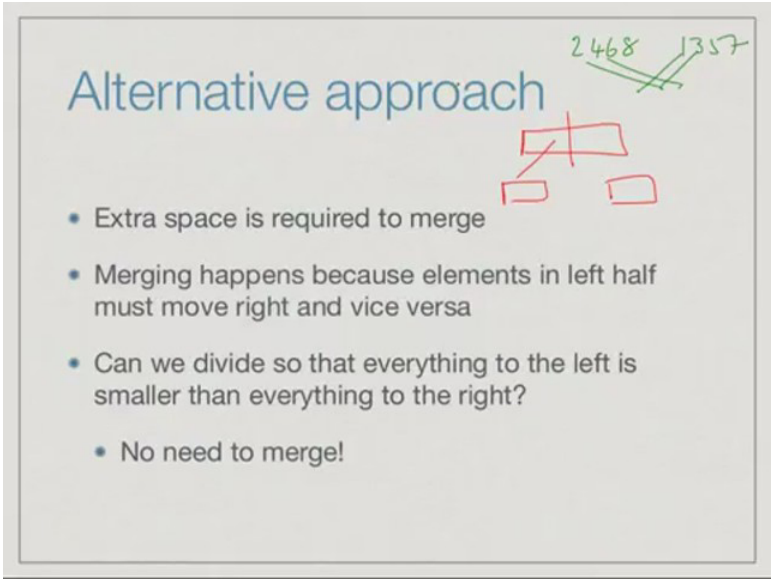
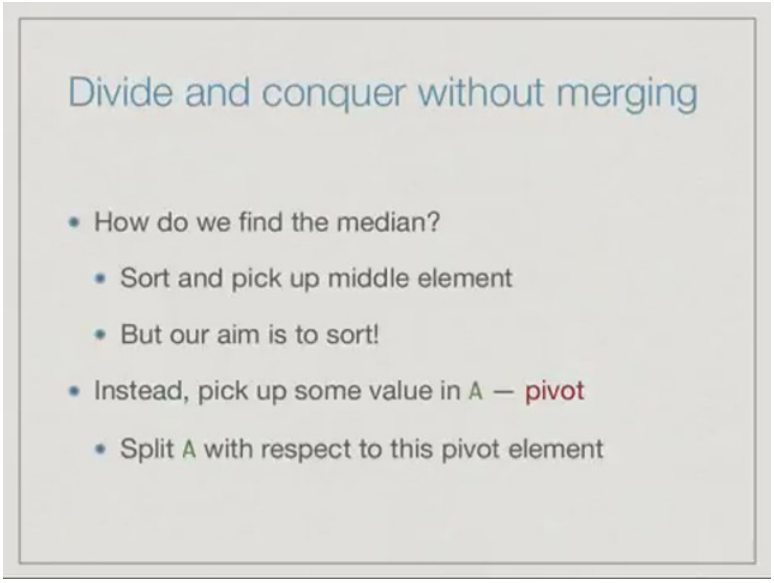
(Refer Slide Time: 00:23)



Let us address the space problem. The extra space required by merge sort is actually required in order to implement the merge function and why do we need to merge? The reason we need to merge is that when we do a merge sort, we have the initial list and then we split it into two parts, but in general there may be items in the left which are bigger than items in the right.

For instance, if we had say even numbers in the left and the odd numbers on the right then we have to merge by taking numbers alternatively from either side. So, if we could arrange that everything that is on the left side of our divided problem is smaller than everything on the right side of the divided problem, then we would not need to merge at all and this perhaps could save us this problem of requiring extra space for the merge.

(Refer Slide Time: 01:19)



How would we do divide and conquer without merging. Assume that we knew the median value; remember the median value in a set is the value such that half the elements are smaller and half are bigger. We could move all the values smaller than the median to the left half and all of those bigger than the median to the right half. As we will see this can be done without creating a new array in time proportional to the length of the list.

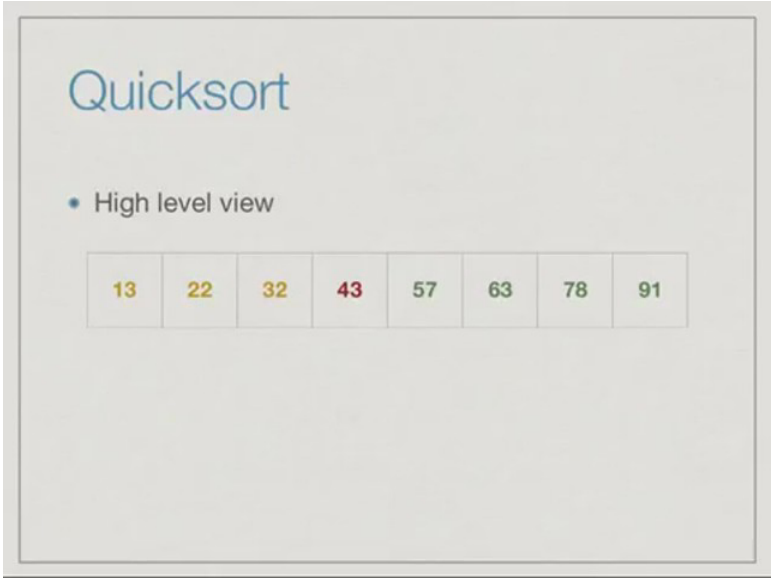
Having done this rearrangement moving all the smaller values to the left half and the bigger values to the right half then we can recursively apply this divide and conquer strategy and sort the right and the left half separately and since we have guaranteed that everything in the left half is smaller than everything in the right half, this automatically means that after this divide and conquer step we do not need to combine the answers in any non trivial way because the left half is already below the right half. So, we do not need to merge.

If we apply this strategy then we would get a recursive equation exactly like merge sort. It would say that the time required to sort a list of length n requires us to first sort two lists of size n by 2 and we do order n not for merging, but in order to decompose the list so that all the smaller values are in the left and in the right. So, rearranging step before

we do the recursive step is what is order n, whereas merge was the step after the recursive step which was order n in the previous case, but if we solve the recurrence, its the same one, we get another order n log n algorithm.

The big bottleneck with this approach is to find the median. Remember that we said earlier that one of the benefits of sorting a list is that we can identify the median as the middle element after sorting. Now here, we are asking for the median before sorting, but our aim is to sort, it is kind of paradoxical. If we are requiring the output of the sorting to be the input to the sorting. This means that we have to try the strategy out with a more simplistic choice of element to split the list. Instead of looking for the median we just pick up some value in the list A, and use that as what is called a pivot element. We split A with respect to this pivot so that all the smaller elements are to the left and all the bigger elements are to the right.

(Refer Slide Time: 03:58)



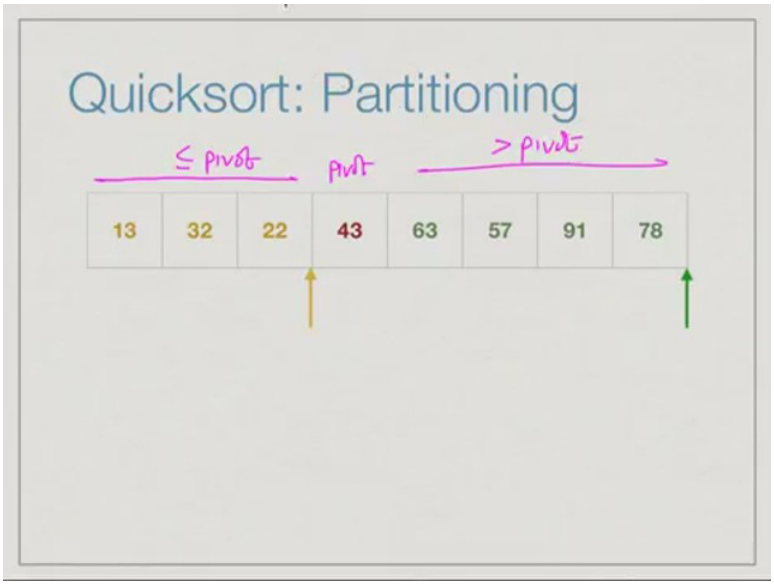
This algorithm is called Quicksort, it was invented by a person called C.A.R Hoare in the 1960s and is one of the most famous sorting algorithms. So, we choose a pivot element which is usually the first element in the list of the array. We partition A, into the lower part and the upper part with respect to this pivot element. So, we move all the smaller elements to the left and all the bigger elements to the right with respect to the choice of

pivot element, and we make sure the pivot comes between the two because we have picked up the first element in the array to pivot. So, after this we want to move it to the center between the lower and the upper part and then, we recursively sort two partitions.

Here is a high level view of how quicksort will work on a typical list. Suppose this is our list, we first identify the beginning of the list, the first element as the pivot element. Now, for the remaining elements we have to figure out which ones are smaller and which ones are bigger. So, without going into how we will do this, we end up identifying 32, 22 and 13 as three elements which are smaller and marked in yellow and the other four elements which are marked in green are larger.

The first step is to actually partition with respect to this criterion. So, we have to move these elements around so that they come into two blocks. So that, 13, 32 and 22 come to the left; 63, 57, 91 and 78 come to the right and the pivot element 43 comes in middle. This is the rearranging step and now we recursively sort the yellow bits and the green bits, then assuming we can do that, we have a sorted array and notice that since all the yellow things are smaller than 43 and all the green things are bigger than 43, no further merging is required.

(Refer Slide Time: 05:51)



So let us look at how partitioning works. Here, we have the earlier list and we have marked 43 as our pivot element and we want to do a scan of the remaining elements and divide them into two groups; those smaller than 43, the yellow ones; those bigger than 43, the green ones and rearrange them. What we will do is we will keep two pointers; a yellow pointer and a green pointer and the general rule will be that at any given point we will have at some distance, the yellow pointer which I will draw in orange to make it more visible and the green pointer.

These will move in this order; the orange pointer or the yellow pointer will always be behind the green pointer and the inductive property that we will maintain is that these elements are smaller than or equal to 43, these elements are bigger than 43 and these elements are unknown. What we are trying to do is, we are trying to move from left to right and classify all the unknown elements; each time we see an unknown element we will shift the two pointers so that we maintain this property that between 43 and the first pointer we have the elements smaller than or equal to 43; between the first pointer and the second pointer we have the element strictly greater than 43 and to the right of the green pointer, we have those which are yet to be scanned.

Initially nothing is known then we look at 32, since 32 is smaller than the 43, we move the yellow pointer and we also push the green pointer along. So, the unknown things start from 22, and there is nothing between the yellow and the green pointer indicating we have not yet found the value bigger than 43, same happens for 22. Now, when we see 78, we notice that 78 is bigger than 43. Now, we move only the green pointer and not yellow pointer, we have these three intervals as before. Remember that this is the part which is less than equal to 43; this is the part that is greater than 43 and this part is unknown. We continue in this way.

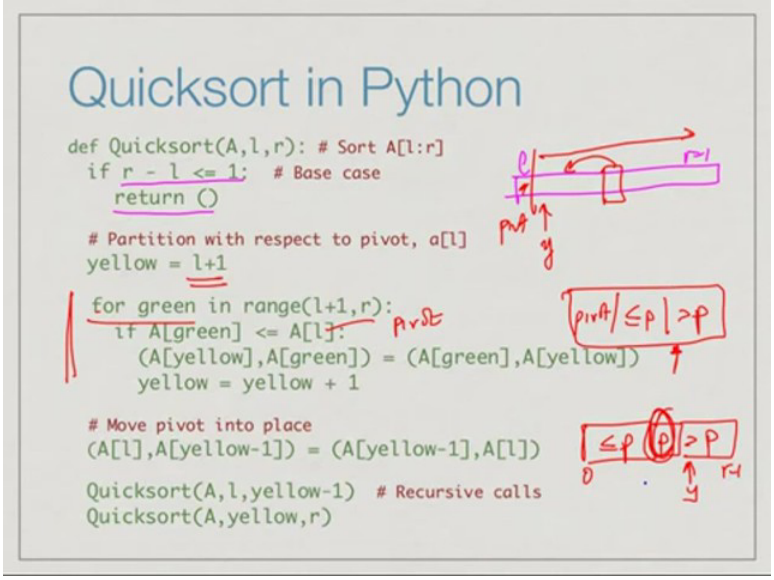
Now, we look at 63, again 63 extends the green zone, 57 extends the green zone, 91 extends the green zone. Now, we have to do something when we find 13. So, 13 is an element which has to be put into the yellow zone, one strategy would be to do a lot of shifting. We move 13 to where 22 is or after 22 and we push everything from 78 onwards to the right, but actually a cleverer strategy is to say that 13 must go here. So, we need to make space, but instead of making space we can say, it does not matter to us, we are

eventually going to sort the green things anyway.

How does it matter which way we sort that, we will take this 78 and just move it to 13. So, instead of doing any shifting, we just exchange the first element in the green zone with the element we are seeing so far, that automatically will extend both yellow zone and the green zone correctly. So, our next step is to identify 13 as smaller than 43 and swap it with 78. Now, we have reached an intermediate stage where to the right of the pivot we have scanned everything and we have classified them into those which are the smaller ones and those which are the bigger ones.

Now, it remains to push the yellow things to the left of 43. Once again we have the same problem we saw when we included 13 in the yellow zone. If we move 43 to the correct place then we have to move everything here to the left, but instead we can just take this 13 in the last element to the yellow zone and replace it there and not shift 32 and 22. This disturbs the order, but anyway this is unsorted, it just remains unsorted. So, we do this and now we have the array rearrange as we wanted, all of these things to the left are smaller than the pivot the pivot is in the middle and everything to the right is bigger than the pivot.

(Refer Slide Time: 09:47)



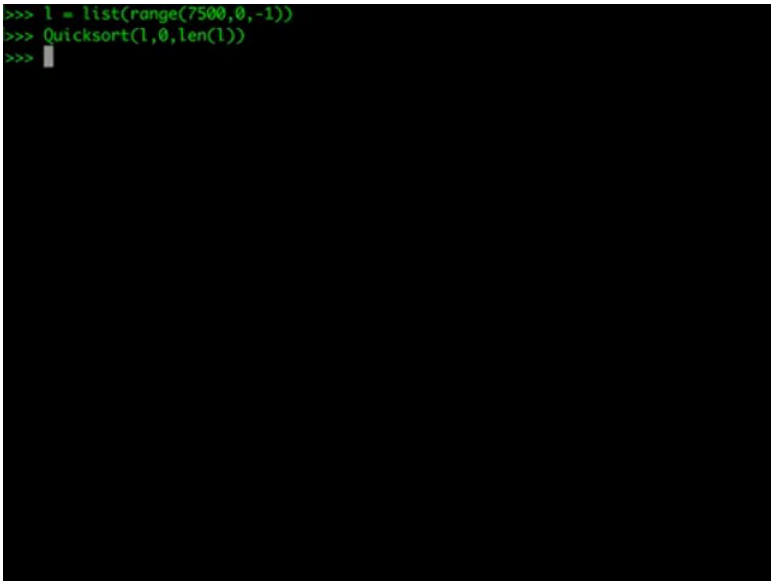
Here is an implementation in Python. So, remember that quicksort is going to have like merge sort and like binary search, we repeatedly apply it in smaller and smaller segments. In general, we have to pass to it the list which we call A, and the end points to the segment the left and the right. If we have something that we are doing a slice l to r minus 1, if this slice is 1 or 0 in length, we do nothing otherwise we follow this partitioning strategy we had before, which is that we are sorting from l to r minus 1. The position l, this is the pivot.

We will initially put the yellow pointer here, saying that the end of the yellow zone is actually just the pivot, there is nothing there. So, yellow is l plus 1 and now we let green proceed and every time you see an element in the green the new green one which is smaller than the one which is the pivot. Remember this is the pivot, if ever we see a green the next value to be checked is smaller than or equal to A[l] we exchange so that we bring this value to the end of the yellow zone.

This is what we did to 13 and then we move the yellow pointer as well, otherwise if we see a value which is strictly bigger, we move only the green pointer which is implicitly done by the for loop and we do not move the yellow. At the end of this, we have the pivot then we have the less than equal to pivot and then we have the greater than. So, this is that intermediate stage that we have achieved at the end of this loop. Now, we have to find the pivot and move it to the correct place.

Remember that the yellow, yellow is pointing to the position beyond the last element smaller than that. So, yellow is always one value before, beyond this. So, we take the yellow minus 1 value and exchange it with the left value and now what we need to do is we have now less than p, p, greater than p and this is where yellow is. So, we need to go from 0 to yellow minus 1, we do not want to sort p again. Because p is already put in the correct place, so we quicksort from l to yellow minus 1 and from yellow to the right end.

(Refer Slide Time: 12:11)



Here, we have written the Python code that we saw in the slide in a file. You can check that it is exactly the same code that we had in the slide. We can try and run it and verify that it works. So, we call Python and we import this function. Remember that this is again a function which sorts in place. If you want to sort something and see the effect we have to assign it a name and then sort that name and check the name afterwards. Let us, for instance, take a range of values from say 500 down to 0 then if we say quicksort(l) then we have to of course, give it the end then l gets correctly sorted.

So, you cannot see all of it, but you can see from 83, 84 up to 102 up to 500. Now, we have the same problem that we had with insertion sort. If we say 1000 and then we try to quicksort this, we will get this recursion depth because as we will see, in the worst case actually, quicksort behaves a bit like insertion sort and this is a bad case. So, to get around this we would have to do the usual thing - we have to import the sys module and set the recursion limit to something superbly large, say 10000, maybe 100,000 and then if we ask it to quicksort there is no problem.

This is another case where this recursion limit in python has to be manually set and one thing we can see actually is that quicksort is not as good as we believe because if we were to, for instance, sort something of size say 7500 then it takes a visible amount of

time. We saw that merge sort which was n log n could do 5000 and 10000 and even 100,000 instantaneously.

So, clearly quicksort is not behaving as well as merge sort and we will see in fact, that quicksort does not have an order n log n behavior as we would have liked and that is because we are not using the median, but the first value to speak. We will see that in the next lecture as to why quicksort is actually not a worst case order n log algorithm.