

Programming, Data Structures and Algorithms
Prof. N.S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Module - 03

Lecture – 30

Searching

Unordered linear search and analysis

Ordered linear search and analysis

Binary search on an ordered array

Binary search illustration and code

Analysis for binary search

So, today let us continue our study of algorithms with an exploration of this area of searching, as most of us no searching is actually a fairly common word now associated with computer science, because of search engines which are accessible to most of us. And let start off with simplest of searching exercises, where we want to search a data structure for a key which is given as input.

(Refer Slide Time: 00:50)

Searching

The process used to find the location of a target among a list of objects

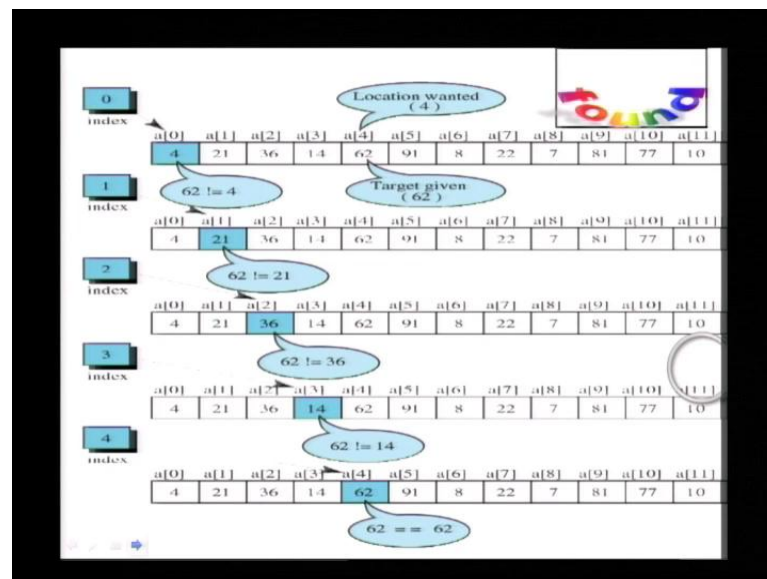
Searching an array finds the index of first element in an array containing that value

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	21	36	14	62	91	8	22	7	81	77	10

So, in this lecture we are going to look at this issue of searching an array. Of course, one could consider the problem of searching other data structures. We will come to it as we progress. So, if you look at the question of searching. Searching is a process that is used to find the location of a given key or a target among the list of objects. So, when you search an array, the search algorithm, is expected to return the first element, in the array

that contains the given key. In this picture we can see that the target key that is been given is a key 62, and it occurs in the location 4, and we want to design an algorithm which efficiently gives us the value 4. The approach to achieve this particular task is given to us in the coming slide.

(Refer Slide Time: 01:47)

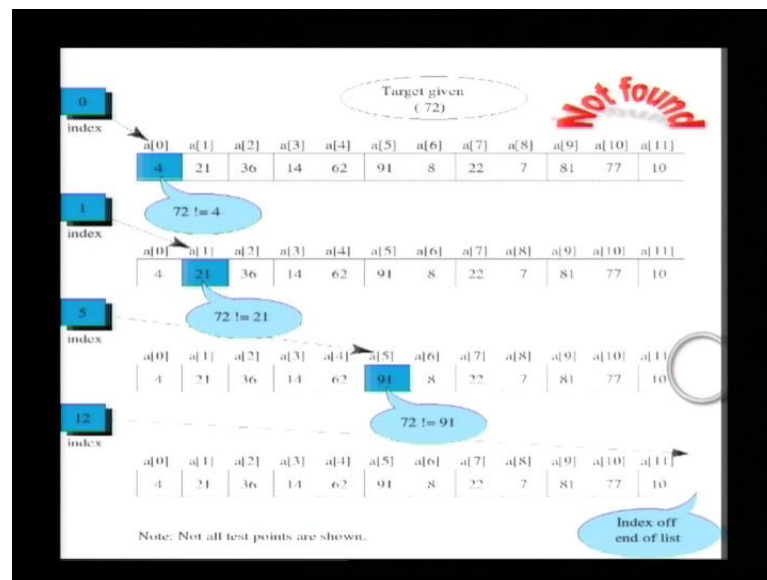


So, now let us explore the procedure for the search of the given key 62, which we know from this visual is at location 4 or it is in the array index within number 4. And it is important to note that the indices of the arrays are from index 0, all the way up to index 11; that is there are 12 elements in this array, and we want to search for a given key which is 62. Let us look at the basic steps that have to be done. We start by comparing with the element which is at index 0, and the element there is 4, the comparison of 62 and 4 definitely results in the fact that they are not equal. And next comparison would be with the element at index one, which is 21 and so on till we come to the third element for example, where again a comparison with 62 is made with a value which is their, which is 14. And after 5 iterations they remain 62 is found for the first time in the array index four, and this is considered as a discovery or a successful search, where the search key has been found.

Now, it is a act to indeed call such a search procedure and linear search, because in every

iteration we queried index in the array, increases by 1. If you plot the indices of the locations which have been probed or searched, you will find that this plot against iteration number; that is the x axis been the iteration number, and y axis be the index of the location search. You will see that this plot is a straight line, and into the natural to call is a linear search. This is very important to understand, why this is called linear search. The array indices as searched in a linear fashion. It starts off with 0 then one then 2 and 3 and so on and so forth.

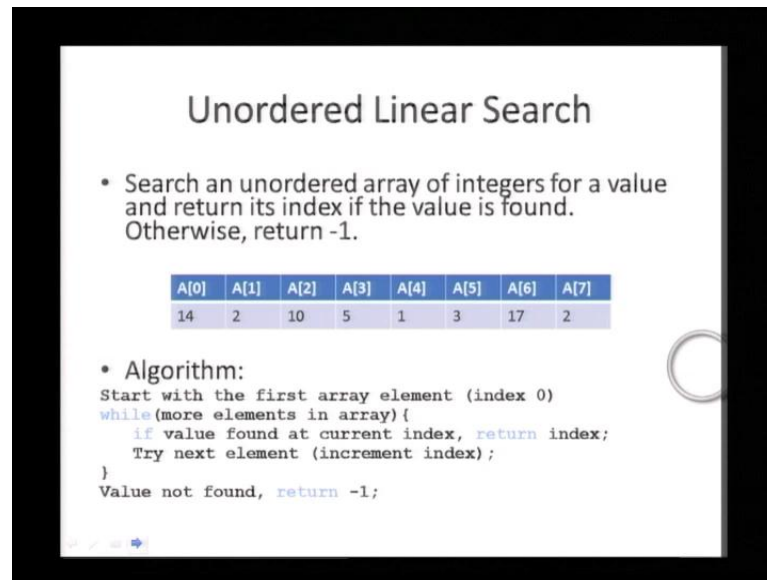
(Refer Slide Time: 04:01)



It is considered the running time to search for a key which is seventy 2, which as you can see does not occur in this array. And there are 12 elements in this array, and we do not show all the comparisons. This is definitely not necessary. As you can see all the comparisons will fail in this linear search, till the index value exceeds 11; that is it becomes 12. At which point if time you have an exit condition; that is you have searched the array, compared every element is a array with a given key, and you have exceeded the total number of elements that are there in the array, and therefore, the element is not present in the array. And the algorithm at this point of time can report that the given key is not present in the array. It is very important to note that we have observed two of the exit conditions of this algorithm, when the key is found, and when the key is not found. The key is not found when you compared it with all the elements in the array. And when

the first time the key is found in the array, the algorithm exits. These are two exit conditions, and this can be encoded.

(Refer Slide Time: 05:25)



The slide is titled "Unordered Linear Search". It contains a bullet point describing the search process: "Search an unordered array of integers for a value and return its index if the value is found. Otherwise, return -1." Below this is a table representing an array with 8 elements, indexed from A[0] to A[7]. The values are 14, 2, 10, 5, 1, 3, 17, and 2. Below the table, there is a section titled "Algorithm:" followed by a pseudo-code block. The pseudo-code starts with "Start with the first array element (index 0)", then enters a "while" loop that continues as long as there are more elements in the array. Inside the loop, it checks if the value is found at the current index; if so, it returns the index. If not, it increments the index and tries the next element. After the loop, it returns -1 if the value was not found.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
14	2	10	5	1	3	17	2

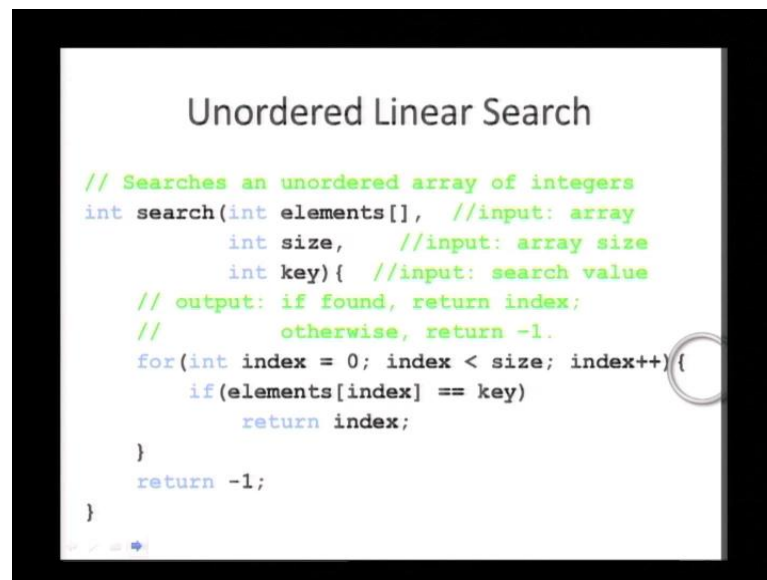
```
Algorithm:  
Start with the first array element (index 0)  
while(more elements in array){  
    if value found at current index, return index;  
    Try next element (increment index);  
}  
Value not found, return -1;
```

In the following algorithm, and to make a distinction which something that we are going to study we call this unordered linear search. The algorithm is linear search, and it is unordered linear search, because the algorithm does not use any structure of the data elements which are present in the array. For example, the data elements in the array could be sorted, but the description of the algorithm does not use that fact, and therefore, this is called an unordered linear search. In other words unordered linear search is applicable, when you have searching for a given key in an array, in which you have no apriory information about the organization of the data elements. Data elements could be in a sorted order, or they could be in unsorted order.

And we have seen the conditions under which, a straightforward linear search algorithm will exit, and this is encoded in this pseudo code which is described. It says that while there are some more elements in the array. If the value is found at the current index then you return the index of the current location; otherwise you increment the index, and continue in the loop. At the end if the value is not found at all. A return value of minus one is given. Of course, if a programs is one has be very careful, and ensure that the

array indices are between, or at least as largest 0, and minus one is not an array index. The programming language like c, this is definitely the case where there the indices start at 0.

(Refer Slide Time: 07:04)

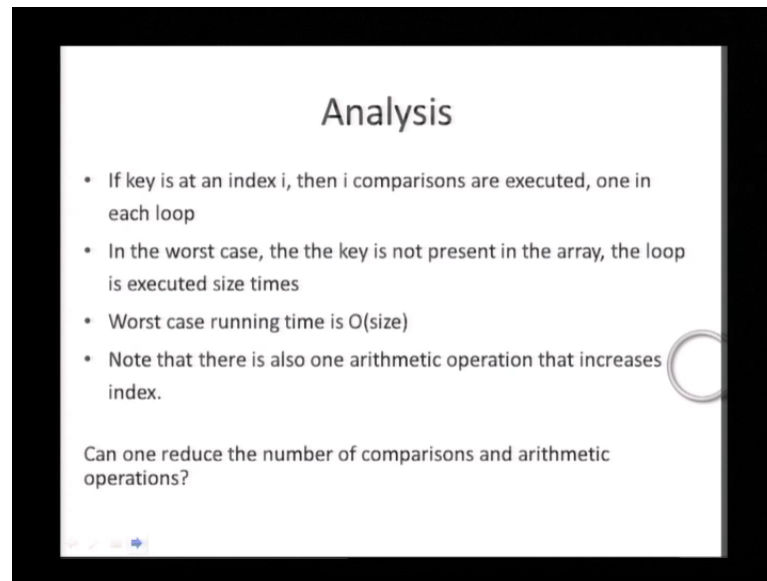


```
Unordered Linear Search

// Searches an unordered array of integers
int search(int elements[], //input: array
           int size,      //input: array size
           int key){ //input: search value
    // output: if found, return index;
    //           otherwise, return -1.
    for(int index = 0; index < size; index++){
        if(elements[index] == key)
            return index;
    }
    return -1;
}
```

So, this is a snapshot of a c program which is included in the slide, to understand the complexity, or the amount of time that is spent in executing this particular algorithm which is unordered linear search. So, the function that we have written here that you see here is called search function. The arguments to this function is an array, which is called elements, and the size of the array is given as an additional argument, just to illustrate this example, and the key, the desired that is being search for in elements is also passed as an argument search. The for loop there that initializes and indexed to 0 on search as up to the size of the array, and in every iteration, in every loop the element access at a particular indexes compared with a given key, and if a successful match is made then the indexes returned, as the location where the key is present. If the key is not present in the whole array, which is discovered after the loop has run for as many steps has the size of the array. The control exits from the loop, and a minus one value is return by the search function, which informs the calling function that the key has not been found. The use of this c program mainly, is for us to understand where the effort in computation is, for us to be able to, say something about the running time of this algorithm.

(Refer Slide Time: 08:53)



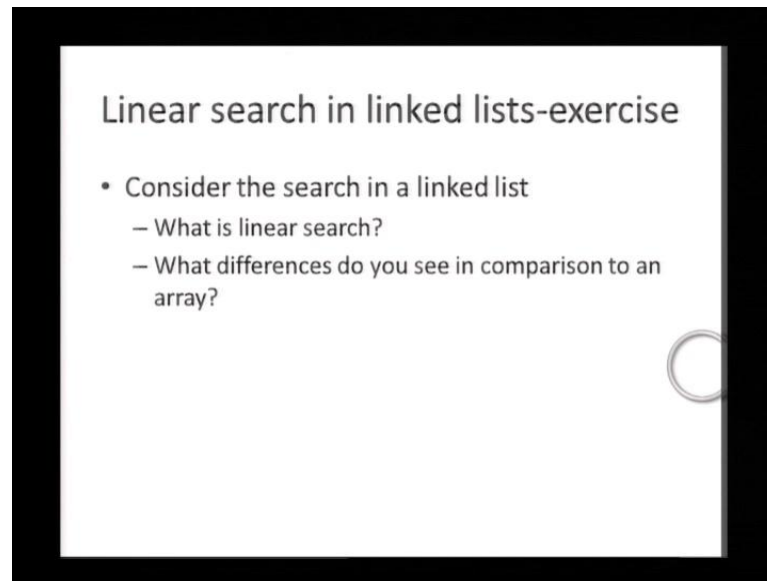
The slide is titled "Analysis" and contains the following text:

- If key is at an index i , then i comparisons are executed, one in each loop
- In the worst case, the the key is not present in the array, the loop is executed size times
- Worst case running time is $O(\text{size})$
- Note that there is also one arithmetic operation that increases index.

Can one reduce the number of comparisons and arithmetic operations?

Here is the analysis, if the key is at index i then clear the i comparisons are executed; one and every loop. So, the other words, if the key is the index 0, then one comparison is definitely executed; that is, in the first loop. In the worst case, if the key is not present in the array, then the loop is executed as many times as a size of the array which we have said. Therefore, the worst case running time of this algorithm, is order of size of the array. Note that we have one arithmetic operation also, which increases the value of the variable called index. A natural question now, is the following; can we reduce the number of comparisons in a number of arithmetic operations performed by a search algorithm to find a key in a set? And we also consider cases, in which the data in the array is, in a sorted order, and then we see if it is possible to design better algorithms, better in the sense that the number of comparisons in number of arithmetic operations is reduced. Here is an exercise, at this point of time.

(Refer Slide Time: 10:01)

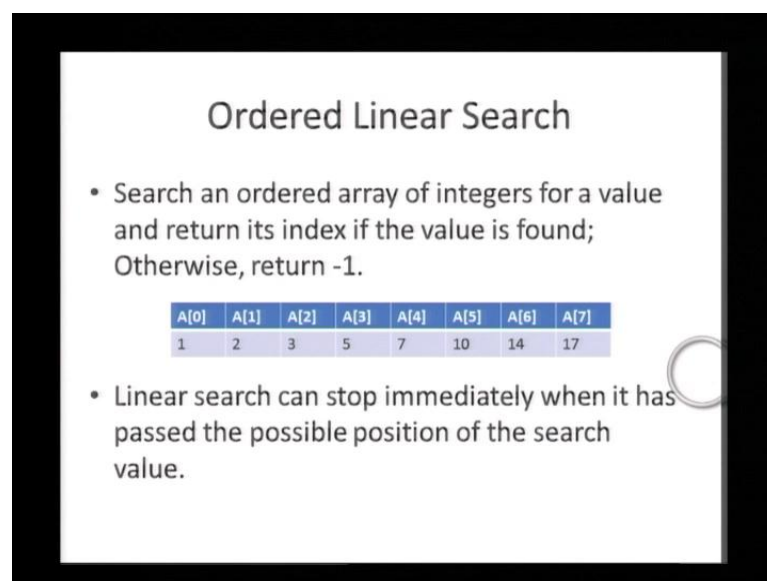


Linear search in linked lists-exercise

- Consider the search in a linked list
 - What is linear search?
 - What differences do you see in comparison to an array?

So, what happens if one does a linear search in a linked list. And what is linear search. In this case, as you can see linear search involves incrementing the array index, starting from the smallest index value to the largest possible index value in the array, and searching or comparing for the presents of a key. So, it is important as an exercise to understand, what linear search in a linked list is.

(Refer Slide Time: 10:38)



Ordered Linear Search

- Search an ordered array of integers for a value and return its index if the value is found; Otherwise, return -1.

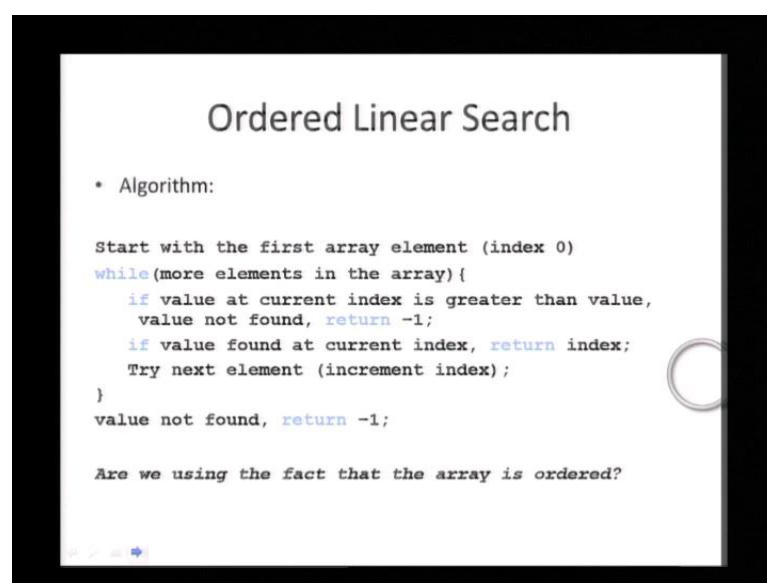
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	5	7	10	14	17

- Linear search can stop immediately when it has passed the possible position of the search value.

That is an exercise, and let us move ahead, and explore this question of, what order linear searches. In other words what is linear search when the data is ordered in the given array. In other words the array contains the data elements in say sorted order. Let us say in this case as you can see it is in ascending order, and how much time does it take, or how good in algorithm can be design, to be able to find a target key in this particular array. And of course, if the key is not found in the array, we should return a value minus one, and we make assumptions as we have been making so far that all the data items are in the range, sorry all the indices are at least as large as 0.

So, one of the properties of ordered linear search, is that linear search can stop immediately, when it has passed the possible position of the search value. For example, if you see the slide, if the queried value is the value 8, then one can perform a linear search up to the value 5, up to the value ten which is found in the location index by the number 5; that is the element a of 5 in this array a, and we find the ten is larger than the queried value 8, and we already know that the array sorted in ascending order. Therefore, we are not going to ever find 8 after the array index 5, because 10 is the value which is sitting in that location. So, this is one way in which we can use the fact that, the array, the data elements in the array are in sorted order.

(Refer Slide Time: 12:30)



Ordered Linear Search

- Algorithm:

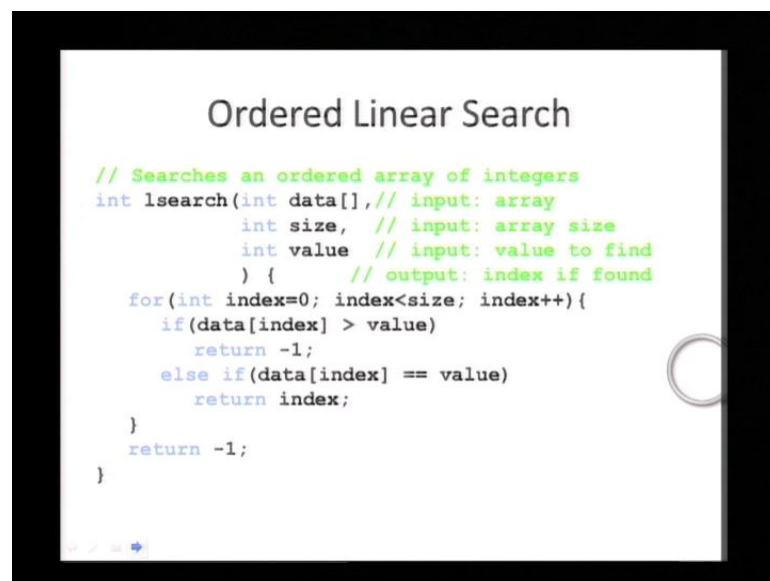
Start with the first array element (index 0)

```
while (more elements in the array) {  
    if value at current index is greater than value,  
        value not found, return -1;  
    if value found at current index, return index;  
    Try next element (increment index);  
}  
value not found, return -1;
```

Are we using the fact that the array is ordered?

So, this is exactly the algorithm that is implemented. Observe that there is one another check, which is there, which is run first inside the while loop. If the value is, at the current indexes greater, then the value that we are searching for, then the value will not be found, and you can return a minus one immediately. Now while this is one way of using the fact that, the elements of the array are ordered. In this case in ascending order, and this is indeed the c code for this.

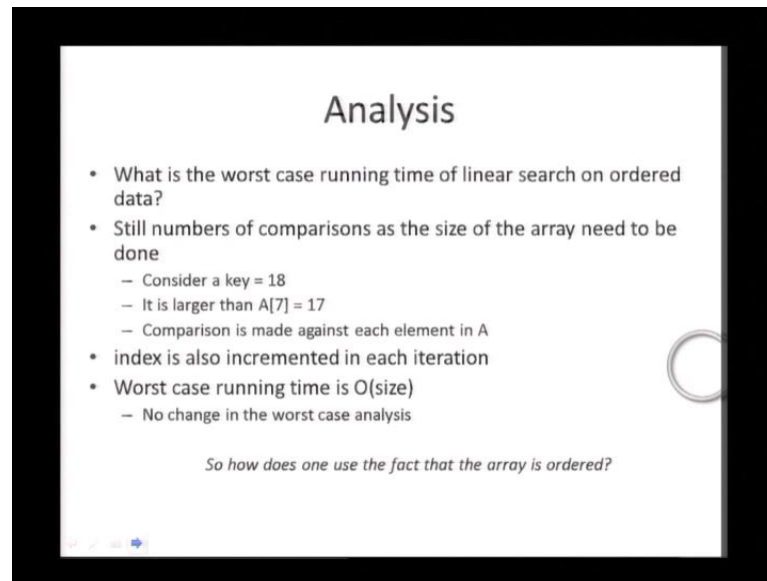
(Refer Slide Time: 13:04)



```
Ordered Linear Search

// Searches an ordered array of integers
int lsearch(int data[], // input: array
            int size, // input: array size
            int value // input: value to find
            ) { // output: index if found
    for(int index=0; index<size; index++){
        if(data[index] > value)
            return -1;
        else if(data[index] == value)
            return index;
    }
    return -1;
}
```

(Refer Slide Time: 13:14)



The slide is titled "Analysis" and contains the following content:

- What is the worst case running time of linear search on ordered data?
- Still numbers of comparisons as the size of the array need to be done
 - Consider a key = 18
 - It is larger than $A[7] = 17$
 - Comparison is made against each element in A
- index is also incremented in each iteration
- Worst case running time is $O(\text{size})$
 - No change in the worst case analysis

So how does one use the fact that the array is ordered?

Let us perform an analysis, what is the worst case running time of linear search on ordered data. In the worst case, as we can construct by an example, no matter what the array is. If one considered the target key to be a value which is larger than the element, which is present in the largest index. In this example consider the key 18. 18 is larger than 17 which is a value, which is present in the index 7. Therefore, an execution of linear search to look for 18 in the array, will compare 18 with each of the 8 elements which are present in the array, and it is a same for loop as we have seen in the previous slide. Therefore, there is an arithmetic operation every iteration in the worst case, and there is also a comparison that happens in the worst case. Therefore, in the worst case, the running time is, the order of the number of elements in the array. Therefore, there is really no change in the worst case analysis. Therefore, how does when use the fact that the array is ordered, to get better algorithms? Is it possible at all?

(Refer Slide Time: 14:30)

Binary Search

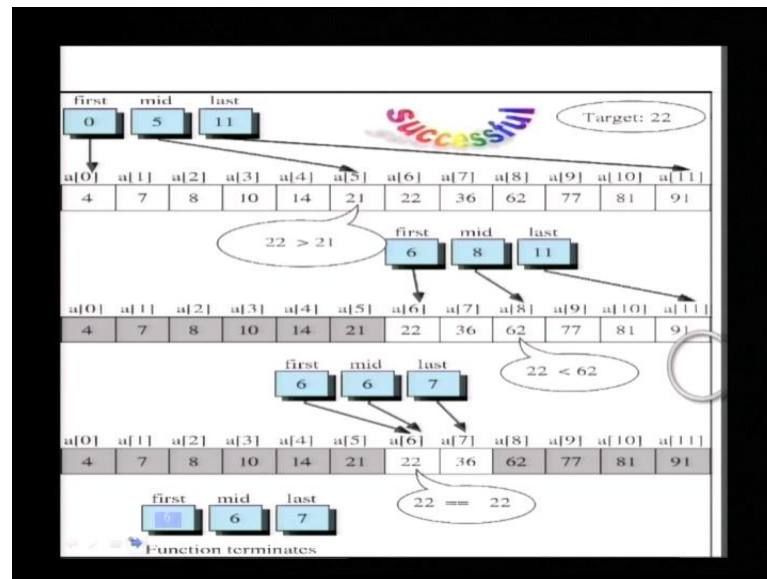
- Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	5	7	10	14	17

- Binary search skips over parts of the array if the search value cannot possibly be there.
- If key is 18, check $A[3] = 5$. It is smaller so need to search only among $A[4], A[5], A[6], A[7]$. Search region has reduced by $1/2$

That is the focus of the next search algorithm that we explore, which is very well known as the binary search approach, and we will see why this is called binary search approach. And here the most important principle, is that the search key does not have to search, or does not have to be compared with every element in the array. In other words, by making certain comparisons, we can deliciously discard, certain parts of the array from the effort that we have to put into compare the given key with the elements. For example, if the given key is 18. If we end up checking, comparing 18 with the element 5 which is present at the array location three. If 18 is present in the array, then it could be present only among the indices 4 5 6 and 7. And we can see that, the search region is kind of reduced by half or approximately by half. This is exactly what we are going to encode into a alternate procedure. And let us look at one run of this algorithm.

(Refer Slide Time: 15:44)

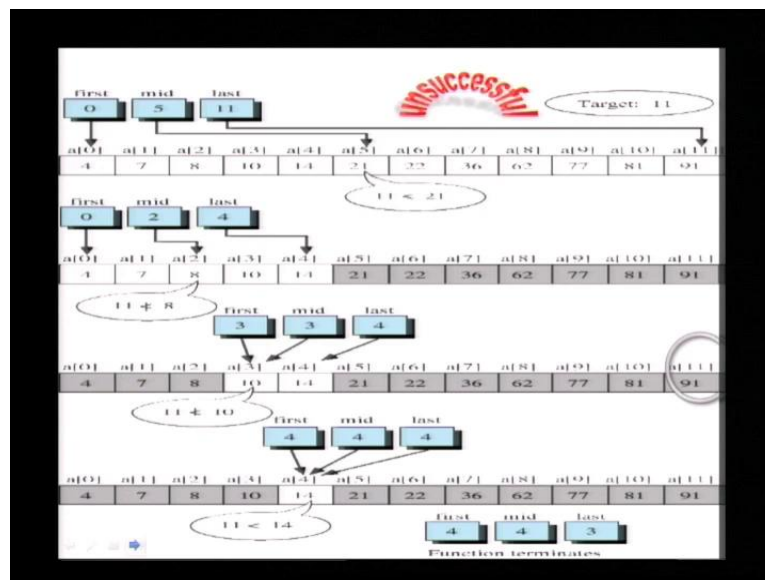


So, a is the array here, and the target key is 22 and there are 12 elements in this array. As you can see 22 is present in this array, it is in the array location index by the value is 6. So, the algorithm is very simple. It keeps track of three values which are called; first, mid, and last. First and last are extremely important. They keep track of the sub array that we want to search. The sub array that we want to search is, in this example, the unshaded part; the shaded part, is the part that we do not want to search. So, let see this run of this algorithm. Initially, first is the value 0, last is the largest array index which is 11, and mid is the midpoint, which is the first plus last divided by 2, and we take the floor of the division. In this case the floor would be 5. So, 11 plus 0 divided by 2 is 5 and a half, and we take the value 5. A comparison of the given key is made with the data item which is located at the array index 5, which is 21, and 22 is greater than 21, and because the array is sorted in ascending order. It is clear that 22 must be present, only in the array indices 6 through 11, and definitely is not present in the array indices 0 through 5.

The array indices 0 through 5 are now shaded in grey, and first and last are now used to encode, the first and last index values of the relevant part of this array a , which is now 6 to 11 and the midpoint is now. You can do the calculation is 8 is 6 plus 11 by 2 which is 17 by 2 is floor of it is 8. And the algorithm repeats this step of comparison, comparing 22 with 62 which is a value in the array index 8. And of course, 22 is smaller than 62 and

therefore, 22 cannot be present in the indices 9 10 and 11. It is not present in the array index 8, as our comparison shown. Therefore, it can only be present among the array indices 6 and 7; that is among the array indices first and mid minus 1. This is most important thing that it is present in the array indices first and mid minus 1 in this case. And as you can see now, first and last have become 6 and 7 respectively and mid is now 6; 22 is successfully found and the algorithm terminates reporting the index of the location where 22 has been found which in this case is 6.

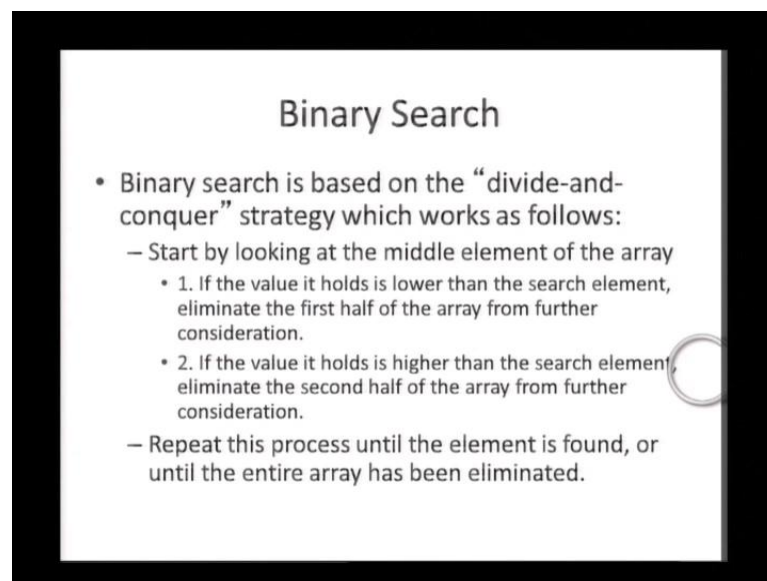
(Refer Slide Time: 18:55)



Why do we see the condition and which this algorithm exits and reports that the key is not present. In this case, the target value is 11, and as you can see 11 is not present in the array, and again the gray shaded part is the irrelevant part of the array for the search algorithm, and the unshaded part is the relevant part. First and last as usual end code have, do encode the relevant part of the array, keeps track of the indices, initially it is 0 and 11. A comparison is with mid, 11 is smaller than 21; therefore if 11 is to be found, it can be found only to the left of the index 5 that is among the array indices 0 to 4, which is now captured by the modified value of last, which has now become 4. Mid is now recalculated in the next iteration to be 2, a comparison is made with a value which is sitting in the array index 2 which is 8. 8 indeed is not larger than 11; therefore, 11 has to be to a right of 8 if it is at all present in the array.

Therefore, now in this case, 11 should be present if at all, in the array indices last and mid minus 1. Sorry last and mid plus 1. So, mid was the value 2, and now observe that the value of first is now 3; that is, mid plus 1 and last. As you can see that in one more query it is discovered that, the search key 11 is not present in the array, and at termination condition you can see that, first has become larger than last, and this is a termination condition for the algorithm. So, this is very important the termination condition for the algorithm is, when first exceeds the value of last. At which point return you can report that the key has not been found, and the key is found, the value of mid. The location where the key has been found will be reported as a return value of the algorithm. These are the two invariants, which precisely map the presence or absence of a key from the array.

(Refer Slide Time: 21:18)



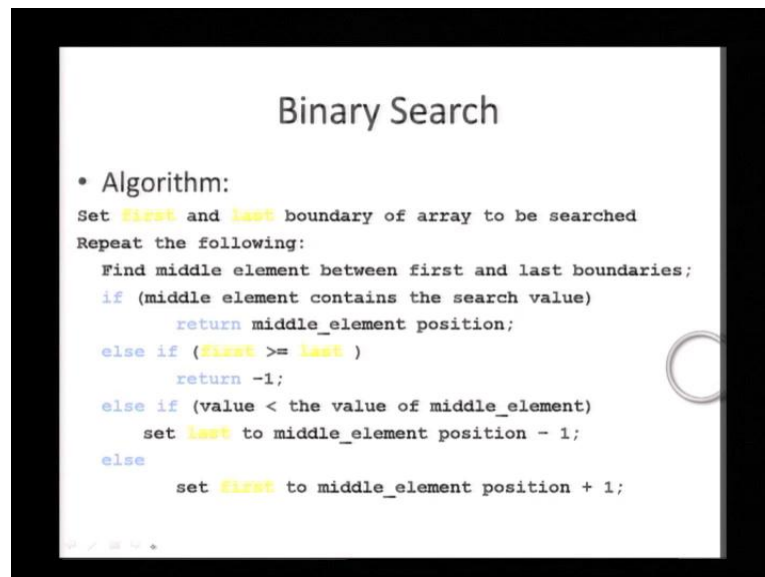
Binary Search

- Binary search is based on the “divide-and-conquer” strategy which works as follows:
 - Start by looking at the middle element of the array
 - 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.
 - 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.
 - Repeat this process until the element is found, or until the entire array has been eliminated.

So, in generic terms, binary search is a paradigm of solving the problem by what is called a divide and conquer strategy. In this divide and conquer strategy, the search space, in this case the array is repeatedly divided into smaller and smaller portions with the guarantee that, the search value would be present in the region that is being searched at every level or in every iteration. As we have seen in with every comparison, the size of the array has been reduced by a factor of 2. In other words the array size becomes smaller and smaller is halved in every iteration, and this kind of gives us a clear handle on, an

understanding of the binary search algorithm.

(Refer Slide Time: 22:14)

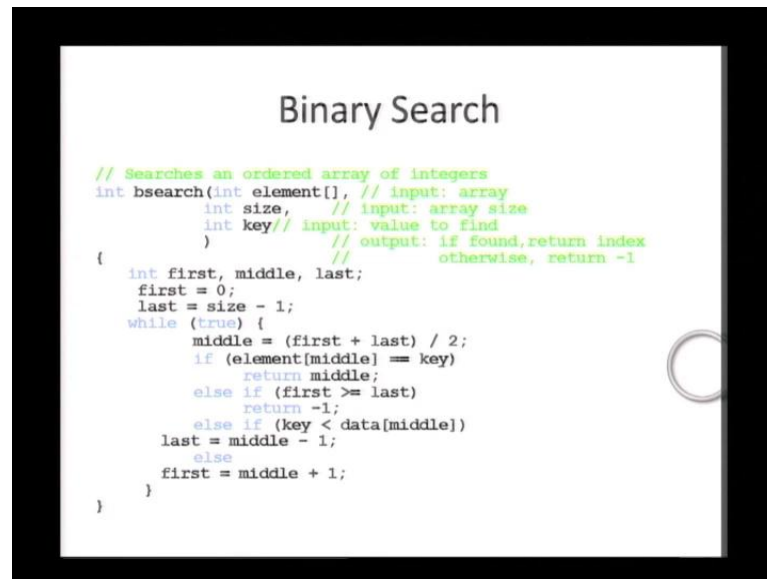


Binary Search

- Algorithm:
Set **first** and **last** boundary of array to be searched
Repeat the following:
 Find middle element between first and last boundaries;
 if (middle element contains the search value)
 return middle_element position;
 else if (**first** >= **last**)
 return -1;
 else if (value < the value of middle_element)
 set **last** to middle_element position - 1;
 else
 set **first** to middle_element position + 1;

Let us see the pseudo code. And the pseudo code is very important, because it tells us how the first mid and last have to be modified. So, initially first and last are set to take the values of the boundary of the array. The mid value is calculated. The search value or the key is compared with the element in the mid value. If it is in the present the mid element is returned as a value. If it is not present, a comparison is mid whether the value was. If indeed the mid value does not contain, but first is more than last then the value is returned. First is at least as large as last then the return value is minus 1, same that the key has not been found. Otherwise if the value is smaller than that in the middle element, last is now make to take the index which is mid element position minus 1; and if value is larger, than first is taken to be the mid element plus 1. That is, in other words one would visualize the search to be moving either to the left or to the right of the midpoint in the array, provided the key is not found.

(Refer Slide Time: 23:41)

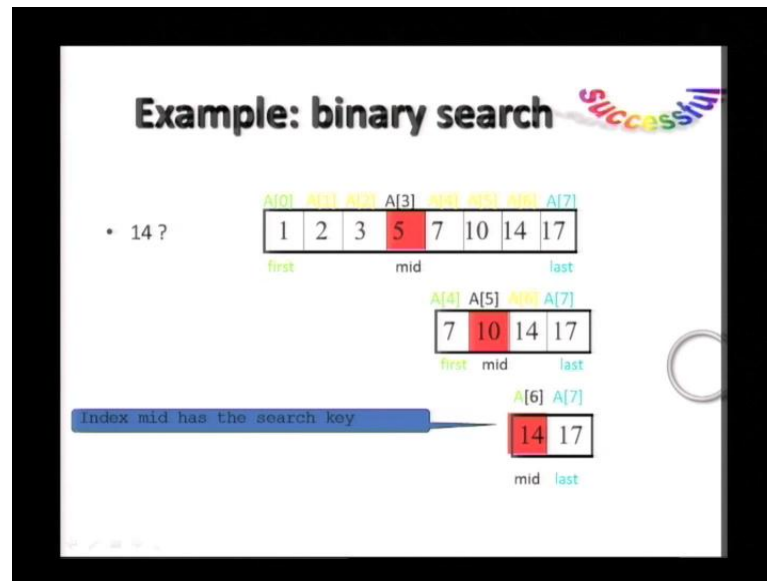


```
Binary Search

// Searches an ordered array of integers
int bsearch(int element[], // input: array
            int size,      // input: array size
            int key)       // input: value to find
{
    // output: if found, return index
    //          otherwise, return -1
    int first, middle, last;
    first = 0;
    last = size - 1;
    while (true) {
        middle = (first + last) / 2;
        if (element[middle] == key)
            return middle;
        else if (first >= last)
            return -1;
        else if (key < element[middle])
            last = middle - 1;
        else
            first = middle + 1;
    }
}
```

Here is the pseudo code for this function which is binary search. It is actually cut and paste from a c program. Right now the arguments are as usual element an array which contains sorted set of elements. The size of the array is given, and key is also given as part of the input. And first is taken to be the value 0, and last is taken to be the value size minus 1. And you can check the integer division which is middle takes the value of first plus last divided by 2, and whatever comparisons we have discussed so far are made. The value of this particular piece of code is that it gives us an idea as to the number of operations there are perform in a run of this particular algorithm.

(Refer Slide Time: 24:38)



So, let us use go through the example of binary search in this case where the array has the elements 8 elements in ascending order and the search key is 14. 14 at present in the array index 6, and one can see the conditions and which the algorithm exits. So, the first value that is searched, is the array index 3. And array index by the value 3 which is 0 plus 7 divided by 2 on the floor of it gives you the value 3, and the search value is 14 and clearly 14 is to the right of 5, and first is the one whose value is updated to take the value 4; last remains unchanged. In a next iteration mid look becomes 5 and 7 plus 4 divided by 2, and this search succeeds by computing the mid value 6. As you can see the order of the elements in the array, guides the choice of the array locations which are propped by the algorithm.

(Refer Slide Time: 25:49)

UNSUCCESSFUL

Example: binary search

• 8?

A[1]	A[2]	A[3]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	5	7	10	14	17
	first		mid				last

A[4]	A[5]	A[6]	A[7]
7	10	14	17
	first	mid	last

A[4]
7
f m l

First and last are the same and middle does not contain the key. Failed search.

So, here is an unsuccessful binary search, and the unsuccessful binary search can terminate in to conditions where first and last are the same, and the middle value does not contain the search key. This is very important. There are three cases here; the first case we have already seen, which is successful search. Here is an unsuccessful search, where the exit condition is because the key is not present, and first and last of the same value. right here 8 is a search key, and it is clear that, the sequence of searchers finally, queries the element 4, which has the value 7, but first and last take the same value at this point of time, and because 7 is not equal to 8 and there is nothing else to be searched, the search returns a failure.

(Refer Slide Time: 26:42)

Example: binary search unsuccessful

• 4?

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	5	7	10	14	17

first mid last

A[0]	A[1]	A[2]
1	2	3

first mid last

A[2]
3

f m l

Here first equals last and mid does not contain the key. Fail.

Here is a case where first exceeds the value last, in the next iteration, and the key is not present here. So, this can also be checked.

(Refer Slide Time: 27:03)

Binary Search Analysis

$T(n)$ → Array size
↳ Time

$n = 2^k$

A[0]	A[1]
1	3

$T(n) = O(\log n)$

$T(n)$ for $n=2$
 $T(2) = 2$

$A[0] \dots A[2^k-1]$
 $A[2^{k-1}] \dots A[2^k-1]$

$\lfloor \frac{2^k-1}{2} \rfloor = 2^{k-1} - 1$

$T(n) = T(\frac{n}{2}) + 1$

The focus now is to analyse binary search, which is the algorithm that gives analysis we postponed, to the end. Let us look at this analysis of binary search, where our goal is to

estimate the total number of comparisons of the binary search algorithm. So, let us write the formula for binary search by t of n ; where t stands for time, and n stands for the array size. For the moment let us assume that n is a power of 2. This simplifies the analysis, and therefore, we use n to be a power of 2. Let us look at the case when t of n , for n is equal to 2. So, let us just focused on the array, it just has two elements, and let us assume that the elements are 1 and 3 in sorted order, and let us count the number of steps that it takes to check the query element. So, this is the array index with 0. This array index by 1. And it is clear that a queried element, can be resolved for whether it is present or not in the array, in two comparisons. The maximum of two comparisons. So, for example, what could happen is, that if the search is for a value half, then the array index which should be compared is, half would be compared with 1, half is compared with 1, and then the first mid and last are updated, and half will not be found, so that takes only one comparison. If the value is three then in two comparisons it could be found. And if the value is larger than 3 then again it would be found, that if the value is absent in at most two comparisons. Therefore, t of n for n is equal to 2 is written as t of 2 which is just two comparisons.

So, let us considered the case for n is a power of 2, and let us look at the array indices. The array indices are a of 0; that is, sorry the indices are 0 to $2^k - 1$, and the value of the array index; that is taken to be relevant, is the value $2^k - 1$ divided by 2, which will be $2^k - 1$ minus 1; that is the floor of this is $2^k - 1$ minus 1, which would be the value that would be compared. Therefore, as you can see, after the first comparison the relevant part of the array would be the range a of 0, to a of $2^k - 1$ minus 1, or a of $2^k - 1$, to a of $2^k - 1$ there are only two possibilities, for the rest of the search space, after one comparison. So after the first comparison with a given key, these are the two range of values. As you can see that the range of values is now down by a factor of 2.

We can write this. We can encode this using the following recurrence, which is t of n is t of n by 2. The time taken to search for the key and an array of size n by 2 plus 1 and the boundary condition is given by t of 2 is equal to 2, and the solution for this recurrence is t of n is order of $\log n$. Indeed it is $\log n$ to the base 2, but the constants in the order take care of it. So, this is the analysis of binary search, and observe that we have use the fact

of the array is sorted, to come up with an algorithm with just use as order of $\log n$ comparisons, as suppose to ordered search which in the worst case, was using linear number of comparisons, or order of the size of array number of comparisons. And in the case when the array is unordered, we were already using the linear number of comparisons which are unavoidable. So, with this we stop this discussion on search and we will continue on the next lecture.