Programming & Data Structures

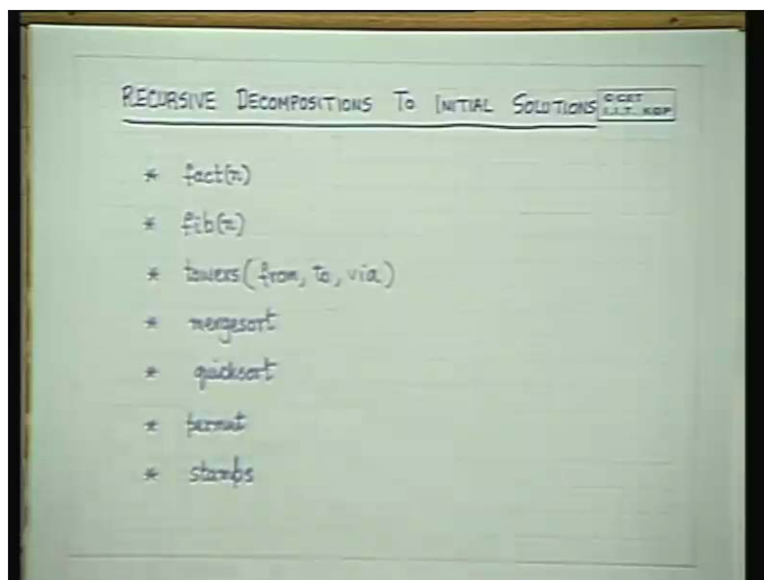
Dr. P.P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 11

Merge sort And Quick sort

We shall continue our discussion on designing algorithms from recursive definitions. And we have seen that, what we were trying to do so long was given a recursive definition of a problem, we are trying to get an initial solution.

(Refer Slide Time: 01:20)



The final solution will be obtained after a more detailed analysis as we saw in the previous days. So, today we will continue with some of the problems which we were tackling in the previous day and we had seen how we obtained the final program for the factorial problem, the Fibonacci numbers problem and the towers of Hanoi problem in the previous class. Today we will concentrate on the two important sorting algorithms namely merge sort and quick sort and try to see how we can use whatever knowledge we have of arrays and functions and recursive programming in C to obtain these programs.

(Refer Slide Time: 02:45)

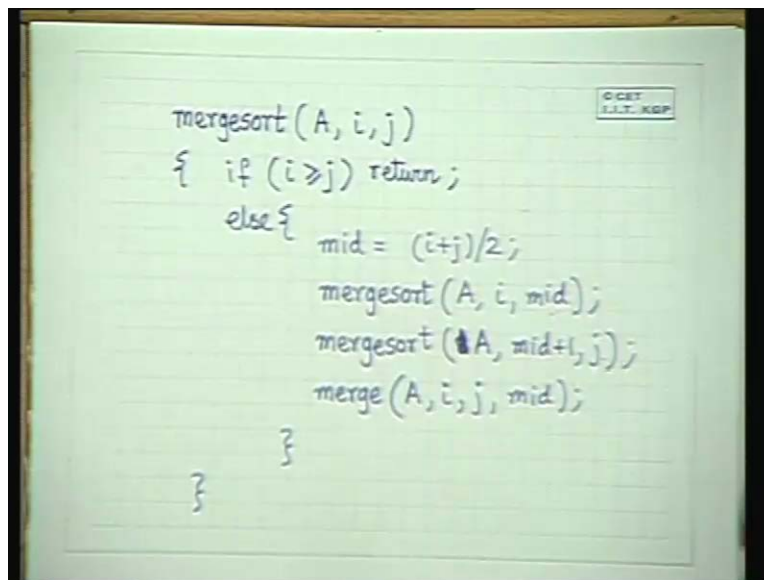
If you will recall the merge sort algorithm goes like this. The recursive decomposition was given a list L, this will produce a sorted list L, a new sorted list, this will return a new sorted list. And the base condition was if the size of the list is 1 then the returned list is the original list itself otherwise we split L into 2 non-empty sets L_1 and L_2. We recursively merge sorted L_1 to get L_1 dash, merge sorted L_2 to get L_2 dashed and then did a merge routine of two sorted sub lists of L_1 and L_2 and this is what we returned as L dash. So, we understood how merge sort this recursive decomposition works but we still do not know how to make lists and how to return arrays or how to make a list of elements. So with our knowledge of arrays, we will be trying to solve this problem.

(Refer Slide Time: 05:46)

So how do we, given an array A or data or A which contains our elements so say 4 3 2 1 15 and 9 and n is 1 2 3 4 5 6, n is equal to 6. Now our data structure for storing a list will be an array and in order to indicate which part of the array we are going to use, we will use the indices of the array. For example this is array index 0, this is 1, this is 2, this is 3, this is 4 and this is 5. So whenever we have to pass this list we will just pass the indices of the list and we will make sure that when we split the list, we just split it in the middle or in a position so that we are not required to interchange the elements at all. So what we will do is initially we will instead of passing a list, we will call the merge sort routine like this. The array A will be passed as an array name and the indices i and j for which merge sort requires to be done will be passed.

Now suppose I pass A, 2 and 4 then this means this corresponds to this list, this part of the list, A, 2, 4 corresponds to this. Similarly A, 0, 3 corresponds to this list, the first 4 numbers. So we can use the array and the indices of the array to pass out the list that we require and instead of returning anything we will sort the elements in the array itself. So if we sort in the array itself, we need not return anything. So based on these two ideas, we can now rewrite our merge sort routine, the original merge sort routine this way.

(Refer Slide Time: 09:16)



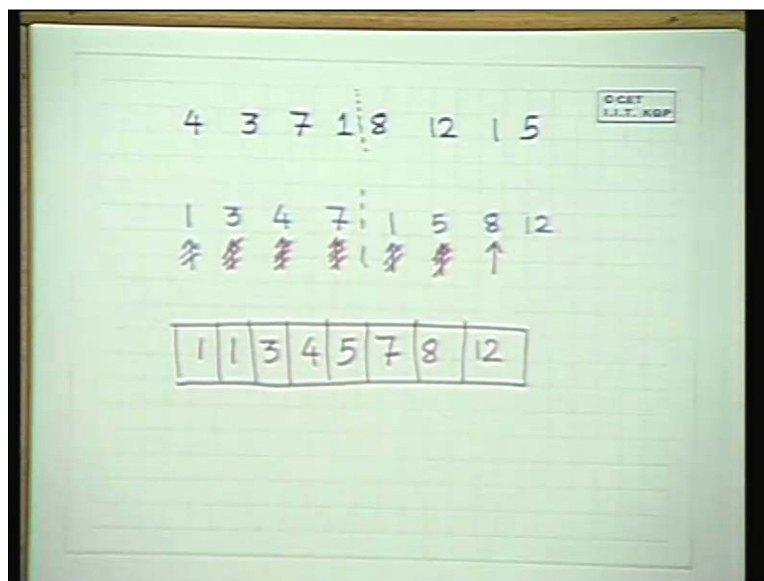
So the new merge sort routine will be like this A, i, j. Now what is the base condition when i is, when there is only a single element. So when i is equal to j, we get the base condition. So if i is greater than equal to j you are supposed to return that element. Now that element already exist in the array, so we need not return anything, we just simply write return else the next step was to split the list into two equal parts or two, any two parts. So we will have to choose an index between i and j, so we will have to choose any index between i and j, suppose for the time being we choose the middle element.

So we write mid is equal to i plus j by 2. So now we have obtained the middle element of the list. Suppose initially it was 0 to 5, so it will be 0 plus 5 by 2 which is 2. So you will

get two here and now we are left to sort the two parts of the list, so you can simply call merge sort A, one half will be from i to mid, the other part will be A mid plus 1 to j. So now we have split the list into two equal parts and then called merge sort on one part, one half called merge sort on the other half and we expect the array from i to mid will be sorted after the return of this merge sort. We expect the array from mid plus 1 to j will be sorted after we return from this. And then all we are left is to call the merge routine. The merge routine will again the whole list and the position where it was split up or the two different lists. To pass the two different lists, what we have to pass? We have to pass the array A, we have to pass i, we have to pass j and we may pass mid but if we know that we are going to calculate the middle element, we can compute mid in the merge routine also.

So either we pass mid here or we compute the mid in the merge routine and do not pass it. So if we just use an array then we can implement merge sort by manipulating the array indices which will denote the lists that we have. So this is the idea of the basic merge sort routine. We are yet still left to write down the merge routine. So, any questions regarding this?

(Refer Slide Time: 12:15)



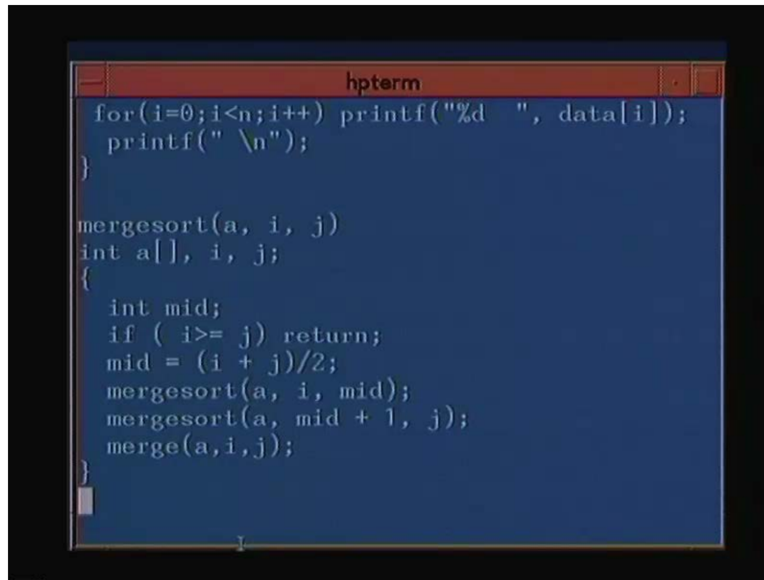
Now let us come to the merge routine. The merge routine takes the array A, the index i, the index j and mid. And what does it supposed to do? It is supposed to sort it. Now let us see how we would do it by hand, just a quick recollection. Suppose the initial list was 4 3 7 1 8 12 1 and 5. So from array index 0 to 7, we had it and we split it into two parts here and after we merge sort this, we are supposed to get back 1 3 4 7. After we merge sort this we are supposed to get back 1 5 8 12.

Now comes the merge routine. Now in the merge routine, we will start comparing the first two elements and as we discussed earlier, the smaller of the two will be the first element. So here since there is a tie by default let us take this one then whichever becomes, whichever gets chosen the index of that is incremented and then again this

index and this index is compared, the smaller of the two gets chosen and the corresponding index gets incremented. Again the smaller of the two gets chosen and the corresponding index gets incremented, again the smaller of the two gets chosen and the corresponding index gets incremented, the smaller of the two gets chosen and again this index is… Then this is chosen but once this crosses the mid value then the rest of the elements from here can just be copied out.

So we are to solve it in two parts, one is we compare the indices and increment it and then whenever we reach the end of list, we copy into end we copy the rest of the elements at the end and we can use an additional array to store the sorted list of this part getting merge sort, alright. So using this technique we can write out the merge sort routine, the merge routine and based on this the merge sort routine will get completed.

(Refer Slide Time: 14:12)

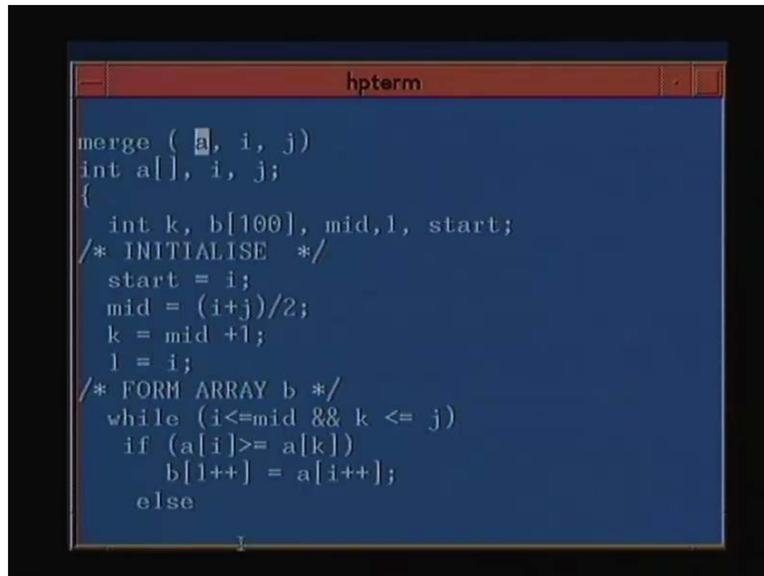


So let's go to the computer and see how the program looks like in more details. This is the main program where we input the data, this is not so relevant. The main part is here this part, can you see. This part where after reading in the elements you call merge sort with the array called data here, the array is called data, 0, n minus 1 and after this call we print the result. Now we come to the merge sort routine. The recursive routine as we discussed just now, it has got an array a, the index i and the index j. a is declared to be an array, you need not give the values as we have discussed earlier i and j are integers, mid is the value, this k I don't think it's required.

If i is greater than equal to j return otherwise mid is i plus j by 2, merge sort a, i, mid, merge sort a, mid plus 1, j and merge a, i, j here we did not pass mid because we know that mid is i plus j by 2 we will compute mid in the merge routine. So the recursive merge sort routine is fairly simple, it does, this is the recursion breaking condition here, this is the splitting of the two lists, these are the two recursive calls. It does not matter in which order you do it, you can call, you can put this first and this second or this first and this

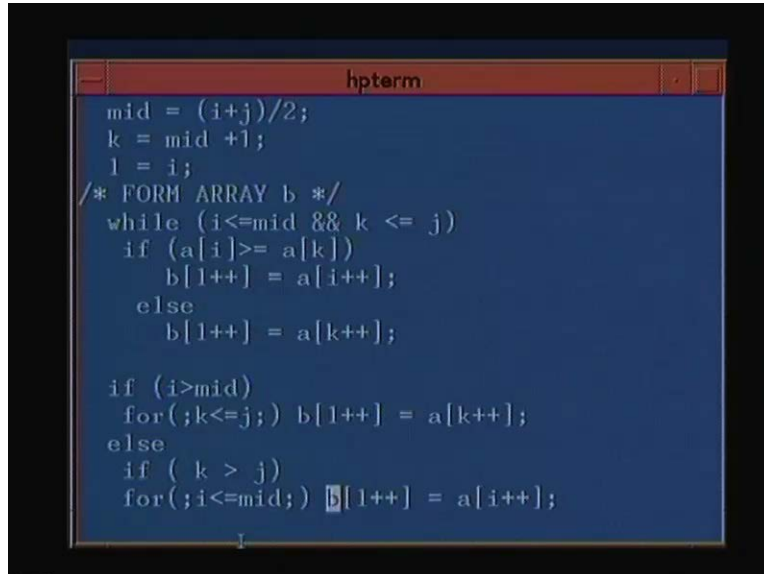
second. It does not matter because they are going to work on different portions of the list and then you merge a, i, j.

(Refer Slide Time: 14:18)



So let's go to the merge routine. The merge routine takes in as parameters the array a, the index i, the index j. It uses a temporary array b to store the elements during merge sorting, the mid is computed, L and start are temporary variables. So we do some initialization, the start is i, i is the start of the list for which merging is to be done, mid is computed i plus j by 2. The first pointer, we mention that we will start with two comparisons. Here we start with k which starts from mid plus 1 and L which starts from i. So we will continuously compare a k with a l and appropriately increment or decrement and we will form the array b. So this is the part where we form the array b. And what did we say? As long as i is less than equal to mid and k is less than equal to j, alright.

(Refer Slide Time: 17:45)



As long as i is less than equal to mid and k sorry this should be ya, so the array b will be starting from l, the array b will be starting from l. So if a [i] is greater than equal to a [k] then b [l] is made to a [i] and l is incremented and a [i] is incremented, otherwise b [l] is assigned a [k] and l and k both are incremented. Is that okay? So as long as i is less than equal to mid and k is less than equal to j that is both the arrays are not, both the sub parts of the arrays are not filled up or not exhausted. We continuously compare a [i] with a [k] and we put in b [l] the value which is greater because we are sorting in descending order, so that is why we are doing it this way.

Now after this part is over either i will be greater than equal to mid or j will be greater than equal to k or k will greater than equal to j. This will come out based on any two of these conditions. If i is greater than mid, what does it mean? It means that we have completed from a[i] to a mid that part has already been put in b. So the elements which are still remaining that is from k onwards up to j will be now put into the array b in the for loop. So b [l plus plus] is equal to a [k plus plus] that means b [l] is equal to a [k] and l is equal l plus 1 and k equal to k plus 1. If this is not the case then the other array, this must have happened, k must be greater than j. If k is greater than j then for i equal to whatever is the current value of i and as long as i is less than equal to mid, we put them these values back into the array b. So, b [l plus plus] equal to a [i plus plus]. And now the last part, we copy back these relevant values of b back into the array a.

(Refer Slide Time: 17:55)

```
hpterm
      b[l++] = a[i++];
    else
      b[l++] = a[k++];

  if (i>mid)
   for(;k<=j;) b[l++] = a[k++];
  else
   if ( k > j)
   for(;i<=mid;) b[l++] = a[i++];

 /* COPY BACK TO ARRAY a  */

   for(l=start; l <= j; l++)
    a[l] = b[l];
}
```

So what we do is copy back into the array a and we will copy back from the start position to j. The start was stored as initial value of i if you recall here. The start was stored as the initial value of i and so we will copy back from l equal to start till l is less than equal to j l plus plus, we copy back into the array. So this completes our merge routine. So in the merge routine, we have first initialized and found the middle value. We have formed the array b by comparing the indices with a [i] and a [k] and correspondingly put the element in the array b. If one of the arrays get exhausted, if one of the sub parts get exhausted the rest of the elements are put back and then array a is filled up with the current values of array b as after sort.

(Refer Slide Time: 19:45)

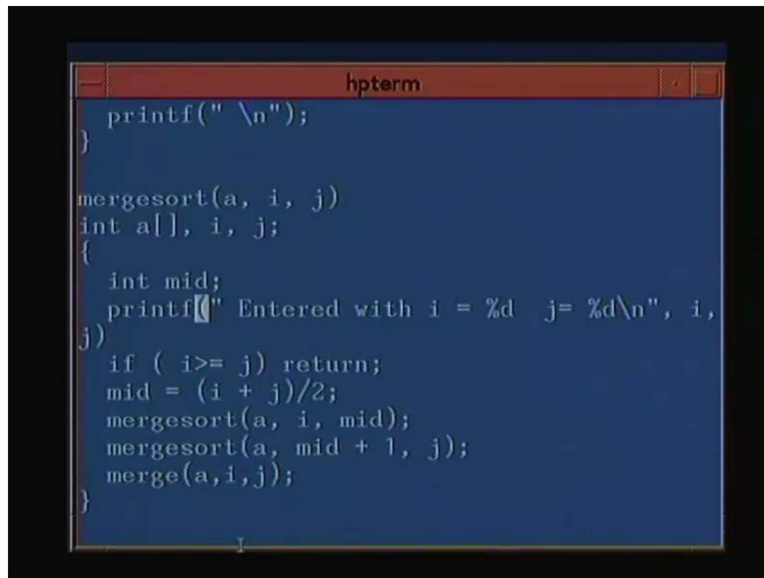
```
hpterm
:wq
"mergesort.c" 62 lines, 1062 characters
$ ls
a.out          mergesort.c   qsort2.c
fib.c          permut.c      towers.c
m2.c           qsort1.c      tt
$ cc mergesort.c
$ ./a.out
Give n :4
 n = 4
3 5 2 4

 Numbers read are:  3  5  2  4

 Sorted numbers are:  5  4  3  2
$
```

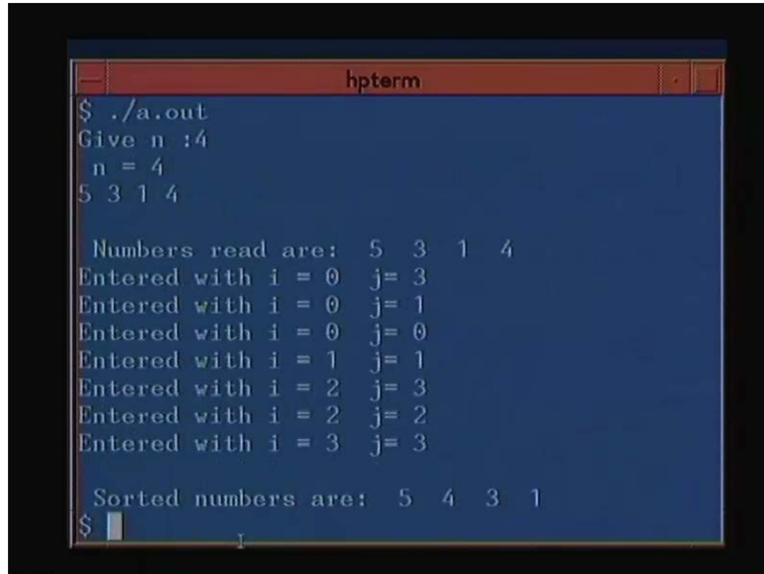
So this completes merging. So after it is merged that portion of the array a from i to j is now sorted a i to a j is now sorted. So once a i to a j is sorted, this means we have now, we can return back from the merge sort. So we are now executing it, n let us take 4 elements. So we get the sorted numbers, the numbers read actually 5 2 4 and the sorted numbers are 5 4 3 and 2.

(Refer Slide Time: 20:26)



Now in order to understand how the splitting and the recursive calls are made, we can simply put in a right statement here, here entered with i equal to… and j equal to… So as soon as we enter the merge routine we will see what is the value of i and what is the value of j. So this way we can find out exactly how the split occurs. This is how it ran, we initially ran it with 5 3 1 4. So initial call was i equal to 0, j equal to 3 then it split up into two parts, mid was computed and 0 1 was one case and 1 2 3 was the other case.

(Refer Slide Time: 21:00)



So 0 1 was called first because that's what we did. So 0 1 was called, the mid was computed and mid of 0 and 0 plus 1 by 2 gives 0. So 0 0 was called, 0 0 is a recursion breaking condition so after that it came back to this routine which I have called it and this routine now called it with other one 1 1. So this also completed because this is a recursion breaking condition, so it came back here because this originally I have called it 0 1 and 2 3, so the 2 3 call now came. So this is a 2 3 call, the 2 3 call in turn made at 2 2 call and when the 2 2 call was over this 2 3 call made a 3 3 call.

When the 3 3 call was over, it returned back to the 2 3 call and this 2 3 call returned back to 0 3 call and we got the execution completed. And whenever you execute, you can put in print statements at various places to understand exactly how the recursion takes place. So let's get back now. So I hope the merge sort routine is fairly clear, how we can use arrays and indices of arrays to do it.

(Refer Slide Time: 23:32)

```
list QUICKSORT (L)
{ n = size(L);
  if (n==1) L'=L; /BASE/
  else {
        X = select(L);              } DECOMPOSE
        L1 = split_ge (L, X);
        L2 = split_lt (L, X);
        L1' = QUICKSORT(L1);
        L2' = QUICKSORT (L2);
        L' = CONCAT (L1', L2') /RECOMPOSE/
  }
  return (L')
}
```

So now we will see how the quick sort routine can be done. Let us recall the original quick sort algorithm. the quick sort algorithm took in a list l and returned a sorted list. similarly the base condition was if the size of the list was 1 that list itself was returned, otherwise we selected an element from the list and we split this list l based on this element x into two parts. L_1 which is greater than equal to and L_2 which is less than… then we quick sorted this and we quick sorted this and we realize that if we just recompose by joining them up, we are going to get back the final sorted result. This part of the decomposition splitting is often called in textbooks, it is called the partition routine.

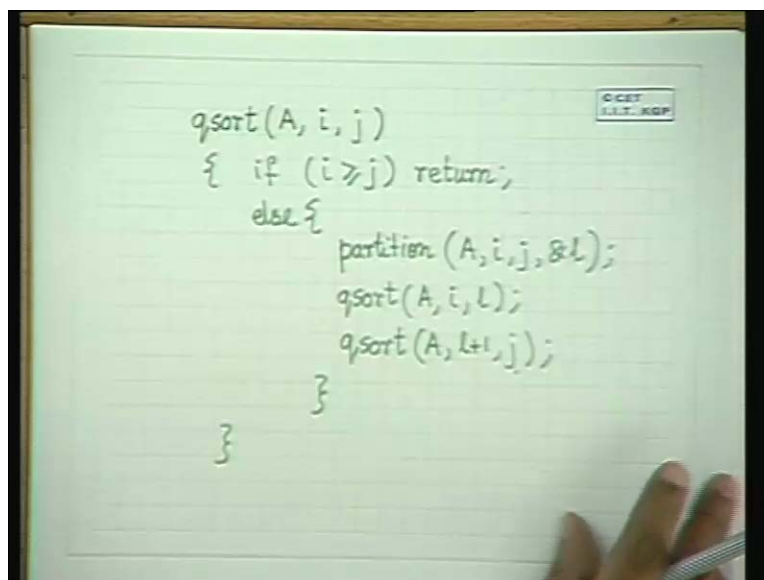
(Refer Slide Time: 24:15)

```
list QUICKSORT(L)
{ n = size(L);
  if (n==1) L' = L;
  else { <L1,L2> = partition(L);
        L1' = QUICKSORT (L1);
        L2' = QUICKSORT(L2);
        L' = concat(L1', L2')
  }
  return(L')
}
```

So if we write it down with the partition function then that same function will look like this, list quick sort L if n is equal to size of L. if n is equal to 1 else L_1 and L_2 are obtained by partitioning L, the selection of element can go inside the partition routine itself and then we quick sort this, quick sort this and concatenate.

So now just like in the merge sort, we will use arrays and array indices to define our quick sort routine. So how will we do that now? The quick sort A of the array A, the list is indicated by the array indices. If i is greater than equal to j return else partition A and the list has to be denoted, so from i to j and say where the list is partitioned is returned in l.

(Refer Slide Time: 27:20)

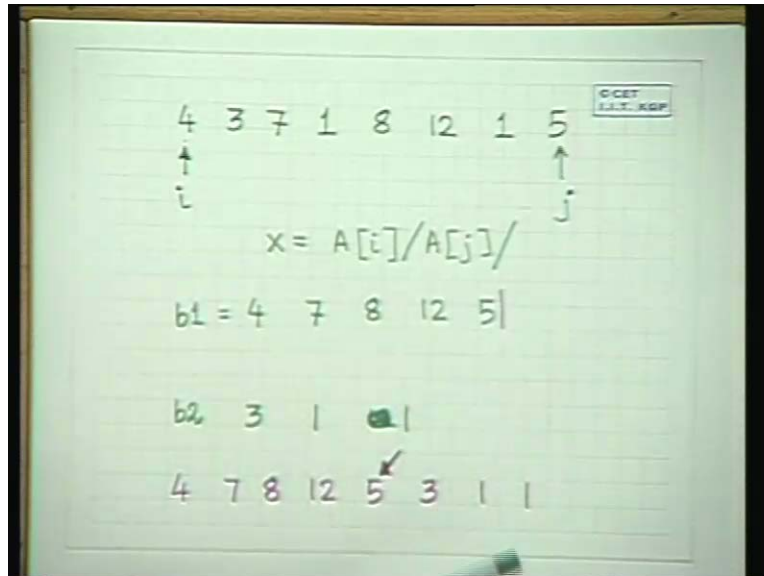


Now in order to return the value of a variable you have to put and in C that is why I purposely put it here also otherwise you could have made the partition routine return a value, anything you could have done. So we will return this here and then we will call quick sort. Now this partition routine unlike the split routine which does actually nothing, this partition routine will actually modify will exchange the values in the array A to make sure that those from i to l are all less than equal to those from l plus 1 to j based on the element that we have selected. Whatever element we have selected for partitioning, the elements which are less than equal to the selected elements will lie between i to l and from l plus 1 to j we will have those elements which are greater than the selected element x that is the idea of this partition routine.

That is the position l will be returned after the function and you call quick sort A, the rest is quite easy L, A. what else do we have to do? We have to do concatenation. Now i to l is sorted in the array, l plus 1 to j is sorted in the array, so automatically the array is sorted concatenation need not be done anymore because the array itself is sorted from i to l and l plus 1 to j, so concatenation is automatically existing there so we just simply return.

So this is the idea of the basic quick sort algorithm where we have replaced the list by the array A and the indices of the arrays. This is how we have done it. So once this part is clear, we are now left to solve the problem of partitioning. Now again we can use the ideal like we used in merge sort to partition.

(Refer Slide Time: 30:45)



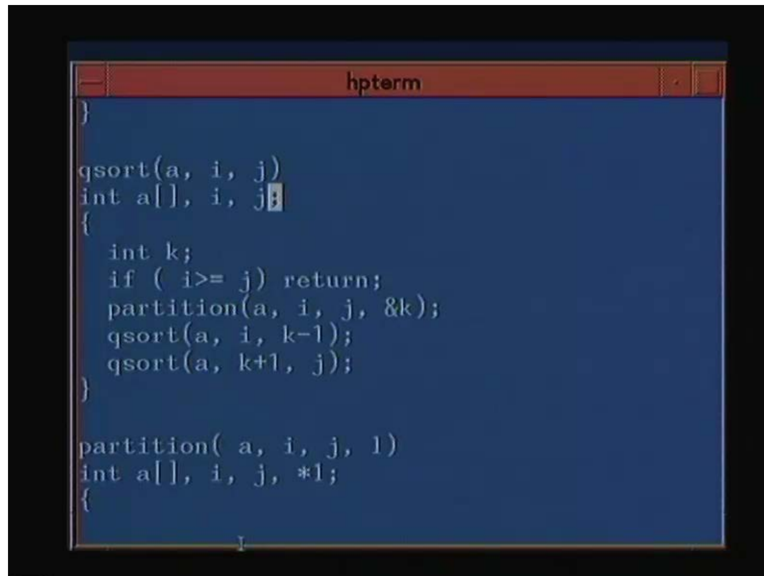
Let us take an example and understand how we will be partitioning. Suppose 3 7 1 8 12 1 5 and we have to partition this, this is i, this is j. The first thing that we have to do is select an element on the basis of which partitioning will be done, alright. So let us say we will select x is equal to A [i], you could have selected A [j] also it does not matter. Some element we will select A [i] you can select, A [j], any element you can select, whether selecting the element makes a difference or not we will come to it later but the question is the algorithm is correct by selecting array element.

Let us suppose here we selected A [i] then we now start putting into the array elements which are greater than equal to 1. So we use two arrays let us call it b1 and b2 and we start from here. If this is less than the selected element, we put it in b2. If it is greater than the selected element we put it in b1 or greater than equal to.

So first you start from here 4 goes here, 3 goes here, 7 goes here, 1 goes here, 8 goes here, 12 goes here sorry 12 goes here, 5 goes here, one more 1 had come sorry 4 3 7 1 8 12 1 5. After having done this now what do we do? We put back these elements back into the array A and we put it like this. We put 4 here, the first we put back 4, then 7 then 8 then 12 then 5 then 3 then 1 then 1. And what do we return back, which index do we return back? We return back this index, the place where the partitioning occurs. The partitioning occurred at this place that is the size of b1 will tell you where the partitioning occurs. So this is one very simple routine to do partitioning, you take two temporary arrays, you select any one elements from here, take two temporary arrays and just put them the elements greater than equal to on one part less than on another part and then put

them back into the original array and you will get what you want. And you have to return this. So this is one function, so let us have a look at this quick sort routine very quickly.

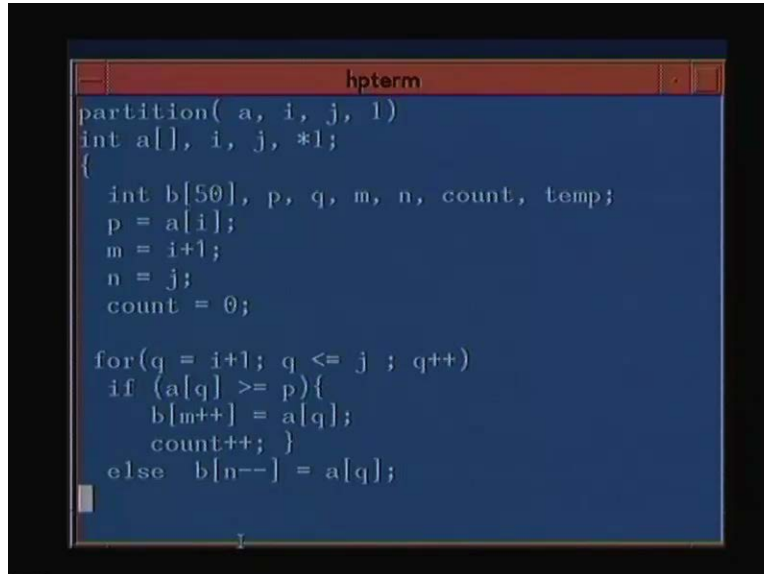
(Refer Slide Time: 31:25)



The main quick sort function, this is all reading and all this similarly quick sort is called here q sort (data, 0 and n minus 1) just like in merge sort. The main quick sort routine is here and this is what we were discussing q sort a, i, j, a is an array, i and j are indices, k is a temporary variable which will be the location which is returned. Well, if i is greater than equal to j return, partition a, i, j and get the value of k but since we have to get the value and it is called by value we pass the address to get the value and here interestingly we have done it k minus 1 and k plus 1.

Can anybody tell me why we have done k minus 1 and k plus 1? Because after we do the partitioning, the element on the basis of which the partitioning was done can already be put in the array index k. So the element on which the basis that element x which was selected will be put into the array element k and once that is put into the array element k, the rest of the elements can be done from k minus 1 to k plus 1, that is why. If you had kept this as k also it won’t have mattered. And here is our function for partitioning and in this partitioning function, we have used one array instead of two arrays.

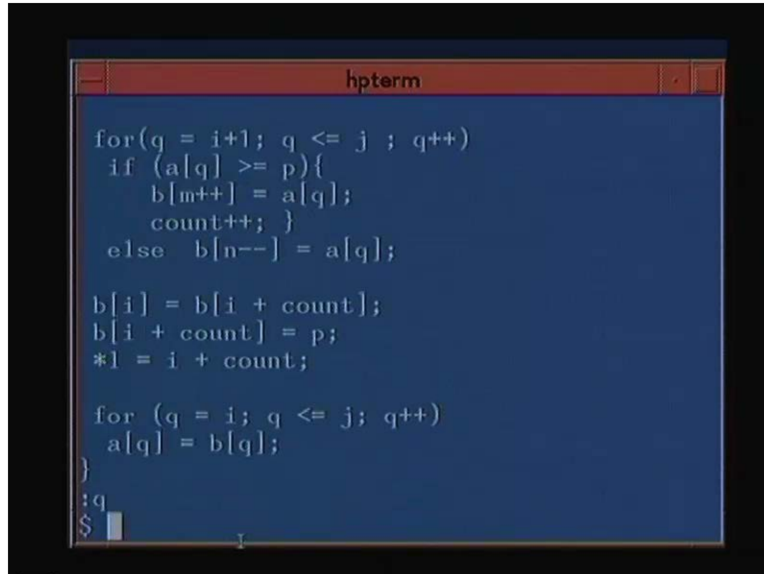
(Refer Slide Time: 32:38)



What have we done by using this one array? In the two arrays that we discussed, we put b1 and b2. Now the elements which were greater than equal to we put in b1 and the elements which were less than we put in b2. Here we just use one array and what we do is the elements which are greater than equal to we put from bi onwards and the elements which are less than equal to we put backwards from bj. So in one array we can handle the whole thing. So that's exactly what is done here, p is initialized to a[i] that is this is the elements which is selected, m is initialized to i plus 1 because p will be put in its proper position after partitioning and n is initialized to j, so one starts from the beginning, the other starts from j. This is the size of the list n, so for q equal to i plus 1 till q is less than j, we compare if a [q] is greater than equal to p we put in bm, alright. Otherwise we put a [q] in b [n], n is the end of the list as you noted. So those which are less than are put backwards from the end of the list and this n is decremented as soon as one element is put in.

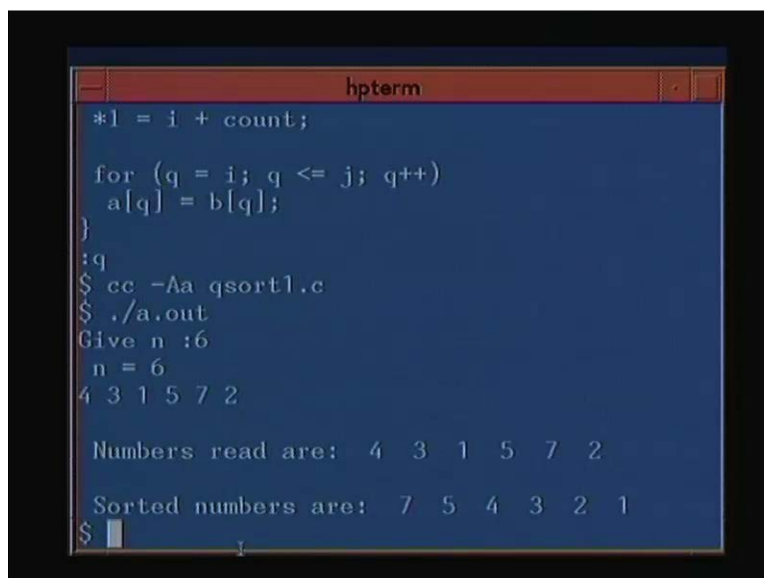
Those elements which are greater than are put in the front of the list b and m is incremented as and when they are put in. And the counter will tell you what are the elements which are finally put in but this ends the for loop. The for loop ends here and this counter will tell you how many elements are actually going into the front of the list.

(Refer Slide Time: 35:45)



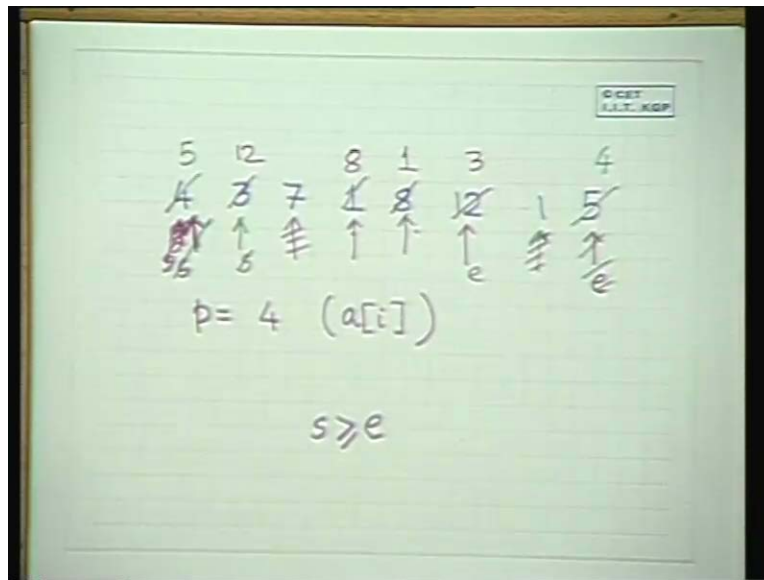
So at the end of this you know where the first original element which was selected where that element, this will be bi plus count that is the first element will be put in. And the content of l, l will be returned as i plus count because l will get back the value here where the partitioning occurred. If count as we mentioned say in b1, the number of elements in b1 was the value of count, so the content of l is i plus count and that is what is returned. And before returning again we copy the values of array b back to the array a. So this way we can write out this quick sort routine. Sorry, have to compile. We give in the elements and we can get it sorted.

(Refer Slide Time: 36:15)



Similarly you can just put in lot of statements inside to find out exactly how the partitioning is done but we will come back to a more important issue here. In quick sort we have used here instead of this two arrays, we actually showed how we can use one array. In that array from the front we will put in these values and from the back we will put in these values that’s what we did. But quick sort can be written without an additional array very easily. We can do it by using this array itself. So let us see how we would do that. We choose our element p or x here to be 4 that is a[i], alright. We choose it to be a[i] and we start with this index x and two indices the start and the end.

(Refer Slide Time: 40:40)

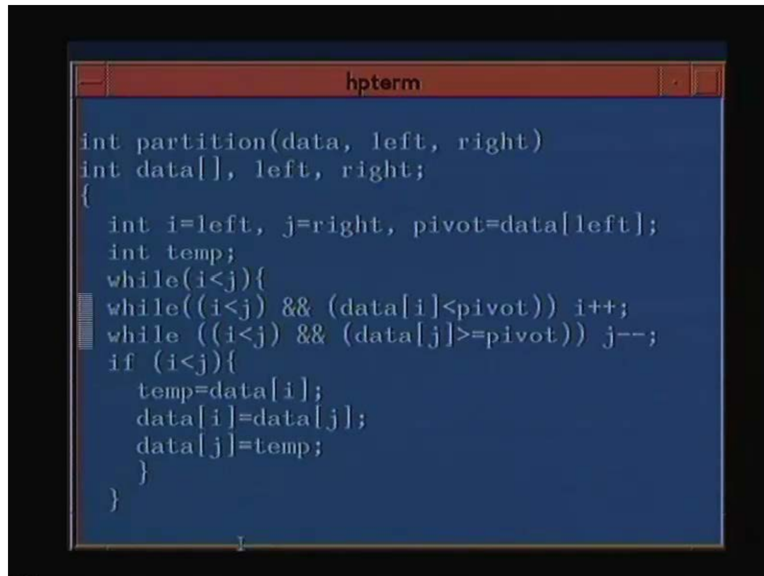


And we compare for the element here, we go on comparing as long as, suppose we are going to sort in ascending order for the time being. So suppose if this is less than p, we proceed because these sides we won’t sorry, we are sorting in descending order. So, as long as this is greater than p we proceed and this side we go on proceeding because as long as it is less than or equal to p. So here this is not greater than p, so we continue. This is greater than p, sorry as long as it is greater than p we continue. So we wait here, we stop here, this is not greater than p and here as long as it is less than or equal to p we will continue but as soon as it is greater than p we will stop. And once we see these two, these are two candidates for exchange because this will exist this side and this will exist this side. So we simply exchange them and after exchanging this becomes 5, this becomes 4 because the elements which are this side are less than or equal to elements which are this side are greater than, let us say this is our argument.

And s now comes here and e now comes here. Now since this is still less than or equal to p, it will wait here but this one is less, so this will continue. It will come here, it will wait here because this condition is that this element is greater than p. So, now we exchange these two and proceed. So here what do we do? This is greater than, so we continue. This is less than and this is greater than, so we exchange this two.

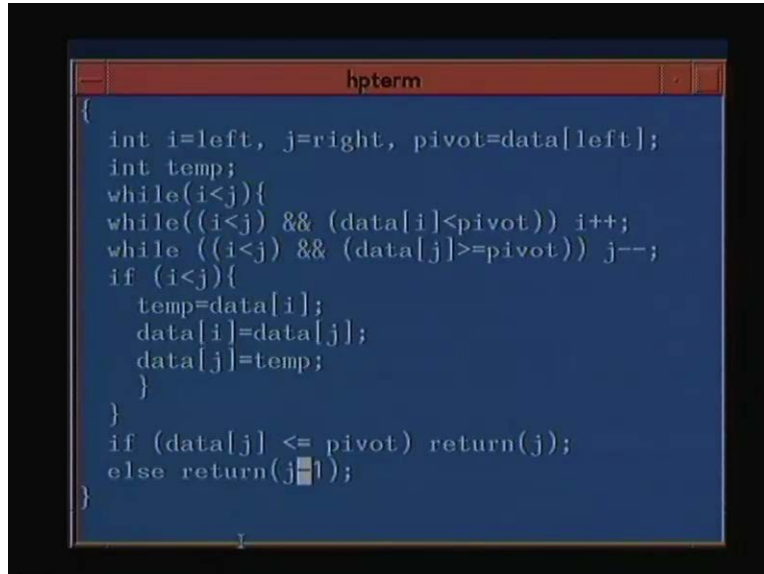
And when do we stop? When s becomes greater than or equal to e. When this index crosses this index we can stop. So by the sequence of comparisons and exchanges, we can do the partitioning in the array itself and we need not do the partitioning with them by using an additional array. So the program for that is not difficult to write, I will just show you the partitioning routine. This is the partitioning routine which takes in an array data, index left and an index right. It initializes the i which we call s to left, j to right and the pivot element on the basis of which it will be done is data left.

(Refer Slide Time: 40:55)



And it will go on doing this while i is less than equal to j. and as long as i is less than j and data i is less than pivot, here we have done sorting in the other way it will increment i. And for j what will it do? As long as data j is greater than equal to pivot, it will decrement j and after these two positions are decremented and incremented and if i is still less than j these are two candidates for exchange, they will be exchanged. And this will continue in this outer loop as long as i is less than j. So that way we will be able to partition the routine and the pivot position that is you can now return if data j after the, see at the end of it i will be equal to j.

(Refer Slide Time: 42:22)



So if data j is less than equal to pivot, we return j otherwise if data j is greater than pivot we return j minus 1. So this way we can return the element by the partitioning routine without using an additional array. So this way we see that we have now been able to write down using arrays and indices of arrays, two routines the quick sort routine and the merge sort routine. And these are as we discussed first, these are only initial definitions where we have still to answer question as to where to exactly split in the merge sort routine to get the optimal result or in quick sort which element do we select, so that quick sort works best. We have not answered these questions and to answer these questions we have to do much more analysis for which we have to study some more mathematics which we will do in our subsequent class.

So today we will stop here and I will request you to just quickly go through textbooks to see the exact routines of merge sort and quick sort which may vary from text book to textbook and just implement them in your lab and see exactly how the recursion occurs.