Binary Search

→Slide1:

Hi everyone, and welcome back to Algorismo!

Today, we're going to dive into a powerful and efficient algorithm, known as Binary Search. Unlike Linear Search, which checks every element one by one, Binary Search cuts the search space in half at every step, making it a much faster approach—especially for large, sorted datasets.

We'll explore how this algorithm works, why it's so efficient, and how you can implement it in Python.

So, let's get started!

[Next]

→Slide2:

Before we dive into Binary Search, let's quickly review what we covered with Linear Search. In the best case, Linear Search finds the target on the very first try, making it extremely efficient with a time complexity of O(1). **[Next]**

However, in the worst case, the algorithm has to go through every element in the list before finding the target or determining that it's not in the list. **[Next]**

This gives us a time complexity of O(n), which can become quite slow as the list size grows.

[Next]

The average case is when we find the target somewhere in the middle. **[Next]**

Although, the number of comparisons can be regarded as n/2 on average, however, ignoring constants, we still get O(n) as the complexity for the average case. **[Next]**

These scenarios highlight the limitations of Linear Search, particularly in larger datasets, setting the stage for why we need more efficient algorithms like Binary Search.

[Next]

→Slide3:

So Linear Search checks each element of a list one by one. **[Next]**

This simple approach works well for smaller datasets, but it starts to struggle with larger lists. **[Next]**

As the size of the list grows, the time it takes to complete the search increases linearly—meaning it grows in direct proportion to the size of the list. In the worst case, Linear Search will need to check every single element.**[Next]**

So what if we have a large list containing 1 Million records? **[Next]**

We will need to make 1 Million comparisons in worst case. **[Next]**

The question is, can we do any better? **[Next]**

This is where Binary Search comes in! **[Next]**

Unlike Linear Search that checks each element individually, Binary Search cuts the list in half with each step, drastically reducing the number of steps needed. **[Next]**

But there is a catch – it works only on sorted lists. **[Next]**

How efficient is binary search compared to linear search. Well, instead of taking n steps, Binary Search can complete the search in log of n steps. **[Next]**

For example, if you're searching a list of 1 million elements, Binary Search only needs about 20 steps, compared to the 1 million steps Linear Search might need. **[Next]**

So how does this work in practice? Let's explore Binary Search through a simple challenge—a number guessing game! **[Next]**

Imagine thinking of a number from a list containing 16 numbers sorted in an increasing order . **[Next]**

You have up to 4 attempts to guess it. Think you can do it? Let's find out as we dive deeper into the magic of Binary Search!

[Next]

→Slide4:

Let's play this game using Binary Search to guess a number in a sorted list. **[Next]**

Here's our list, with 16 elements **[Next]** indexed from 0 to 15 **[Next]**

The target number we're trying to guess is 12. **[Next]**

Here's how Binary Search narrows down the possibilities step-by-step:

We start by calculating the middle index by adding the first and last index and doing integer division by 2. By integer division we mean to truncate the decimal part without rounding it off. So this gives us 7. **[Next]**

The element at index 7 is 35. **[Next]**

Since 12 is less than 35, we know 12 must be on the left side of the list. And why should it be to the left? Yes, because list is sorted. So, we discard everything from index current mid index, i.e., 7 to 15. Now our range is from index 0 to 6. **[Next]**

We calculate the new middle index which is 3. **[Next]**

The element at index 3 is 18. **[Next]**

Since 12 is less than 18, we again move to the left half of the remaining list. Now our range becomes index 0 to 2. **[Next]**

With our range now from index 0 to 2, we calculate the middle index once more which is 1. **[Next]**

The number at index 1 is 7. **[Next]**

Since 12 is greater than 7, we narrow our search to the right side of the current range discarding the left part of the list up to current mid index. Our new range becomes index 2 only. **[Next]**

Now we have a single index left, from 2 to 2. **[Next]**

We calculate the middle index. **[Next]**

The element at index 2 is 12, **[Next]**

which matches our target! And there we have it! We found 12 at index 2 using Binary Search, by dividing our search space in half with each step. **[Next]**

Starting with a list of size 16 **[Next]**

This efficient method took only four steps to find the number. **[Next]**

You may recall that the number of comparisons in linear search are directly proportional to the input size. **[Next]**

If the there is one element in the list, **[Next]** it will take one comparison, **[Next]**

likely there will be 10 comparisons for 10 elements. **[Next]**

In general, we can say that for a list of size n, **[Next]**

linear search has to do n comparison in worst case. **[Next]**

This is expressed as O(n) and is also regarded as linear time complexity. **[Next]**

How about binary search. **[Next]**

We just witnessed that for a list of size 16, **[Next]** we had to do 4 comparisons. **[Next]**

Now, what is the generalization? What will be the number of comparisons for a list of size n? Let’s begin with the straight answer and develop our intuition onwards. **[Next]**

For a list of size n, the number of comparisons that binary search will need to do are O(lgn) in worst case – Well, how? Let’s break down why Binary Search has a complexity of O(logn). **[Next]**

What is n here? **[Next]**

N is the size of our list. For our example, it is 16. **[Next]**

And how many comparisons or steps? **[Next]**

In the game we just played, it took us 4 comparisons to find the target number 12 in a list of 16 elements. **[Next]**

Now , what is log n or log 16 to be more specific? **[Next]**

We cannot calculate the log until we know base of log. **[Next]**

Binary Search is based on dividing the list by 2 repeatedly, so we use base 2 for logarithms. **[Next]**

So, what is log 16 with base 2? **[Next]**

This means, "What power of the base (i.e., 2) will give us the number, 16?"

What power of base (2) will give 16? **[Next]**

It is 4. **[Next]**

Because 2 raised to power 4 is 16, so the power is 4. This means we can halve the list 4 times to narrow down to a single element in a list of size 16. **[Next]**

Therefore, log n base 2 or specifically log 16 base 2 is 4.

This tells us that in a list of 16 items, Binary Search will take at most 4 comparisons in the worst case, as each step divides the list by 2. This is why the complexity is O(log n), or in this case, 4 when n or the input size is 16. **[Next]**

You'll often see L O G written as L G, especially when referring to base 2. These are just notational differences, with L G commonly used in computer science. **[Next]**

One last thing: in some cases, you may see $\log_2 n + 1$ instead of log n for the maximum comparisons in binary search, because it rounds up the divisions if there's an extra element left. Since 1 is a constant, we can ignore it anyway.

In summary, for a list of size 16, Binary Search takes at most 4 steps to find an element because each step reduces the search space by half. This pattern holds as n grows, so Binary Search has a logarithmic complexity— i.e., O(log n). **[Next]**

[Next]

→**Slide5:**

When comparing Linear Search and Binary Search, it's helpful to look at how each algorithm handles input sizes that are powers of two, since Binary Search divides the list in half with every step. So we are talking about the power of two only for the sake of ease of computation and representation. **[Next]**

As you can see in the first row, when the input size is 8, the linear search takes 8 steps, whereas, binary search takes log 8, i.e., 3 steps only. **[Next]**

As the input size grows to 16, 32, or 64, the step count for linear search remains in a linear proportion to the growth in input size, whereas, the same for binary search is growing much slower, by a logarithmic factor only. **[Next]**

This difference continues to grow, for example, searching through a list of 1,024 elements requires 1,024 steps with Linear Search. On the other hand, Binary Search works in 10

steps—that's because $\log_2$ of 1,024 with base 2 is 10. **[Next]**

Finally, in the last row, when the input size is over a million, binary search takes only 20 steps to complete the search compared to linear search which will require over a million steps in the worst case. So the Binary Search is far more efficient as the input size grows. **[Next]**

The plots show the growth in the steps required by both linear and binary search along the y-axis as the input size grows along the x-axis. **[Next]**

You can see, as the input size grows, **[Next]** the curve of linear search keeps getting steeper **[Next]** and the same for binary search gets flat along the x-axis. **[Next]**

However, binary search only works if data is sorted in increasing or decreasing order. So, what if the data is not sorted? Most likely, it will not be possible to benefit from binary search.

[Next]

→**Slide6:**

Now, let's develop the Binary Search algorithm step-by-step with a real example. **[Next]**

The inputs are a sorted list of numbers and a target value to search for. **[Next]**

Here is the example sorted list of 8 numbers and a target value of 13. Our goal is to find this target value 13 in the list. **[Next]**

The output of the algorithm will be the index of the target number if found in the input list or -1. **[Next]**

In this specific case, we expect -1 as output since 13 is not listed in the input listed. **[Next]**

Let's begin the step-by-step process.

Set low to 0 (the index of the first element) and high to 7 (the index of the last element). High is set to 7 because the length of the list is 8 but the indexes starts from 0 and not 1. **[Next]**

The algorithm will continue until there are elements to process in the list. This is indicated as a general condition to continue until low is less than or equal to high, or simply until low and high do not cross each other. **[Next]**

Next, we calculate the middle index by adding low and high and dividing by 2. **[Next]**

This turns out to be 3.5 but integer division truncates the decimal and we get 3 only. **[Next]**

The element at mid index is 18. **[Next]**

We need to do the first comparison between the element at the current mid-index and the target. If the current element and the target match, we return the index right here and the

search is complete. **[Next]**

However, the element at index 3 is 18, which is not equal to our target 13, so we move on. **[Next]**

Next, we check if the element at the mid-index is larger than the target. **[Next]**

Since 18 is larger than 13, **[Next]** we update high to mid-1, which is 2. **[Next]**

This eliminates the right half of the list starting at index 3. Now our search range is from index 0 to 2. **[Next]**

Let's calculate mid for the second time now. With low set to 0 and high set to 2, the new middle index is 1. **[Next]**

The element at mid is 7. **[Next]**

7 is not equal to 13. **[Next]**

The second case does not apply either as 7 is not greater than 13. **[Next]**

Therefore, we must handle the third case when the element at the current middle index is less than the target. This simple else condition handles that case and we simply update low to mid + 1, which is 2. **[Next]**

This eliminates the left part of the list. Now our search range is narrowed to a single index, 2. **[Next]**

Now we calculate the mid for the third time, with both low and high set to 2. **[Next]**

The middle index now turns out to be 2. **[Next]**

The element at the current mid is 12. **[Next]**

12 is neither equal **[Next]** nor greater than the target. **[Next]**

Since 12 is still less than our target 13. **[Next]**

We update low to mid + 1, which is 3. **[Next]**

In the final iteration the condition in the while loop is false because now, low is 3 and high is 2, meaning low is greater than high. **[Next]**

This signals that the target 13 is not in the list, and to handle this we need to add one more statement after the end of the loop that returns -1. **[Next]**

How many iterations did it take? Only 3, for a list of size 8. And yes, the log of 8 with base is exactly 3 because 2 raised to power 3 is 8. **[Next]**

Let's summarise:

Through each step, Binary Search systematically narrowed down the search range, but we confirmed that target 13 isn't present in the list. This process highlights how Binary Search efficiently checks only the necessary elements, halving the search space with each iteration.

[Next]

→Slide7:

Now, let's translate the Binary Search algorithm into a Python function:

We define a function called binary_search that takes two inputs: a sorted list A and the target value. **[Next]**

The initial boundaries, low and high, are set to the first and last indices of the list. **[Next]**

The while loop runs as long as low is less than or equal to high. **[Next]**

Inside the loop, we compute the middle index and compare the element at that index to the target. **[Next]**

If the middle element equals the target, we return the index. **[Next]**

If the middle element is greater than the target, we reduce the search space to the left half. **[Next]**

If it's smaller, we shift the search space to the right half. **[Next]**

If the target is not found after the loop, the function returns -1. **[Next]**

Finally, we demonstrate the function with a list of numbers and a target, **[Next]**

showing how to call it **[Next]** and handle the result in the same way **[Next]** as we did for linear search. **[Next]**

This Python code mirrors the algorithm step-by-step, making Binary Search efficient and easy to implement. Let’s demonstrate its execution using Python Tutor.

[Next] **STOP**

→Slide9:

When solving problems, we often use iteration or loop to perform repeated computations. We have used this approach in all three algorithms covered so far. To recall, the three algorithms we have covered so far are, find max, linear search, and binary search which we are discussing right now. Iteration is the most common approach to solving problems. It uses loops to execute a set of instructions repeatedly. This method is efficient in terms of memory and is great for large datasets.

But there's an alternative called recursion. Instead of using a loop, the function calls itself, breaking the problem down into smaller, more manageable parts. It might sound too abstract for now. So let’s make it simple with an example. **[Next]**

You may have known the factorial problem. The factorial of a number is the product of all positive numbers from 1 to n. **[Next]**

Mathematically, the factorial of n is n multiplied by n - 1, multiplied by n - 2, and so on

multiplied by 1. **[Next]**

To make it more concrete, the factorial of 4 is 4 multiplied by 3, multiplied by 2, multiplied by 1. The product is 24. **[Next]**

We can write a simple function in Python, named factorial that takes as input the number n, computes, and returns its factorial. **[Next]**

To do so, we initialize a variable result to 1 that will finally hold the value of the factorial of n. **[Next]**

We need to loop over the range of numbers from 1 to n to compute the factorial. You see n + 1 in the range because the loop will stop before n + 1, i.e., n exactly. **[Next]**

We keep multiplying every number i, starting from 1 to n with the result variable accumulating the desired product. **[Next]**

As the loop ends, the result variables return the desired value of the factorial of the input number n. **[Next]**

The loop in this function starts at 1 and increments until it reaches n. **[Next]**

We can do these multiplications in the reverse direction as well. In this case, the iterations start at n and decrement down to 1. The effect of both loops or iterations is the same. **[Next]**

If you remember, we intended to define recursion as an alternative design to iteration or loop. Let's get on with it and define factorial in a different way. So what is the factorial of n? **[Next]**

Well, it is n multiplied by a factorial of n - 1. **[Next]**

Okay, so n is simple, but what is the factorial of n - 1 though? **[Next]**

Well, the factorial of n - 1 is n - 1 multiplied by the factorial of n - 2. **[Next]**

It might seem a little weird. But there is a pattern here. For eah next definition, we take the input from the previous definition out and multiply it with the same definition with input number one less than the previous. If we continue like this, where will we end? Yes, we need to define an explicit end to this process. And we will end this process with input decrements down to 1. So once the input is 1, we simply return 1 instead of calling the same function again. **[Next]**

Let's make it clear with an example. What is factorial of 4? **[Next]**

Following the same recursive definition, it is 4 multiplied by factorial of 4-1 or 3. **[Next]**

And what if factorial of 3? It is 3 multiplied by factorial of 3-1 or 2. **[Next]**

And what if factorial of 2 then? It is 2 multiplied by factorial of 2-1 or 1. **[Next]**

Finally, what if factorial of 1? Well, that's where we define our explicit stop indicating that we are no more continuing to redefine factorial what is input is decreased down to 1 and returns 1 as the answer. **[Next]**

As long as we are recurring, or calling the factorial function repeatedly with smaller input, it is called the recursive case or recursive step. **[Next]**

Whereas, when we opt for a simple answer instead of calling the function again, it is called the base case. So the base case is where recursion stops actually. **[Next]**

Let's unroll how the factorial is computed using this recursive process by propagating backward. **[Next]**

We know when we call factorial with 1 as input, it simply returns 1. **[Next]**

So we replace the call factorial of 1 with simple 1. Multiplying this 1 with 2 results in 2.

[Next]

Next, we unroll factorial of 2 which was defined as 2 multiplied by factorial of 1. **[Next]**

This is already evaluated to 2 on line below. So we replace factorial of 2 with simple 2. Multiplying this 2 with 3 results in 6.

[Next]

Next, we focus factorial of 3 which was defined as 3 multiplied by factorial of 2. **[Next]**

This is already evaluated to 6 on line below. So we replace factorial of 3 with simple 6. Multiplying this 6 with 4 results in 24.

[Next]

Finally, we need to know the factorial of 4 which was defined as 4 multiplied by factorial of 3.

[Next]

This is already evaluated to 24 on line below. And there is no further step we need to carry out. So 24 is the final answer.

[Next]

With that, it's now time to write this recursive logic in a python function. Let's define a function recursive_factorial with n as input. **[Next]**

We first translate the base case into python syntax. That is, if n is 1, return 1 as answer.

[Next]

You may notice how this simple logic represents the base case from the logic we developed.

[Next]

Next, we develop the recursive case. The case that will work when n is not 1. So if we are giving 1 as an answer in base case, the answer in recursive case will be n multiplied by recursive factorial of n - 1. This means, whatever n is provided as input, the answer will be

that n multiplied by the factorial of the number 1 lesser than n. **[Next]**
You may notice that this line correctly represents the recursive case in general.
Recursion might sound a little overwhelming and abstract to begin with. But as you spend time with it, you seem to appreciate and enjoy the simplicity of the design and the luxury of solving a problem in terms of the problem itself with smaller input.
[Next]

→ **Slide10:**

Let’s sum up the basics of recursions once more.
When solving problems, we often start with iteration, the most common approach. **[Next]**
It uses loops to execute a set of instructions repeatedly. **[Next]**
This method is efficient in terms of memory usage and is great for large datasets. **[Next]**
But there's an alternative design approach called recursion. Instead of using a loop to repeat the steps, the function calls itself, breaking the problem down into smaller, more manageable parts. Each recursive call works on a smaller version of the same problem until it reaches a base case—the simplest version of the problem that can be solved directly.
[Next]
While recursion can make the code more elegant and easier to follow, **[Next]**
it also has its challenges, such as potential memory overhead and stack depth issues.
[Next]
For example, in Binary Search, we can use either iteration or recursion. In the iterative version, we use a loop to divide the search space repeatedly. In the recursive version, the function calls itself to divide the search space until the target is found.
Both approaches can solve the same problems, but depending on the context, recursion can sometimes offer a more elegant solution.
[Next]

→ **Slide11:**

Let’s take the iterative version of our binary search function that we just developed and use it to create a recursive version of the same. **[Next]**
In addition to writing the binary search function, we also wrote this code to make use of the function including the inputs, function call, and handling the output. **[Next]**
Here is the first change we intend to do. Instead of initializing the values of low and high inside the function, **[Next]**
We provide these values as input to the function directly. You can see, we have added 0 and length of A - 1 as parameters to the function call. **[Next]**

We must receive these parameters in the function definition too. Let's name these arguments the same, low and high here, **[Next]**

Low receive 0, and high receives length of the input list - 1. **[Next]**

With this change, let's write code to call the function for the recursive version that we will write next. Not surprisingly, this is code is the same as written for the iterative version. The only change we make is in the name of the function to recursive binary search. **[Next]**

Next, we write the function header which is exactly the same as written for the iterative version of binary search but with a slight name change. **[Next]**

The while loop converts to a mere if condition because the iterations are not based on using any loop. In fact, we shall be using repetitive function calls (that is, function calling itself) to achieve the effect of iterations. **[Next]**

And the last return statement executes only if the search completes without finding the target, and returns -1 will now execute under this if statement. **[Next]**

The calculation of the mid-index remains the same. **[Next]**

The first case that returns the index of the list if and where the target element is found is also exactly the same. **[Next]**

Both these cases are the base cases for the recursion. The first base case causes the recursion to terminate when there are no more elements in the list, or say when low crosses high. The second case is when we find our target any the current mid-index.

Having written both bases cases, the cases that cause the recursion or repetition to end, we now turn towards the recursive cases, the cases that will cause iteration, or looping, by calling the function itself with varying smaller problem size, and will continue until we reach any of the base case. **[Next]**

This is the first one, when the element at the current mid index is larger than the target, or say the target is smaller than the element at the current mid index, we shift high to mid-1 because the search will now continue in the left half. We were explicitly setting high to mid-1 in the iterative version. **[Next]**

However, in the recursive version, we shall call the recursive binary search function with modified values of low and high. **[Next]**

Both these parameters are left blank for you to fill yourself in the exercise. Although it is easier to guess that the value of low will be the same as before and the value of high is clearly indicated in the iterative version.

This is where the function calls itself from within and gives the effect of iteration.

The second and last of the recursive case is when the element at the current mid-index is

smaller than the target and we drop the left half of the list by shifting low to the index next to the current mid. The parameters will be filled in the same way as in the previous recursive case. **[Next]**

Now, we have successfully written both base cases, **[Next]**

And recursive cases to complete the logic of binary search using recursion. You will find this template code and solution on our github page. **[Next]**

→ **Slide12:**

In this video, we covered a lot of ground:

First, we reviewed the best, worst, and average cases for Linear Search **[Next]**

and shifted focus to Binary Search, which is much faster due to its efficient, half-by-half search strategy. **[Next]**

We analyzed Binary Search's complexity, defined the algorithm, and implemented it step-by-step in Python. **[Next]**

Next, we introduced recursion as an alternative to iterative solutions, discussing its pros and cons, **[Next]**

and we walked through a detailed sketch of the recursive Binary Search implementation.

[Next]

For hands-on practice, visit our GitHub page—the link is in the description—and try implementing the recursive Binary Search yourself. You just need to fill in a few words to make it work. In case, it doesn't work, the solution is also provided. **[Next]**

In the next video, we'll dive into Sorting Algorithms, **[Next]**

starting with some foundational sorting methods and why they're so essential for organizing data efficiently. We hope you're excited to learn more!

[Next]

→ **Slide13:**

That's all for now. Thank you for watching and joining me on this journey through algorithms. Don't forget to check out the code examples and exercises on our GitHub page. Your feedback is important to us, so please share your thoughts and suggestions in the comments. See you next time for our lesson on a new problem as we shift our focus from searching to sorting. Happy coding!

**********END**********