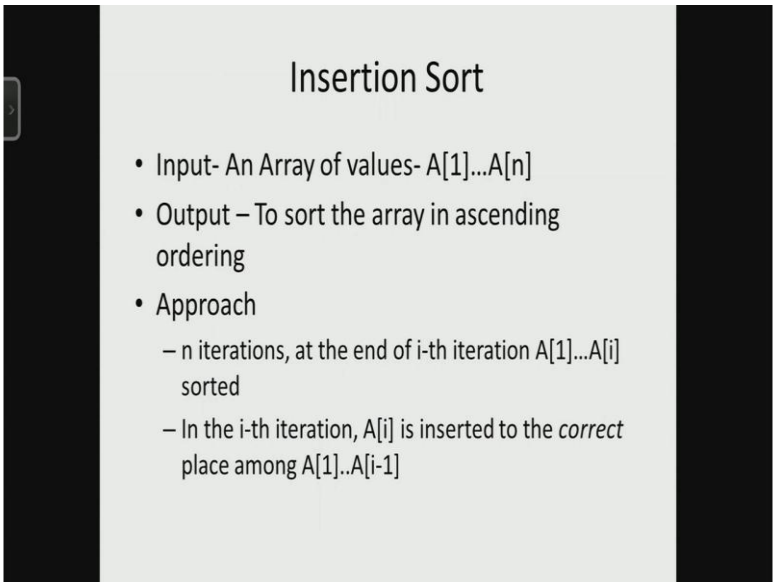
Programming, Data Structures and Algorithms
Prof. N.S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 05
Lecture - 32
Sorting
Insertion sort-Correctness and running time analysis
Merge sort Correctness and running time analysis

So this lecture is on algorithms for sorting, and the focus of the lecture is going to be not just in the algorithms, but the sequence of steps that go into arguing that the algorithm is indeed correct and analyzing the running time.

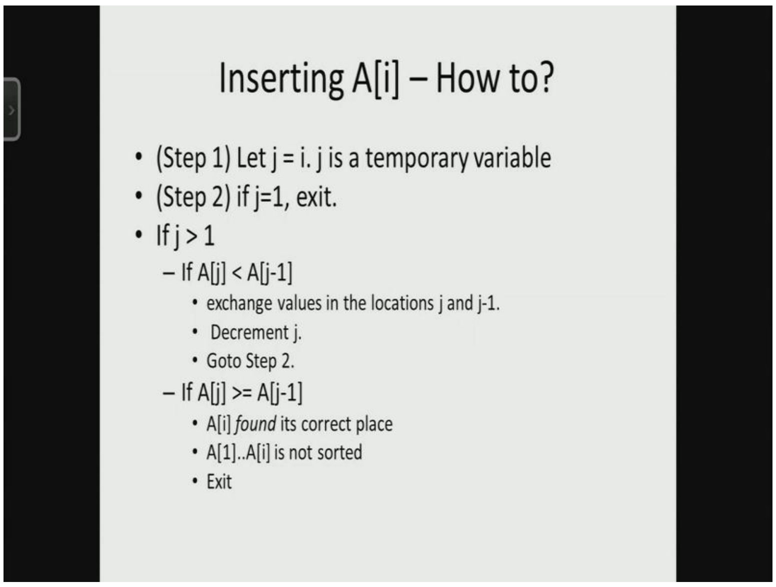
(Refer slide Time: 00:31)


Insertion Sort

- Input- An Array of values- A[1]...A[n]
- Output – To sort the array in ascending ordering
- Approach
  - n iterations, at the end of i-th iteration A[1]...A[i] sorted
  - In the i-th iteration, A[i] is inserted to the *correct* place among A[1]..A[i-1]

Another simplest algorithms to sort a given data set, and in this case we assume that the input is given in an array of size n, with the elements are a of 1 to a of n. The data type of the elements in the array is not so important for the rest of the presentation, as long as it is clear that there is a very clear order between every pair of elements. The output of sorting algorithm is to sort the array in ascending order, and insertion sort is a very simple and popular algorithm to understand the challenge of sorting. The approach that is taken by insertion sort, is the following. It is an iterative approach, which means there is a loop that is running n times. Recall that n is a number of elements in the array that need to be sorted, and at the end of every iteration. Let us say at the end of the ith iteration, the invariant that is maintain by the algorithms, is that the elements form the first i elements in the array; that is elements a of 1 to a of i are sorted in ascending order. Further in the ith iteration, the elements a of i is inserted into the correct location, among the elements a of 1 to a of i minus 1, and thus this invariant is maintain one iteration after another. This kind of guaranty is the correctness of the insertion and sort algorithm, to sort an array of n elements.

(Refer Slide Time: 02:14)



Inserting A[i] – How to?

- (Step 1) Let j = i. j is a temporary variable
- (Step 2) if j=1, exit.
- If j > 1
  - If A[j] < A[j-1]
    - exchange values in the locations j and j-1.
    - Decrement j.
    - Goto Step 2.
  - If A[j] >= A[j-1]
    - A[i] *found* its correct place
    - A[1]..A[i] is not sorted
    - Exit

Of course is challenge here, is to understand how the insertion happens. Recall that in the previous slide we mentioned that, in the ith iteration the elements a of i is inserted into the correct location, so that a of 1 to a of i is sorted. Given that a of 1 to a of i minus 1 is sorted when the control enters the ith iteration. So, how does one insert a of i. One can look at the step by step procedure. We said first j to be a temporary variable, and then whenever j takes the value one you exit. We will see what the motivation for this condition is, and as long as j is more than one; a comparison is made of the elements a of j and a of j minus 1.

In other words, the elements which are in the array locations j and j minus 1 are compared, and if a of j is smaller than the a of j minus 1. Then there is an exchange that is effective, between the elements and the location j and j minus 1. And now j is decremented by one and then we repeat this procedure all over again. Indeed it is also possible that the elements at the jth location is at least as large as elements at the j minus one th location in which case, it is clear that a of j is the correct place, given that the rest of the elements are in sorted order. At this point of time the control exit's, and now we can report that the elements a of 1 to a of i are in sorted order.

(Refer Slide Time: 04:15)

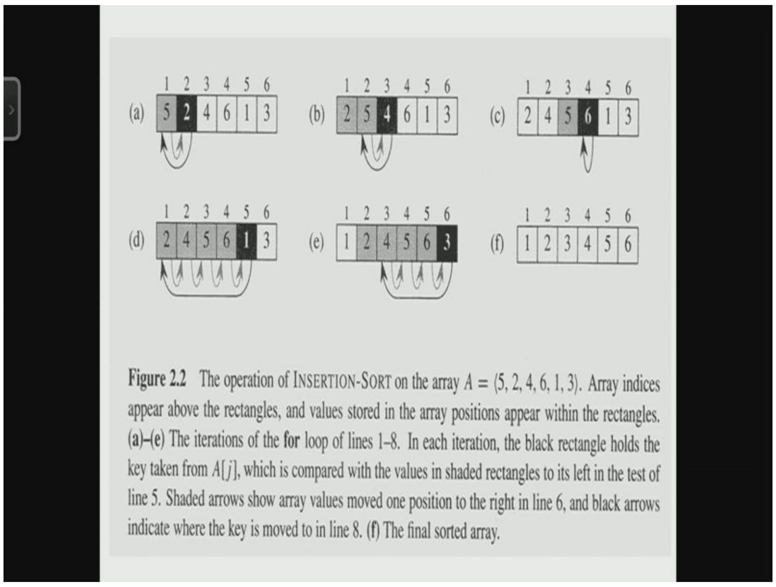
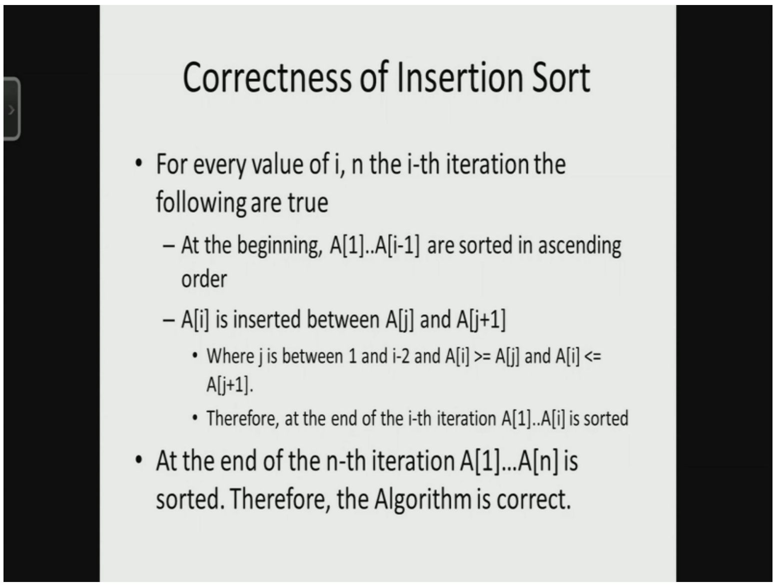


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

In other words we achieve the goal of inserting a of i into the correct position, which is look at this pictorially, this is taken from corner license and drive is standard text book. And let us look at the starting array, which is array index by the letter a and observe that, there are six elements in this array which sort out as 5 2 4 6 1 and 3, observe that in the first step. In our cases would be the second step the way of presented the algorithm, the elements two and five are exchanged. In the array index by the letter b, observe that the first two elements are in sorted order, and the elements four is not being inserted into it is correct position, and this is required a single comparison with four and five, but of course, the comparison with two should also happen.

Therefore, it actually requires two comparison. Similarly if you look at six, six is larger than five and this one comparison keep six exactly where it is, and then observe that one is no inserted into the correct position after four exchanges, six first exchanged with one, then five with one then the value four with one, and finally, the value two is exchanged with one. Similarly three now is move to it is inserted into it is correct place in the figure index by the letter e. At the end of this whole procedure we find that the array is in sorted order. This is essentially the idea behind the insertion sort, and one can even imagine this as a very natural way of arranging a deck of cards

(Refer Slide Time: 06:20)

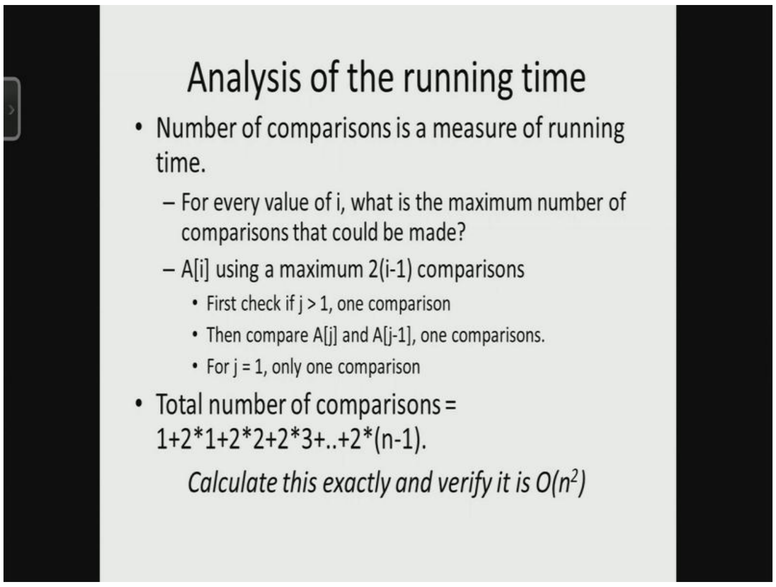


Correctness of Insertion Sort

- For every value of i, n the i-th iteration the following are true
  - At the beginning, A[1]..A[i-1] are sorted in ascending order
  - A[i] is inserted between A[j] and A[j+1]
    - Where j is between 1 and i-2 and A[i] >= A[j] and A[i] <= A[j+1].
    - Therefore, at the end of the i-th iteration A[1]..A[i] is sorted
- At the end of the n-th iteration A[1]...A[n] is sorted. Therefore, the Algorithm is correct.

How does one guarantee the correctness of the insertion sort algorithm. The way to guarantee the correctness of the sorting algorithms, is to know argue that in the ith iteration the elements a of i will move to it is correct position, among the elements a of a of one to a of 1 minus 1 within i comparisons. In other words we prove this by induction. We observe that at the end of the ith iteration the elements a of 1 to a of i minus 1 are sorted in, a of i are sorted in ascending order. We prove by this induction on the value of i observe then when i equal to 1 which is the base case, the array just cons containing a of 1 is just in sorted order. So, let us assume that a of 1 to a of i minus 1 are sorted in ascending order, and let us complete the induction step. So the hypothesis now is a of 1 to a of i minus 1 are sorted in ascending order. Now a of i let us assume that a of i is now inserted between the elements a of j and a of j plus 1. So, let us look at the indices, that j can take. J can take a value between 1 and i minus 2, and where ever it is inserted. On termination it is clear that a of i is at least as large as a of j and a of i is less than or equal to a of j plus 1. In other words, as the algorithm progresses, the prefix of the array which is sorted, it is length keep increasing, iteration by iteration. Consequently at the end of the nth iteration it is clear that the elements a of 1 to a of n occur in sorted order. Therefore, insertion sort is indeed correct in the senses that, we terminate s and termination the array is in sorted order.

(Refer Slide Time: 08:36)



Analysis of the running time

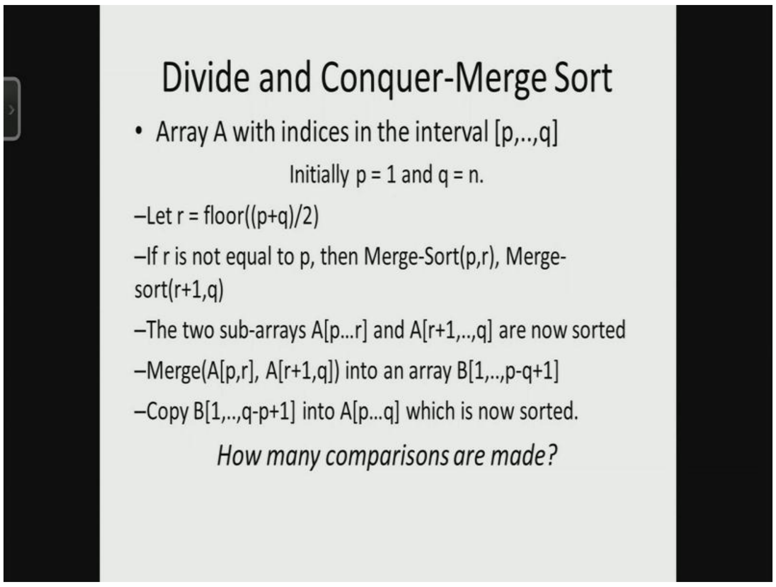
- Number of comparisons is a measure of running time.
  - For every value of i, what is the maximum number of comparisons that could be made?
  - A[i] using a maximum 2(i-1) comparisons
    - First check if j > 1, one comparison
    - Then compare A[j] and A[j-1], one comparisons.
    - For j = 1, only one comparison
- Total number of comparisons = 1+2*1+2*2+2*3+..+2*(n-1).

  Calculate this exactly and verify it is $O(n^2)$

The analysis are the running time is based on the observation that for the elements a of i to be inserted in this correct location in the worst case, which is need 2 times i minus one comparisons, we use comparison as a measure of the running time. We count the total number of comparisons that need to be made, to ensure that i is inserted into the correct place. And here we place an upper bound. In other words we do our worst case analysis of the number of comparisons which are required. And observe that if the elements, the worst case is the case where the elements a of i should eventually enter the location a of 1. In other words in the elements a of 1 to a of i a of i turns out to be the smallest elements, in this sub array of consecutive elements.

Then a of i indeed has to be compared with each of the elements; that is i minus 1 all the way have to, the element index set i minus 1 up to the element index by y by 1 .Further in each of this iterations if you notice the code, that we have written, there is a comparison of whether the indexes is equal to 1 or not. So, that is why we get 2 into i minus 1 comparisons. Therefore, the total number of comparisons made, is one comparison for the first element, and subsequently it is an even number of comparisons up to the nth element. Therefore, the total number of comparisons that is made is given by summation 1 plus 2 plus four plus six and so on up to 2 time n minus 1. Here is the small exercise which is in the toy exercise, calculate this exactly and verify that it is of the order of n square. So, this brings to end the discussion of insertion sort, and it is instructive to see on what input insertion sort performs the most number of comparisons. This is also left as a small exercise to the student.

(Refer Slide Time: 11:07)



Divide and Conquer-Merge Sort

- Array A with indices in the interval [p,..,q]

Initially p = 1 and q = n.

–Let r = floor((p+q)/2)

–If r is not equal to p, then Merge-Sort(p,r), Merge-sort(r+1,q)

–The two sub-arrays A[p...r] and A[r+1,..,q] are now sorted

–Merge(A[p,r], A[r+1,q]) into an array B[1,..,p-q+1]

–Copy B[1,..,q-p+1] into A[p...q] which is now sorted.

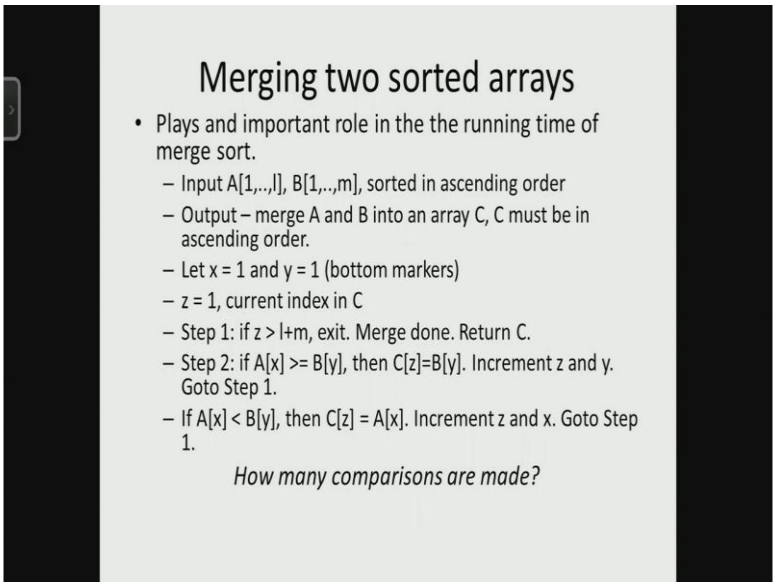
How many comparisons are made?

Natural question now, is that can one reduce the number of comparison made by our sorting algorithm, from order of n square to something significantly smaller than n square. We now show an algorithm which uses a divide and conquer pharadine, recall that in the last lecture, where we studied binary search. We encounter the divide and conquer paradine, where in that case we were searching for a particular element in a sorted array, and in an every iteration the region of the array which is sorted, became smaller by a fraction of two. In this case we do something similar we create two sub problems of almost equal size.

Sort the two sub problems recursively, and then merge the sorted arrays. So, this is the whole idea, this is the divide and conquer paradine, and this is what merge sort does. The sequence of steps are as follows; for the goal of sorting an array, whose indices are in the interval p to q, where p is smaller than q. Initially the first time when merge sort would be called, the value of p would be 1 and q would be n; that is we want a sort the array of size n. The algorithm takes a middle element in the given array. In this case it computes the index, given by the formula r which is the floor of p plus q by 2. In other words, the range p plus q is taken which is the sum of the two elements divide by two gives a middle index in the array, and we take the floor just in case the p plus q is odd. If r is not same as p. In other words, if the array has more than one element, or more than three elements. If the array has more than three elements, then merge sort of p comma r is called followed by a recursive call to merge sort of r plus 1 to q.

Observe that here is a boundary condition of what happens when r is equal to p. Observe that when r is equal to p, there at most two elements in the array. And there is no need to recursively sort the elements of size one. Now the two arrays a of p to r and a of r plus one to q are now sorted recursively, using merge sort itself, at the end of which the array a of p comma r and a of r plus 1 comma q are sorted arrays. We call a function called merge which we will shortly discussed. The output of merge is should take these to sorted arrays, and insert them into third array, which is called b here. Peak b contains all the elements of both the arrays in a sorted in ascending order. b is now copied into the array p comma q which is now sorted, and this is description of the recursive algorithm. The natural question now, is how many comparisons are made by this merge sort algorithm.

(Refer Slide Time: 14:54)



Merging two sorted arrays

- Plays and important role in the the running time of merge sort.
  - Input A[1,..,l], B[1,..,m], sorted in ascending order
  - Output – merge A and B into an array C, C must be in ascending order.
  - Let x = 1 and y = 1 (bottom markers)
  - z = 1, current index in C
  - Step 1: if z > l+m, exit. Merge done. Return C.
  - Step 2: if A[x] >= B[y], then C[z]=B[y]. Increment z and y. Goto Step 1.
  - If A[x] < B[y], then C[z] = A[x]. Increment z and x. Goto Step 1.

How many comparisons are made?

Before we go there, let us we look at a key step, right which place in important role in a merge sort algorithm, which is a time taken to merge to sorted arrays. Let us look it as independent exercises, not just in the contacts of merge sort, but just the question of; the input consisting of two sorted arrays and the output should be the third array which contains, all the elements of two arrays in sorted order. This operation is called the merging operation of the two sorted arrays a and b. So, let us look at the time taken to merge the two arrays a and b into a sorted array c. The idea is quiet intuitive. So, initially x and y are taken to be one. They are smallest elements in the two arrays respectively. And z is the first element, the index of the first element in the array c. The first step is a following which is very important, this is a beginning of an iteration. If z exceeds the total number of elements in the array, in the two arrays, then you exit merge has been completed you written the value of c.

The algorithm returns the value c. In a second step, the smallest of the two elements a of x and b of y are identified, and the smallest of the two elements is now assigned, to be the element present in the array index z in the array c. This is iteratively done right as follows; a of x is compared with b of y. The smaller the two elements as you can see, is assigned to c of z. z is an incremented and depending on where the smaller element came from, whether it came from a or b, the indices x and y are incremented, the indices y and x are incremented. And for example, if the smaller element came from the array y array b, then the element assigned to c of z is b of y and y is now incremented and z is also incremented. If is smaller elements came from the array a, then x and z are incremented and the array element a of x is copied into the array index z in c. Now we will argue that this indeed, merges the two arrays and we will also count the total number of comparisons made.

(Refer Slide Time: 18:06)

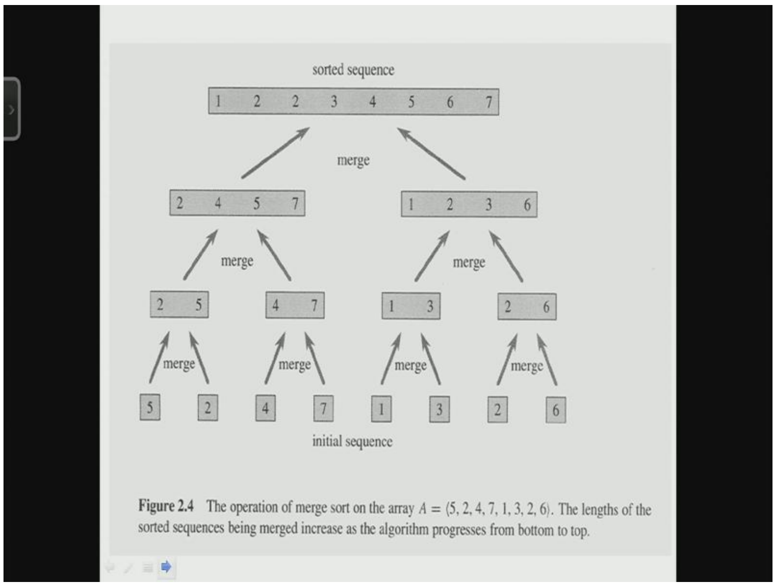
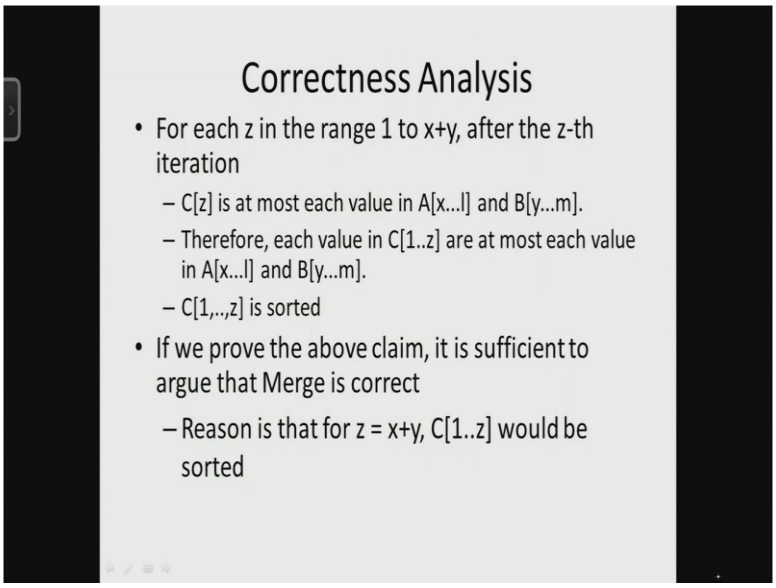


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

At this point of time, before we go into that argument, let us does look at the execution of the merge sort algorithm, and again this image is taken from the text book or by (18:26), and let us look at the run of this algorithm. The leaf levels in this tree tell us the initial array. The initial array has had eight elements, and the elements are 5 2 4 7 1 3 2 and 6, and observed that in this view every intermediate array that you see as you view this in a bottom of fashion, is a sorted array. And the parent of two arrays is obtain by merging the contains of two sorted arrays. Let us look at the leaf level, let us look at, the elements five and two, individually they are in sorted order, and they are merged to result in the parent array which has an elements two and five. If you look at the sibling, the right sibling of the array 2 5, it is 4 7 which is obtained by merging the two arrays, which have just a single element 4 and 7 respectively. Now 2 5 and 4 7 as you can see, is merged it to give the array 2 4 5 and 7. And eventually the whole sorted sequence obtains at the root of this recursion tree. It is important to observe that this data structure is present, only in the analysis of the algorithm, and the algorithm indeed does not run in this order. It runs in a top down order as oppose to the analysis which is happening or which is being presented in a bottom up fashion.

(Refer Slide Time: 20:13)

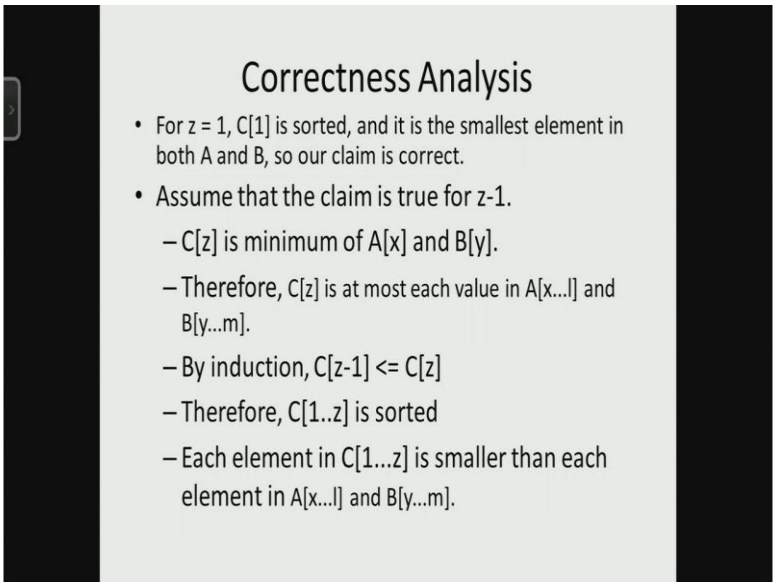


Correctness Analysis

- For each z in the range 1 to x+y, after the z-th iteration
  - C[z] is at most each value in A[x...l] and B[y...m].
  - Therefore, each value in C[1..z] are at most each value in A[x...l] and B[y...m].
  - C[1,..,z] is sorted
- If we prove the above claim, it is sufficient to argue that Merge is correct
  - Reason is that for z = x+y, C[1..z] would be sorted

Let us look at the correctness analysis of the merging step. The merging step is indeed correct. So, let us just observe a primary invariant which says that for every z in the range; 1 to l plus m, after the zth zth iteration c of z satisfies some nice properties. c of z is smaller than all the values in the array a among the region x to l, among the indices x to l and it is smaller than all the values in the array b among the indices y to m. Therefore, it follows that c of 1 to z all the values are, at most each values in the array indices x to l in a, and the array indices y to m in the array b. The third property that is the guaranteed by the algorithm is that, the array c is sorted for every z. So, is it remembered that z is the iteration number. To prove the above claim, it is sufficient. If we prove the above claim, this prove that the merge algorithm is indeed correct, because when you take the value z is equal to l plus m, the array c of 1 to z would be sorted; that is the whole array is sorted, because of the range of value is that z takes from 1 to l plus m.

(Refer Slide Time: 21:56)



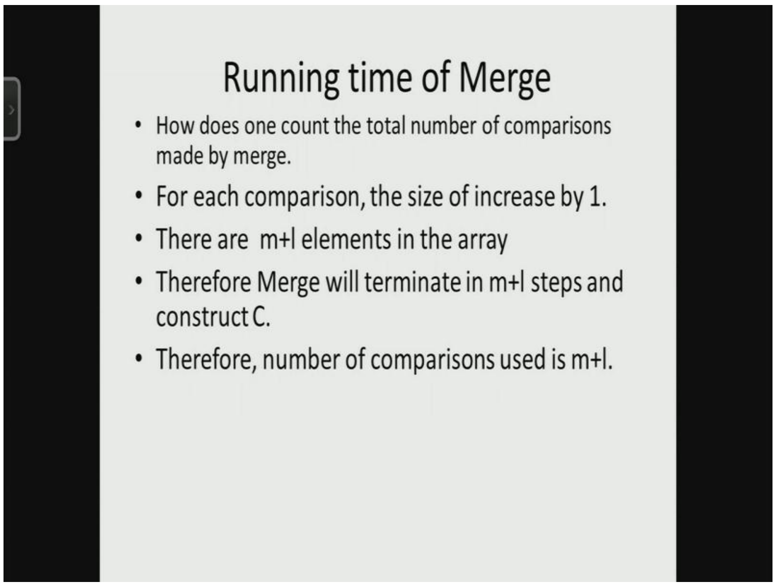
Correctness Analysis

- For z = 1, C[1] is sorted, and it is the smallest element in both A and B, so our claim is correct.
- Assume that the claim is true for z-1.
  - C[z] is minimum of A[x] and B[y].
  - Therefore, C[z] is at most each value in A[x...l] and B[y...m].
  - By induction, C[z-1] <= C[z]
  - Therefore, C[1..z] is sorted
  - Each element in C[1...z] is smaller than each element in A[x...l] and B[y...m].

Continue in the correctness analysis, we set up the proof by induction. Let us assume that z is equal to 1, and in this case the single ton array. The array containing just one element is sorted. It is indeed that smallest element in the arrays a and b; therefore, our claim is correct, in this base case. Let us assume, and let us make the induction hypothesis that the claim is indeed true for a value z minus 1 ,and let us prove this for z this would complete the induction step. So, the first claim is that c of z, is the smallest elements in a of x and among a of x and b of y. This is indeed correct, because if you look at the steps in the algorithm, we choose a minimum of the elements a of x and b of y and assigned it into the value c of z, and therefore, c of z is indeed the minimum.

Since a and b are sorted, it is clear that c of z indeed smaller than the elements in the array location x to l in the array a, and the elements in the array locations y to m in the array b. By induction we know that c of z minus 1, is indeed s smaller than or equal to c of z. In other words we know that by induction that c of z minus 1 is smaller than or equal to all the elements in the arrays a and b put together, and we have taken a minimum of all those elements and put them into c of z. Therefore, c of z minus 1 is smaller than or equal to c of z. Consequently it follows that c of 1 to z is in the sorted order, given that c of 1 to z minus 1 was in the sorted order by the induction hypothesis. This proves the induction step; and therefore, we have proved that merge in the succeed, that after l plus m iterations the whole array c is in the sorted order, and therefore, the merge algorithm indeed succeed.

(Refer Slide Time: 24:16)

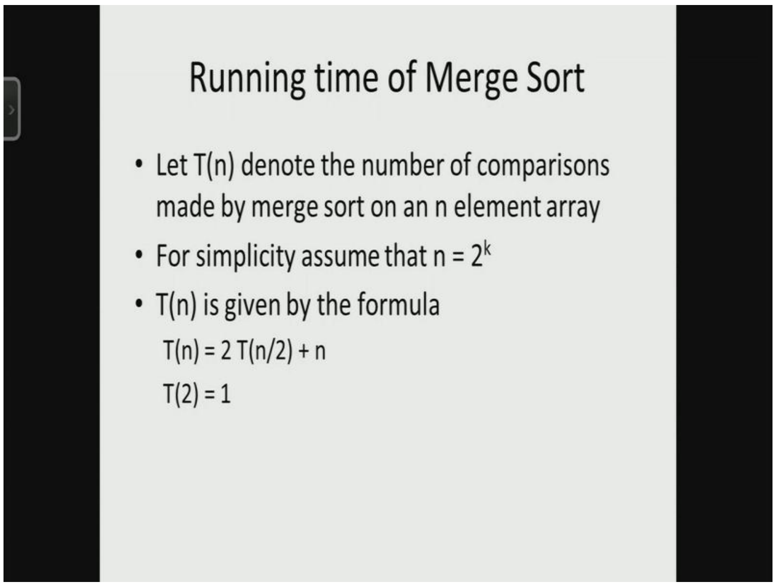


Running time of Merge

- How does one count the total number of comparisons made by merge.
- For each comparison, the size of increase by 1.
- There are m+l elements in the array
- Therefore Merge will terminate in m+l steps and construct C.
- Therefore, number of comparisons used is m+l.

Let us now analyze the running time of merge. It is very important to get an understanding of how much time it takes for merge to run, it is crucial to understand, how does one count the running time, and in this case we count the total number of comparisons. The most important observation that we make, is that every time a comparison is affected, the size of the array c keeps increasing by exactly one. As a consequent of this observation, after l plus m comparisons there will be l plus m elements in the array c, and therefore, the total number of comparisons used, is a sum total of a total number of array elements in two arrays a and b. Therefore, the number of comparisons used by the merge algorithm is l plus m. In other words it is just linear in the size of, the number of elements in the two arrays put together.

(Refer Slide Time: 25:21)



Running time of Merge Sort

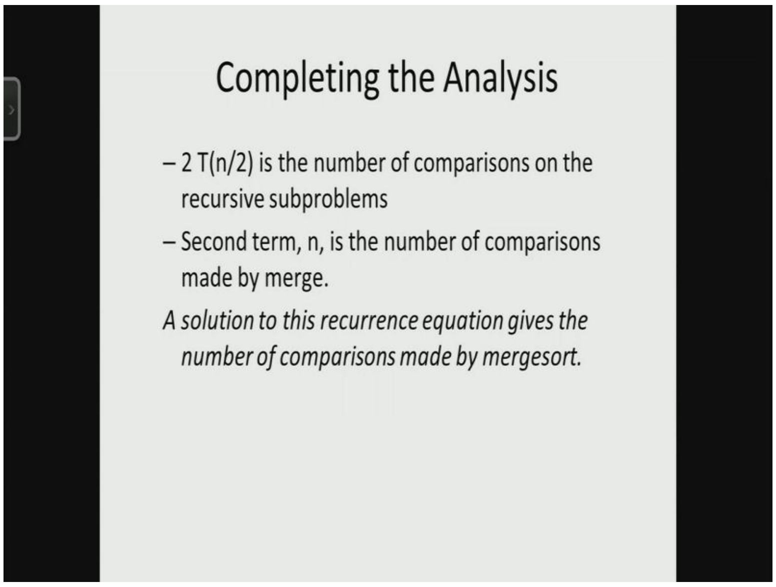
- Let T(n) denote the number of comparisons made by merge sort on an n element array
- For simplicity assume that $n = 2^k$
- T(n) is given by the formula

  $T(n) = 2\,T(n/2) + n$

  $T(2) = 1$

This gives us a handle to analyze a running time a merge sort. Again here we count the total number of comparison made by merge sort. Let t of n denote the number of comparisons made by merge sort on an n element array. For the sake of simplicity, just for the argument let us assume that n is a power of two for some value k; that is n is equal to two power k. Now t of n is given by the formula, t of n is two times t of n by two plus n, and t of two is just equal to 1.

(Refer Slide Time: 26:28)



Completing the Analysis

- – 2 T(n/2) is the number of comparisons on the recursive subproblems
- – Second term, n, is the number of comparisons made by merge.

A solution to this recurrence equation gives the number of comparisons made by mergesort.

Let us understand this formula. This is called a recurrence equation, and two times t of n by two, is the number of comparisons on the recursive sub programs, to sort the recursive sub problems; that is, the sub problem consisting of one half of the array, and the second sub problem consisting of the second half of the array. The second term which is n, is the total number of comparisons made by merge which we just very recently analyze. A solution to this recurrence equation, gives us a solution to the number of comparisons made by the merge sort algorithm, and this will be completed in the next lecture.