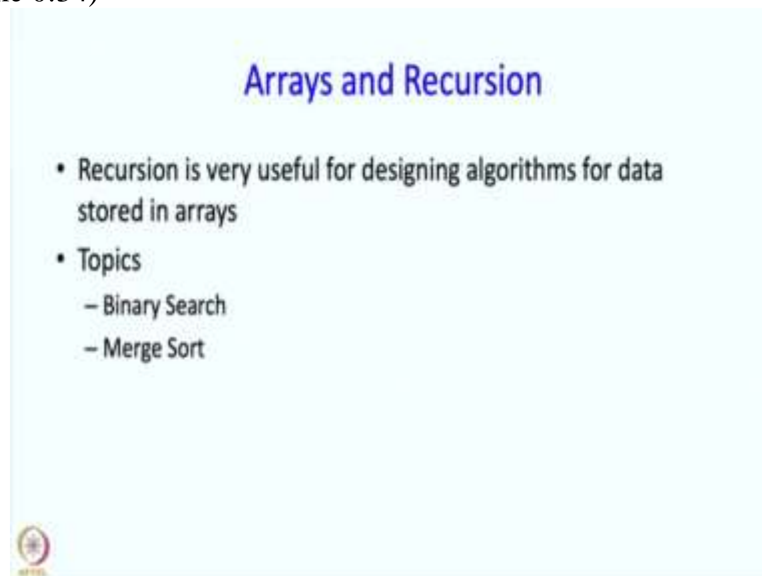
An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay
Lecture 18 Part 1: Arrays and Recursion
Binary Search Introduction

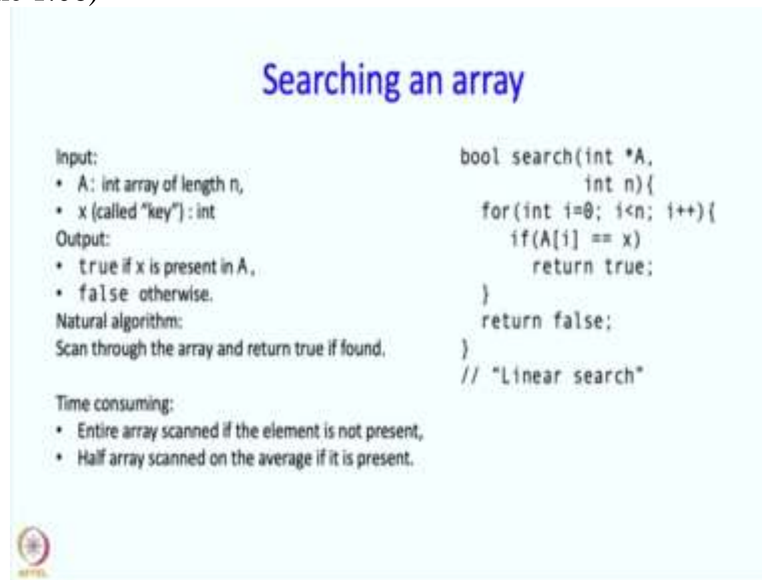
Hello, and welcome to the NPTEL course on an introduction to programming through C++. I am Abhiram Ranade and this lecture is about arrays and recursion. The reading for it is Chapter 16 of the text. So recursion is very useful for designing algorithms for data stored in arrays and as we saw for data stored elsewhere as well, but also for stored, also for data stored in arrays.

(Refer Slide Time 0:34)



So we are going to talk about two very useful recursive algorithms for which you work on arrays and these are the binary search algorithm and a sorting algorithm called merge sort.

(Refer Slide Time 1:06)



So let me begin by discussing the notion of searching an array. So the input to this is an array of length n and then we are given a key or an element of a value x which is called, which is often called the key. And say it is also an integer and we want to know whether x is present in A, okay? We may also know things like at what position it is present, but for now just for simplicity, let us say, we just want to know whether this x is present in A.

And if it is present, we should return true, otherwise we should return false. Now this is not a new problem. When we did the marks display problem, we already did this. So we were given a roll number and we went through our area of roll numbers and tried to check, tried to find if the roll number was present somewhere.

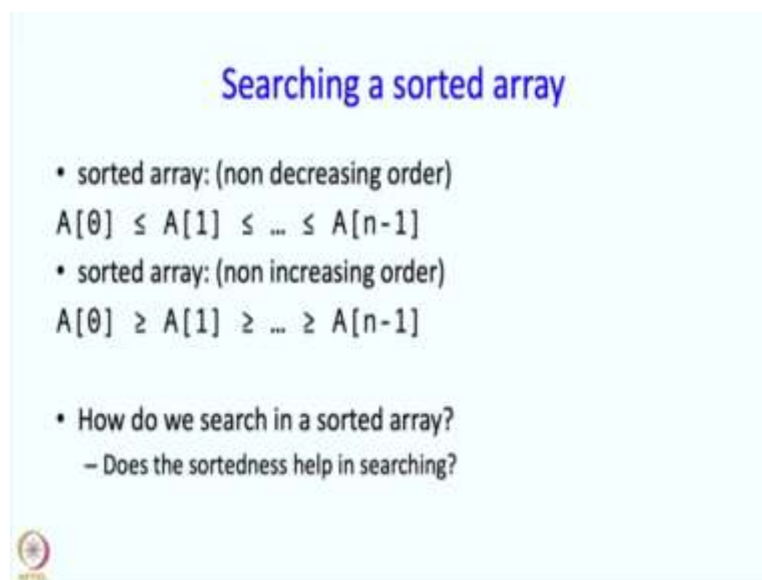
So this is just, this is just a simpler, simpler version of it. And I am doing this primarily to remember to, to jog your memory as to what we have already, what we already did. But then we are going to do something more interesting with it. So the natural algorithm that you have already seen is to scan through the array or go through every element starting at the beginning, and return true if you find it.

But if you go to the end without finding it, then you return false. So here is the algorithm or here is the program. So as argument, we take this array, array name A. I could have written this as int A[]. But I can also write as, write it as int *A. Then we are going to start at zero and n is assumed to be the length of the array over here.

And so we are going to go through all the elements. If we ever find that x is one of the elements, then we return true otherwise we just continue going through the array until the end of the array. And if we get to this point then that means, we have not, we have not seen x in the array at all and therefore, we return false. So you have seen this a number of times actually, but as I said, just to set the context that is why I am saying this. And this is called linear search, I think we have also used that term. So I just want to observe that this is somewhat time consuming, to decide whether x is present we essentially have to scan the entire array. We certainly have to scan the entire array the element is not present.

And say on the average if the element is present, we will scan half the array. So, so it, our time might be small if the array, if the element is present at the beginning. It might be large if the element is present towards the end, but say on the average it takes us, we have to go through half, half the array. So if the array is large, this can be this can take quite some time.

(Refer Slide Time 4:32)



So let us now consider us a different problem that of searching in a sorted array. So what is the sorted array? Again you know it. But let us just, let me just define it because we are going to use it over here and sorted when we talk about a sorted array. There can be two orders, one of two possible orders. So the non-decreasing order is A[0] is less than or equal to A[1] less than or equal to A[2].

And so on until A[n-1]. So there are, so the elements are increasing but with equal elements allowed. So that is why this order is called a non-decreasing order. We could have a non-increasing order and that simply means A[0] is greater than A[1] and so on until A[n-1]. So now we want to search in such an array, say whatever non-increasing or non-decreasing.

We know of course what it is, but yeah, so we want to search in it. So the interesting question of course is that does the sortedness help us in the search. And if so how much? And naturally if it helps us a lot, then maybe we will say that look let us keep our array sorted. Because if you are doing searching, then that our searching might happen very fast. All right, so let us consider the problem, we are searching for x in a non-decreasing sorted array A[0] through A[n-1].

(Refer Slide Time 6:13)

Searching for x in
a non decreasing sorted array A[0..n-1]

Key idea: First compare x with the "middle" element A[n/2] of the array.
Suppose x < A[n/2]:
- x is also smaller than A[n/2..n-1] , because of sorting
- x if present will be present only in A[0..n/2-1].
- So in the rest of the algorithm we will only search first half of A.

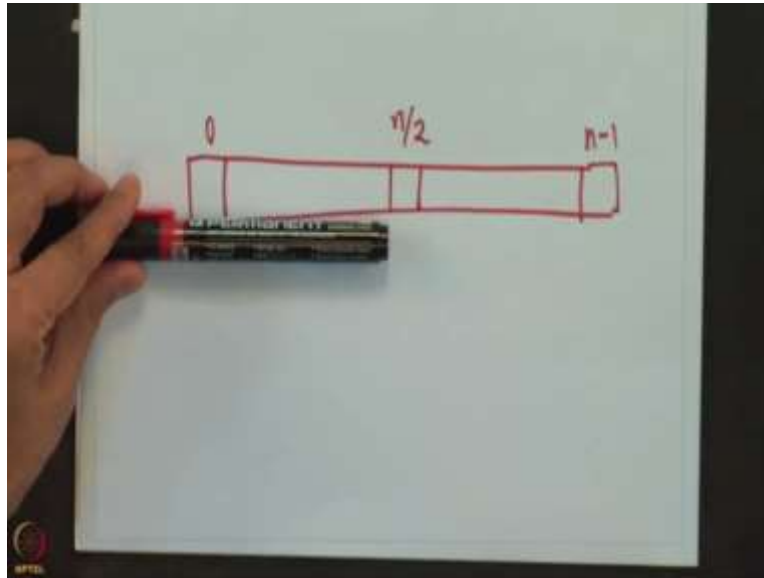
Suppose x >= A[n/2]:
- x if present will be present in A[n/2..n-1]
- Note: x may be present in first half too,
- In the rest of the algorithm we will only search second half.

How to search the "halves"?
- Recurse!

In one comparison we have eliminated half the array!

Now here is the key idea. We are going to compare x not with A[0] as we did in the case of linear search, but our very first comparison is going to be with the middle element of that array.

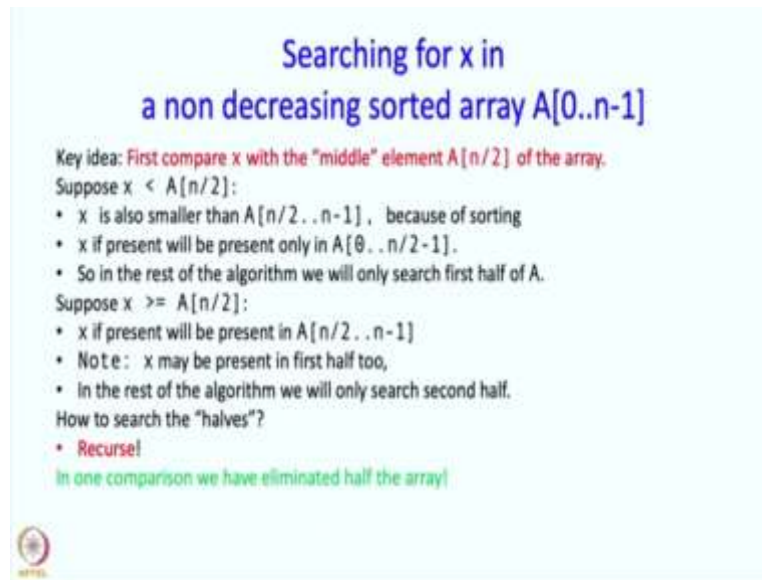
(Refer Slide Time 6:43)



So let me draw a picture. So this is my array A. So this is 0 and this is n-1. In linear search we compared x to this, then to this, then to this and so on. The new idea is that we are going to compare it to this middle element. Well what is the middle element? This is the (n/2)th element element with index n/2. Now this is not exactly the middle element if say n is even it is sort of on the, on the right side. But we want where I am talking about an approximate middle and of course division is integer division.

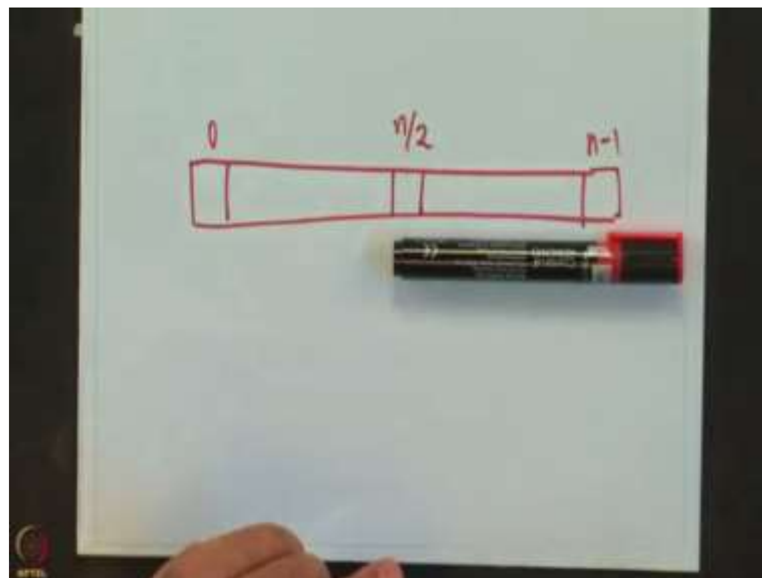
So we get an integer. So x is compared to this element. Now let us see, so what if x turns out to be smaller than this element. What do we know. If x is smaller than this element, then x is also smaller than all of these elements. So why? Because these elements cannot be smaller than this, these elements can only be larger. And therefore it means that our x must lie only on this side, must lie only in this region. So x can be present only in this region.

(Refer Slide Time 7:56)



And therefore, our subsequent search can be restricted to this region. So the first half, again half I am using approximately.

(Refer Slide Time 8:14)

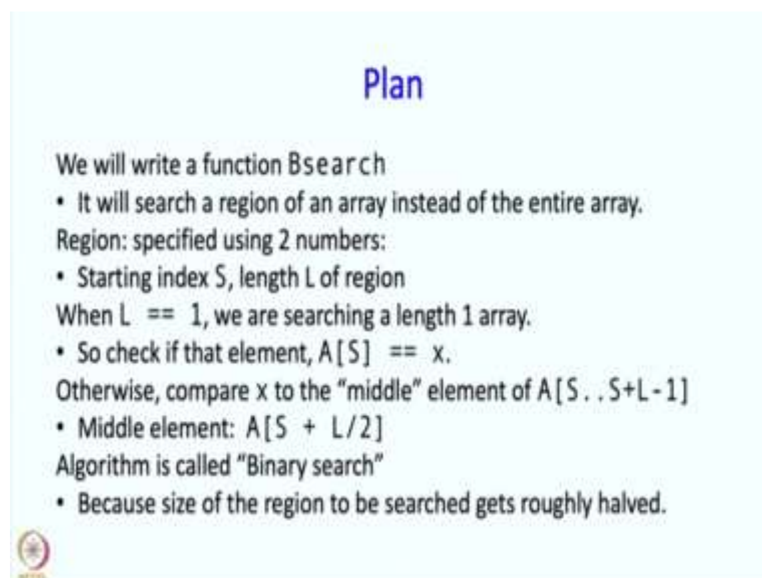


What if x is bigger than or equal to n/2. So if x is bigger than or equal to n/2 then we know that x must be present in this region including this n/2, well x could be present here also, but never mind that, if x turns out to be bigger than or equal to this, then it has to be present in this second half. And therefore, again, we can say, I will ignore the first half and I will just

restrict myself to this half, because if x is present, I am going to, if at all it is present, I am going to find it in this region as well. So x if present will be present in n/2 through n-1.

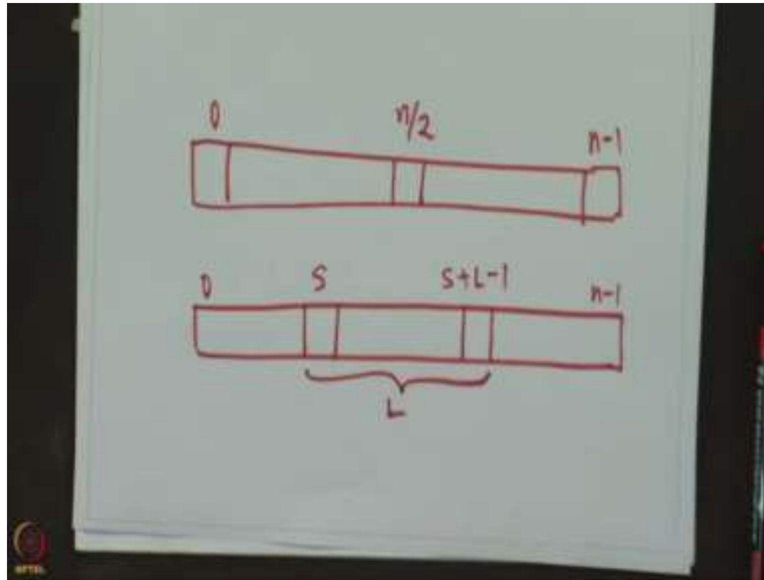
And if it may be present, but I am going to ignore that and in the rest of the algorithm will only search the second half. Now how do we search the halves? This is the beautiful answer. So we are going to recurse and I have really told you the entire algorithm. Well I will tell you, the exact exact function in a minute, but I have really told you whatever I really, whatever the ideas are and let me just make one observation that I have only done one comparison. But in one comparison whichever way the comparison turns out to be, I have eliminated half the elements. So in one comparison, I have sort of reduced my problem to half the size so if I do linear search I will do thousand comparisons. But now even if I do a linear search after this one comparison, I will still only do 500, but that is not what I am going to do in any case, I am going to recurse. So the savings that I have will be enormous.

(Refer Slide Time 10:12)



Alright. So let us sort of develop this idea so what is our plan? So we are going to write a function Bsearch. It will search a region of an array instead of the entire array, of course, the region could be the entire array, but in general it would search some region and the region is going to be specified using two numbers, the starting index S and the length L of the region.

(Refer Slide Time 10:42)



So maybe I should draw a picture here as well. So this is my array again, and this is 0, this is n-1 and let us say this is some starting index S. So the length of the region that I am going to search is going to be some L over here. So there are L elements in this entire thing. So what is this index over here? This is S+L-1. So I could have specified the starting index and the ending index, but I could, I can give the same information by specifying S and L. So both are really equivalent, I am just happening, we are just happening to use L over here.

So Bsearch is going to be given S and L and of course x which is the element to be searched and B search is going to search this, this array but not the full array, it is going to search only within this region.

Now if L is 1, then what happens what is this array? So if L is 1, then these two elements really become the same element. So L equal to 1, so these two elements become the same right. So we are really, the region that we are talking about is really a single element region. So searching a length one array or a single element region is very easy.

So we just check whether this A[S] is equal to x. If it is? We return true, otherwise we return false, as simple as that. Otherwise what do we do? So just as we compared x to the middle element over here, we are now going to compare x to the middle element of this. So what is the middle element?

So the middle element is somewhere over here and this has index. So I will write the index over here, so this has index S+(L/2). So remember this was n/2 and this was 0, so it was 0 plus n/2 and this is, this has index S+(L/2). So I am going to compare x with this and again, the idea is going to be that I will eliminate either this side or eliminate this side. So medium, the median element, middle element is S+(L/2) and the algorithm is called binary search, because the size of the region to be searched gets roughly hard at each step.

(Refer Slide Time 13:29)

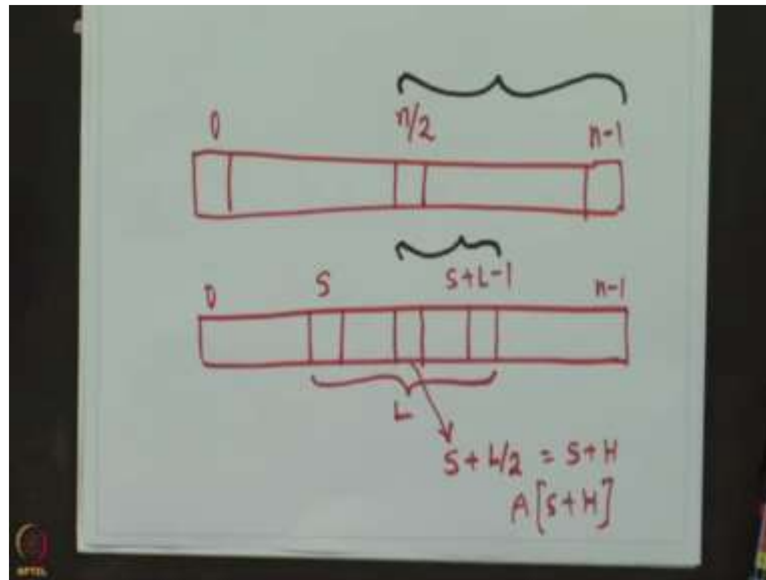
The code

```
bool Bsearch(int A[], int S, int L, int x)
{ // Search x in A[S..S+L-1]
   if(L == 1) return A[S] == x;
   int H = L/2;
   if(x < A[S+H])
     return Bsearch(A, S,   H,   x);  // region A[S..S+H-1]
   else
     return Bsearch(A, S+H, L-H, x);  // region A[S+H.. S+L-1]
}

int main(){
  int A[8 ]={-1, 2, 2, 4, 10, 12, 30, 30};
  cout << Bsearch(A,0,8,11) << endl;  // searches for 11.
}
```

All right so here is the code. So Bsearch is going to take the starting address of the array. And the region is going to be given by the index S and the length of that region is going to be 1/L and x is going to be the element that I want to search. So we are going to search for x in A[S] through A[S+L-1], as we just said.

So if L is equal to 1, we can check whether A[S] is equal to x. So what does that mean? We are returning the result of that condition. So if that condition is true then we will return true, otherwise we will return false. Then we are going to, we are going to look at the middle element and that middle element is L/2 or so, the half, the halfway point or the half of that region, half of the length is L/2. And so we are going to check whether x is less than or equal to S plus H.

(Refer Slide Time 14:50)



So in this picture is, this is, this is S+(L/2) or it is also S+H. So this element is A[S+H]. So we are checking whether x is less than this element or not. If x is less than this element then what happens? If x is less than this element, we have to search which region, well, we have to search the region starting at S and going to S+H-1. So that is what we are searching and indeed if I write search the region (S, H) H is the length, so that is indeed saying S through S+H-1.

Now this has to be written carefully because you do not want to miss out on these minus ones or you do not want to write an unnecessary one. So you have to do this, you have to do this somewhat carefully. Otherwise what do we have to search? Well otherwise we have to search this part. So starting over here you have to search all the way till this point, just as in the previous case we searched over this entire region. So what is that region? So that starts with S+H.

(Refer Slide Time 16:09)

The code

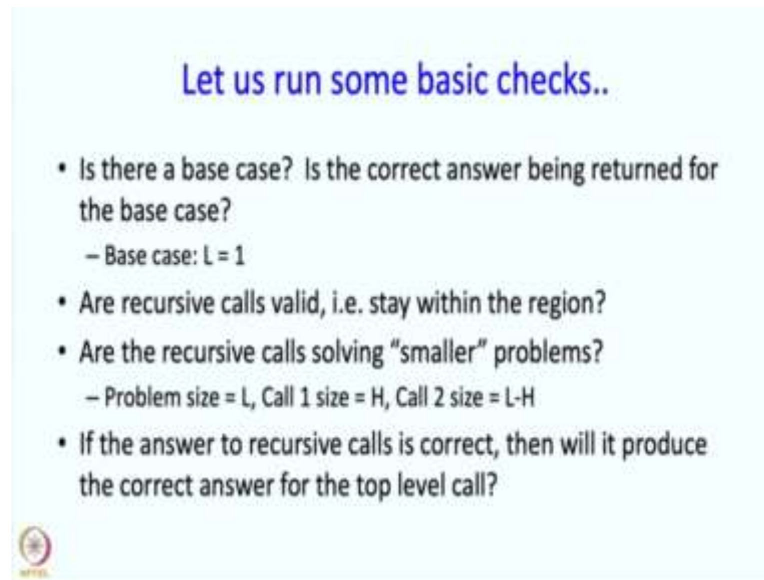
```
bool Bsearch(int A[], int S, int L, int x)
{ // Search x in A[S..S+L-1]
   if(L == 1) return A[S] == x;
   int H = L/2;
   if(x < A[S+H])
     return Bsearch(A, S,   H,   x);  // region A[S..S+H-1]
   else
     return Bsearch(A, S+H, L-H, x);  // region A[S+H.. S+L-1]
}

int main(){
  int A[8 ]={-1, 2, 2, 4, 10, 12, 30, 30};
  cout << Bsearch(A,0,8,11) << endl;  // searches for 11.
}
```

So that is what is over here and we go to the end of the region. So that is what the region which I marked over here and in fact by specifying the length in this manner, I get exactly this. So why is that? So the final index of the region is going to be this, plus this minus 1, just as the final index over is this plus this minus 1. So this plus this, is S+L, -1 is exactly the region that I wanted.

So I have written exactly the analogous code of what I had of the algorithm that I described at the very beginning, and what is the main program. So say it is given some input like this. So this is a sorted array and in this sorted array, I want to search for 11. So my call is (A, 0, 8). Why 0, 8? Because I want to start at the 0th index and the length that I want to search through is the entire array or 8 and then x is 11.

So that is the call that is, that is the value that I am searching for the key that I am searching for. So this will print a 1 or 0 depending upon whether 11 is present or not present in the array. In this case 11 is not present. So it will print a 0, it should print a 0. Now this is our, this is our function and it is a recursive function, and, we said earlier that the recursive functions have a certain format. So we should check whether that format is there in this case as well.

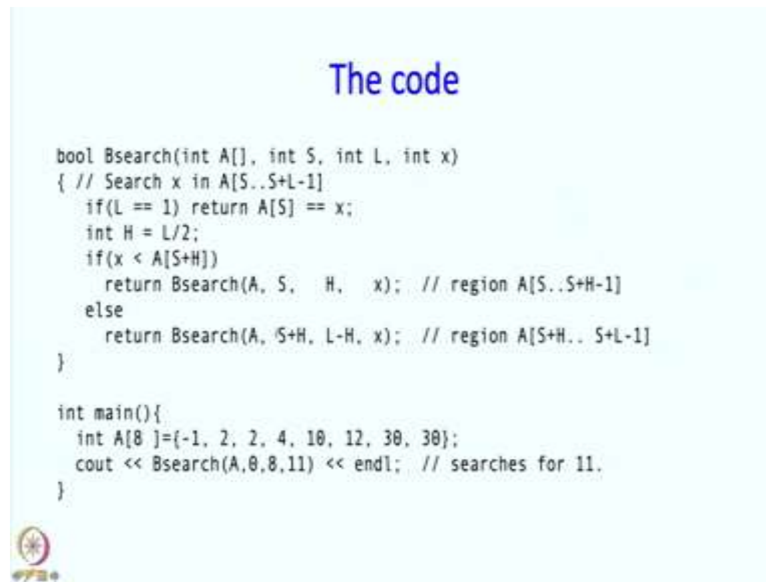
(Refer Slide Time 17:54)



So for example, one question we said that we should be asking is, is there a base case and is the correct answer being returned for the base case. Well let us see this. Is there a base case over here? Yes. So L equal to 1 is certainly a base case. This is the case then the, the program returns without recursing further. In this case, does it return the correct answer? So if L is 1 then that means, we are searching an array of length 1 and therefore, and the array starts at S.

So this is, this A[S] is the only element in the array.So if you want to know whether x is present, we should check whether that only element equals x. So indeed, we are doing that correctly. So that takes care of that first question. Then, you have to check are the recursive calls valid?

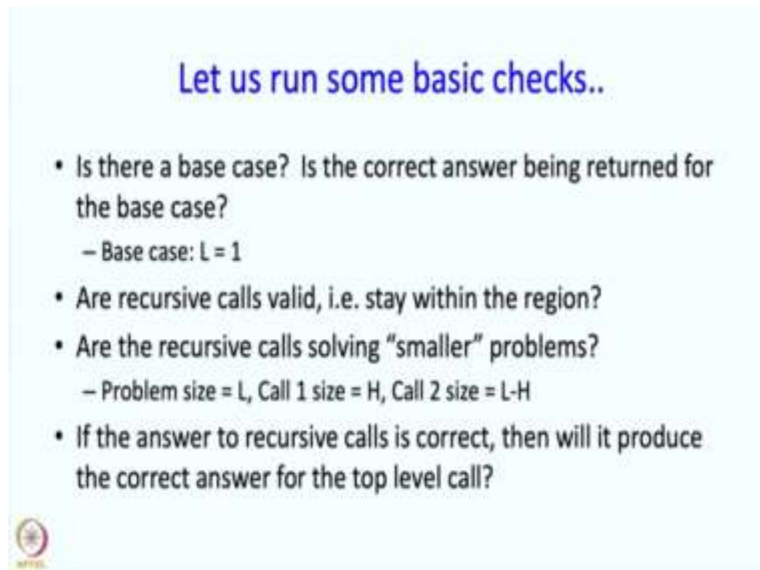
(Refer Slide Time 17:54)



So let us go back. So we are making this as a recursive call. Now as I said earlier, we have to be sure that the indices are given correctly. So for example, this L-H is crucial you should not, I mean at the very beginning for example certainly you should not give a bigger index than 8 over here.

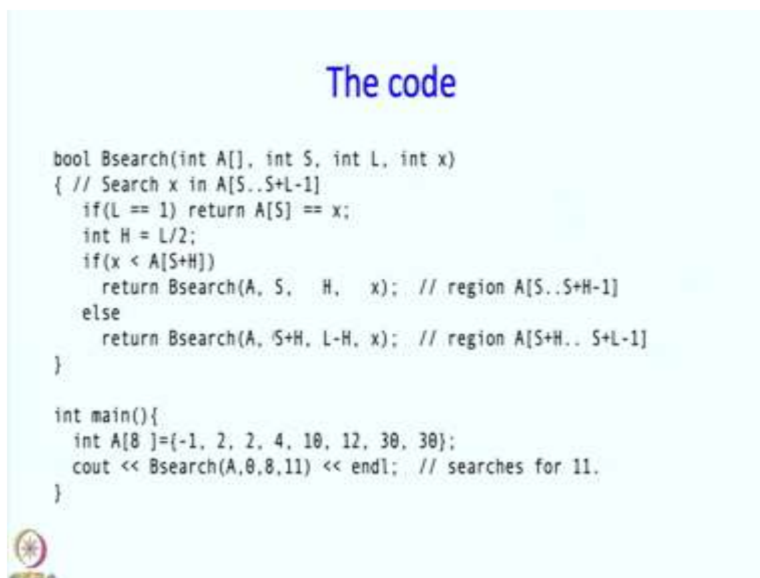
So similarly you should make sure that these array, these indices fall within, within the range that we want, these are actually the indices, these actually define the region that you want searched. So we did this check and yes these are the correct regions that we want searched. No matter what the arguments are, what, no matter what these S and L are. So that is also done.

(Refer Slide Time 19:45)



Now here is an important question. When we are recursing the recursive call should be solving a smaller problem. If they solve the problem of the same size, then we are not making progress, then we are likely to run into infinite recursion.

(Refer Slide Time 20:10)



So we should be checking this. So this is perhaps the most important check. So here, there is a notion, there is a natural notion of the problem size over here. In this case it is captured quite nicely by this argument L. So we are searching in a region of length L. So what we need to be sure about over here, is whether our recursive calls are searching in a smaller region.

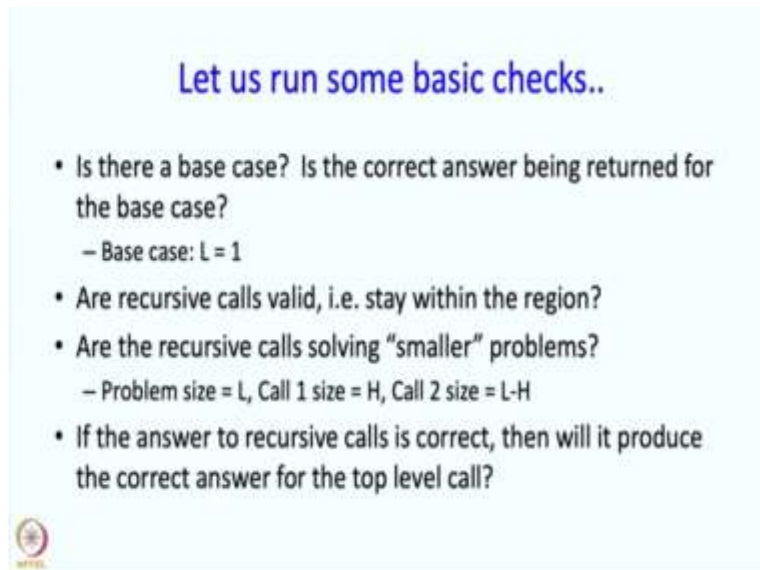
And of course if we keep on doing that, then eventually we will get to a region of size 1, we can certainly not go below go below 1 and in which case we will hit the base case. So we have to make sure whether this call is searching a smaller region and this call as well. Well what we want to know is, is H smaller than L. So what is H? H is L/2. Is L/2 smaller than L? Well this is integer division. So L by 2 is truncating division, so L by 2 is always going to be smaller than L. So this is going to search a smaller region, then what we started off. Well you have to be careful. So L/2 is smaller than L, provided L is bigger than 0. So we have to also make sure that we are never asking to search a 0 sized region. When will that happen? So it will happen if L is 1, so if we happen to run this for n equal to 1 and if we come to this point, then they will be asking to search a 0 size region.

But that is not, that is not happening, because if L is 1, then we are actually, we are actually returning right here. So if L is 1, then this is never reached. So this is reached only if L is bigger than 1. And therefore, this is always a proper call. So a proper call is that this is always larger than or equal to 0, but now we have also checked that it is smaller than L. So yeah, so earlier I said that this call is proper.

But it actually requires this argument that this H has to be bigger than 0 and this will be bigger than 0 because if we start with L=1, then it will be returned right over here. And therefore, here we will come to if L is bigger than 1. So therefore, H will always be bigger than bigger than 0. Now, is this always going to be a valid call? Well here, can H be equal to L? Well L/2 is always smaller and therefore, H cannot be equal to L provided H is bigger than 0.

So this will always be smaller than L and will it always be greater than 0? Well it will always be greater than 0 because H is going to be smaller than L.

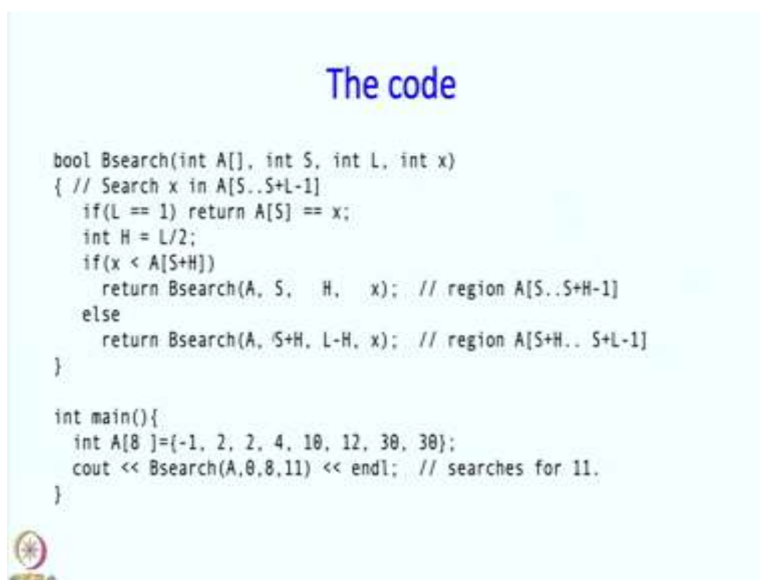
(Refer Slide Time 23:31)



So let us run some basic checks. So we said that if this is a recursive program, then there has to be a base case and a correct answer must be returned for the base case. So clearly in this program, the size of the region being searched if it becomes 1, then we are returning directly.

So this is where we return directly. And so that is happening, so we have a base case and are we returning the correct answer for it? Yes. We are returning the correct answer. Then the next question is, are the recursive calls valid, do they stay within the region? So what does this mean?

(Refer Slide Time 24:20)



So this requires us to check whether these numbers are valid. So this number is a valid index because we started off with a valid index over here, so that is not a problem. Now when can it not be a valid index? Well this could, this is potentially a candidate for not being a valid index. So what do we want this index to be? We want this index to be always bigger than zero. So will it always be bigger than 0? Well we are doing H=L/2. So if L was one over here, there is a danger that this index will become 0.

But note that this will never happen because over here. We checked whether L is equal to 1 and we immediately returned in that case. So if we come over here, we know that H is always going to be bigger than one and therefore, this will never be 0. So what we have checked is that this call is a valid call in the sense that these indices stay within, within that region.

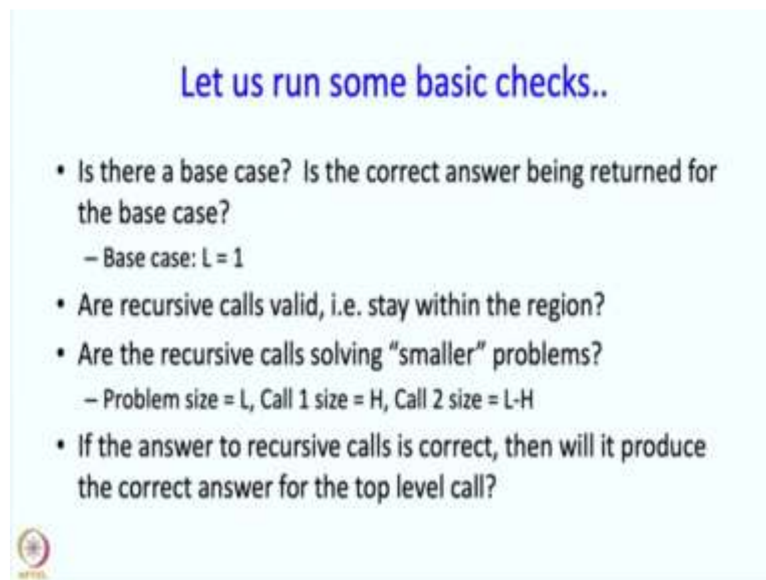
Now what about this index in this call. So is this going to be a reasonable index? Well what do we know about the value of H? So H is L/2, so this could be something like S+L/2. So since L is going to be when we come over here, L is going to be at least 1. So H is going to be certainly non-negative and therefore, this index is going to be bigger than S. But can it be too big? Can it go past the end of the array? Well it will not because, S+L/2 is going to be within this region always.

So this region goes from S to, S+L-1 and you can check that this index will always be within that region. And therefore, that is a perfectly fine index as well. What about this index S+L-H? So H is L/2. So this index is going to be, so, so this index is going to be smaller than L because if H is bigger than 1, then this index is, this index is going to be strictly larger than, larger than 0. And therefore, L-H is certainly going to be smaller than L. And so in this case we can already say that the length is going to be, length is going to be shrinking, but it will not go down below 0, go down to 0. And therefore, again, this is going to be a valid call, this is also going to be a valid call.

So the next thing is are the recursive calls solving smaller problems? Well again let us go back and if we look at this, you have to argue that this is going to be smaller than L. So why is that? Well H is L/2. So L/2 is always smaller. So therefore, this is going to be smaller. L-H is it always going to be smaller? It is going to be smaller if H is always going to be bigger than or equal to 1. Is that going to be true? Well H is L/2 and L is at least 1. So therefore, H is going to be at least 1.

And therefore, this is going to be strictly smaller than L. So yes, so both of our calls are going to be dealing with smaller regions. Now the only reason to go over this so carefully is because these truncations introduced sort of some surprises because we are familiar with doing ordinary division where division is sort of exact division. The integer division sometimes is does things which we do not expect and therefore, it is better to check all these things.

(Refer Slide Time 28:39)



The final question is if the answer to the recursive calls is correct and will it produce a correct answer for the top-level call.

(Refer Slide Time 28:50)

The code

```
bool Bsearch(int A[], int S, int L, int x)
{ // Search x in A[S..S+L-1]
   if(L == 1) return A[S] == x;
   int H = L/2;
   if(x < A[S+H])
     return Bsearch(A, S,   H,   x);  // region A[S..S+H-1]
   else
     return Bsearch(A, S+H, L-H, x);  // region A[S+H.. S+L-1]
}

int main(){
  int A[8 ]={-1, 2, 2, 4, 10, 12, 30, 30};
  cout << Bsearch(A,0,8,11) << endl;  // searches for 11.
}
```

So again if you look at this, if this answer is correct, then it should be a solution for this region and that is exactly the region we wanted to search, because x was greater than or equal to S+H. Similarly this was exactly the region which we wanted to search if x was less than or equal to S+H. So if these answers are correct, then we will get the correct answer over here. So, so we have run every basic checks and in fact they seem to be okay.

(Refer Slide Time 29:33)

What we discussed

- The search problem: decide if a certain key is present in an array.
- If array is not sorted, we do Linear Search.
- If array is sorted, we do Binary search.
- Next: Sample execution of binary search, analysis

So now we can sort of run, all right. So what did we discuss? We discussed the search problem then we said that if it is the array is not sorted, we will do a linear search then we

said if the array is sorted, we do binary search. Next we are going to do an execution, we are going to see how this program executes and then will also analyze that program. So we will take a quick break.