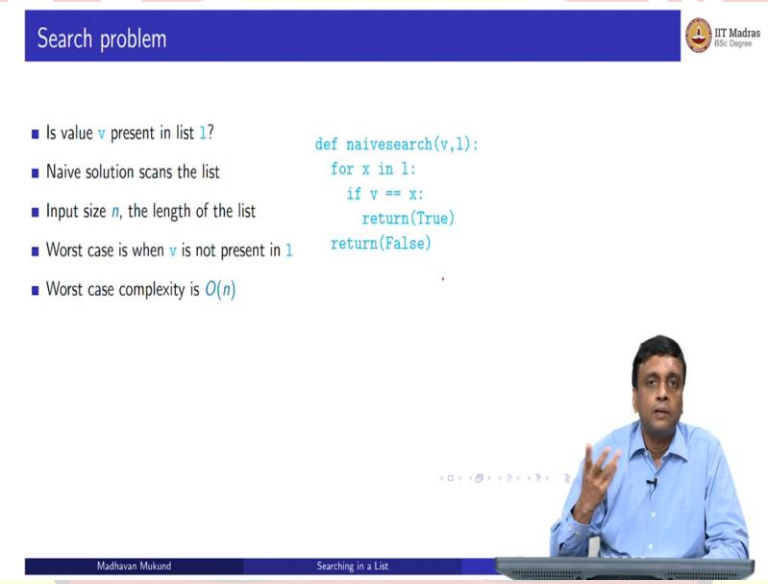
IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhvan Mukund
Searching in a List

So, with this background about analysis of algorithms, let us start looking at some basic algorithms, starting with one set we already examined. So, the first problem we looked at in the SIM card case was to search for a value in a list.

(Refer Slide Time: 00:22)



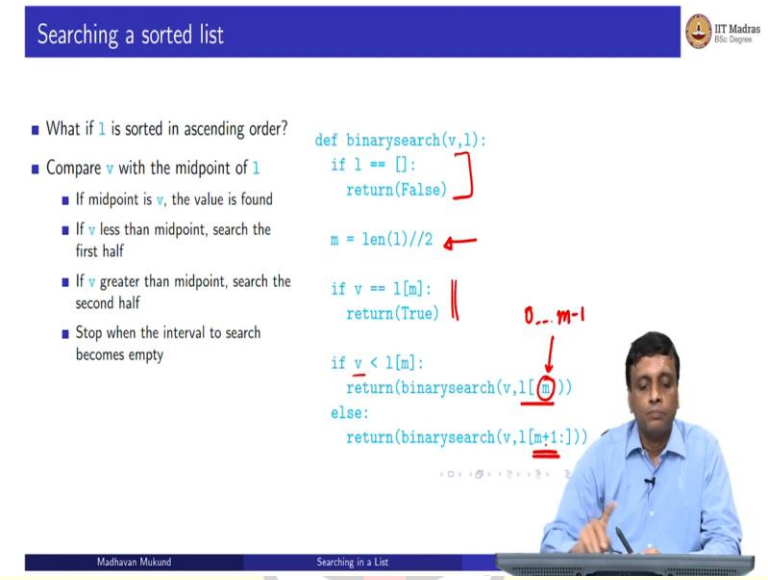
So, the search problem says given a value v and given a list l, is v present in l. So, as we said if we know nothing about the list, then the Naive solution is just the simple Python loop, which says for every x in l. Check if the value of v is a value x that you are looking for; if so, you return True. If you have exhausted all the elements in l and not found in v, then you return False.

So, this is a Naive solution, which scans the entire list. So, the input size as we said for a list is the length of the list; and in this case the worst case as we have discussed before will happen, when the value v is not present in the list, because you will have to go through every element of the list.

So, x will pass through every element in the list, and each of this x will not be v; and eventually you will come out of that loop and then return False. So, here using this analysis in our earlier terminology, we argued that the worst-case complexity of this particular algorithm is big O of n. So, notice this is a worst case, because I could very well have situations, where v is found in a

very first position. So, there is kind of best-case scenario, but I am not interested in best case scenarios; because they do not tell me much, they are like lucky shots. So, I am looking at what is the worst thing that the algorithm do; so, the worst case scenario here is O of n.

(Refer Slide Time: 01:36)



So, then we said what if this list is sorted in ascending order? So, then we came up with this strategy that guess my birthday strategy; where I compare v with the midpoint of v, interval I am searching in. So, I have 0 to n minus 1 as positions in my list, such that the values are in ascending order; so, I search at the midpoint.

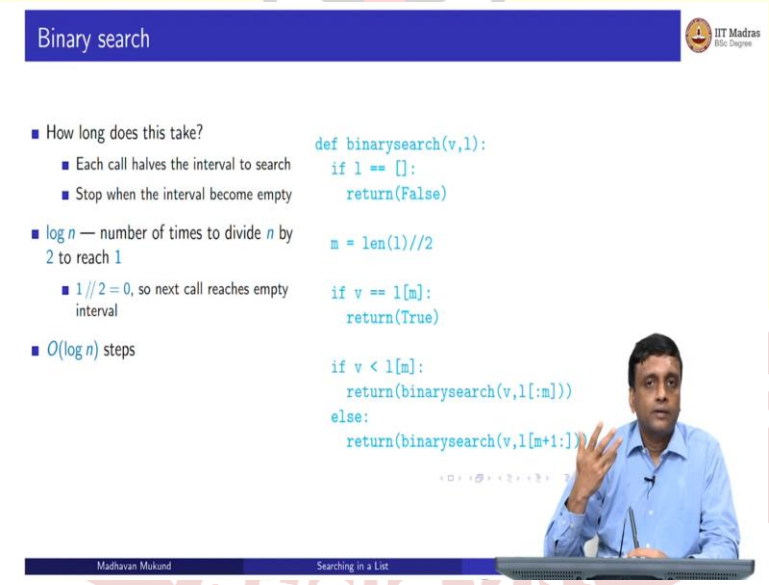
Now, if the midpoint happens to be the value I am looking for, I am done. If it is not the value I am looking for, then the value I will looking for is either beyond the midpoint or before the midpoint. So, depending on whether it is bigger than the midpoint value or smaller than the midpoint value; I either search the second half of the list, or the first half of the list. So, that is what this function binary search as it is called list doing.

So, if we first checks that if I have run out of values to look at; remember I am shrinking this thing. So, when I shrink it down from n to n by 2 to n by 4 and so on; eventually I will come to a list whose length is 1. Now, if it is not the value 1, then I will look at left of 1, which will be the empty list; or the right of that one element, which will again be the empty list. So, I will have to search in an empty list; but obviously I cannot assign v in an empty list. So, if the empty list is where I have reached and return False.

Otherwise, I use this integer division to get the length divided by 2 as a midpoint. I check whether the midpoint is the valid one. If the midpoint is equal to v, then I return True; if this is not the case, then I must now decide whether to go left or right. So, if the value I am looking for, it is not the midpoint; so, it is either strictly less than or strictly greater than. So, if it is strictly less than, then I will search in the slice up to, but not including the midpoint. So, remember in Python, this will go up to m minus 1; it will go from 0 to m minus 1. And not hit m, which is the midpoint which we have already seen.

Otherwise, if it is not smaller, it is bigger; and if it is bigger, then I want to start beyond the midpoint. So, m is the midpoint, so I start which I have already checked; so, I go to m plus 1 till the end of the list. So, this slice notation is very convenient in Python, to express the beginning of the list to m; or m plus 1 to the end of the list. So, in both cases I am skipping m, here because m is not included in this range calculation. And here because I am starting at m plus 1; so, this as we said is called binary search.

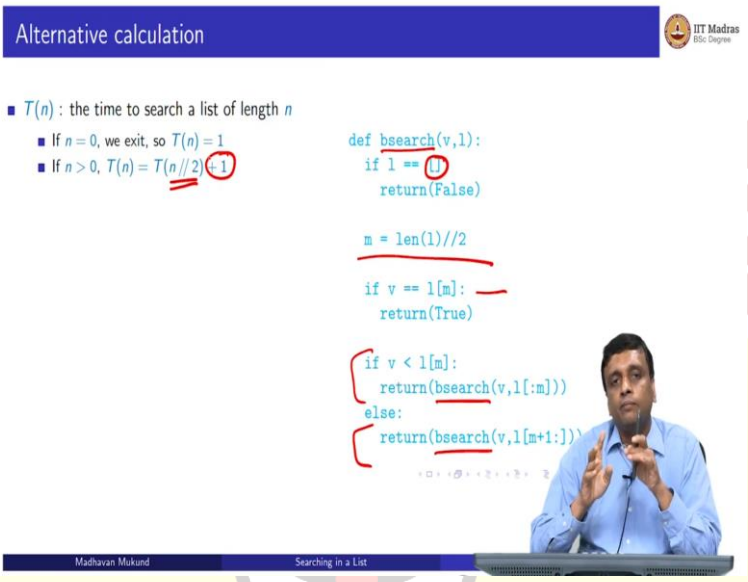
(Refer Slide Time: 03:55)

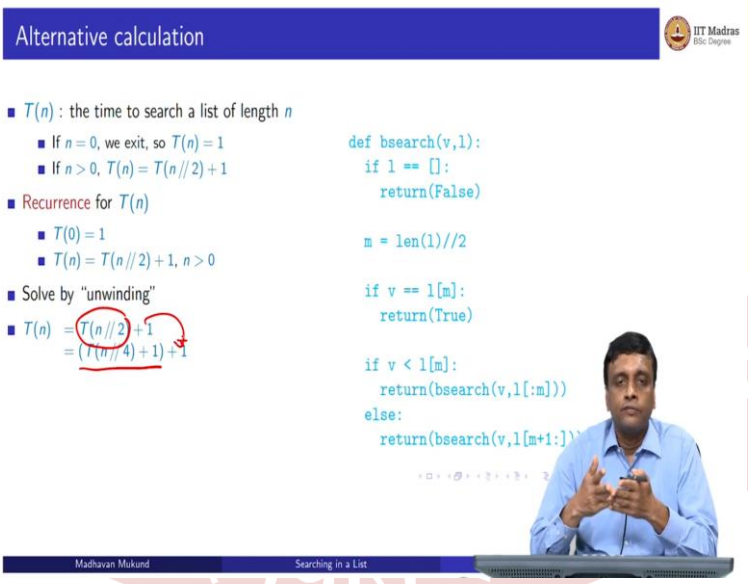


So, how long does this take? We have argued before many times, as this takes time proportional to how many steps you need to halve the interval down to 1. So, each step halves the interval and so we stopped when this interval becomes empty. So, log of n remember log of no with no base is always to the base 2 by assumption. Log of n is a number of times we divide n to get 1; and now I have an interval of size 1 and I divide by 2 again, I get 0. So, log of n plus 1, but we will ignore

the plus 1; so, this is going to be big O of log n. So, that is one way to do the calculation, just to analyze at a kind of at the level of the meaning of the program. Try to argue what it is doing and make the kind of high level of analysis of that.

(Refer Slide Time: 04:41)





Now, there is a more explicit way to do this, especially for functions like this, which essentially, I just shortened to make it fit on the page; so, binary search uses itself. So, this is a recursive function. So, function on the full list calls itself one half the list. So, let T of n as we normally say be the time it takes to search a list of length n. So, we have two cases if the length is 0, it is an

empty list; then we just check and leave. So, this is one basic operation checking whether the list is empty or not; so, in one step I get my answer, which is always False.

On the other hand, if the answer is not that, then I have to do some work. So, I have to compute the midpoint, check this and all that. So, in the worst case remember this is always worst case; what I have to is I have to search either the first half or the second half. So, I have to solve the same problem for half the list; so that is T n by 2. And I have already spent one step deciding which half to do and all that, more than one step.

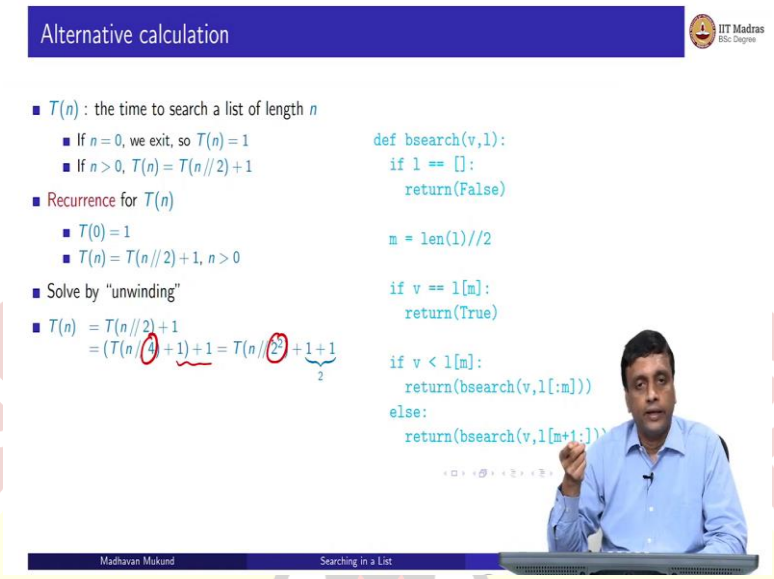
But, as I said we have collapsed it all that saying this whole thing is of three steps or for steps or whatever it is that, you want to count for these comparisons and divisions and all that. So, we have spent certain amount of time, but a fixed amount of time to decide which half to call, and then we have to solve that.

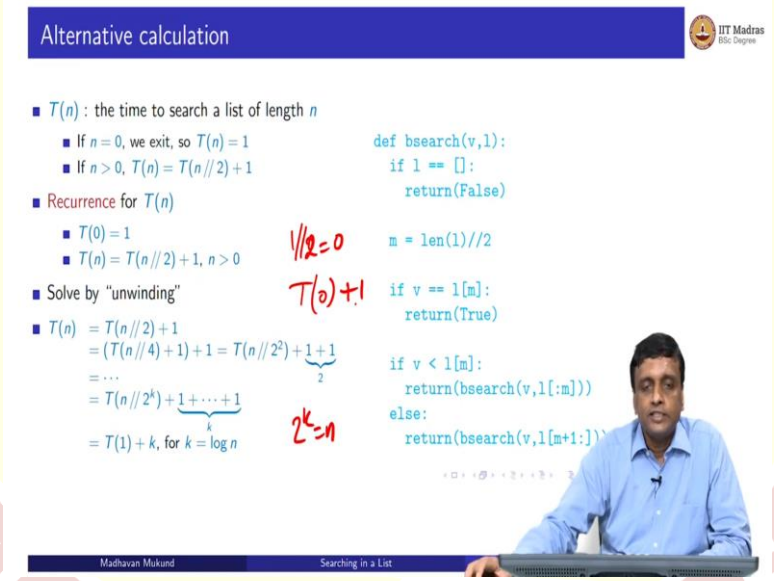
So, it is T n by 2 plus 1; so this is what is called the recurrence. So, T of n for binary search is 1 if n is 0; and if n is bigger than 0, it is T n by 2 plus 1. So, how do we solve this? The easiest way to solve this, if you can manage to find out a pattern is to unwind it. So, unwind it means you take the definition and you keep expanding it again and again.

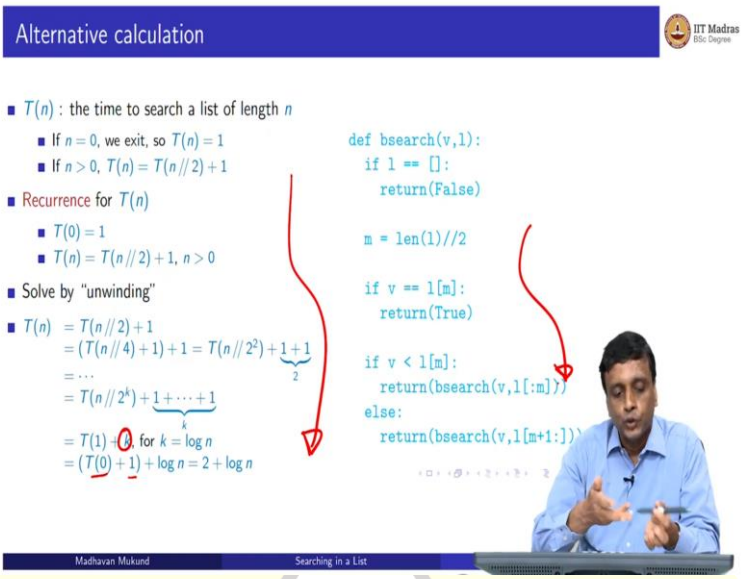
So, we start with T of n, so what is T of n? by definition it is T of n by 2 plus 1. So, I do not know what this value is? So, I again apply the definition to it. So, if I trick n by 2 as the input, then it will expand as T of n by 4, half of it plus 1. So, this is an expansion of this part and then I have this old plus 1 coming here; so, this is what mean by unwinding.

You take the definition whenever you have a expression which you do not know the answer to apply the definition again. So, T of n, I apply the definition, got T of n by 2; T of n by 2, I got the definition; I apply the definition again I got T of n by 4.

(Refer Slide Time: 07:00)







So, now if I kind of simplify this and rewrite it; I will take this two once together and say there are 2 ones. But I will also rewrite this and saying that I divided by 2 twice. So, instead of writing it as 4, I will write it as divided by 2 to the power 2; showing that I divided twice. I started with n and divided by 2, started with n by 2 and divided by 2 again. So, in general now if I do this k times, that is why if I applied this binary search case k times, I have gone left and right left and right. Then the interval would have been divided by 2k times; so I would have an interval of size n by 2 to the k.

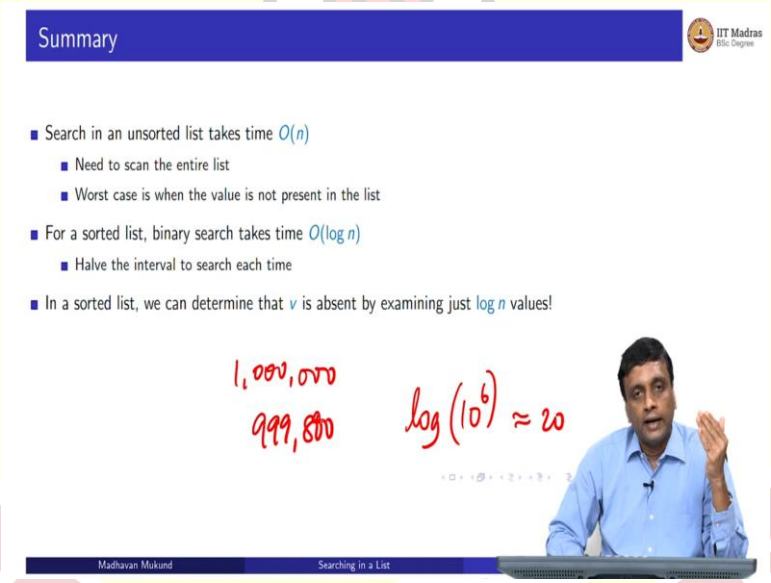
And each time I would have spent one step of work to try and decide whether to go left or right. So, I would have spent k times one work; so I would have T n by 2 to the k plus k times 1. Now, of course we said and we have seen many times, as when will n by 2 to the k reach the limit that we want, when it becomes 1. So, after log n steps, I will have T to the 1; so 2 to the k is equal to 1. So, after log n steps, I will end up with T to the 1 plus log n of course. So, T to the 1 plus k, where K is log n. but, what T of 1? T of 1 is T of 0, because 1 by 0, 1 by 2 is 0 plus 1. So, if I expand this T of 1 one more time, because I do not have an expression T of 1.

I get T of 0 plus 1 plus log n; log n because I stopped with k equal to log n. So, I get this plus 1 that is 2, because T of 0 is 1, I have a 1 here; so, 2 plus log n, so, again it is big O of log n. So, the interesting thing here is that without trying to analyze what is happening; so notice that I am not this this whole thing here that I have done on the left hand side. It was merely a kind of algebraic manipulation based on what is happening in the code. So, I have not try to understand what is

happening in the code. So, the earlier analysis, which I gave you said, I am halving the interval and the interval keeps halving; and at thumb point interval become the trivial interval, and how many times it will take.

So, here I am not asking anything about what the algorithm is doing. I am just saying this is the call; if I call it with n, it ends up calling itself with n by 2. And in order to do that, I have to do a certain amount of work in between. So, this is a more syntactic way of analyzing an algorithm; it does not require me to understand at a higher level, what process the algorithm is doing and yet I arrived at same answer. So, this is useful to know that when you have a kind of recursive algorithm like this; it is very important to be able to write this recurrence. And then usually you can solve it by unwinding it like this.

(Refer Slide Time: 09:48)



So, to summarize the first non-trivial algorithm that we have seen in this course is searching. So, given a value, given a list, search for this value in the list; if the list is not sorted, you cannot do anything better and big O of n. So, we have to scan the entire list and the worst case is when the value is not present. On the other hand, if the list is sorted, then we can use its binary search by looking at the midpoint. And an order log n time by, halving the interval to search at each time; I will find or not find the value in log n steps. So, this is really remarkable if you think about it; because it says that log of say 10 to the power 6 is roughly 20.

So, that means if I give you one million sorted values, and I give you a new value which is not present in these one million sorted values; you will only look at 20 of them. So, ninety nine (thous), nine hundred and ninety nine thousand nine hundred and eighty values, you will have to you can ignore. So, these many values can be ignored; you have start with one million values. So, this truly a remarkable property that almost the entire list you can ignore; and still, you can convince me that the value I am looking for is not there. So, this is truly remarkable.