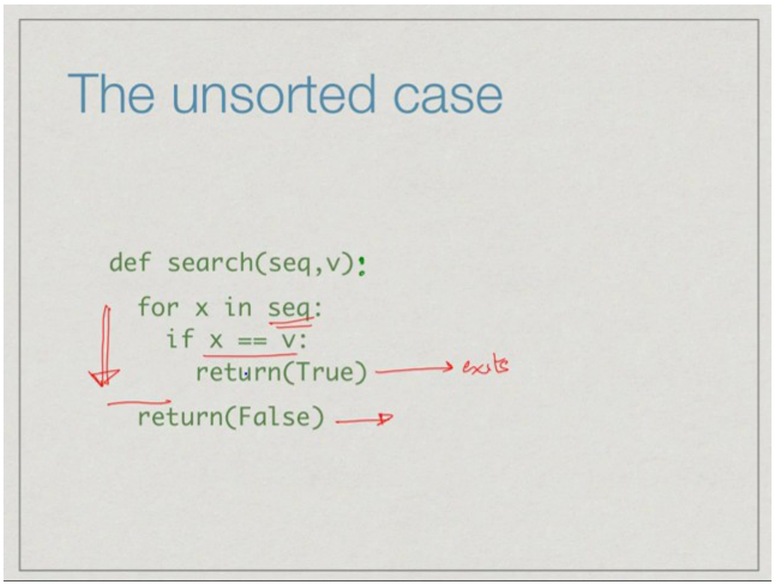
(Refer Slide Time: 10:37)



The unsorted case

```
def search(seq,v):
  for x in seq:
    if x == v:
      return(True)
  return(False)
```

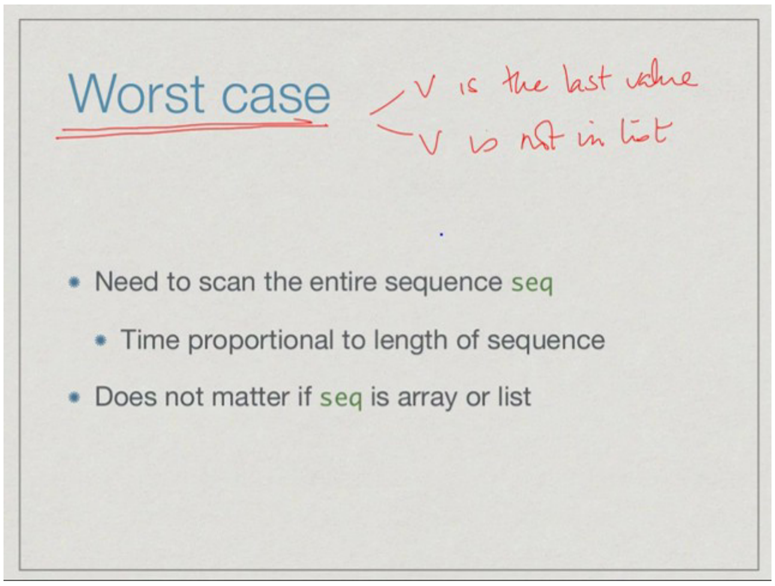
Here is a very simple Python program to search for a value in a unsorted sequence. This is a similar to what we saw before where we are looking for the position of the first position of a value in a sequence, which is we not we do not even need the position we only need true or false, is it there or is it not, it is a very simple thing. What we do is we loop through all the elements in the sequence and check whether any element is the value that we are looking for.

Once we have found it we can exit, so this exits the function with the value true. And if we have succeeded in going through the entire list, but we have not exited with true that means we have not found the thing, so we can unambiguously say after the for that we have reached this point we have not found the value v that we are looking for and so we should return false.

Since we are not looking for the position we have much simpler code if you go back and see the code we wrote for findpos, so there we had first of all keep track of the position and check the value at position i rather than the value itself. And secondly, when we finish the loop we had to determine whether or not we had found it or we had not found it, whether we had remember we use the break to get out of the loop for the first time we found it.

We used to detect whether we broke or not, if we did not have a break then we had found it, if we did not had a break we did not find it. Accordingly either the value of pause was set or it was not set and if it is not set we should make it minus 1. So that was more complicated, this is very simple.

(Refer Slide Time: 12:07)



Worst case

- Need to scan the entire sequence seq
  - Time proportional to length of sequence
- Does not matter if seq is array or list

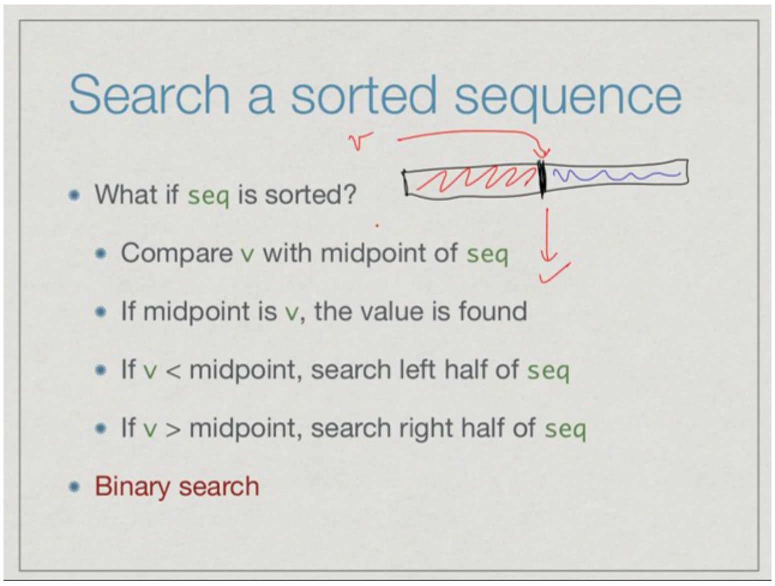
v is the last value
v is not in list

The main point of this function is that we have no solution to search other than to scan from beginning to end. The only systematic way to find out v occurs in the sequence or not is to start at the beginning and go till the end and check every value, because we do not have any idea where this value might be. This will take time in general proportional to the length of the sequence.

We are typically interested in how long this function would take in the worst case. So what is the worst case? Well, of cause one worst case is if we find the value at the end of the list. So, v is the last value then we have to look at all. But more generally v is not in the list. v is not in the list the only way we can determine the v is not in the list is to check every value and determine that that value is not found.

And this property that we have to scan the entire sequence and therefore we have to take time proportional to the sequence to determine whether v is in the sequence or not it does not matter if the sequence is an array or a list, whether it is an array or a list we have to systematically go through every value the organization of the information does not matter. What matters is the fact that there is no additional structure to the information, the information is not sorted in any way at no point can we give up and say that since we have not seen it so far we are not going to see it later.

(Refer Slide Time: 13:26)



Search a sorted sequence

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

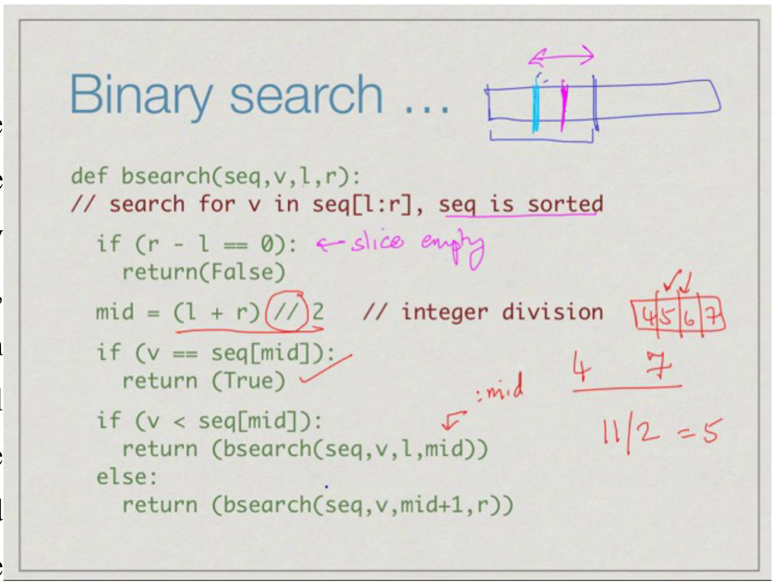
On the other hand, if we have a sorted sequence we have a procedure which would be at least informally familiar with you. When we search for a word in a dictionary for example, the dictionary is sorted by alphabetical order of words. If we are looking for a word and if we open a page at random, supposing we are looking for the word monkey and we open the dictionary at a page where the values or the word start with i, then we know that m comes after i in the dictionary order of the English alphabet. So, we need to only search in the second half of the dictionary after i, we do not have to look at any word before i.

In general if we have a sequence that efficient way to search for this value is to first look at the middle value, so we are looking for v, so we check what happens here. So, there are three cases either we have found it in which ways which case we are good, if we have not found it we compare the value we are looking for with what we see over there. If the value we are looking for is smaller than the value we see over there, it must be in this half.

On the other hand if the value we are looking for is bigger it must be in this half. So we can halve the amount of space to search and we can be sure that the half we are not going to look at positively does not have the value because we are assuming that this sequence is sorted. This is called Binary search.

This is also for example what you do when you play game like twenty questions, if you play that when somebody ask you to guess the name of a person they are thinking of then you might first ask the question whether the person is female, if the person is female then the persons and their answer is yes then you only need to think about women, if the person says no then you only need to think about m, so we have men. So, you have half number of people in your imagination we have to think about. At each point each question then further splits into two groups depending on whether the answer is - yes or no.

(Refer Slide Time: 15:22)



Binary search …

```
def bsearch(seq,v,l,r):
// search for v in seq[l:r], seq is sorted
  if (r - l == 0):
    return(False)
  mid = (l + r)//2   // integer division
  if (v == seq[mid]):
    return (True)
  if (v < seq[mid]):
    return (bsearch(seq,v,l,mid))
  else:
    return (bsearch(seq,v,mid+1,r))
```

← slice empty
:mid
4 5 6 7
4 7
11/2 = 5

Here is some Python code for binary search. So, binary search in general will start with the entire list and then as we said it look at the midpoint and decide on the left, so we will have to again perform binary search on this. How would we do that? Again we will look at the midpoint of this part then we are again look at say the midpoint of the next part that we look at and so on.

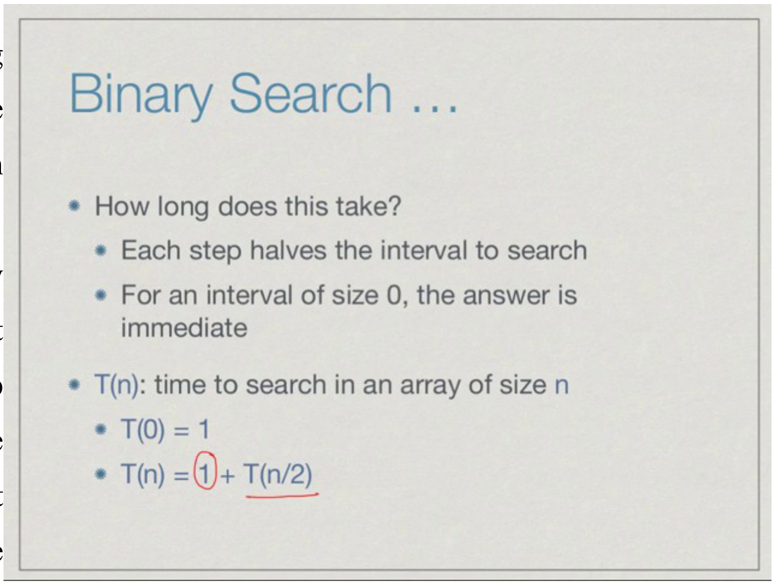
In general binary search is trying to do a binary search for a value in some segment of the list. So we will demarcate that segment using l and r. So, we are looking for this slice sequence starting with l and going to r minus 1, we are assuming that sequence is sorted and we are looking for v there. First of all if the slice is empty, so this says the slice is empty that is we have gone halving the thing and we have eventually run out of values to look at. The last thing we look at was the slice of length 1 and we divided it into 2 and we got a select of slice of length 0. Then we can say that we have not found it yet, so we are not going to ever find it and we return false.

On the other hand if the slice is not empty, then what we do is we first compute the midpoint. An easy way to compute the point is to use this integer division operation. Supposing, we are have currently the slice from 4 to 7 then at the next point we will take 11 by 2 integer wise and we will go to 5. Remember 4, 5, 6, 7. We could either choose 6 or 7 then next to split it into two parts, because we are going to examine 6 and then look at 4, 5 and 7 or look at 5 and then 4, 7. If we do integer division then we will pick the smaller output. So, we find the midpoint. Now we check whether the value is the value at that midpoint if so we return true, if it is not then we check whether the smaller, if so we continue our search from the existing left point till the left of the midpoint.

Now we are using this Python, think that this is actually means this is a slice up to mid and therefore it stops at mid minus 1. So, it will not again look at the value we just examined. it will look at everything strictly to its left. If the value that we are looking for is not the value with the midpoint and it is smaller than the midpoint, look to the left, otherwise you look strictly to the right, you start at mid plus one and go up to the current right line.

This is a recursive function. It will keep doing this at each point the interval will half, so eventually supposing we have a slice of the form just one value, so 5 to 6 for example, then at the next point right we will end up having to look at just a slice from 5 to 5 or 6 to 6 and this will give us a slice which is empty because we will find at the right point at the left point are the same.

(Refer Slide Time: 18:08)



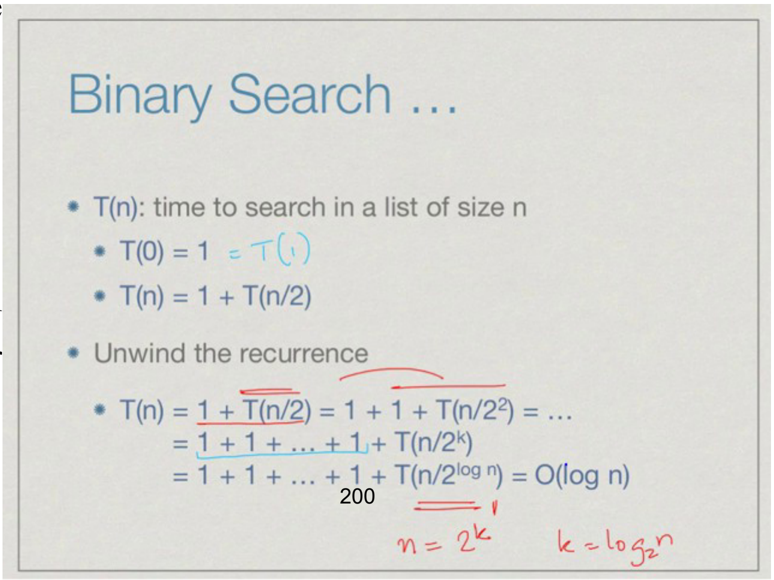
Binary Search …

- 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 an array of size n
  - T(0) = 1
  - $T(n) = \boxed{1} + \underline{T(n/2)}$

So, how long does the binary search algorithm take? The key point is that each step halves the interval that we are searching and if we have an empty interval we get an immediate answer. So, the usual way we do this is to record the time taken depending on the size of the sequence or the array or the list, so we have written array here, but it would be sequence in general. If the sequence has length 0 then it takes only one step because we just report that it is false we cannot find it if there are no elements left.

Otherwise, we have to examine the midpoint, so that takes one abstract step you know computing the midpoint and checking whether the value is we will collapse at all into one abstract step. And then depending on the answer, remember we are computing worst case the answer in the worst case is when it is going to be found in the sequence. So, the worst case it will not be the midpoint we will have to look at half the sequence. We will have to again solve a binary search for a new list which is half the length of the old list, so the time taken for n elements is 1 plus the time taken for n by 2 elements.

(Refer Slide Time: 19:18)



Binary Search …

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

$n = 2^k$ $k = \log_2 n$

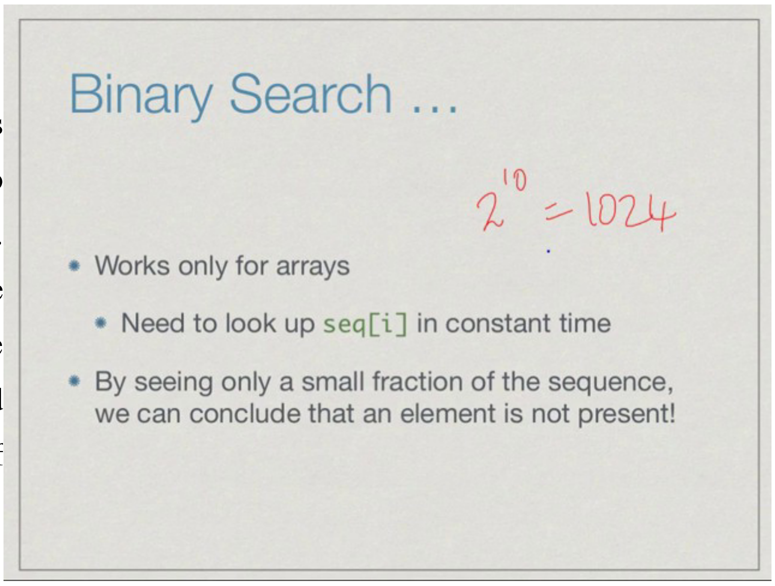
We want an expression for

T of n which satisfies, so this is what is called a recurrence normally. What function T of n would satisfy this? One way to do that is just keep substituting and see what happens. We start unwinding as itself, so, we have this by the same recurrence should be 1 plus T of n by 4, because I take this and halve it. So, T of n is 1 plus 1 plus T of n by 4. So, we start with 1 plus T of n by 2 and I expand this. Then I get 1 plus 1 plus T of n by 2 squared and in this case I will again get 1 plus 1 plus 1 by T of n by 2 cube. In general after k steps we will have 1 plus 1 plus 1 k plus 1 times or k times and t of n by 2 to the k.

Now when do we stop? We stop when we actually get T of 0 or we can also say that for T of 1 it takes one step just we want to be careful. So, when this expression becomes 1 so when n is equal to 2 to the k. So, when is n equal to 2 to the k, this is precisely the definition of log right. How many times do I have to multiply 2 by itself, in order to get n and that is the value of k that we want. After log n steps this term will turn out to be 1. We will end up with roughly log n times 1 added up and so we will get log n steps.

So what we are saying is really, if we start with the 1000 values, in the next step we will end up searching 500, next step 250, next step 125, next step 62 and so on. And if we keep doing this when will we get to a trivial sequence of length 0 or 1. Well, be keep dividing 1000 by 2 how many times can we divide 1000 by 2 that is precisely the log of 1000 to the base 2 and that is an equivalent definition of log.

(Refer Slide Time: 21:20)



Binary Search …

$2^{10} = 1024$

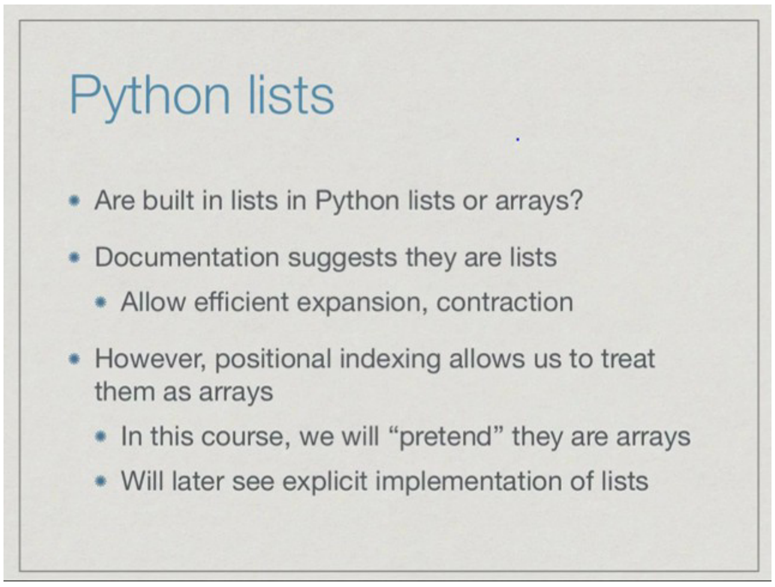
- Works only for arrays
  - Need to look up `seq[i]` in constant time
- By seeing only a small fraction of the sequence, we can conclude that an element is not present!

This comes back to another point. Now we have said that if we had a sorted sequence of values we can do this clever binary search, but remember that it requires as to jump in and compute mid which is fine and we need to then look at the value at the midpoint and we are assuming that this computation of mid and comparing the value of the midpoint to constant amount of time, that is why we said that it is 1 plus T n by 2 this 1 involves computing mid and looking up the frequency at the midpoint. But this can only be done for arrays because only for arrays can we take a position and jump in and get the value at that position in constant time, it will not work for lists, because we need to look up the sequence at the ith position in constant time.

Of course, one important and probably not so obvious thing if you think about binary search is that by only looking at a very small number of values, say for example we give you a sorted list of 1000 entries as I said if a value is not there we only have to search 10 possible entries, because we keep having after log n which is about to remember the 2 to the 10 is 1024 right two times, two times, two ten times is 1024. After 10 halvings of 1000 we would have come down to 0 or 1. We would definitely be able to tell quickly whether it is there or not. So, we only look at 10 values out of 1000, 999 values we do not look at all unlike the unsorted case where we have to look at every possible value before we solve.

It is very efficient binary search, but it requires us to be able to jump into the ith position in constant time therefore if I actually did a binary search on a list even if it is sorted and not on an array where I have to start at the 0th position and walk to the ith position by following links unfortunately binary search will not give me the expected bonus that I get when I use an array.

(Refer Slide Time: 23:18)



Python lists

- Are built in lists in Python lists or arrays?
- Documentation suggests they are lists
  - Allow efficient expansion, contraction
- However, positional indexing allows us to treat them as arrays
  - In this course, we will "pretend" they are arrays
  - Will later see explicit implementation of lists

So having discussed this abstractly, we are of course working in the context of Pythons. The question is, are built in lists in python are they lists as we have talked about them or are they arrays. Actually, the documentation would suggest if you look at the Python documentation that they are lists because you do have these expansion and contraction functions so we saw we can do an append or we can do a remove of a value and so on. They do support these flexible things which are typical of lists, however Python supports this indexed position right so it allows us to look for a to the i.

If you try it out on a large list you will find that it actually does not take that much more time to go say it construct a list of a hundred thousand elements, you will find it takes no more time to go to the last position as to the first position as you would normally expect in a list we said that it should take longer to go to the last position.

Although they are lists as far as we are concerned we will treat them as arrays when we want to, and just to emphasise how lists work when we go further in this course we will actually look at how to implement some data structures. And we will see how to explicitly implement a list with these pointers which point from one element to another.

For the rest of this course whenever we look at a Python list we will kind of implicitly use it as an array, so when we discuss further sorting algorithms and all that we will do the analysis for the algorithms assuming they are arrays, we will get give Python implementation using Python's built in list, but as far as we are concerned these lists are equivalent to arrays for the purpose of this course.