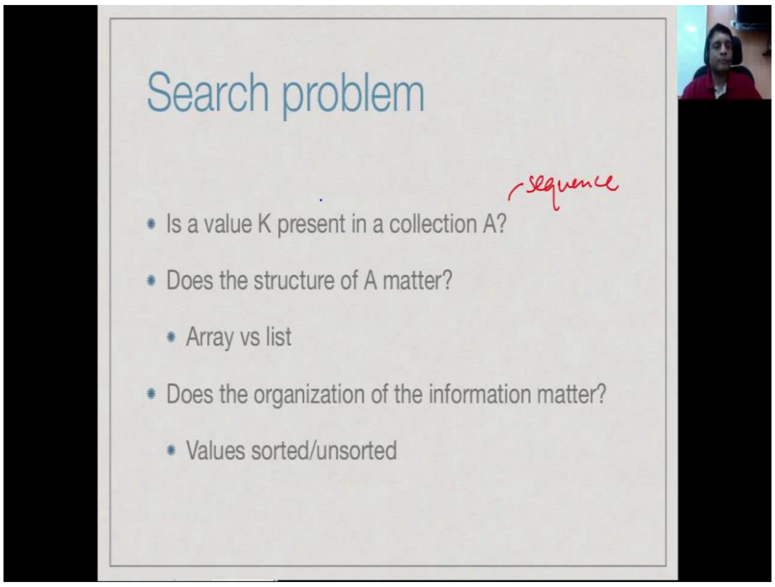
Design and Analysis of Algorithms

Prof. Madhavan Mukund
Chennai Mathematical Institute

Week – 02
Module – 02
Lecture - 10
Searching in an array

Let us look at the problem of searching for a value in an array.

(Refer Slide Time: 00:05)



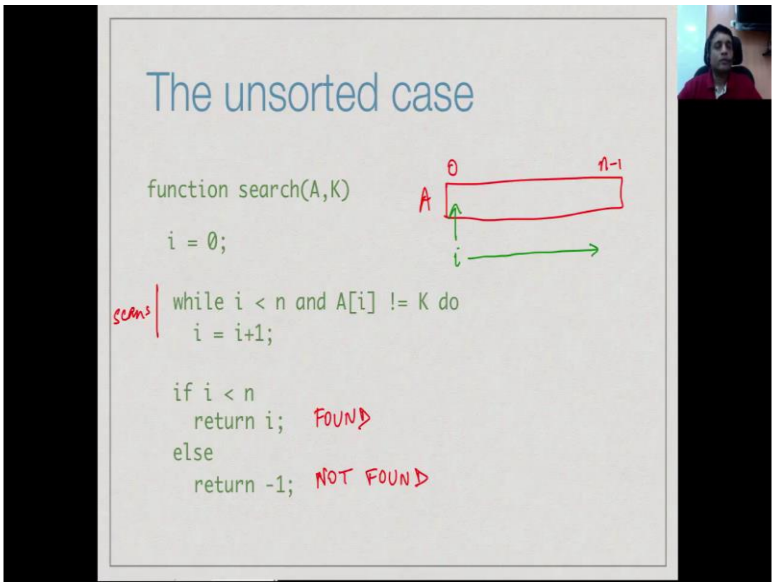
Search problem

- Is a value K present in a collection A? sequence
- Does the structure of A matter?
  - Array vs list
- Does the organization of the information matter?
  - Values sorted/unsorted

So, in general the search problem is to find whether a value K is present, present in a collection of values A and in our case we will think of A is generally as a sequence of values. And moreover, we also assume that the sequence is something like integers, where we can talk of one value being less than another value. So, the values can be ordered with each other. So, we have already saw that we can keep such sequences in two different ways, as a arrays and as lists.

And depending on whether we keep it as an arrays or list, the way we can access the elements is different. So, the first question we might ask is, whether searching makes a difference in a list verses an array and the second question we might ask is, is there some importance to how the values are arranged in the sequence, does it help if they are in ascending or descending order or it does not matter. It is equally the same to search for something in a randomly ordered collection of values or when it is structured in some particular way.

(Refer Slide Time: 01:03)



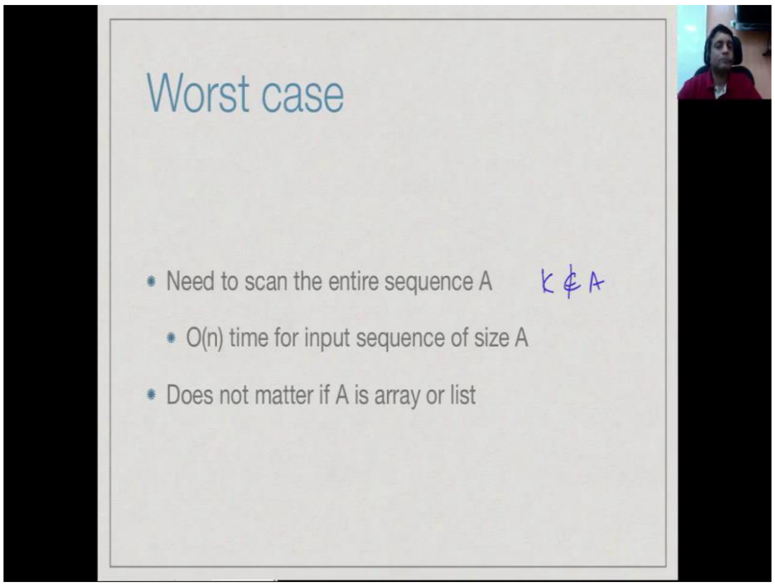
The unsorted case

```
function search(A,K)

  i = 0;

  while i < n and A[i] != K do
    i = i+1;

  if i < n
    return i;      FOUND
  else
    return -1;     NOT FOUND
```

So, in the unsorted case we have no choice basically, we have a sequence A which runs from 0 to n minus 1. So, we must look at all the values, because we have no idea where K maybe. So, systematic way to do it is to start with the position 0 and just scan all the way to n minus 1. So, we have this loop here, which scans and this scan either terminates when you reach the end without finding it or when at some position i we find that A i is equal to K.

And then depending on that we either say that it is found, in which case we return the position or we have reached i equal to n, which means we are gone beyond A n minus 1 and so, we return naught 1, minus 1 which is a invalid position indicate that it is not found.

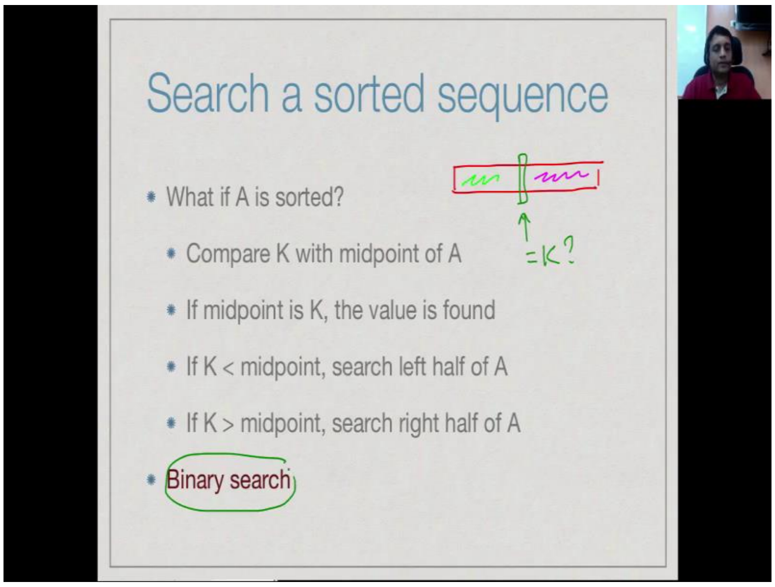
(Refer Slide Time: 01:53)


Worst case

- Need to scan the entire sequence A
  - O(n) time for input sequence of size A
- Does not matter if A is array or list

So, we saw before that the worst case actually happens when K is not an element of A, K does not come in A, we have to scan A 0 to A n minus 1 in order to determine the case not that. Because, we have no evidence in advance which position is likely to be. So, this means that in the worst case searching for an element and non sorted array takes linear type. And of course, it does not matter now whether it is an array or a list, because in a list we could also linear times start from the first element and follow the links all the way to the end, in an array we start with A 0 and go all the way to A n minus 1, both of them take linear time.

(Refer Slide Time: 02:29)

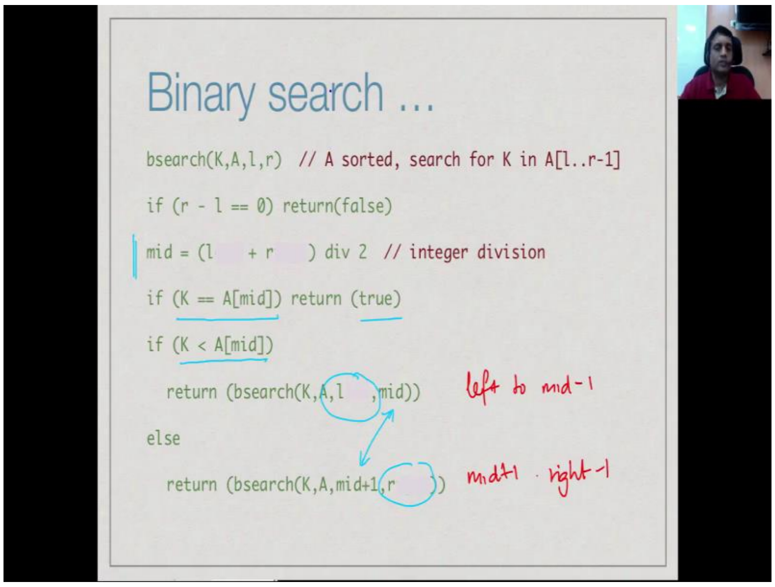

Search a sorted sequence

- What if A is sorted?
  - Compare K with midpoint of A
  - If midpoint is K, the value is found
  - If K < midpoint, search left half of A
  - If K > midpoint, search right half of A
- Binary search

On the other hand if the sequence is sorted and in particular if it is an array, we can be a little more intelligent. So, what we know is that the values are assigned in ascending order. So, if you probe the value in the middle and check is this equal to K, if the value that we have is equal to K of course, we have found it. If it is smaller than the value here, then we only need to search this half, and if it is larger than we need to search in this half.

Now, this is something that we intuitively do all the time, this is how we look for say words in a dictionary or when we play 20 questions, we try to ask questions about the age of a person, when you says this person less than 40, this person does not greater than 65 and so, on. So, this is something we know intuitively, but we can formalize. So, we take the midpoint of the range we are searching for. If the midpoint is a value that we want we found it; otherwise, we depending on the value we are looking for and what the value is that midpoint, we search either the bottom half or the top half. So, this has a name which many of you may already know, this is called binary search.

(Refer Slide Time: 03:38)



Binary search ...

```
bsearch(K,A,l,r)  // A sorted, search for K in A[l..r-1]

if (r - l == 0) return(false)

mid = (l    + r    ) div 2  // integer division

if (K == A[mid]) return (true)

if (K < A[mid])

  return (bsearch(K,A,l   ,mid))

else

  return (bsearch(K,A,mid+1,r    ))
```

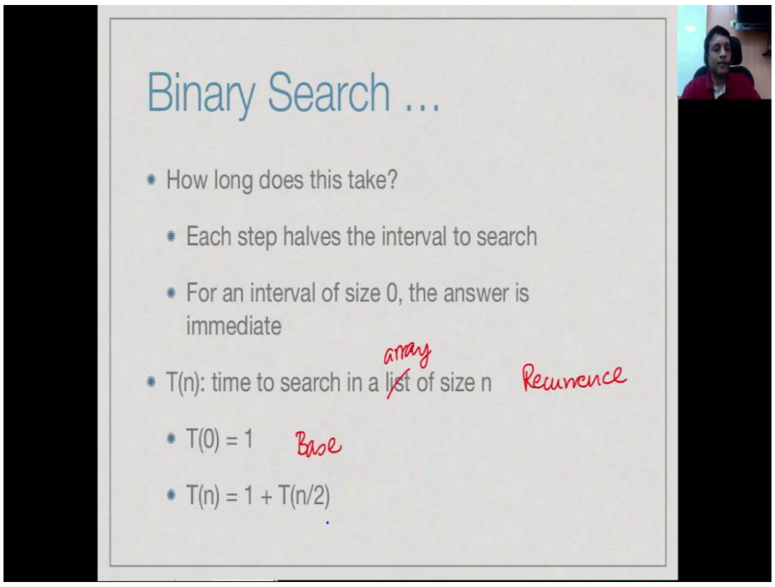
So, here is a simple recursive algorithm for binary search. So, in general it searches an array, remember that when we do the search we might be searching different segments depending on how far we progress the search. So, in general binary search takes a value K to search an array and two end points, left and right and just to make sure that we get a everything right, you will have the convention that it searches from the index l to the index r minus 1, let me searches from l to r, but not including r itself.

So, now if l and r are actually the same, then we have an empty array, because l to r minus 1 is actually something that there are no elements in the fields. So, we say we have not found it. So, when the interval that we are searching for becomes empty, the array definitely does not contain the value ((Refer Time: 04:27)). Otherwise, we compute the midpoint between l and r by taking the sum and dividing by 2 and because this might be an odd number, we use integer division.

Now, we examine at this point we are found the midpoint. So, now, we examine if the value that we want is there at the midpoint, if. So, we return true. Otherwise, if the value that we want to smaller than the midpoint, then we go to the left and the value that we want is bigger than the midpoint, then we go to the right. So, this either goes from left to mid minus 1 or mid plus 1 to right.

In other words we exclude mid from our search. So, this, the first case runs the search from left to mid minus 1 because that is our assumption, we call it to mid, it goes to mid minus 1. This one starts at mid plus 1 and goes to right minus 1. So, the original thing was from left to right minus 1 and we have now excluded mid from this and we have also have the entire bit to search.

(Refer Slide Time: 05:25)



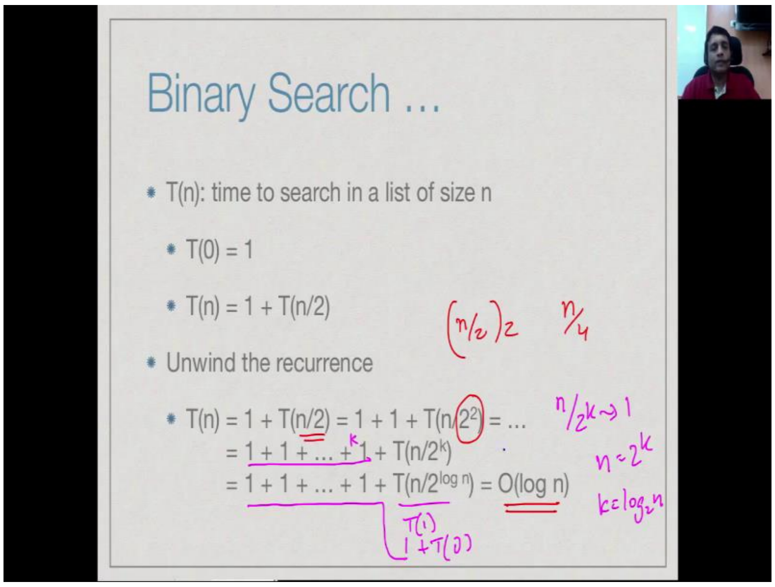
Binary Search ...

- How long does this take?
  - Each step halves the interval to search
  - For an interval of size 0, the answer is immediate
- T(n): time to search in a ~~list~~ array of size n Recurrence
  - T(0) = 1 Base
  - T(n) = 1 + T(n/2)

So, the crucial advantage of binary search is that each step we have the interval to search and at some point we will reach 1 and then when we have 1, we will get an interval of size 0 and so, we will get an immediate answer. So, we can write as we saw for such recursive functions, we can write what is called a recurrence. Recurrence is just an expression for the time, in terms of smaller values with same expression.

So, the base case is that when we have T of n we mean the time taken to search in a list or an array actually of size n. So, T of 0 is 1. So, if we have an empty array we have nothing to do and T of n in general is 1 step to find and compare. So, this one is actually a constant number of steps to compare with the midpoint and decide to go up, down and all that. So, those operations plus the time taken to search which the half we focus on, the left half and right half. Remember, that we look at the left half, we never going to look at the right half, you can.

(Refer Slide Time: 06:25)



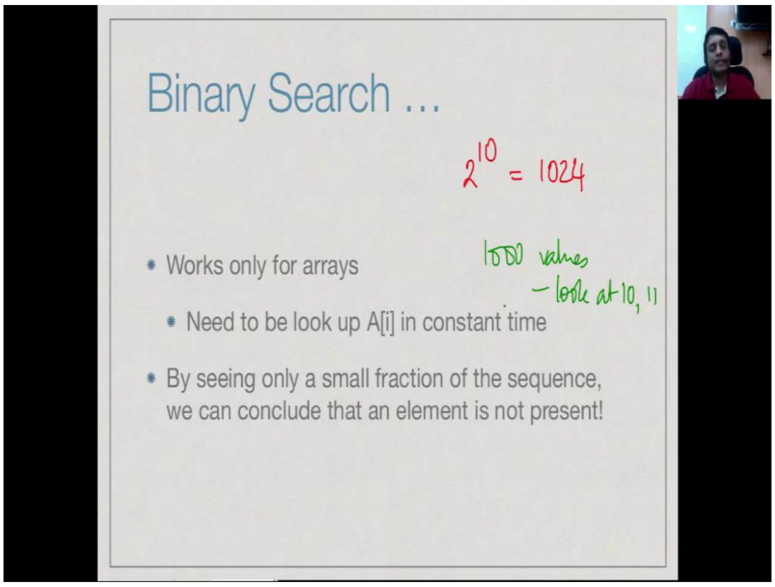
Binary Search ...

- T(n): time to search in a list of size n
  - T(0) = 1
  - T(n) = 1 + T(n/2)
- Unwind the recurrence
  - $T(n) = 1 + T(\underline{n/2}) = 1 + 1 + T(n/\boxed{2^2}) = \ldots$
    $= \underbrace{1 + 1 + \ldots + \overset{k}{1}} + T(n/2^k)$
    $= 1 + 1 + \ldots + 1 + T(n/2^{\log n}) = \underline{\underline{O(\log n)}}$

So, one way to solve such recurrence is to unwind it. So, we have T n is 1 plus T n by 2. So, we take n by 2 and we do n by 2 divided by 2 and rather than write it is as n by 4 we write is as n by 2 square. And this is because now, if I do one more time it will be 1 plus 1 plus 1 divided by 2 cube and so, on. So, in general you can see that if I do it k times then I am going to have k 1's here, if I do 3 times I have 1 plus 1 plus 1 plus T n by 2 to 3, 4 or 4 plus T n by 2 and so, on.

So, after k steps I have k plus T of n by 2 to the k, now when n by 2 to the k comes 1 at the next step I am going to get T of 0. So, when does this become 1, when n is 2 to the k in other words when k is log 2 of n. So, when I get log 2 of n, then this become T of 1 and at the next step this is going to become 1 plus T of 0. So, this is going to be log n 1s and so, over all the complexity of binary search is just order of log n. So, we are compromise a linear search in the case of unsorted array to a logarithmic search in the case of a sorted array.

(Refer Slide Time: 07:50)



Binary Search ...

$2^{10} = 1024$

- Works only for arrays
  - Need to be look up A[i] in constant time
- By seeing only a small fraction of the sequence, we can conclude that an element is not present!

So, we mentioned in the previous unit about arrays and list that, things that work on list may not work on arrays and vice versa. So, here is an example of something it works only for arrays. The idea of looking up the midpoint and then going left works only, if you can find the midpoint constant time, if you have to spend time looking for the constant for the midpoint, then you cannot get this recurrence anymore which not going to be 1 plus T of n by 2, but it is going to be n plus T of n by 2 plus here n by 2 and then we will actually get a linear.

So, binary search for a list will actually turn out be linear, because of the time it takes us to go to the midpoint. So, this works only for arrays, but really the remarkable thing about binary search is that by only looking at a very small fraction of sequence, we can conclude that an element is not present. So, we know for instance that 2 to the n, 2 to the 10 is 1024.

So, if I give you 1000 values, we can look at 10 or maybe 11 and say that something is not there. So, we overwhelming number or values we do not even have to look at in order to decide whether a value is there or not and this makes binary search to be remarkable procedure, if you think about it here.