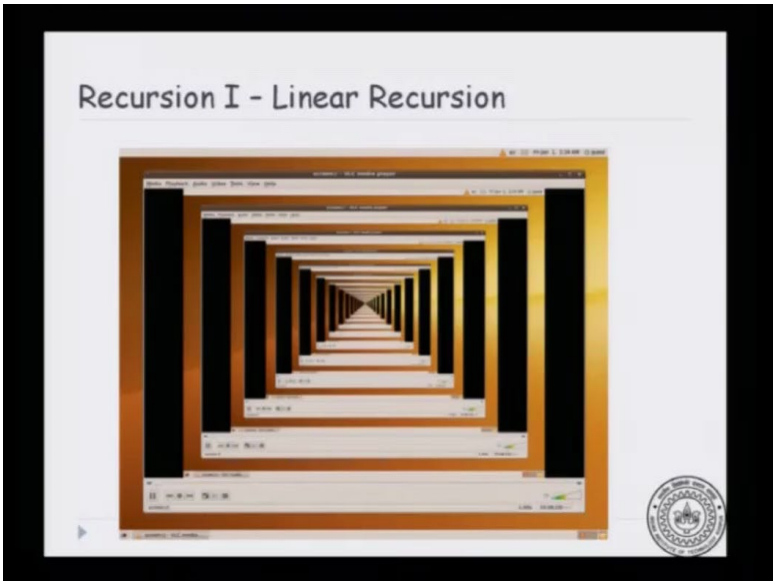
introduction to Programming in C
Department of Computer Science and Engineering

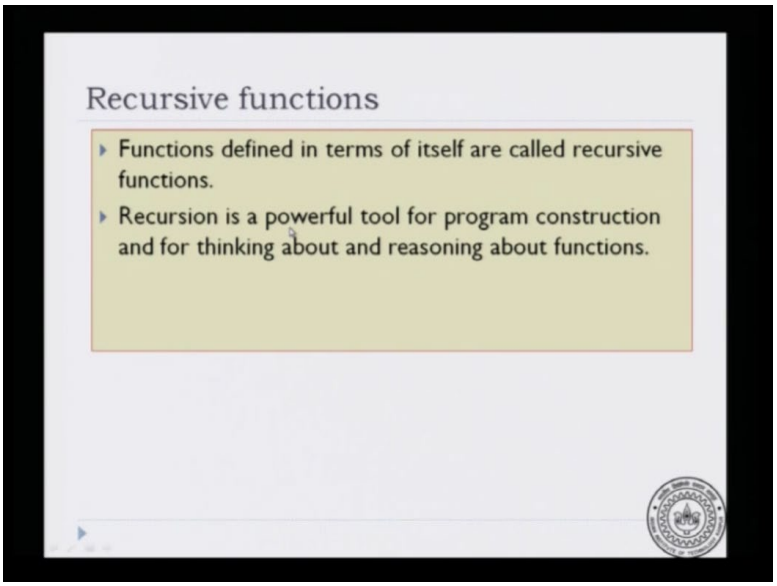
Today's video will talk about an important concept in computer science which is recursion, and we will slowly approach this by looking at various kinds of recursion.

(Refer Slide Time: 00:09)



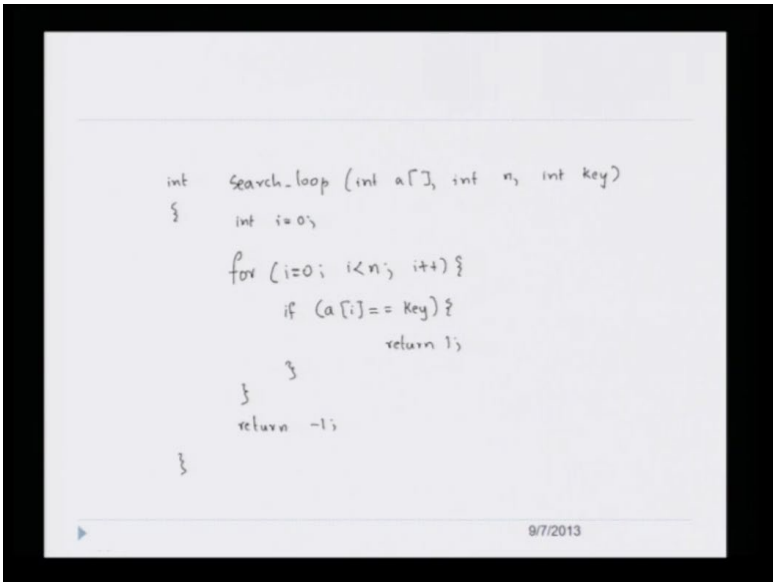
Now, recursion is usually something that is completely new. it is a new way of thinking about problems that might sound unfamiliar at first, but eventually it is a more natural way of solving problems than other techniques. So, we will carefully examine what recursion means. So, this is the video of a media player having a copy of itself inside the video and it goes on forever. We will see what does this have to do with recursion.

(Refer Slide Time: 00:59)



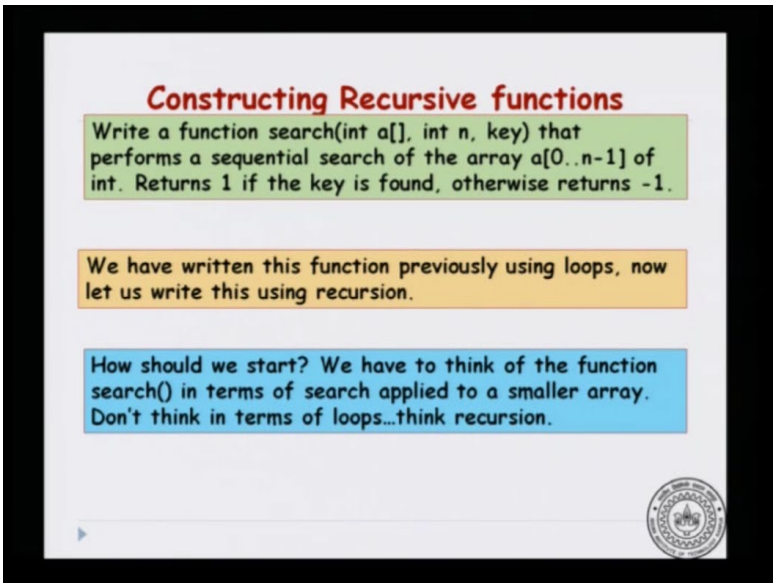
So, recursion in English means roughly say again i am function defined in terms of itself are called recursive functions. Now, this is not completely accurate. We want to say that functions defined in terms of itself in a particular way, these are valid recursions. Recursion is a powerful tool for program construction and for thinking about and reasoning about functions in general. So, it is a general purpose technique of programming, and you can do any kind of program using only just recursion. We will not see such general types of recursion in this course, but we will see fairly common examples for recursions.

(Refer Slide Time: 01:53)



So, for example, let us consider a very simple function which will search for a key within a given array and we know how to write this. What i will do is, i will take an integer, initialize it to 0, for i = 0 to n. N is the size of the array. i will increment i and if at any i, i find the key, i will return 1 indicating that i have found the key. if i have not found the key and i have reached the end of theory, i will return-1. This is a typical way to search for a key inside a given array. Now, we will approach the idea of recursion by looking at a recursive solution to this. Hopefully, while seeing this program, we will get an idea of what recursion means.

(Refer Slide Time: 02:52).

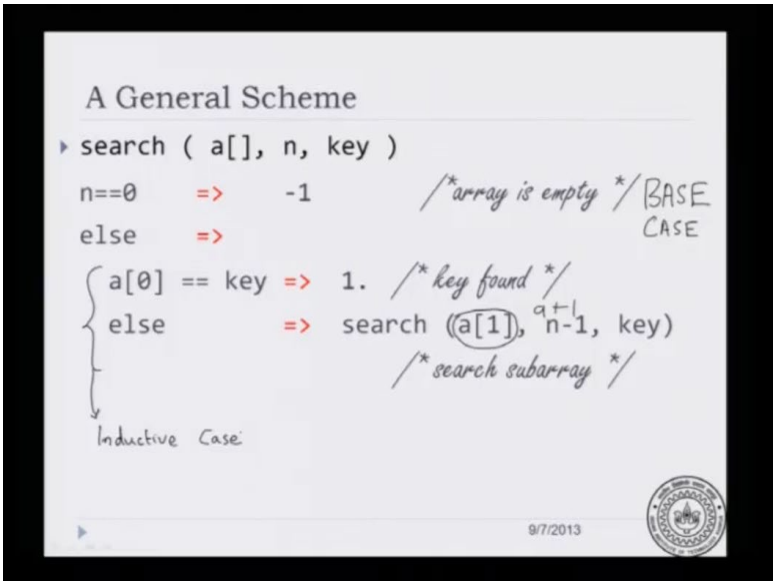


So, what do we mean by a recursive solution to this, right. Rather than defining it and describing abstract properties of recursion, why not let write an actual program which is defined in the recursive manner and through these kinds of examples will eventually get the hang of recursion. So, we have to write a function search it will return whether a key is found or not. if the key is found, it returns 1. if the key is the not found, it returns a-1 and you have to search an array a of size n for the key. Now, we have written this function just now using loops. Now, let us write this using recursion.

Now, what do we mean by solving it in recursive manner? We have to think of the function search in terms of the same function applied to a smaller instance of the problem. So, we have to solve the problem of searching for a key in an array of size n. Can we think of this, in terms of solving the sub problem for a smaller array? This is the

basic question that you have to ask when you want to design a recursive function. So, let us try to in very abstract terms think of how to solve this in a recursive manner.

(Refer Slide Time: 04:24)



So, let us say that i will in some unspecified syntax, this is not going to be valid c, but this is just so that we see the idea in a very clear manner. i have to search for an array of size a of size n for key. Now, if the array is empty that is n is equal to 0, you can have more conditions here. N can be < 0 as well, but let say that empty array is n is equal to 0, then you say that i have not found the key because it is an empty array. So, you give back the value-1. So, n = 0 implies the value to be returned this-1. That is what this notation is supposed to stand for.

Suppose n is not 0, so, this means that an array is non-empty. Now, how do we solve this, recursively right. So, this look for the first element whether it is the key or not. if the first element is the key, we do not have to do anything further. We know that the key is present in the arrays, so you return 1. So, the key has been found and you return 1, and now is the big step for recursion. How can we search for the key in an array of size smaller than n? So, if a[0] is not equal to key, then this means that key can be somewhere in a 1 through a n-1 or it is absent in the array. in any case what we now have to do is search for the arrays starting at a 1, so by a 1 this is not strictly c notation. What i mean is the sub arrays starting at a + 1. So, search in the sub array starting at a + 1. Now, the sub

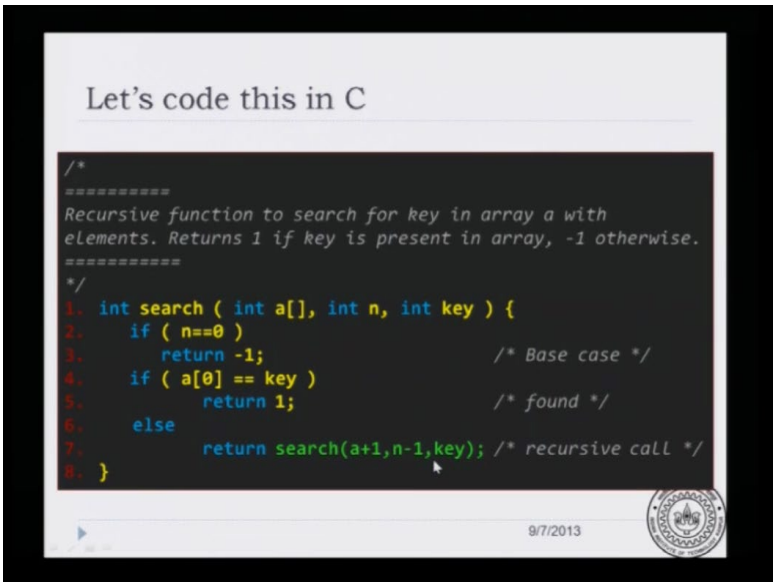
array has one element less because we already know if you are here that a 0 is not equal to key, so there are only n-1 element in the smaller sub problem.

What do we have to search for, we have to search for the key. So, this says that either the key is present as the first element of the array or you have to solve the sub problem of searching in the sub array of size n-1 for the same key. So, here is the key to thinking about a problem in recursive terms. What you first do is, consider the case when you have the trivial array which is the empty array in this case. So, we have the base case and then, these are the recursive case. So, the recursive case consists of doing something at size n. So, in this case, it is search whether the first element is the key or not. if it is true, then we do not have to do anything further, we have found the key, otherwise solve the sub problem.

Now, the sub problem is a smaller copy of the old problem. So, this is what is known as the inductive case or the recursive case. The reason i am calling it inductive case is that recursion has very tight connections to the idea of mathematical induction. if you know how to write a proof by mathematical induction, what you normally do is you consider a base case. So, you have a theorem and you want to prove this by mathematical induction. You consider the base case probably n = 1 or n = 0. These will be the base cases for an association about natural numbers and then, if the base case is true, then you say that i assume that the problem is true for size n and now, i want to prove that the theorem is true for size n + 1. This is how a mathematical induction proof looks like and in the case of a recursive program, there is a very tight analogy.

Recursion in fact is just a mathematical induction in the context of writing programs. We have to solve a problem. First we will see what is a problem in the base case and the base case is a very trivial case usually, but it is important that you think about base case. You say that if the array is empty and then, i will return-1 because the key cannot be in the array. Then, you say that i will now define the problem of size n in terms of a sub problem of size n-1 for example. So, we will solve the same. We will solve the bigger problem in terms of a smaller copy of itself and this is the key to thinking about recursive programs.

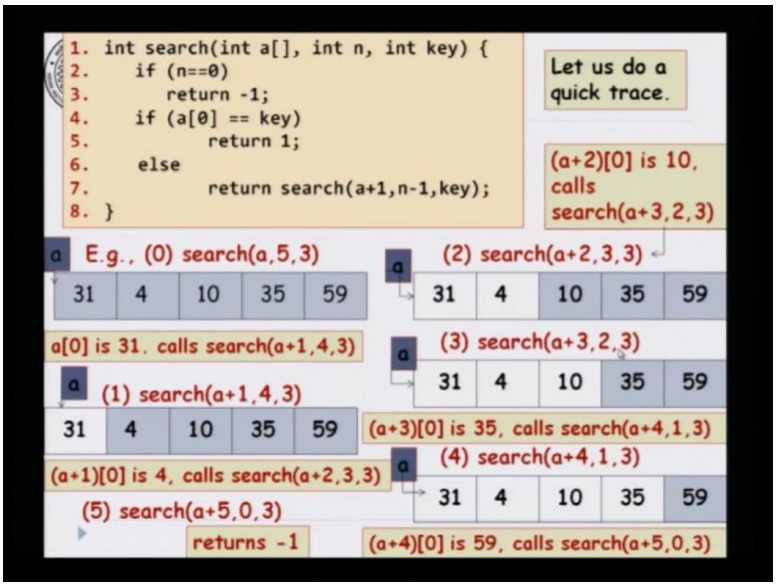
(Refer Slide Time: 09:52).



Let us code this in c. So, we code this in a very straight forward manner. i will write a int search, int a[], int n which is the size of the array a, int key which is the key we are searching for. if n = 0, then return-1 because the key is not found. This is the base case and otherwise n > 0, so you can search for a 0 is equal to key or not. So, you can search for whether the first element is the key. if it is, then you have found the key, otherwise what you do is you call search a + 1 which is the sub arrays starting at size 1. The sub array has size n-1 and key.

So, when you search or write a recursive program, there are a few things that you want to check. The first is that the base case is properly handled. The second is that when you define the sub problem, you want to ensure that it really is a sub problem because if you solve the problem in terms of an equal size problem or even a bigger size problem, your program may not terminate. We will see this in a moment. So, this part which is highlighted in green which is calling search itself, but on a smaller sub problem is a + 1 n-1. This is what is known as a recursive call to the same function. So, we have seen functions that can call other functions. Now, we have seen functions which can call themselves and this is what is known as recursion.

(Refer Slide Time: 11:46).

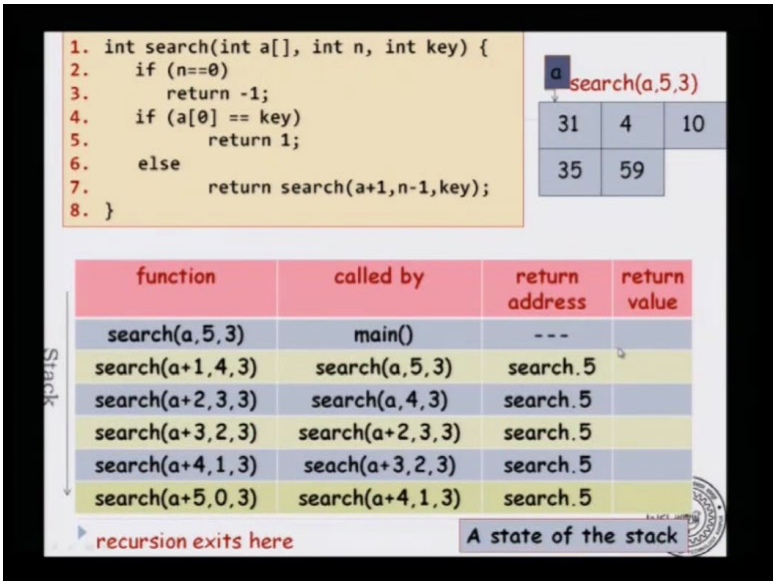


Let's see how this function behaves. Now, before we go into the execution trace of this function, i want to add a word of caution. The actual way to understand recursion is not to think about the stack and how functions are calling other functions. The real way to understand recursion is to think about this program as a problem defined in terms of sub instance. But in any case we will just see the execution of this function through the stack trace just to get comfortable with what happens at the back of all of this. So, let us do a quick trace.

Suppose we have an array 31 4 10 35 59. it is an array of size 5 named a, and we are searching for the key 3. Now, we know that this key is not present in the array, but let see how the function executes. So, first we call search(a,5,3). A 0 is 31 which is not the key. So, it calls search a + 1 4 because now we are searching in the sub array of size 4 for the same key. So, that is in effect, the same as calling the same search function on this sub array highlighted in grey. This is because the answer to search in the whole array is now the same as answer to the search in the sub array. That is what the recursive statement is. Now, (a+1)[0] is 4 this is the first element of the sub array. A 4 is not 3 and a, and at this point you call the sub sub problem which is search a + 2, the sub array of size 3 for the key 3. Here is the sub array of size 3 and you are searching for 3 in this sub array. Again the first element of the array is 10, which is not 3. So, you call the sub problem of this which is a + 3. Now, the array is of size 2 and you will search for 3 and this goes on until you find that you have exhausted the array. Finally, the array is of size

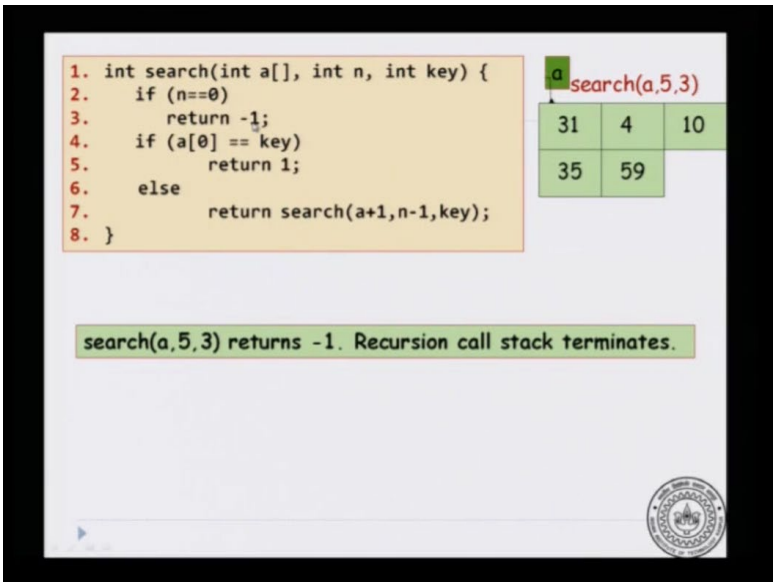
0 and you will finally say that since the array is of size 0, i have not found the key. So, you return-1.

(Refer Slide Time: 14:30)



Let us just look at this stack of function cause and see how it looks like. Search(a,5,3) is called by mean and let say that it has some return address. We do not care about it right now, but search(a,5,3) calls search(a+1,4,3) and the place to return is some line in search function. This calls the sub sub problem a + 2 3 that calls a + 3 2 that calls a + 4 1, and that calls a + 5 0 at which point you realize that the sub problem now is empty and then, you return a-1. So, at this point you have reached the base case. if n = 0, return-1. So, that will return a-1. Where will it return to? it will return to the function which immediately called it which is search(a+4,1,3). So, this guy gets a-1. Therefore, it just returns that-1, return the value of whatever is returned by the sub problem, ok. So, it is-1 and that-1 gets returned. So, it gets bubbled up all the way back to main, and main you can realize that the element is not present in the array because the return value of search(a,5,3) was-1. At this point, the call stack terminates.

(Refer Slide Time: 16:26)



So, what was special about the recursion call stack? it was just that most of the stack was involved by a function calling itself over and over, but each time the function called itself, it was calling on a smaller version of the problem. Here is how, you think about a very simple program in terms of recursion. Earlier, we saw how to solve this using iteration which was using a loop and we have seen the problem how to be solved using recursion.

Now, a word of caution, we will see this in further examples. it is very important that you handle the base case properly. Now, this is something that we are not used to in normal way of thinking. When we think about solving a problem, we are thinking about solving substantial sizes of the problems. We are not concerned too much with what happens with an empty array, what happens when n is-1 and things like that, but even in this problem, we know that when we call search(a + 5,0,3) we know that the function terminated because we had a base case which said that if n equals to 0, then return-1. if we did not have this case, you could see that probably it will go on calling itself infinite number of times. So, just like when you are writing for loop or a while, loop you have the case of infinite loops. in the case of recursion, you can have an infinite recursion and you have to guard against that. The only way to guard against that is to get the base case correct. So, here is something in counter intuitive about programming recursive functions. You know almost half of your intellectual effort is in handling the base case properly, and only the remaining is involved in solving the recursive case.