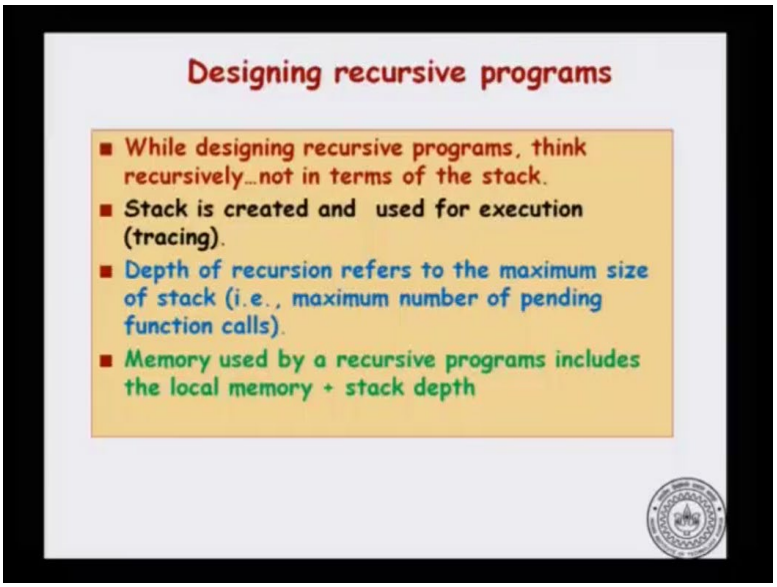
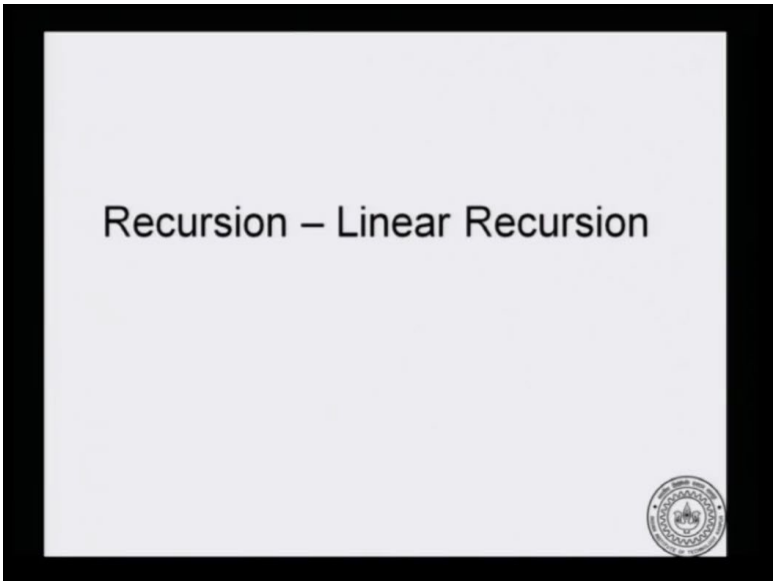
Introduction to Programming in C
Department of Computer Science and Engineering

(Refer Slide Time: 00:06)



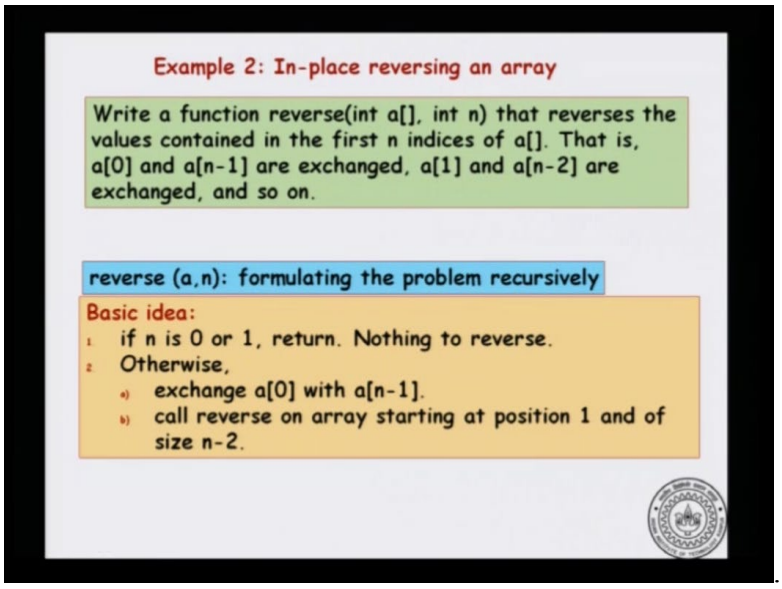
In this video will look at linear recursion in a bit more depth, while I describe what I mean by linear recursion. We have mentioned earlier that when designing recursive programs, think about the problem in recursive terms, do not think in terms after stack that is used in execution. When it is actually executed that will be a stack created and use for the execution, and that depth of recursion is a term which means the maximum size of the stack, while you execute the program on given input. The memory used by the programs includes the local memory of the function, + the depth of the stack.

(Refer Slide Time: 00:51)



So, let us look at linear recursion in a bit more detail. By linear recursion I mean problems which can be solved by calling, and instance of the sub problem, exactly one instance of some sub problem. We will see more general kinds in later videos.

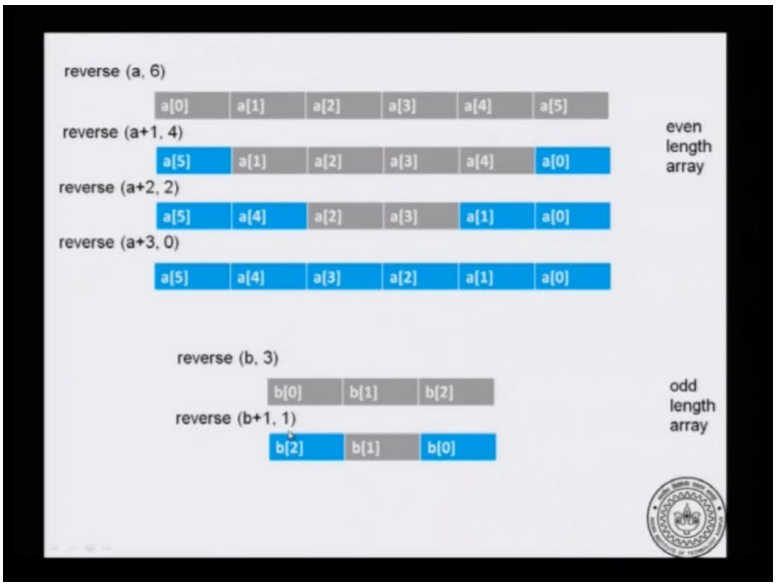
(Refer Slide Time: 01:11)



Let us look at an example that we have seen before, which is reversing and array in place. So, we had to reverse an array a with n elements, and it is suppose to reverse the values contain in the first n indices of a; that is a[0] is exchange with a[n-1] a[1] is in exchange with and so on. So, we have to do n/2 exchanges approximately. So, let us look

at the problem recursively, we had earlier solve it using loops. Now the basic idea of the recursive solution to, in place reversal, is the following; if n is a[0] or 1 if thus arrays either empty are it contains exactly one element, then we do not need to do any think to reverse theory; otherwise it contains at least two elements. In this case exchange a[0] with a[n-1]; that will be the first layer. Now call this sub problem. We have to solve one more sub problem which is reverse on an array, which is from a[1] through a[n-2]. Notice that we had already solved the problem of swapping a[0] and a[n-1]. So, earlier we had seen a program which involved linear recursion, which just bend left to right. In the case of reversal f on array, it is still linear recursion in the sense that there is only one call to a sub problem, but the way in which you call the sub problem is slightly different.

(Refer Slide Time: 03:02)

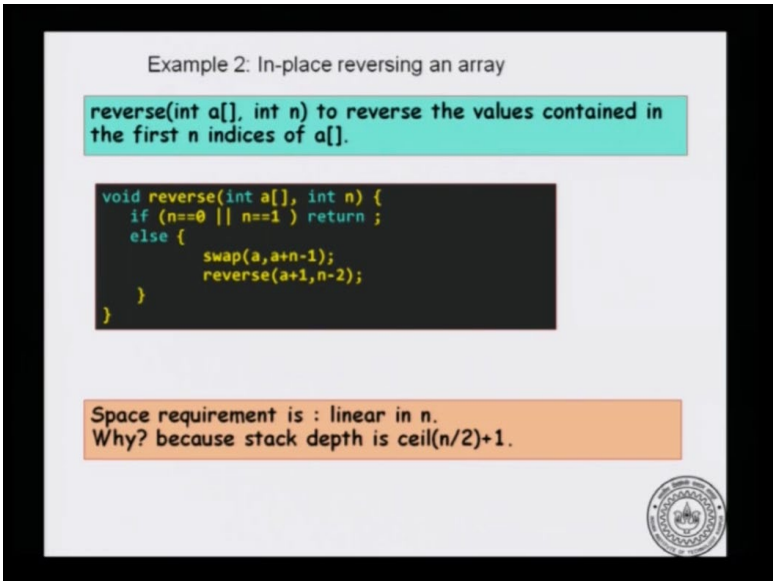


Let see with an example. We will consider actually two examples; one for even length array, and another for an odd length array. Let say that a is an even length array with six elements and we want to reverse it, using the function reverse a[6], and we have to do it in a recursive way. So, what you do is first swap a[0] with a[5]. And now what is the sub problem left be solved. We have to solve reverse of this intermediate array, which starts from a[1] and contains four elements. So, we have to reverse the array with starts from a + 1, and there are four elements to be reversed. So, in one step even though we have only a single call to a sub problem, we have actually reduced the size by 2. Now use a reverse or rather swap a and a[1] and a[4], and now the sub problem that remains is, to reverse this sub routine which is a + 3, and you have two elements to reverse. So, you do this,

and in this point you have a sub array which starts at a + 3 and has zero elements to reverse. At this point that is nothing but.

Now, for in odd length array let us take a very small array which contains three elements, and we have to reverse it. What you do is, you reverse you swap b[0] with b[2]. At this point you have a sub problem which has exactly 1 element, and you do need to reverse that array, that arrays it is soon reverse. So, the problem just stops there. So, notice that difference between the even length array the odd length array. In the case of even length array, the step, just before the last step involved an array of size 2, and you still had to reverse that is array. In the case of an odd length array this, the last of involves has a single length array, which is soon reversed. So, you do not have to do anything. So, there are two base cases to worry about; one is where the sub array is of size zero, and another is where the sub array is of size 1, 0 corresponds to even length arrays, and one corresponds odd length arrays.

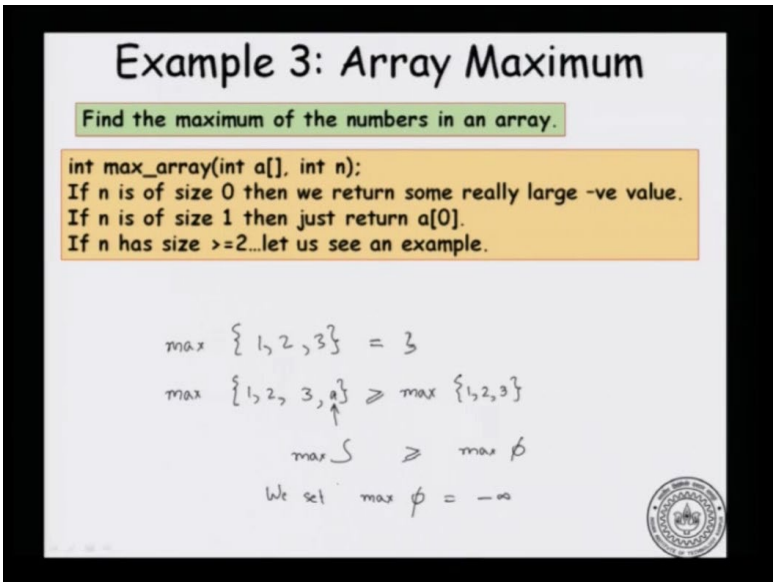
(Refer Slide Time: 05:31)



Let us write this code now. So, we have reverse a containing n elements, and we have return type void, which means that this function is not going to return you value, but it is going to do something. So, if n=0 or n = 1 return, because in that case a is, it is on reverse; otherwise you swap the first element with a last element, that is this operation a and a + n - 1. So, notice that swap is a function that takes two pointers to int and exchanges them. Once you do that you call the sub problem, which is reverse(a + 1, n –

2). Notice that unlike the previous examples we have discussed, the sub problem reduces by 2 insides. Even though you have a only single call, the sub problem is not of size n - 1, it is of size n - 2. So, look at the case of the odd length array and the even length array that we have seen before. And you can notice that the sub problem reduces by 2 in size for every step. Now what is the depth of the stack. you know that ruffle n upon to calls will be done, because you start at a size n, the next call will be of size n - 2 and so on until you hit either one or zero. So, you can work out that there will be about n/2 steps, before you reach one or zero. The accurate expression is, ceiling of the expression n / 2 + 1. So, many calls will be there, before you hit 1 or 0. So, each function call will take, let us a constant among space and there are about n/2 function calls. So, the stack depth is n/2, and therefore, the wholes space which is stack depth times the number of variables at each function that will be about n/2.

(Refer Slide Time: 07:58)

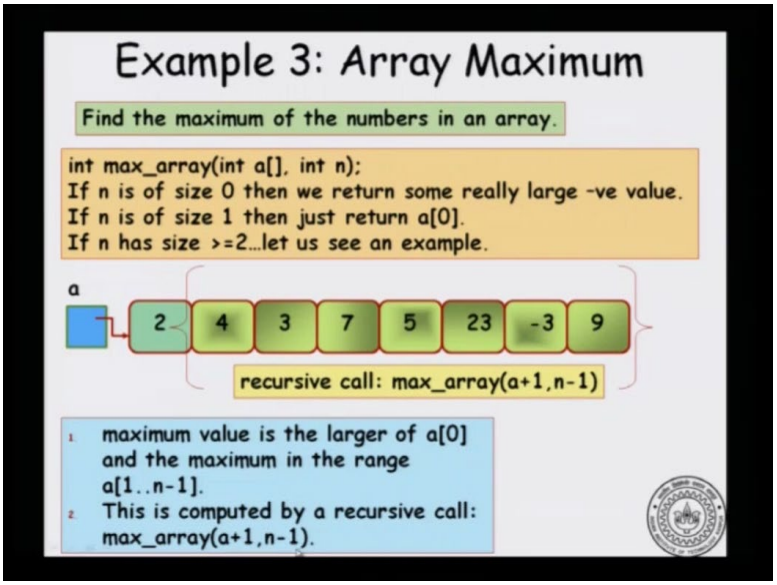


So, now let us consider a third example which is, computing the size, the maximum of a particular array. For concreteness let us consider in integer array, and we have to compute the following function int max array. It takes two arguments; one is the array itself, and the second is m, which is the number of elements in the array. Again let us think about the problem recursively, we have return loops to solve the problem earlier, but now let us think about it in a recursive manner. If the array contains 0 elements, then what is the maximum. So, here it may be slightly counter intuitive if you are saying for the first time. The maximum of an empty array is some large a negative value. Think of

it has minus infinity. Why do we do this this is, because let us take a concrete example 1 2 3. We know that the maximum of this array is three. Now, what happens when you take a larger array or list of numbers. So, what happens if you take, let us keep this unspecified a is an int. You know that if a is less than 3 then the maximum of this array is going to be three. If a > 3 then the maximum of this the second one is going to be greater than that. So, in any case, whatever be the nature of a you can always say that maximum {1 2 3} a is going to be >= the maximum {1 2 3}. Now what; that means is that, if you take a larger set, its maximum is always going to be >= the maximum of a sub set.

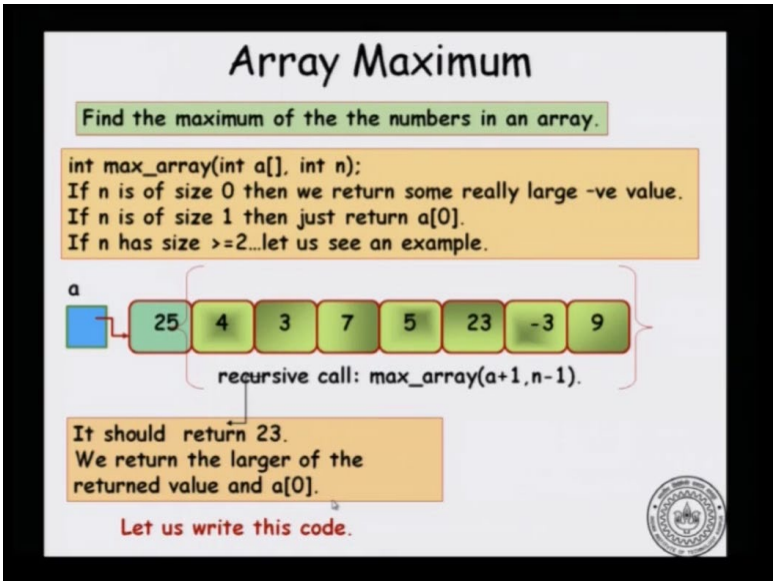
Note that this is independent of a, because you can analysis my cases, if a <= 3 then this maximum will 3 itself and 3 >= 3. If a > 3, then this maximum is strictly greater than the previous maximum. So, maximum is always monotone according to the sub set relation. Now this means that what will be the maximum of the empty set. The empty set is a sub set of every set. So, no matter which s I pick, maximum s has to be >= maximum of the empty set. This means that a reasonable value for maximum of empty set is minus infinity. So, the set... So, this is a reasonable convention; that is why when n is of size zero, we returns some really large negative value. By which I mean the absolute value of the number is really big, because we are trying to say that it essentially minus infinity. If n is of size 1, then you just return a[0], because the array contains only 1 element, it is maximum will be a zero. If n has size at least 2. Now we are in business, we have to solve the problem in terms of a sub problem. So, here was an example where the base cases had to be really thought of, but now we are at the case where we are thinking about the recursion. So, what is the recursive step here.

(Refer Slide Time: 12:10)



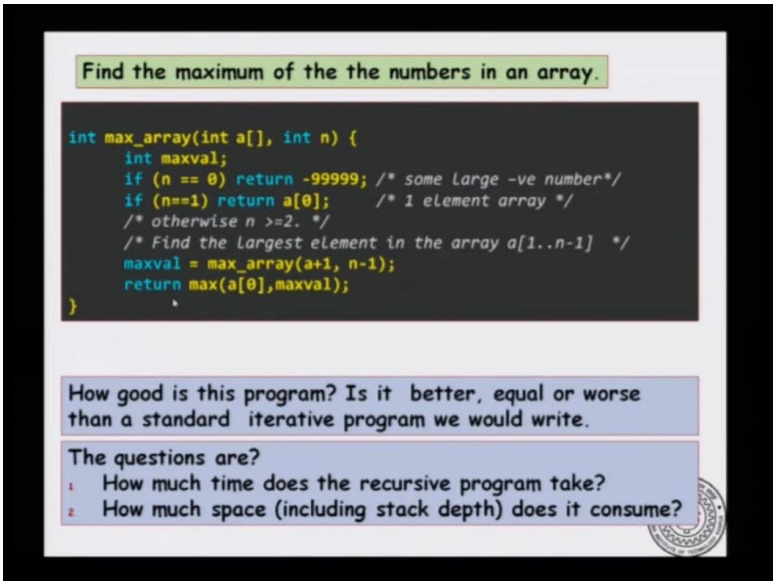
So, let us take a concrete array. We have array a, which contains the numbers 2 4 3 7 5 23, - 3 and 9, some concrete array. And I want to calculate the maximum of the array a in terms of some sub problem. The natural sub problem that we can think of, is the sub problem of finding the maximum of this sub array, which start from a[1] and goes on until the last element. So, recursive call should be something like max array a + 1, and there are n - 1 elements in it, because we omit the first element. Now, maximum value, how can we solve the whole problem in terms of the sub problem. Suppose we note what is the maximum value in the tail; a + 1, 2 containing n - 1 elements. The maximum of the whole array will be the greater of the two numbers, which two numbers, the maximum of this sub array and a[0]. So, maximum value is the large of a zero and the maximum of the tail sub array, which is a + 1 to a + n - 1. Now in order to compute the sub problem we called a recursive call to the same function, looks for the max array from a + 1 containing n - 1 elements.

(Refer Slide Time: 13:55)



So, in this example, the maximum of the tail sub array will be 23. And let say that a zero is 25. So, the maximum of the whole array will be the greater of the two numbers 25 and 23. So, in this case, the maximum value will be 25 which is a[0].

(Refer Slide Time: 14:26)

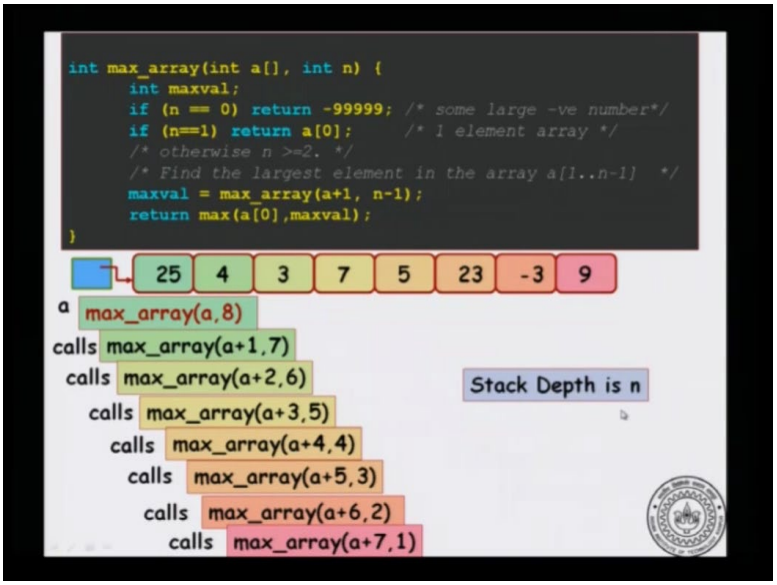


So, now let us write this code. So, the recursive function is very simple, and this is one of the reasons why people like to write recursive functions, because from a recursive function it is very clear what the function is going to do. Usually recursive functions are shorter than their loop versions, and they are easier to understand when you read

someone else code. So, let us solve max array using the recursive function in c. We have int max array, because it is finally, going to return in int value which is the greatest value in the array. Now you have an int array a, and n is the size of the array. Let say that we set some max val if n is 0, then the maximum is simply something like minus infinity. Let us keep it a very large number - 9 9 9 9 9. So, - 5 9, some large value does not matter, and then if n = 1 then the array contains only one element and therefore, it is the maximum. So, you just return a[0]; otherwise n is at least 2. So, in this case, you say that the maximum value of the sub problem is max array a + 1, n - 1. So, this is the maximum of the tail array.

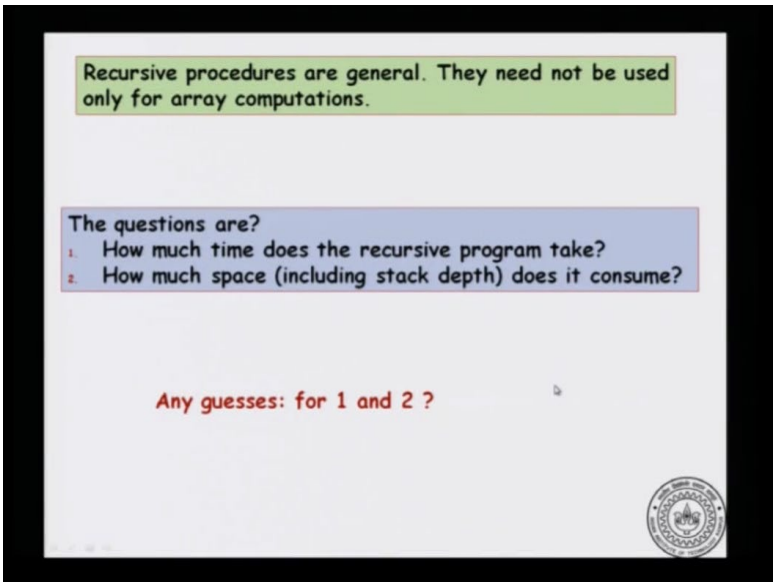
Now once you have the maximum of the tail array, the maximum of the whole array is the grater of the two numbers which is a[0] and max val. So, we return max{a[0], max val}. Now max is a function that is already there in the standard math library in c, but if you want to write it, it is not a difficult function to write it, you can take two integers and return the greater of the true integers. Now we can think about is a better then the loop version of the program. The advantage of the recursive program is that, it is easier and in some sense it contains fewer number of lines then the loop program. The disadvantage is that it takes subs more space while executing. So, the questions are how much time does the function take, how much space does the function take. So, these are things which are concrete and can be measured, there is also software question which is, how you see is set for programmer to look at this function and understand what it does. In the second criteria and it is the recursive function that is course. In the first criteria it is often the iterative function, the loop function that is course.

(Refer Slide Time: 17:39)



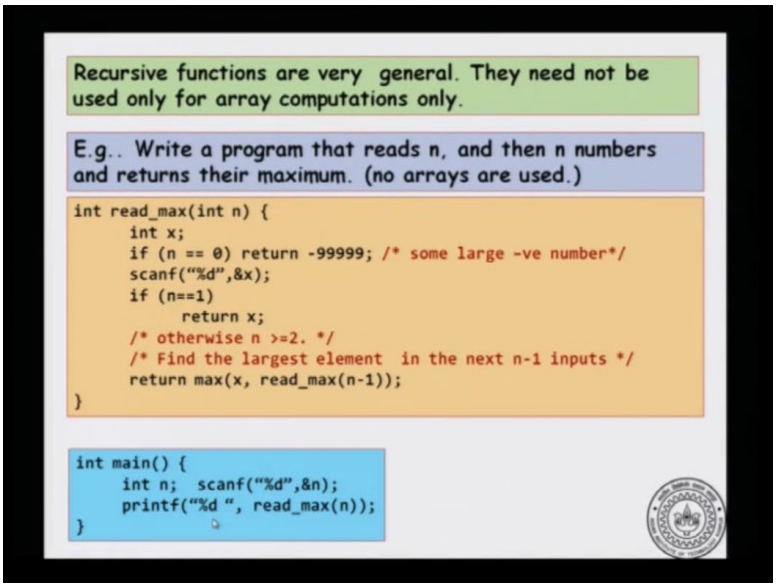
So, please think about the questions, and you can work through it and say that in order to solve max_array of an array of size n. Let us take an array size 8, you will see that these other recursive calls it will make (a + 2,6) so on up to (a + 7,1), and when you hit an array of size 1 you get to one of the base cases, which is that when you have an array which contains a single element, the maximum is the only element in the array. So, once you hit here, you will start returning. So, the maximum depth of function calls in this will be the size of the array. So, you can say that stack depth is n.

(Refer Slide Time: 18:25)



Now, recursive programs are general programs, just like look loop programs are general programs. You have return loops even before you saw what are arrays in C? Similarly you can write recursive programs which deal with general data, not just array data. And in all of these questions, you can ask the following question how much time does the function take, and how much space does the function take.

(Refer Slide Time: 19:01)

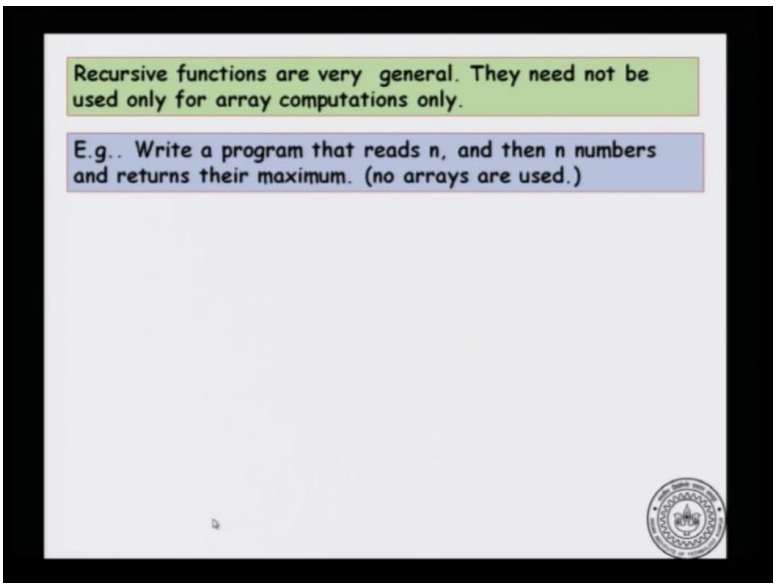


We will see an example for a recursive function, that will read n numbers and returns the maximum. Before we came to know of C arrays, this is the kind of loop functions that we used to write, we would take n numbers. So, first you will read how many numbers to read, then you will read exactly those many numbers and find their maximum using a loop. Now Let us try to do that using recursion. We are not going to use any arrays. So, what we have to do is, write a function to read_max, it takes n elements. And the logic is the same as finding the maximum of an array, but we will do it without using arrays. How do you do this. If you have zero numbers to read then you return minus infinity, or some approximation, some large negative value; otherwise you read the first number. If n = 1; that is we have to read only one number, then you just say that x the maximum; otherwise n >= 2, and we have read one number.

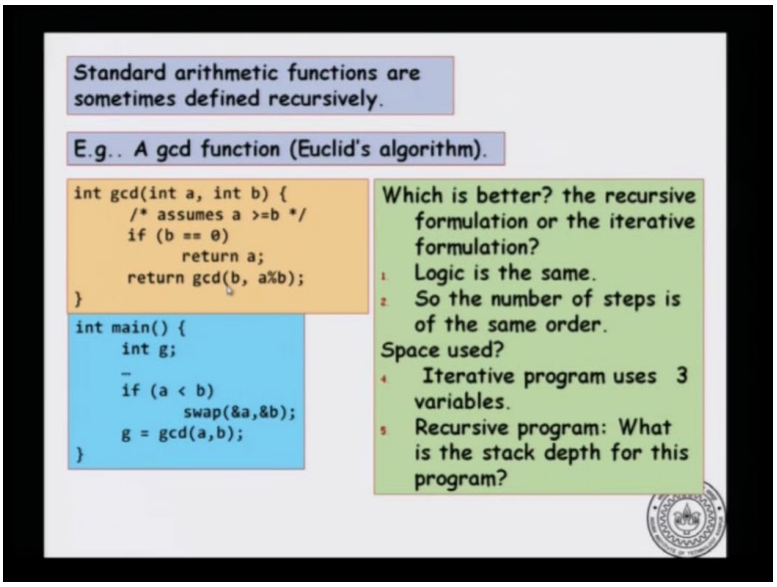
So, you say that return the maximum of the two values, which is x and what goes inside, inside you have to solve a sub problem, which is the sub problem of reading n - 1 numbers and returning the maximum. Go back and compare the program with finding

the array maximum, and the recursion works exactly in the same way. So, we will read n - 1 numbers, and return the maximum of those, and then you compare maximum of the first number and the maximum of the sub problem. This is exactly as before except that we did not use any arrays. And how do you call this function, you just declare a main function with n, you scan it how many numbers to read, and call the function read max n. Finally, it will return the maximum of the n numbers read and you just print the value. So, think about this for a minute, and see why we did not need to use arrays.

(Refer Slide Time: 21:29)



(Refer Slide Time: 21:40)



Now, there are other functions which are typically return in a recursive manner. We just saw that you can use recursion with arrays. We saw problems where, you do not need to use arrays, but you can still write a recursive routine. We now will come to arithmetic functions, and many arithmetic functions are of an recursively defined. For example, let us take to function Euclid's algorithm, and you can write the GCD function as follows. You first ensure that a >= b using the swap function, and then you just call gcd(a,b). And gcd(a,b) is defined a recursively as follows. If b = 0 then gcd(a,b) = a. If b is non zero then you just return gcd(b, a%b). So, this is how you write recursive GCD routine. And I will make the clean that this routine is cleaner, then the iterative routine. In the iterative routine, remember we had to use an intermediate variable, which will store the value of, let say a and then did a careful three way exchange in order to accomplish b, a . a % b.

Here the code is very simple, if b is zero then we know the gcd(a,b) = a. If b is non zero then we know the gcd(a,b) is gcd(b, a%b). So, it is a very concise way of writing the function. Now you can ask the question which is better, is the recursive formulation or the iterative formulation. Logic is the same, so it will take the same number of steps. So, the time taken will roughly be the same. And we have also made the clean that there recursive version is easier to understand. The disadvantage may be the following that, the recursive function may use very deep stack. So you can ask the question like how deep will be the stack in the case of the recursive program. So, in the following video, we will talk about more general kinds of recursion. In this video and the previous video, we have seen recursive problem. So, it can be solve by one call to a sub problem, and we will see more general kinds of recursion.

Thanks.