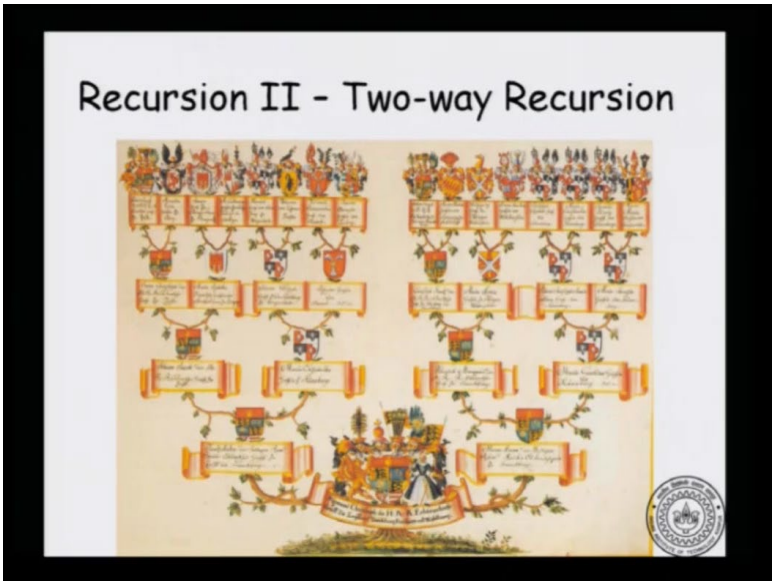
**Introduction to Programming in C**
**Department of Computer Science and Engineering**

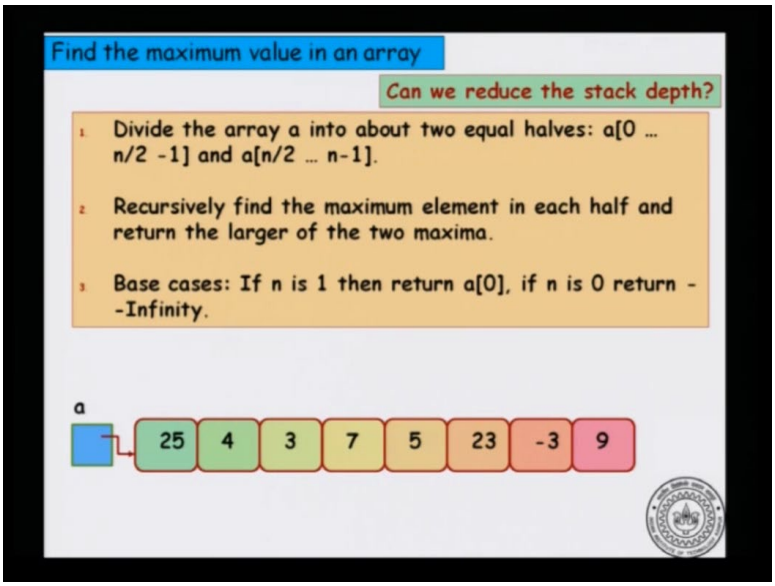
In this video we will look at slightly more general way of defining problems through recursion.

(Refer Slide Time: 00:10)




We will, for the lack of a better name, I will call it just two-way recursion. These are problems which are solved by calling two-sub instances. This is the picture of a family tree, and we will see that the call stack for a two-way recursive functions looks somewhat similar to a family tree.

(Refer Slide Time: 00:32)




Let us revisit a problem that we have seen which is to find the maximum value in an integer array. We saw that the stack depth in our earlier solution was order n, because each problem of size n called once at problem of size n - 1. Now, can we reduce the depth of the stack from something close to n to something smaller than n.

So, here is an alternate way to look at the problem which can be described in a very simple way. Instead of looking at the maximum of the first element and then the tail, what I can do is, take an array of size n and split it roughly in 2 halves. So, there is left half and a right half, each of size n over 2.

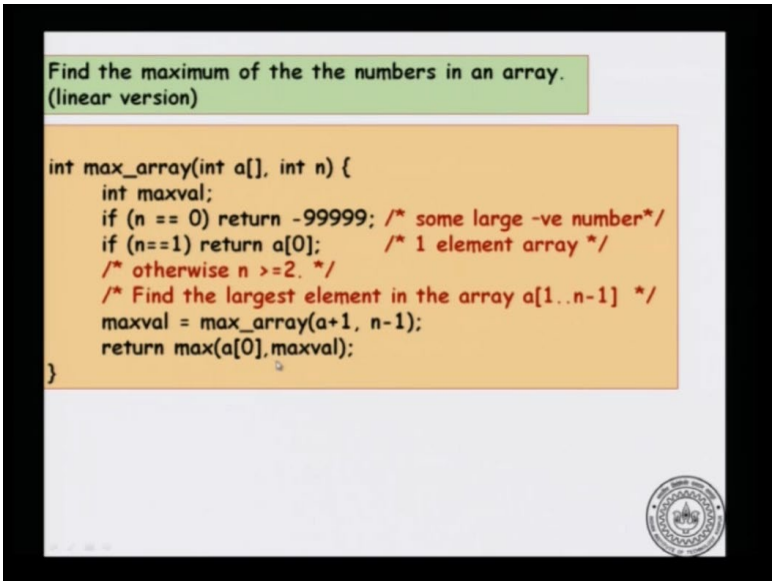
Now, imagine that you have the solution for the greatest element in the first half, let us call that x. And imagine that you have the greatest element of the right half, let us call that y. Now, whichever is greater among x and y, is going to be the greatest in the whole array. And this is the idea that we are going to implement right now.

So, divide the array into about 2 equal halves. The first half is 0 to a n/2 -1; this contains n/2 elements. And the second half is, a n/2, so on, upto n - 1; this is the right half. Now, recursively find the maximum element of each half. And let us say that you have x which is the maximum in the left half and y which is the maximum in the right half, then you just return the larger of x and y, that should be the largest element of the array.

While doing this we have to take care of the base cases. This is as before for the linear case; when n = 1 then the only element in the array is the maximum element, so, return

a[0]. If n = 0 that is the array is empty, you return minus infinity. So, let us consider a concrete array; a, is an integer array with these elements.

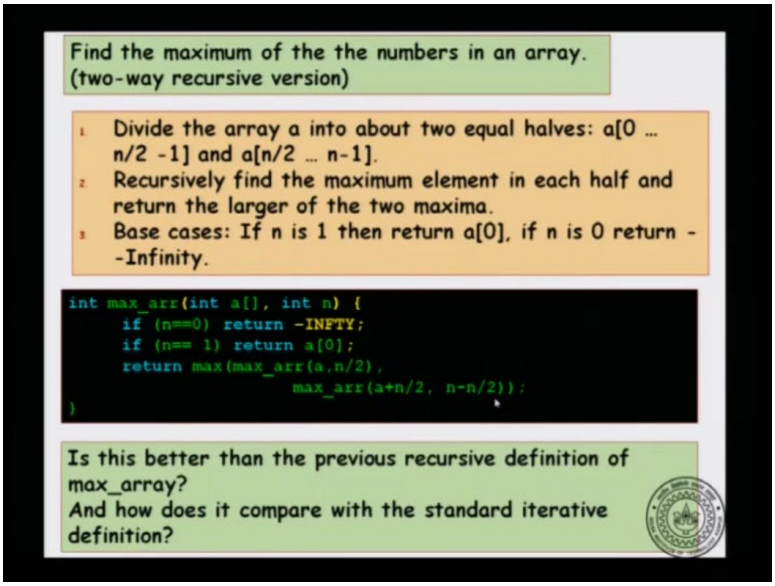
(Refer Slide Time: 02:55)




Just to remind you, the linear version was done as follows: if n = 0, you return something like - infinity, a very large negative value. Now, if n = 1, you return a[0] which is the only element in the array. Otherwise, you have atleast 2 elements. And earlier what we did was, you call the sub problem, a + 1, so, the array which starts with the second element in the array. And now the sub problem has n - 1 elements because you are considering a 0, the first element as a separate thing.

Now, what you want it to return was maximum of whatever was returned in the sub problem. So, let that be some maxval. And whichever is greater, a[0] and maxval, that is going to be the greatest element in the array. Now, we saw that the stack depth for this problem was n because size n problem is being reduced to a size n - 1 problem. So, in each step we are reducing the size of the problem by 1, and increasing the stack depth by 1. So, in total that stack depth would be n because there will be about n calls or n - 1 calls; however, you want to count.

(Refer Slide Time: 04:21)




Now, let us look at the two-way recursive version. So, here is the algorithm that we discussed; and let us just code this up. So, we will have int max_array; and then int a, which is the array containing n elements. And let us say that we have some constant - infinity, we have defined elsewhere in the program. Later we will see how to do this.

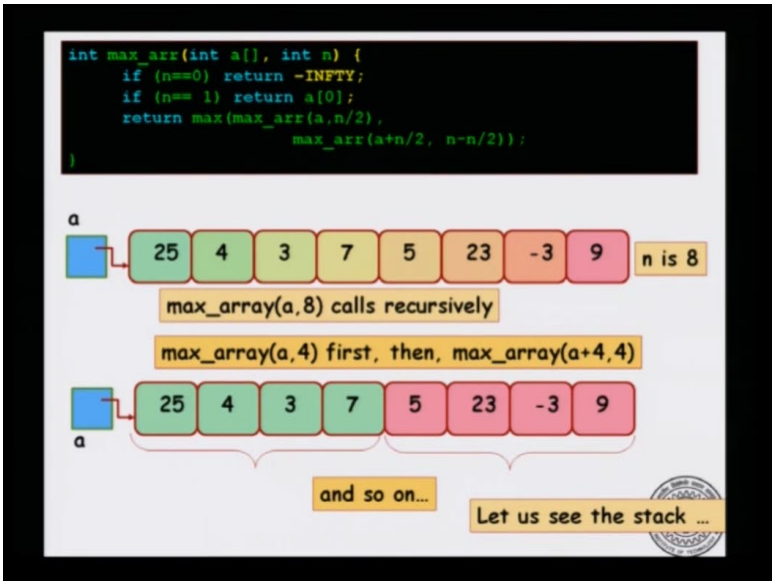
Let us say that if n = 0, you return - infinity, some large number, some large negative value. And if n = 1, you return the only value in the array. So, these are the base cases as before. The changes here; if you have at least 2 elements then you return maximum of the values returned by the 2 sub problems. What are the 2 sub problems? The first is the left half of the array which starts from, a, that is the, at the first location in the array, and contains n / 2 elements.

Then we need to compute the maximum of the right half; how do we find the right half? So, we need to skip, n / 2 elements, which went to the left half, to get to the first index in the right half. So, we do that by saying, a + n / 2. If, a, is the address of the first location of the whole array then a + 1 / 2, is going to be the first address of the first location of the right half.

And how many elements does the right half contain? n / 2 elements went to the left. Therefore, what we are left with is n – n/2. So, notice, how we call the left half starting from, a, and containing n / 2 elements; and right half which is starting from, a + n / 2, and containing n - n / 2 elements.

Now, let us examine whether this is better than the previous recursive call, where we reduce the problem of size n to a problem of size n / 2. It was called linear recursion because we called one sub problem in order to solve the whole problem. Here, we have, we are roughly dividing it into halves and then calling 2 version, 2 sub problems, each of size about n / 2. Now, surprisingly, we will see that there is a huge improvement if you do this. And this is one of the most elementary tricks in computer science which is called divide and concrete; and here is a very simple example of that.

(Refer Slide Time: 07:00)



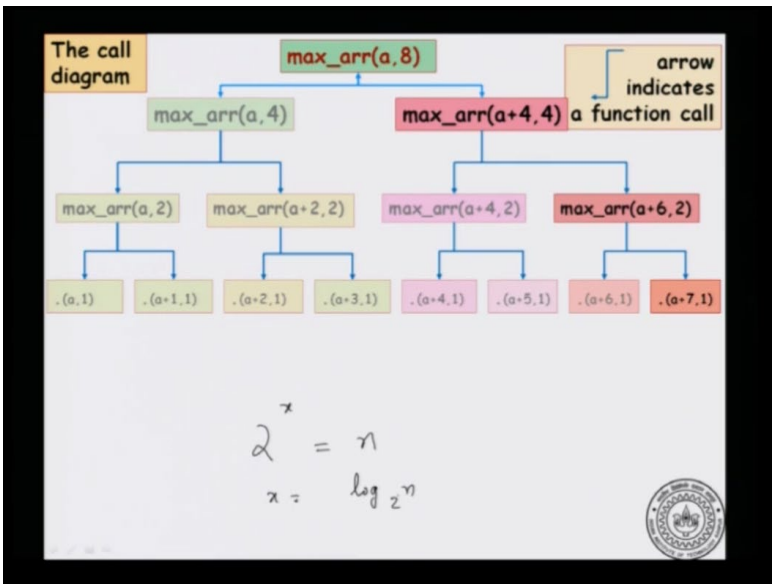

So, if you look at the concrete array that we had, and we call, max_array(a, 8), because this contains 8 elements. Now, we say that it will recursively call 2 sub problems which is maximum, max_array(a, 4). So, that will be the first 4 elements starting from, a 0. And then max_array(a + 4, 4), which are the 4 elements starting from, a 4 which is the fifth element in the array.

Now, let us just look at the stack. Now, notice what types I repeatedly mentioned which is that in order to think about a recursive problem you just think about the formulation of the problem, and then what you have to convince yourself is if I solve the sub problems correctly then I will get the correct solution to the main problem. So, I will have, I will divide my work into 2 sub problems.

So, both of them will report their results back to me. Now, what I have to do is to figure out how do I put these 2 solutions together in order to solve the whole sub problem. So,

think about it in terms of the design of the algorithm, and not about the execution stack. But, we will show why this is a major improvement over the linear recursion version of the same solution by looking at the stack.

(Refer Slide Time: 08:33)




So, let us just look at the stack; max_array(a, 8), calls, max_array(a, 4). Now, the way function calls in c works, you will go to the second half of this problem which is, (a + 4, 4), only after max_array(a, 4), is completely done, right. So, let us now see how, max (a, 4), will execute? It has 2 sub problems again. And let us look at the first sub problem which is, max_array(a, 2), that itself has a sub problem, max_array(a, 1). In order to abbreviate I will just put at dot there, but that dot is supposed to signify max_arr.

Now, once you have solved this, suppose this is the base case, now it contains only 1 element, so, the only element is the maximum; so, it returns that value to, max_array(a, 2); that is one of the sub problems for, max_array(a, 2). So, now, this, max_array(a, 2), calls the second sub problem that it has, which is, max_array(a +1, 1). Again, it is a base case; it contains only 1 element; that single element is the greatest element in that.

So, you have 2 values now - one coming from the left and one coming from the right. And you just compare these 2 values, and that will be the greatest value in the first two elements of the array. So, once you do this, you return; and one you return, you get the value, max_array(a, 2). So, suppose, all of that happens, and then you return to,

max_array(a, 4). At this point, this function will call its second component which is, max_array(a +2, 2), and the recursion continues.

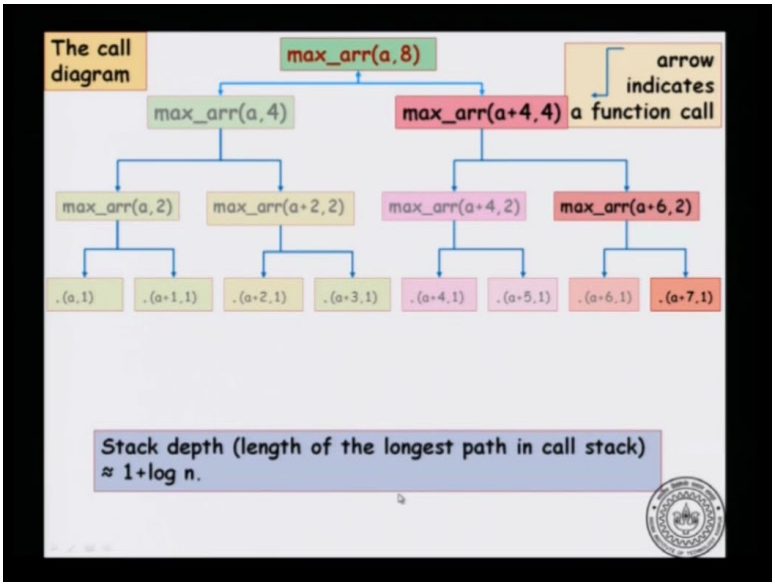
So, as soon as a function returns its stack will be erased; I am showing that by dimming out that particular function call, ok. And this proceeds. So, once this value is obtained you can return to, max_array(a + 2, 2). Now, this function is finished because it has called both its sub problems. So, this will return. And this problem has returned, has finish with both its sub problems. So, you will, after this function is done you will eventually unwind all the way back up to the top.

And now, we are ready to call the second sub problem of, max_array(a, 8), which is max_array(a + 4, 4). And. you do it similarly. Now, one thing you can notice here, is that, at any point the active path, what are active on the stack, the functions which are not yet returned are the highlighted entries in the call tree, ok.

So, for example, at the very end the call stack contains 4 functions. Before you eventually return and compute the loss, compute the maximum of the whole array, the worst case depth of the stack is 4. And we had 8 elements, so, you would thing that based on this experience that the depth of a stack is about n over 2. But, if you think more carefully about it what happens is that, at every sub problem, at every level, I am dividing the problem by 2.

So, the depth of the stack is the maximum length part in this tree. And at every step of the tree I am dividing the problem by 2. How many times do I have to divide in by 2 in order to reach 1, that will be the depth of the tree. Equivalently, you can think about, how many times do I have to double in order to reach n if I start from 1, that is the bottom of way. So, if I start from 1 and I double every level, how many times do I have to double in order to reach n, that is the solution to the equation $2^x = n$. So, what I have to find is, how many times do I have to double? So, how many times do I have to multiply 2 with itself in order to reach n? And you will see that the solution is $\log_2 n$. So, this is going to be the height of the call graph or the call tree.

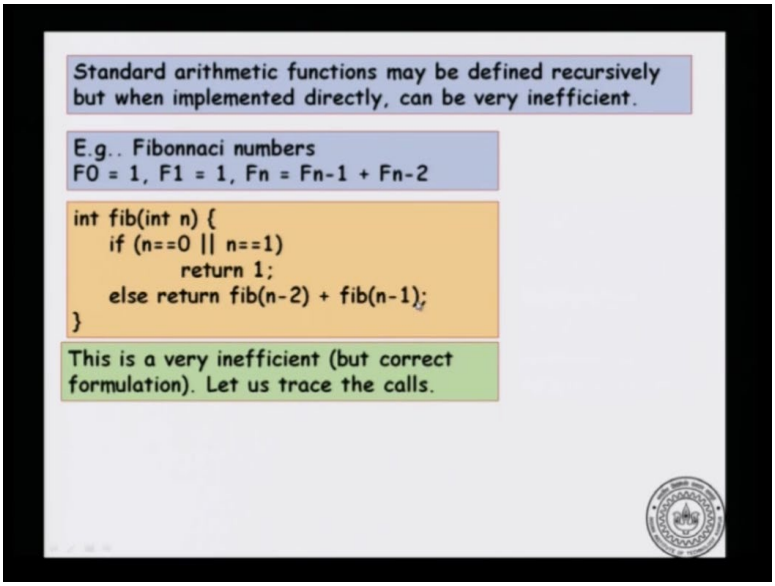
(Refer Slide Ti me: 13:29)




So, the stack depth here is about, 1 + log n, that is approximately correct, which is a huge improvement over n. If you think of n as something like 1024 which is $2^{10}$, we are saying that the stack depth is about 10. Notice that, in the linear case we would have a stack depth of about 1024, instead we are doing about 10. So, this is the huge improvement in the case of stack depth.

So, with a very simple idea which is instead of solving one sub problem of size n - 1, what if you split it into 2 halves, roughly about size n/2. You will see that you get a huge improvement in the stack depth. This is one of the simple ideas that we repeatedly use in computer science.

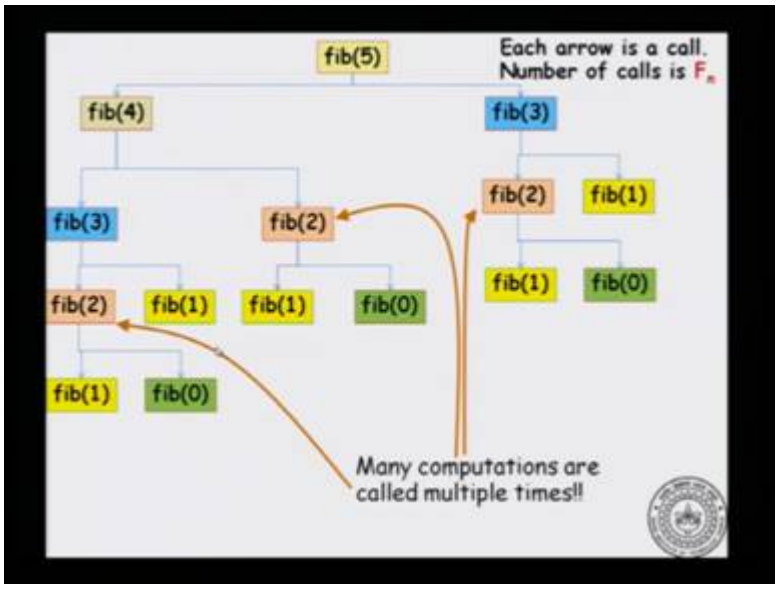
(Refer Slide Time: 14:25)




Now, there are standard arithmetic functions also which can be defined in terms of the two-way recursion. A very classic example is Fibonnaci numbers. So, for example, they are defined as, F0 = 1, F1 = 1. And for n >= 2, they are defined as Fn = Fn - 1 + Fn - 2. So, if you code this out, so, a very simple function, int fib(int n), if n = 0 or n = 1, you return 1. Otherwise, you return Fibonnaci, so, fib(n - 2) + fib(n – 1).

So, it is a very simple arithmetic sequence which is defined in terms of a two-way recursion. So, this is the very simple way to write it, but it is a very inefficient way to do it. So, we will see why it is inefficient in a moment.

(Refer Slide Time: 15:20)




If you just think of how you trace the function, in the case of a later, of a concrete Fibonnaci number; let us say, we want to calculate the fifth Fibonnaci number. Now, that depends on fib(4) and fib(3); fib(4) depends on fib(3) and fib(2); fib(3) depends on fib(2) and fib(1), and so on. So, this is the call graph that you will have, the call tree that you will have if you consider the calculation of Fibonnaci 5.

Now, what is the problem here? You will see that many computations are unnecessarily done multiple times. So, if you look at Fibonnaci 2 in the call graph, it is evaluated multiple times. So, Fibonnaci 2 is evaluated when fib(3) is called. It is also called when fib 4 is called. And fib(3) is called in a different context. When you want to calculate fib(5), even there fib(2) is called. So, you will see that fib(3) is called 2 times, fib(2) is called 3 times, and fib(1) is called 5 times, and so on.

So, we are unnecessarily repeating the work. And there is tricks in computer science to alleviate, to remove this kind of unnecessary work. But, that is strictly, it is not an idea that strictly falls into the concept of recursion, and is slightly outside the scope of this course. So, we will not cover this in this course, but I just want to point out that even though it is natural to consider this arithmetic sequence in terms of two-way recursion it may not be the most efficient way to do it.