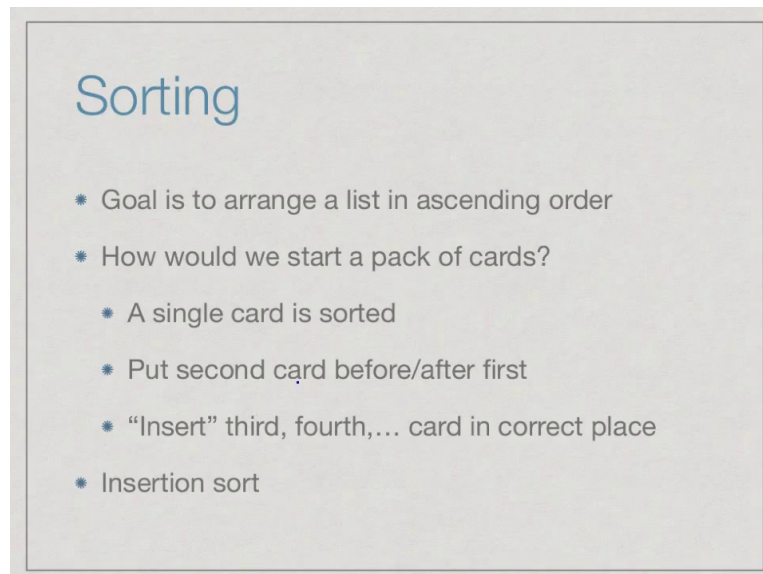
Module # 04

Lecture - 02

Sorting

Sorting a list is often an important prerequisite to doing other useful things on it. 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)



Sorting

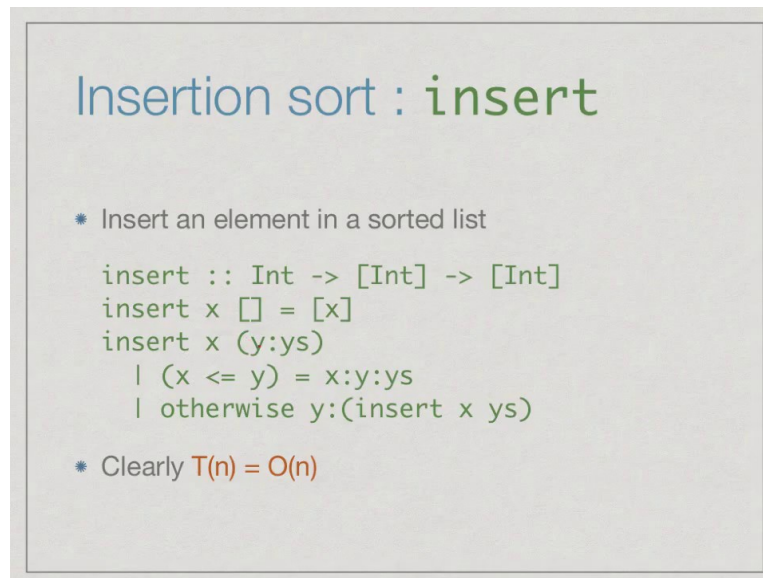
- * Goal is to arrange a list in ascending order
- * How would we start a pack of cards?
 - * A single card is sorted
 - * Put second card before/after first
 - * "Insert" third, fourth,... card in correct place
- * Insertion sort

So, our goal is to arrange a list in ascending or in descending order of values. So, let us just focus on ascending order because; obviously, in descending order everything will be symmetric using greater than instead of less than. So, to 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 that 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 its value with respect to the first card we picked up, we either put it above or below. So, continuing in this way we take the 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 built up, this algorithm is quite naturally called insertion sort.

(Refer Slide Time: 01:37)



Insertion sort : **insert**

- * Insert an element in a sorted list

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
  | (x <= y) = x:y:ys
  | otherwise = y:(insert x ys)
```

- * Clearly $T(n) = O(n)$

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 put in the correct place. So, the base case is to insert a value into an empty list which just produces a one element list consisting of that type.

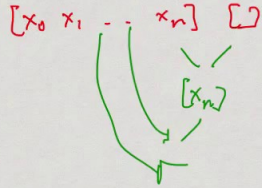
On the other hand, if you want to insert `x` into a non empty list of `ys`, we look at the first value and check whether or not `x` should come before it, if `x` is smaller than the smallest `y`, then we just take it up front, if `x` is not smaller than the smallest `y` then overall between the `x` and the `ys` the first `y` is the smallest value. So, we pull that out in front and recursively insert the `x` into the remaining `ys`. So, this in general would require us 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 $T(n)$ is $O(n)$.

(Refer Slide Time: 03:07)

Insertion sort : `isort`

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively
`isort = foldr insert []`



Now, we can express insertion sort in terms of this auxiliary function `insert` 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 sorted the tail we insert `x` into it in the appropriate position. So, the `insert` function does all the work. An 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 `[x0, x1 ...xn-1]` then we start with the empty list and then we insert `x n` into this and we get `xn`. And now we will take `xn-1` and insert it into this, then take `xn-2` and insert it into this and so on. So, we are just folding this `insert` function from right to left. So, a 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)

Insertion sort : `isort`

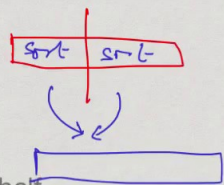
```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively
`isort = foldr insert []`
- Recurrence
 $T(0) = 1$
 $T(n) = T(n-1) + O(n)$
- Complexity: $T(n) = O(n^2)$

So, what is the complexity of insertion sort? Well, for the empty list $T(0)$ is 1 because in one step we get back the empty list and for the non empty case, we have to first sort $n-1$ elements and then having sorted $n-1$ elements, we have to insert a value into this list which takes $O(n)$ time. Because, remember the complexity of insert is $O(n)$ and therefore, the recurrence for insertion sort is say $T(0)$ is 1 and $T(n)$ is $T(n-1) + O(n)$. Now, we have seen this recurrence before and we know that if we expand it out, you will get $T(n)$ is $O(n^2)$, because we get something like $1+2+3+\dots+n$

(Refer Slide Time: 05:00)

A better strategy?



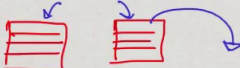
- Divide list in two equal parts
- Separately sort left and right half
- Combine the two sorted halves to get the full list sorted

So, can we do better than $O(n^2)$ 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 lists into a single over all sorted list.

(Refer Slide Time: 05:29)

Combining sorted lists

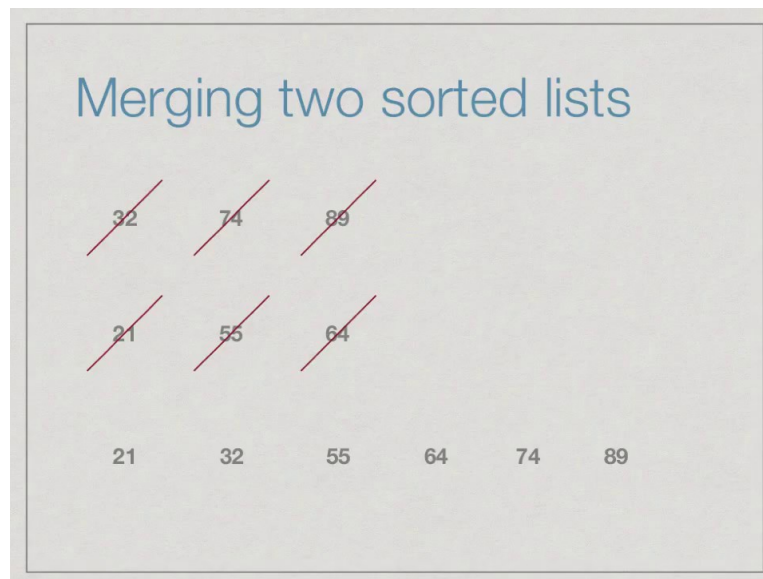


- * Given two sorted lists **l1** and **l2**, combine into a sorted list **l3**
- * Compare first element of **l1** and **l2**
- * Move it into **l3**
- * Repeat until all elements in **l1** and **l2** are over
- * Merging **l1** and **l2**

So, the final step requires us to combine two sorted lists l1 and l2 into a sorted list l3. Now, this is easy to do, because once again 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 lists 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 would work and note that this is l1, so this is sorted $32 < 74 < 89$ and this is l2, so this is also sorted. So, we start from the left we are looking at the left most element. So, since 21 is the smaller of the two, 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 two elements and we get 55 and then 64. And now of course, we have 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.

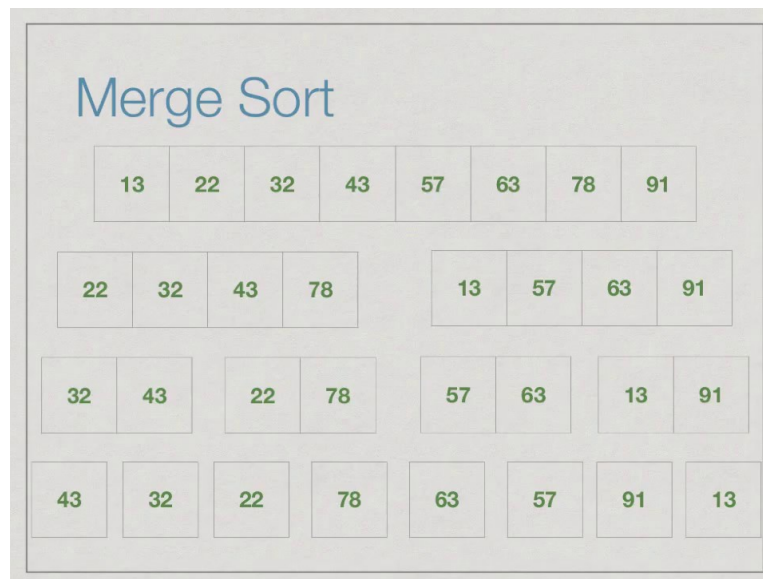
(Refer Slide Time: 06:59)



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 i th element. So, it says sort from 0 to the midpoint, the midpoint to the end, so this should be a round bracket. So, we sort from 0 to $n/2 - 1$, from $n/2$ to $n-1$, and merge sort combines sorted halves into a new list l' . 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)



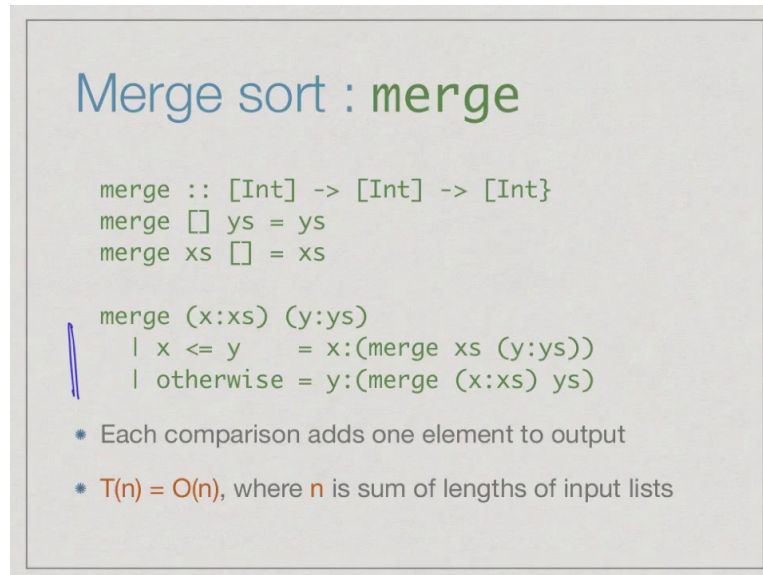
So, let us run through how merge sort would work on an arbitrary list, so the first step you will divide at the midpoint. So, we will divide it into two lists and try to sort them recursively. So, we will get the left list and the right list, now in turn we will divide each of these into two, because we are applying the same strategy. So, we get two lists from the left and two lists from the right, now we pretend that we still don't know how to sort. So, we will move it one more step. So, we will split each of this list of length 2 into two, so we will get individual lists of length 1.

And now remember that a list of length 1 is by definition sorted, because it is only one value and it must be in order and now we can start merging. So, you want to merge these two, merge these two, merge these two, merge these two to get merged list which are sorted of length 2. So, if we merge 43 and 32, we get 32 followed by 43, the second pair gives us the same lists as before, the third pair again gets inverted, and 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 of length 2 to get two sorted lists of length 4. And finally, we merge the two sorted lists of length 4 to get sorted list of length 8. So, this is how merge sort works we

divide, divide, divide until we get singletons and then we work backwards merging, merging, merging until we get back the final sorted list.

(Refer Slide Time: 09:15)



The slide is titled "Merge sort : merge" in a blue and green font. It contains Haskell code for a merge function. The code is as follows:

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs

merge (x:xs) (y:ys)
  | x <= y    = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)
```

Below the code, there are two bullet points:

- Each comparison adds one element to output
- $T(n) = O(n)$, where n is sum of lengths of input lists

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 to be sorted and produces a new output which is the combination of the two sorted lists. So, we have base cases . If the first list or the second list is empty, we can just copy the other list without doing anything. Because, it is already sorted we just tag it along. On the other hand if we have something to do and 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 xs with the remaining ys and then we stick x as the new first element of the overall list.

So, x is the smallest over all, if x is not the smaller one then y is the smallest over all, so this is a very direct translation of our 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 lists put together. So, we can say that the complexity of merge is $O(n)$, where n is the sum of the lengths of the two input lists.

(Refer Slide Time: 10:42)

The slide is titled "Merge sort" in blue. To the right of the title, there is a handwritten diagram in blue ink. It shows a list `[x]` with an arrow pointing down to `[]`, which then points to `[x]` again, forming a loop. Another arrow points from `[x]` to a circle containing `[x]`. Below the title, there is Haskell code for a merge sort function. The code is as follows:

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort (front l))
                   (mergesort (back l))

where
  front l = take ((length l) `div` 2) l
  back l = drop ((length l) `div` 2) l
```

Handwritten annotations include a blue 'X' next to the `mergesort [x] = [x]` line and a blue arrow pointing to it from the right.

Once we have 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 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 of the elements. And remember that take and drop supplied with the same argument exhaustively enumerate all the elements. So, take ++ drop is always the list itself.

So, we can be sure that we are not losing any elements or missing out or duplicating elements. we are taking the length of l integer divided by 2. And then we dropping the same number of elements and using that as the back half of the list and having now constructed the sorted versions of the front and the back we are merging it. 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 need a base case for the singleton list as well; otherwise, we will end up with the an infinite loop when we come to the singleton and try to split into back and front.

(Refer Slide Time: 12:15)

Analysis of Merge Sort

- $T(n)$: time taken by Merge Sort on input of size n
 - Assume, for simplicity, that $n = 2^k$
- $T(n) = 2T(n/2) + n$
 - Two subproblems of size $n/2$
 - Merging solutions requires time $O(n/2 + n/2) = O(n)$
 - Solve the recurrence by unwinding

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

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

(Refer Slide Time: 13:02)

Analysis of Merge Sort ...

- $T(1) = 1$
- $T(n) = 2T(n/2) + n$
 - $= 2 [2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n$
 - $= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n$
 - ...
 - $= 2^j T(n/2^j) + jn$ $j = \log n$ $2^{\log n} = n$
- When $j = \log n$, $n/2^j = 1$, so $T(n/2^j) = 1$ $+ \log n$
- $T(n) = 2^j T(n/2^j) + jn = n + (\log n) n = n + n \log n = O(n \log n)$

So, $T(1)$ is 1 and $T(n)$ is $2T(n/2) + n$, so now we recursively expand $2T(n/2)$ and we get $[2T(n/4) + n/2]$ and we will combine these twos and rewrite this fours. So, we will write this two times 2 as 22, we will write this 4 also as 22 for a reason that will become clear in a minute. So, now, we expand this n by 22 and we will get another division by 2. So, we will get $2T(n/23) + n/22$.

Now, notice that this and this cancels so we get another n . we already had 2 n , this n plus 2 n times n by 2. So, now, we will have 3 n and then 22 into 2 this product will give us 23. Start with three steps we have 23 $T(n/23) + 3n$. So, now, we see a pattern emerging that this is three steps and we have a 3 here and 3 here and 3 here. So, you can check that if you do this j steps you get $2^j T(n/2^j) + jn$.

Now, this keeps going until this becomes 1. when does this become 1? So, $(n/2^j)$ is equal to 1; that means, 2^j is equal to n and other words j is the $\log_2 n$. So, when j is $\log n$, then $n/2^j$ is 1 so we get $T(1)$, so this point we have $2\log n$, because j is $\log n + 1 * \log n + n * \log n$ rather, because we have got j times n so now j is $\log n$.

So, we have $2\log n$ from this term we have $T(1) + \log n * n$ and $T(1)$ is 1. So, this goes away, so we have $2\log n + (\log n) * n$. $2\log n$ by definition is n . So, $n+n \log n$, but then this is the smaller terms ,we throw this away and we get $O(n \log n)$. So, merge sort takes time $O(n \log n)$. We 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 $O(n)$ than to $O(n^2)$. So, merge sort is a significantly more efficient sorting algorithm than insertion sort or any other $O(n^2)$ sort.

(Refer Slide Time: 15:42)

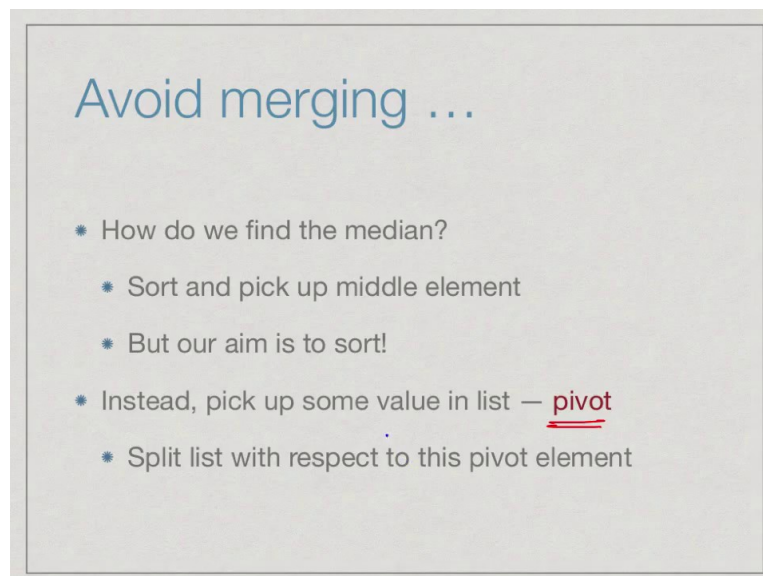
Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?
- Suppose the median value in list is m
 - Move all values $\leq m$ to left half of list
 - Right half has values $> m$
- Recursively sort left and right halves
- List is now sorted! No need to merge

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 into two. So, can we ensure that everything on the left is 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 remember is the value such that half the values are smaller and half the values are bigger.

Suppose, the median value is n , now if we take all the values $\leq n$ and move it to the left and then we have half the values $> n$ on the right and sort them then because n is the median value. The right hand side lies strictly to the larger side than the left hand side. So, we can just use ++ to combine these two from left and right, so I do not have to do any merge.

(Refer Slide Time: 16:49)



Avoid merging ...

- How do we find the median?
 - Sort and pick up middle element
 - But our aim is to sort!
- Instead, pick up some value in list — pivot
 - Split list with respect to this pivot element

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 are going to sort the list by finding the median. So, instead we will do this kind of splitting of the list with respect to some arbitrary value, we won't use the median exactly we will just pick up some value in the list and divide it into values just smaller than this pivot and larger than this pivot. So, we pick up some value in the list call it a pivot value and split the list with respect to this pivot element.

(Refer Slide Time: 17:28)

Quicksort

- * Choose a pivot element
 - * Typically the first value in the list
- * Partition list into lower and upper parts with respect to pivot
- * Move pivot between lower and upper partition
- * Recursively sort the two partitions

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 the two parts and move the pivot in between and once you sort the two parts with the pivot in between everything is automatically in order.

(Refer Slide Time: 18:02)

Quicksort

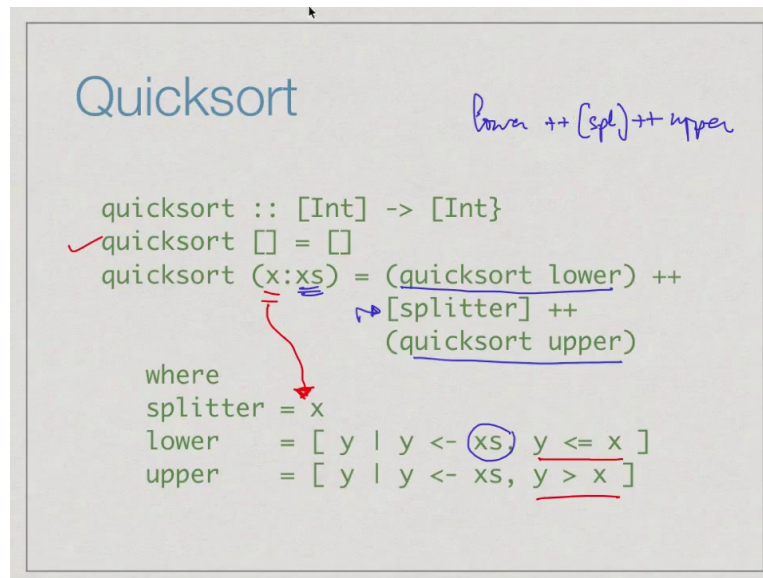
- * High level view

13	22	32	43	57	63	78	91
----	----	----	----	----	----	----	----

So, typically this is how quick sort would work, so you start with some arbitrary list and then you pick the first element say as the pivot. Now, you analyze the rest of the list and decide which ones are to the left and which ones are to the right. So, here for instance the yellow

values 32, 22 and 13 are smaller than the pivot and the green values are larger than 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 its called the splitter, but it could also be called the pivot and then I take list comprehension to take all the values which are smaller than or equal to the pivot or the splitter and all those that are greater than pivot. Now, one important thing to note is that this comprehension is operating on the tail of this list.

So, the actual splitter though it is less than or equal to itself is not be included in lower, this is crucial; otherwise, we will end up as you might 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 we take 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 than the splitter no further merging is required.

(Refer Slide Time: 19:56)

Analysis of Quicksort

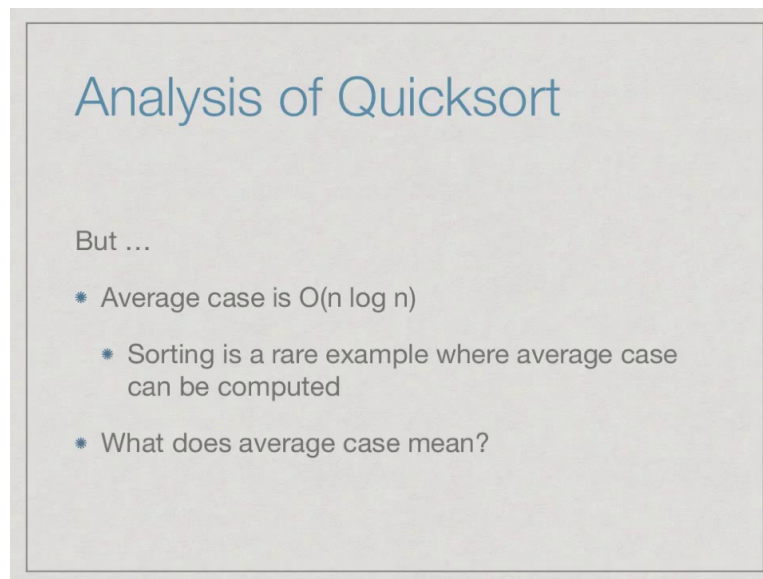
Worst case

- Pivot is maximum or minimum
 - One partition is empty
 - Other is size $n-1$
- $T(n) = T(n-1) + n = T(n-2) + (n-1) + n$
 $= \dots = 1 + 2 + \dots + n = O(n^2)$
- Already sorted array is worst case input!

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 or 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 $n/2$, but it could be $n-1$. So, we may end up with a similar recurrence to insertion sort which says that in order to sort a list of n elements, we have to first split it.

So, what splitting requires us is to walk down the list and move things to lower or upper, shows that splitting phase requires $O(n)$ time and then we may have to recursively sort something, which is as large as $n-1$. And of course, we know that we expand this out we get this $1+2+\dots+n$ which is $O(n^2)$. So, paradoxically an array which is already sorted for example, where the first element is that smallest value is a worst case input, because it is the smallest value and it will split the list as size 0 and size $n-1$. So, it appears that quick sort has not achieved anything, because we have got a worst case complexity which is as bad as insertion sort, the first naive sorting algorithm that we discussed.

(Refer Slide Time: 21:11)



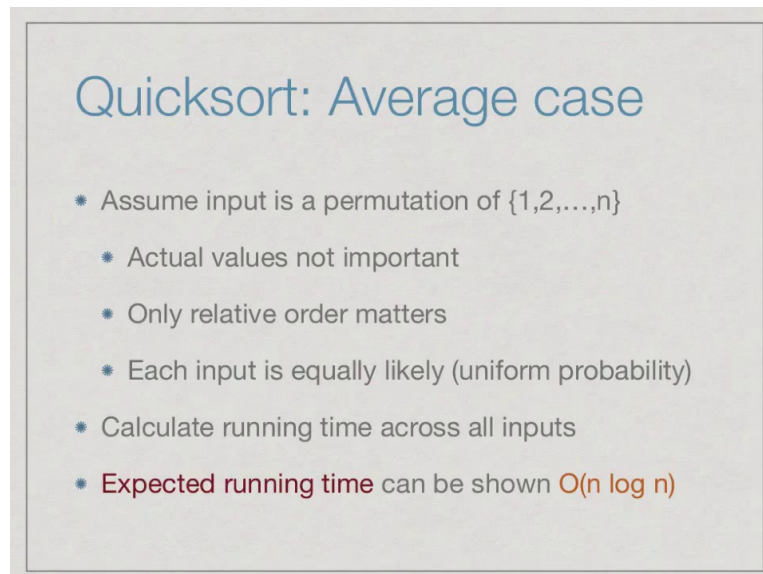
Analysis of Quicksort

But ...

- Average case is $O(n \log n)$
 - Sorting is a rare example where average case can be computed
- What does average case mean?

Now, it turns out that 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 $O(n \log n)$. So, let us just quickly look at what it means to compute the average case.

(Refer Slide Time: 21:30)



Quicksort: Average case

- Assume input is a permutation of $\{1, 2, \dots, n\}$
 - Actual values not important
 - Only relative order matters
 - Each input is equally likely (uniform probability)
- Calculate running time across all inputs
- **Expected running time** can be shown $O(n \log n)$

So, for sorting notice that the actual values that have been sorted are not so important as their relative order. So, we can always assume that if we are sorting a 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 is equally likely. So, we can assign a sensible probability to each input saying it is $1/n!$.

And now we can calculate the running 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 we are looking for is $O(n \log n)$. 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)



Summary

- * Sorting is an important starting point for many functions on lists
- * Insertion sort is a natural inductive sort whose complexity is $O(n^2)$
- * Merge sort has complexity $O(n \log n)$
- * Quicksort has worst-case complexity $O(n^2)$ but average-case complexity $O(n \log n)$

So to summarize, sorting is an important starting point for many functions on list. So, it is good to be able to sort a list efficiently, insertion sort is a natural inductive sort, but its complexity in the worst case is $O(n^2)$. Merge sort on the other hand uses divide and conquer and has a complexity of $O(n \log n)$. Quick sort is a bit 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 lists together, this has a worst case complexity of $O(n^2)$. But, you can actually show that quick sort has an average case complexity of $O(n \log n)$.