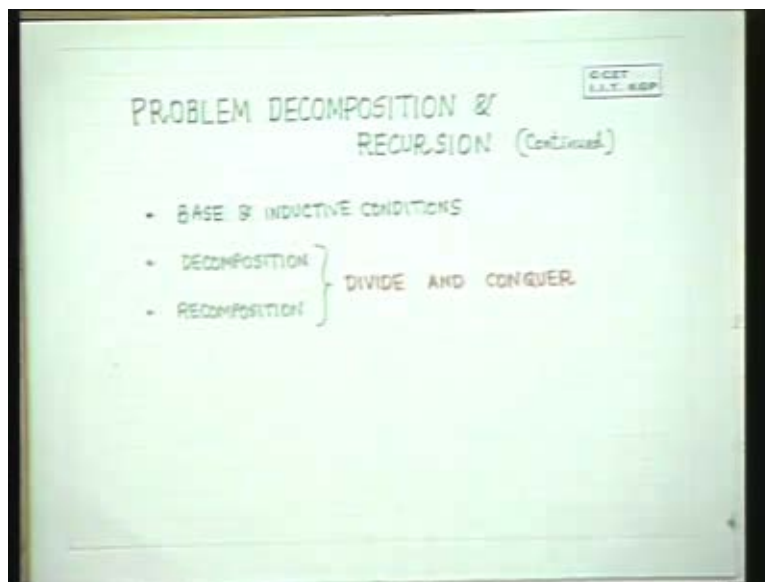


**Programming and Data Structure**  
**Dr. P.P.Chakraborty**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture # 10**  
**Problem Decomposition by Recursion - III**

We will continue our study of problem decomposition by recursion and if we recall that we had decided that our problem will be written in a recursive manner by a principal in which we have got two parts, the base condition and the inductive conditions. The base conditions specifies what will happen on the boundaries of the recursion and the inductive conditions specifies the problem decomposition. And in the problem recursive part we have two aspects of it, one was the decomposition part of the problem, the other was the recomposition part of the problem.

(Refer Slide Time 01:57 min)



And together they form what is well known technique in computer science and is called the divide and conquer principle. And we saw several simple examples in from (Refer Slide Time: 02:04) how to do it. We will just quickly go back to more reasonably written version of those examples. The first was the simplest one in which we have the factorial program where the base condition was when  $n$  is equal to 1, the value is 1 otherwise this was the decomposition fact of  $n$  minus 1 and the recomposition was  $n$  into the return value of the factorial.

(Refer Slide Time 02:35 min)

```
int fact(n)
int n;
{ if (n==1) val=1; /BASE/
  else /* (n>1) */
    { x = fact(n-1); /DECOMPOSE */
      val = n*x; /RECOMPOSE/
    }
  return(val);
}
```

So this was the decomposition and this was the recomposition part of the problem. Next we tackle the Fibonacci numbers and in the Fibonacci series we saw there are two base conditions, when  $n$  is 0, the value is 0 when  $n$  is 1 the value is 1 and these two together form the base condition.

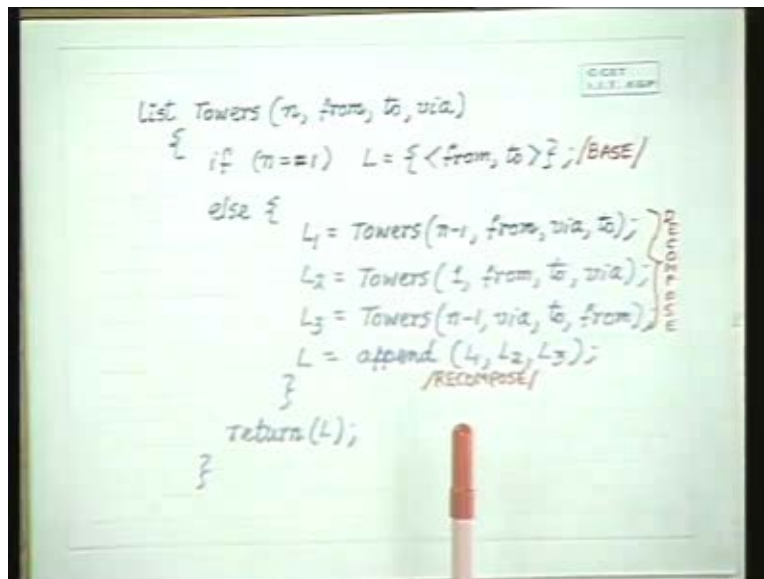
(Refer Slide Time 02:43 min)

```
int fib(n)
{ if (n==0) val=0;
  else if (n==1) val=1; } /BASE/
else {
  x = fib(n-1);
  y = fib(n-2); } /DECOMPOSE/
val = x+y; /RECOMPOSE/
return(val);
}
```

Unlike in the factorial where there was only one sub problem to be solved in the decomposition, here we have two sub problems  $\text{fib } n - 1$  which we called recursively and  $\text{fib } n - 2$  which we called recursively. We got by the two solutions and we recomposed by adding the two of them. And this is what we returned as the value in  $\text{val}$ . When we saw the tower of Hanoi's

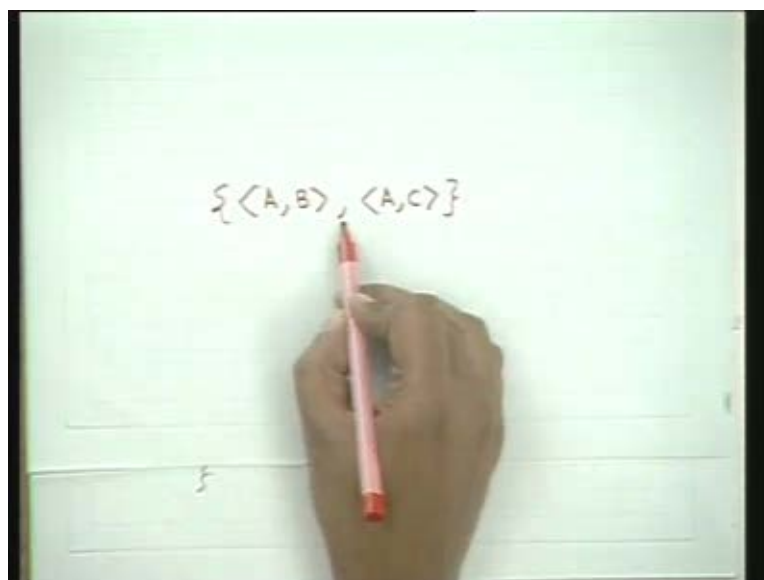
problem which when written down have got  $n$  disks on three (Refer Slide Time: 03:45) and they are called from, to and via and to solve the towers problem you had to transfer the  $n$  disk from this... to this... using this... and these are the three parameters which are character inputs in our case and  $n$  is an integer. But what does it return? It returns a list, it returns a sequence.

(Refer Slide Time 03:27 min)



So it returns a sequence of numbers of moves and a move was represented by a tuple of the form and say two moves were represented like this.

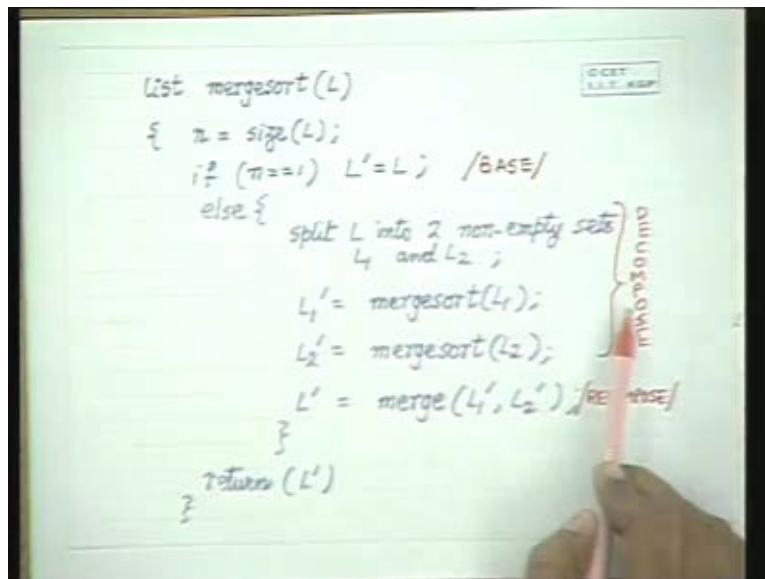
(Refer Slide Time 04:23 min)



So this was the sequence of moves which we used to represent the solution. So what it must return is a list unlike in the previous two cases which was an integer and let us remember that the list is no defined data type in a language like C but when we design the problem, we design it using our own mental structures and as we know it is as a sequence of tuples or we call it a list of tuples. And the base condition was when  $n$  was 1,  $L$  which is the returned list is only one element called from to to and the inductive conditions where when  $n$  was greater than 1. Then there we broke it up into three sub problems, first we said that  $n$  minus 1 from this peck, we move it to the via peck using the to peck.

Then we will move 1 disk from the from peck to the to peck using the via peck and then we will move the  $n$  minus 1 disk back, we had moved it to the via peck, we will move it from the via peck, we required to move it to the to peck from the from peck. And we will get, for this we will get a set of solutions, for this we will get a set of solutions and for this we will get a set of solutions and the final solution will be an appending of these three lists to form your list  $L$ . And this is the recomposition of the solution and we return  $L$  either computed here or computed here. Is that okay? So this is the, now we have written it down in a much more organized fashion. The next was the merge sort, the split sort. Here again the input  $L$  is a list of integers. What is returned is another list of integers.

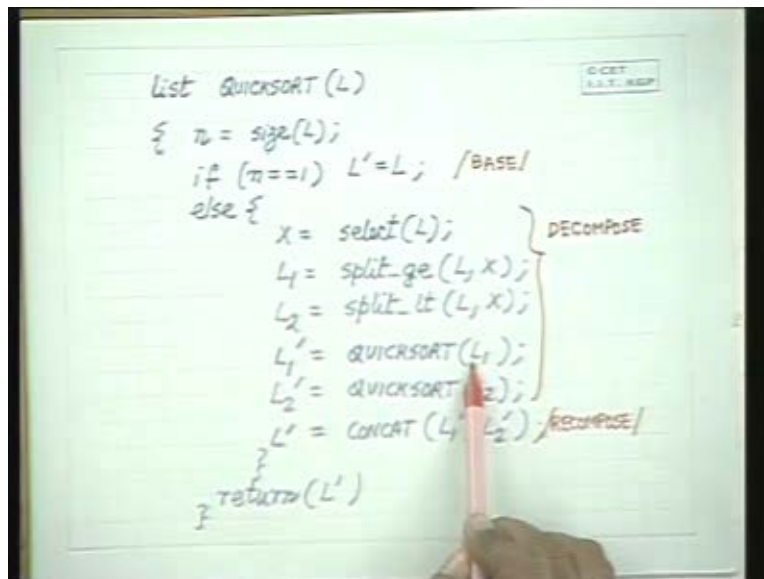
(Refer Slide Time 06:45 min)



And we have got the basis condition when  $n$  is equal to 1, the returned list is the same as the original list. That is when one element is there, it is already sorted. In the inductive condition, we split  $L$  into two non-empty list  $L_1$  and  $L_2$ . And we recursively merge sorted  $L_1$  to get  $L_1$  dash as a return sorted list of  $L_1$ . We recursively sorted  $L_2$  to get a sorted list  $L_2$  dashed and we define what was the combined routine which could combine two sorted list to form a complete sorted list. And let us now call it the routine merge because this is what it is traditionally called. So were the decomposition steps splitting and recursively merge sorted and then the recomposition of the solutions by the function merge and the list  $L$  dash is returned. Is that okay? The other version which we called quick sort which was a slightly different variation.

Here again the input is the list of integers, the output is a list of integers,  $n$  is the size of  $L$ , if  $n$  is equal to 1 the base condition is same otherwise we selected out one element of  $L$  and we created two list, one is  $L_1$  which puts into  $L$ , which puts into  $L_1$  all elements from  $L$  which are greater than or equal to  $x$ . This is what `split_ge` does, `split_lt` does what? It creates, it returns a list  $L_2$  which taking the list  $L$  and the element  $x$  returns all elements which are less than  $L$   $x$ .

(Refer Slide Time 08:38 min)



And then we recursively we quick sort  $L_1$  by the same algorithm, we could have used merge sort here also, any sorting routine but we will recursively use quick sort, we are doing problem decomposition and  $L_2$  to get  $L_2$  dashed,  $L_1$  to get  $L_1$  dashed. And now these are sorted, both are sorted and they are placed such a way that if we want to put them in ascending order, we have to just put  $L_1$  and  $L_2$  second. So we append or concatenate  $L_1$  dashed and  $L_2$  dashed and recompose the solution. So this is the approach that we used, we have just written them down in a more organized step by step fashion. So here again we have the decomposition steps and the recomposition steps.

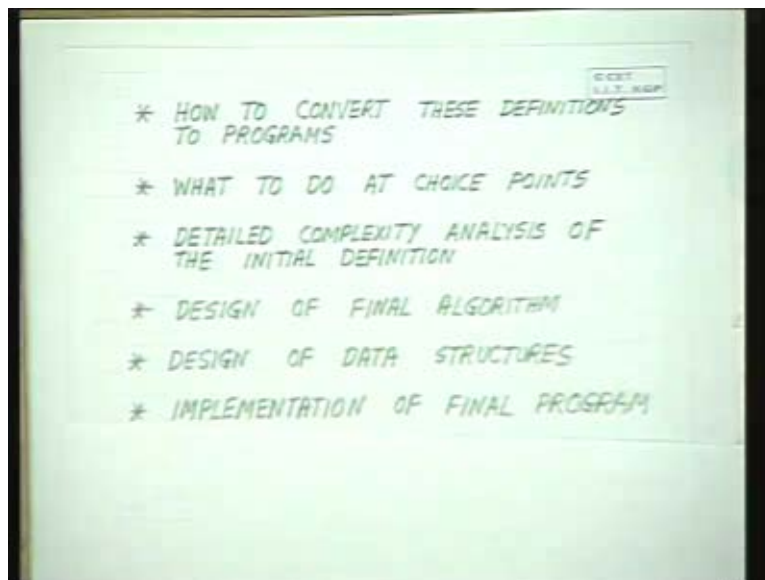
Now what's left? Is this the end of our programming. So the next thing we have to answer is how to convert these definitions to programs. And before deciding on this, we have to answer several issues. The first is before converting into programs, we have to answer several questions. The first question is what do we do with the data structure. Whenever there is integer etc there is no problem but if we end up with things like list, we have to implement this list. With integers and characters and an array, how will we implement a list is something we have to answer or we will do it in some other way. We have to return a list of what, a list of pair sub elements. How will we logically pair up these elements? So how will we write out our programs for describing these lists? Here we have just informally return them down.

So we have to write down, we have to organize our data in form of higher level structures than pure integers or characters or arrays as they are given. So this is the part of data structuring, the simplest data structuring which we shall have to address. The second question which we will

have to answer, so first is directly how do we convert it into programs? So we will have to answer this question and we will see what we will do at each and every place. Second is what do we do at choice points. Now let us see what choice points we had? Split a list into two non-empty list  $L_1$  and  $L_2$ . Where do we split the list? Half half, one third, two third or all are equal or anything. Does it make any difference? We have got a choice of splitting here, what do we do? Does it make have any effect, we have a choice of selecting an element  $L$  here. Which element do we select? Does it make any difference to the final algorithm, if we select this element? So we have to answer these questions. We can do it arbitrarily but as we saw in our example of max and min and max and second max, a deeper analysis really helps to get a better understanding of the solution.

So we have to answer what we do a choice list. In order to answer that this is what we have to do. we have to analyze this initial definition as we call it and do a complexity analysis, complexity means how much time when an algorithm take if we take this choice point, if we take that choice point, if we use this data structure for a list, if we use that data structure for a list. What are we going to do? So unless we do this we are never going to reach our required goal of getting the best possible solution. And only what after we have done this, we have made two things. We have been able to choose the final algorithm by choosing what we do at various choice points and we will be able to design our final data structure. How do we implement that list, how do we implement this elements from and to those pairs, which is the best way to design them?

(Refer Slide Time 13:31 min)

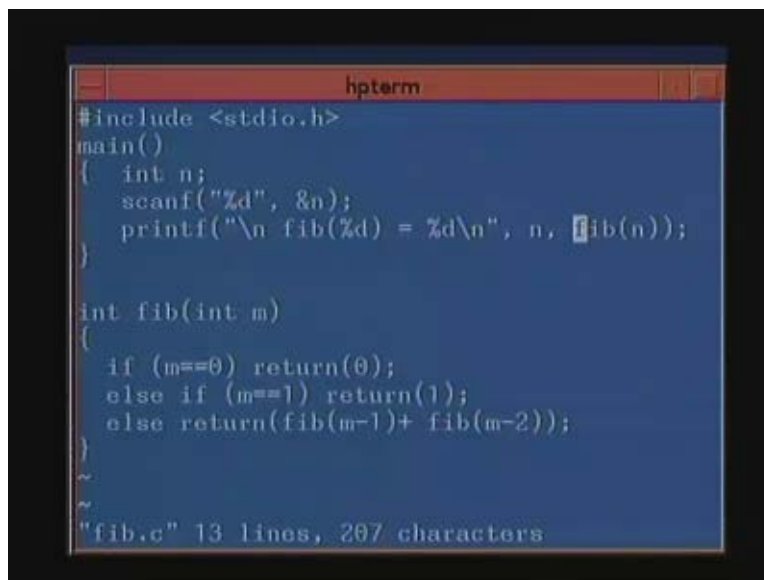


So after having come up to this point, we are ready to design the final algorithm and the final data structures. It is at this point that we turn to writing of the final program. It's a long way to go till we proceed to do this and we have not yet learnt many things to proceed even beyond this point. So, with a possible knowledge that we have at present of arrays and other things only integers and arrays, we will see how we can convert these definitions to programs at the level that we are in. And once we have got an initial working program which is correct then we will go

step by step to understand what we have to do here, what mathematics is involved to do this complexity analysis. But before that we have to learn some more of programming. So today we will quickly see how we can convert these definitions to programs with whatever knowledge we have and then we will see what other knowledge we require to see other avenues and that's what we will study in the subsequent classes.

So today we will just quickly go back and see the programs that we have. The factorial one is obvious, so we just pick the Fibonacci number which is equally obvious. If you have to directly convert it to a program, there is nothing you can just simply write this down in C language and you will get a working program because this is going to return an integer and whatever we have in our knowledge, we can simply convert it to a program. There's nothing to do, we can directly convert it into a program. We are not analyzing the program to see whether this is a best possible, whether there are any choices we have that will come to later but today we will just see how to just directly convert it to a program and this one is obvious. So we will see slightly different variation of this.

(Refer Slide Time 16:02 min)

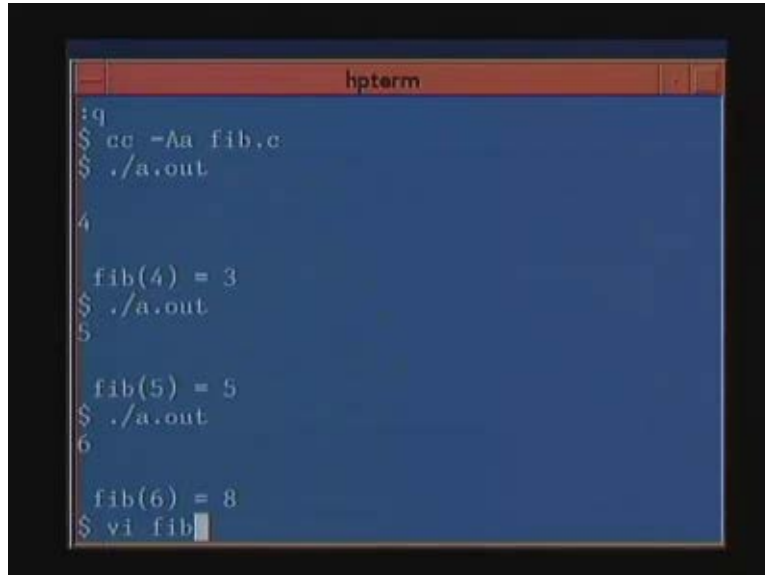
A screenshot of a terminal window titled 'hpterm' with a dark blue background and white text. The code displayed is a C program for calculating Fibonacci numbers. It includes a main function that reads an integer 'n' and prints the result of fib(n). The fib function is recursive, returning 0 for m=0, 1 for m=1, and fib(m-1)+fib(m-2) for m>1. At the bottom, it shows the file size: "fib.c" 13 lines, 207 characters.

```
#include <stdio.h>
main()
{
    int n;
    scanf("%d", &n);
    printf("\n fib(%d) = %d\n", n, fib(n));
}

int fib(int m)
{
    if (m==0) return(0);
    else if (m==1) return(1);
    else return(fib(m-1)+ fib(m-2));
}
~
"fib.c" 13 lines, 207 characters
```

So here is the Fibonacci program, you have the main program. Visible? So, we have the main program and in the main program we just read and in the printf function itself, we called fib n that's possible in C. And this is the fib n program where fib m actually this is the parameter, base condition if m equal to 0 return 0 else if m equal to 1 return 1 else return, I have just written out a more compact program just to show you the way it can be written in C. So if m minus 1 plus fib m minus 2, so this will give a solution even if you have return fib m minus 2 plus fib m minus 1 even that would have given you the solution, both are equivalent.

(Refer Slide Time 17:18 min)



```
hpterm
:q
$ cc -laa fib.c
$ ./a.out
4
fib(4) = 3
$ ./a.out
5
fib(5) = 5
$ ./a.out
6
fib(6) = 8
$ vi fib
```

So you could have, this will return if fib m minus 1 plus fib m minus 2, in a return statement you can put in an expression. And if, so if we just run this, it doesn't ask for and so fib 5 is 5, fib 6 is 8. So whatever you do you will get it, just a point which I wish to mention here. Is that if I replaced this by 2 and this by 1 even that would have been correct. It doesn't matter if I did fib m minus 2 first and then m minus 1 second because both are independents of problem this will return one solution this will return another solution and the solution will get added. This is because addition is a commutative operation because addition is commutative, you can do any one of them in any order. So if I do it in any order also, I will still get the correct solution. So that's fine. So I think that's quite easy to understand if we do this Fibonacci.

So let's come to the tower of Hanoi's problem. The first problem that we encounter here is a list, we have to return a list of elements, we still don't know how to return a list of element unless we declare an array. We can declare an array for the solution and this can return an array alright but do you know the size of the array that you have to declare. What is the solution, if for tower Hanoi's of n the total number of moves is... So the size of the array that you have to declare is exponential to n. Any way you can declare an array, input the solution and return an array and I leave it to you to decide to find out in C language how you declare an array to be return from a function. And then you can declare three arrays and based on these three arrays you can write and append program to combined three arrays into another array, alright.

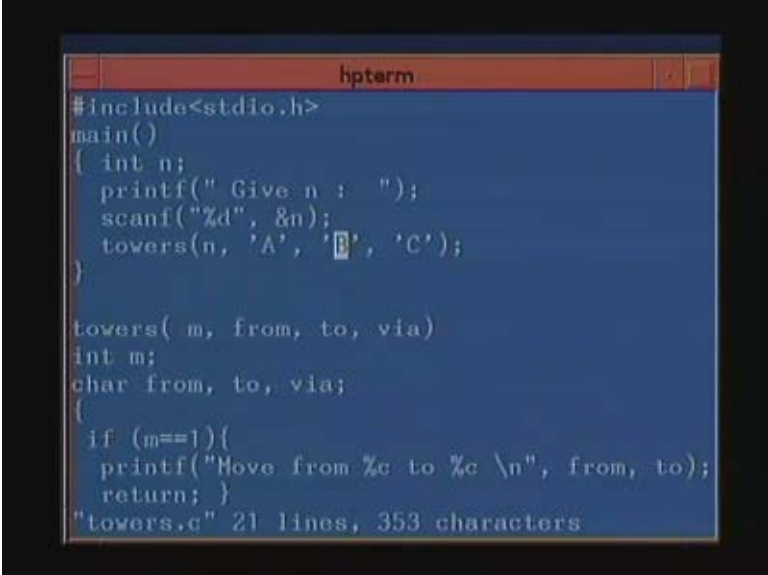
Now if you implement it using arrays, one very interesting point like in the Fibonacci numbers, if I did this one first or this one second it does not matter. If I get back the solution and I do  $L_1$   $L_2$   $L_3$  does it matter if I do this one first, this one if I write it down in another order. I have to just do this append in a proper order. If I do the append in the proper order, I am going to get the solution. Is that okay? Now I just take a different approach to using a list. What I will do is instead of using a list; I will just print the solution. If I have I will not return anything. Tower Hanoi's here if n is equal to 1; it will just print from to.



I don't want to use an array of such a huge size; I just want the result to come on the screen. So if n is equal to 1, instead of 1 is equal to this it will just print from to, otherwise it is recursively call this one, recursively call this one and recursively call this one. This append is no longer required because the solution are printed from inside but once I do that it becomes very important to write these three in proper order because if I write them in reverse order then since the append function is not there and printing is done inside the routine, it will work out in the order in which I ask it to give it to the recursion. Is that okay?

So if I have to just simply print it then I will call this, call this and call this. And instead of this append I will have nothing and instead of this I will just have a print and that will solve my problem. We will go back to using a list when we see is this the best way to solve this problem or not when we have to answer this question, we will come back and see something else but now whatever knowledge we have at our disposal, we will just use it to write a program which works.

(Refer Slide Time 22:06 min)



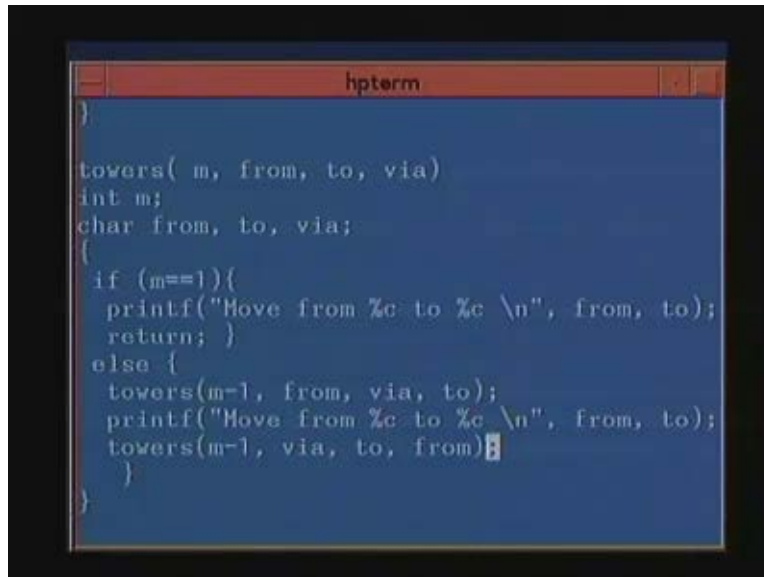
```
hpterm
#include<stdio.h>
main()
{ int n;
  printf(" Give n : ");
  scanf("%d", &n);
  towers(n, 'A', 'B', 'C');
}

towers( m, from, to, via)
int m;
char from, to, via;
{
  if (m==1){
    printf("Move from %c to %c \n", from, to);
    return; }
  towers(m, from, via, to);
  towers(m, via, to, from);
}
"towers.c" 21 lines, 353 characters
```

So let's see how that program will look like. The main program is here, n it reads in n and calls tower n A B and C and this one is the towers routine which takes in m as an integer and three argument as I mentioned before and we have declared them to be characters because if you notice I will call them with characters A B and C. We are introducing character possibly for the first time in this course but I think this part is quite clear and obvious. So we can declare characters by the word char and these are all single characters from is a one character, to is one character and via is one character. And if m is equal to 1, what did we say? We will write a print, printf move from % C is the format to write a character. So move from % C to % C and the first one is the from and the second one is the to or else call towers recursively from via and to and this one you could have called it with m equal to 1 but I have just return out the m equal to 1 case specifically here again. I have just repeated this statement here instead of calling m equal to 1 which is the something and here I have called towers via, to and from. So this is the recursive, direct recursive version of the towers Hanoi where the list input is not taken and we just print the output. And we have to be very careful so that we do it in this particular order because if we do it

in any other order, we will get the problem solved in a different order because recursion will first do this then do this and then do this.

(Refer Slide Time 23:58 min)

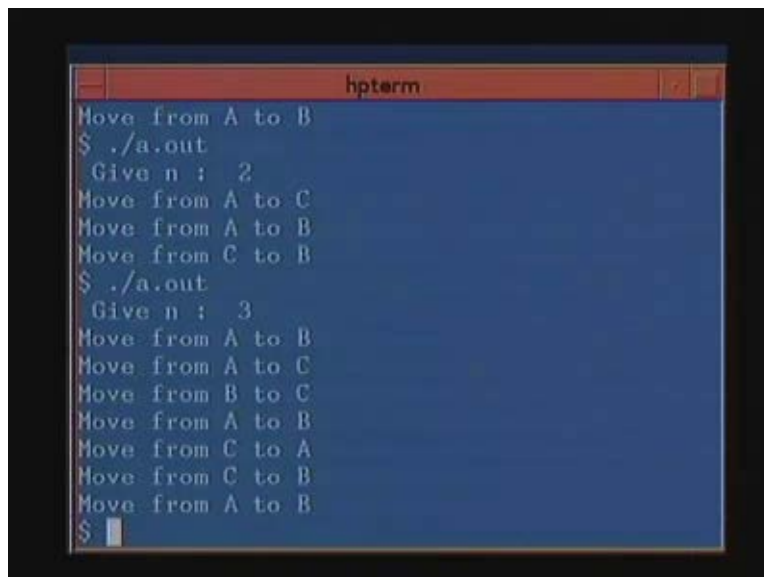


```
hpterm
}

towers( m, from, to, via)
int m;
char from, to, via;
{
  if (m==1){
    printf("Move from %c to %c \n", from, to);
    return; }
  else {
    towers(m-1, from, via, to);
    printf("Move from %c to %c \n", from, to);
    towers(m-1, via, to, from);
  }
}
```

And inside it the printing will take place, so suppose you give 1 you get this, you give 2 A to C, A to B, C to B.

(Refer Slide Time 24:29 min)



```
hpterm
Move from A to B
$ ./a.out
Give n : 2
Move from A to C
Move from A to B
Move from C to B
$ ./a.out
Give n : 3
Move from A to B
Move from A to C
Move from B to C
Move from A to B
Move from C to A
Move from C to B
Move from A to B
$
```

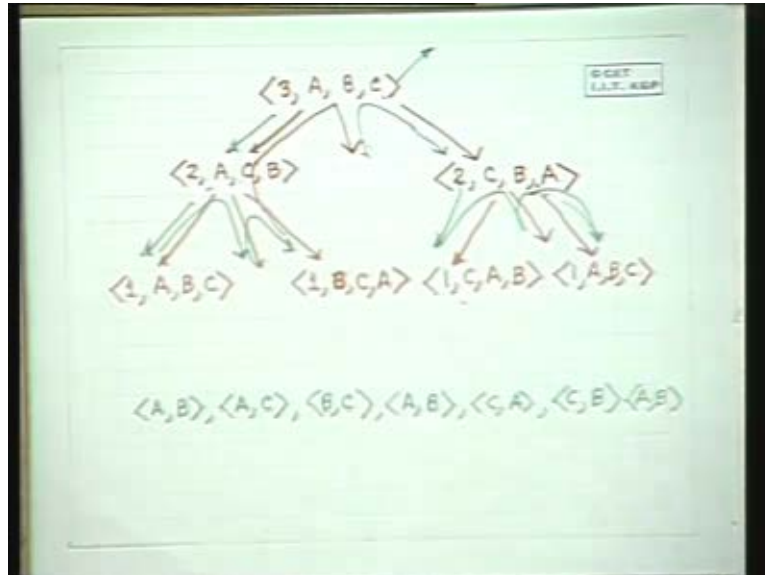
You give 3, you will get the 7 moves and similarly if you give 4 we will get all the moves. So let's quickly go back and see that doing it this way really gives the solution instead of writing out lists and all that. Do I need to go through the recursion once or is it okay? Okay. So what did we

have? We called it with 3 A B C and the recursive program; I will use this to highlight the recursion. We first call tower n minus 1 from via to then we print it and then we do this and the base part we directly print it. So let's see how it works if we do that, exactly how the program executes. So first this part will get called 2 A to C to using B, so the call will get transferred here. So this will again start executing, so this starts executing it is not the independently these are three called. So this start executing as recursion goes, so 1, A, B, C from, to, via. The first call is from, via, to and this starts executing. Now this is the base condition and what happens? You are supposed to return this instead of that we are printing this.

So first we print A B, this gets printed. Remember this was going to be appended in the beginning of the list, this was going to get appended then something, then something that was going to come back here, so this could have come in the beginning of the list and this is going to get printed first. Then so once this completes the second part of this executes which in our program instead of calling one this, this we simply print it A C. Remember if we called recursively also, A C would have been returned and A B would have been the first element here, A C second, this would have come back here and A C would have been a second element in the solution and A C is going to get printed now.

Thirdly we going to call 1 C sorry B to C, this is what is going to happen, via to from. The second recursion is via to from, via to from. This is from, this is to, this is via, so via to from. So after the control initially it came here then it came here, then it came here, then it came here and so what will now get printed because this is the base condition B C and then control will go back, this part is done fully, goes back here and the next sub problem is generated which in our cases 1 A B C which is print A B. Then this part goes back, control goes back to this side and the new sub problem that is generated is 2, via, to and from. This is the inductive condition, so you get a recursion call is met 1 C A B. You are here, so the control now comes here, this is the base condition this gets printed. So then control goes back and the next sub problem gets generated, C B gets printed because 1 from to so C B gets printed.

(Refer Slide Time 30:29 min)



So this part control goes back here and the last sub problem for this which is 1 via to from and this is generated, this is the base sub problem so this gets printed. This completes, control goes back here, this completes control goes back, here this completes control goes here and you got your solution. So this is how we have used recursion to solve this problem where the list instead of the list, we simply printed the solution alright. So control our recursion is very important, you must understand exactly how recursion works in order to convert your problem into recursion. And on other hand if you had used arrays it wouldn't have mattered in which order you solved but it matters how you decomposed them. The recursion that we have implemented, here only the recomposition automatically takes place by printing.

So we have used recursion to solve the problem not only of decomposition but also formation of the solution with our knowledge of recursion. So, problem decomposition recursion in programming are not identical. We have to be careful and you have to use recursion to solve your problem. Therefore the towers of Hanoi's solution which showed in the computer just now is a method by which we used recursion to solve our problem. We are out of time today, so we will just see how quickly in the next class how we will solve the merge sort and the quick sort problems. How we will implement the list and then we will have a discussion as to what we can do about this. So we are now trying to get some initial solutions and once we are able to do that, we will understand what we lack in terms of programming facilities and we will acquired them quickly how to form a list, how to forms structures of elements and then we will be able to solve this problems and analyze them better.