Linear Search

→Slide1:

Hi Everyone!

Welcome back to Algorismo!

Today, we're going to explore Linear Search—quite literally the search we use every day. Whether it's finding your keys, looking for a name in a list, or searching through files, this fundamental algorithm is at the heart of our daily routines.

Let's dive in and see why it's so indispensable.

[Next]

→Slide2:

In our first video, we laid the foundation by discussing what algorithms are—step-by-step procedures for solving problems. We introduced a simple problem to find the student with maximum grade from the list of given grades. **[Next]**

We then discussed how to express the algorithms beginning with simple pseudocode, **[Next]**

and then writing the same using Python. **[Next]**

Then, we organized our code as find_max function. **[Next][Next]**

In the exercise we used the same find_max function **[Next]**

to explore the concept of complexity analysis, **[Next]**

where we saw how the number of steps or operations grow with the size of the input.

[Next]

→Slide3:

Let's quickly revisit the complexity analysis of the find_max function. Here's the code.

[Next]

The key part to note is the loop that checks each element in the list to determine if it's the maximum. **[Next]**

This operation's complexity is directly influenced by the input size because, for every additional element in the list, the algorithm performs one more comparison. **[Next]**

Therefore, the loop at line 3 will execute n times for a list of length n. **[Next]**

Whereas, line 1, 2, and 6 will execute only once no matter what the input size is. **[Next]**

One consideration here is how many times the lines 4 and 5 execute? Since these are part of the loop, let's for the sake of simplicity we assume they also execute n times. We will come back to formalize this concept later. **[Next]**

Summing all these steps, results in 3n + 3 as the total number of steps this algorithm will take to find the maximum number from a list of size n. **[Next]**

To simplify further, we can say that as the size of input grows, 3n + 3 is dominated by n, and the effects of constants does not change. Again, if this looks confusing, don't worry. As we move on, we shall formalize this concepts too. Putting it simply, for a list of size n, the total steps are roughly n. This is what we call a linear relationship—if the list size doubles, the number of operations also doubles. **[Next]**

Formally, we can say that the complexity of find_max algorithm in big O(n) **[Next],**

where we simply ignore any constants and coefficients of the polynomial. **[Next]**

We shall define this big Oh notation soon. **[Next]**

Understanding this notion of step count or complexity helps us evaluate the efficiency of algorithms, which is crucial when working with large amount of data.

[Next]

→Slide4:

Let's write our linear_search algorithm. The algorithm to find the maximum number was named the FindMax algorithm. Similarly, we name our search algorithm LinearSearch, an apt name that clearly reflects its function. **[Next]**

For the FindMax algorithm, the input was a list of numbers. For LinearSearch, the input is also a list of numbers, but we add an additional input—the target value we're searching for within that list. **[Next]**

The output of the FindMax algorithm is straightforward: it returns the maximum number found in the list. In contrast, the output of LinearSearch is the position of the target value within the list, or -1 if the target is not found. This indicates whether and where the target exists in the list. **[Next]**

As both algorithms Begin, in FindMax, we initializes a variable max to the first element in the list. In LinearSearch, there's no need for such an initialization. Our focus is solely on finding a match for the target value, so we skip this step. **[Next]**

Next, in FindMax, we loop through each number in the list. This loop allows us to compare every element to our current max to see if it's larger. Similarly, in LinearSearch, we also loop through each number in the list. However, instead of looking for a larger number, we're searching for an exact match with the target value. **[Next]**

Within the loop in FindMax, we compared the current number to max. If the current number was greater than max, we updated max to this new value. In LinearSearch, rather than checking if the current number is greater, we check if it equals the target value. **[Next]**

If a match is found, we return the index of the current element immediately, not executing the loop any further. But how to return the index? We'll turn to it shortly. **[Next]**
After the loop completes in FindMax, the algorithm returns the max value, which is the largest number found in the list. In LinearSearch, if the loop completes without finding the target, we return -1 to indicate that the target is not present in the list. That's it! **[Next]**
In summary, both FindMax and LinearSearch algorithms share a similar structure, but their goals and internal logic differ slightly. By following a similar approach but adapting the logic, we can see how algorithms can be tailored to solve different problems. **[Next]**
You may pause and notice the differences highlighted in LinearSearch compare to FindMax.
[Next]

→Slide5:

Let's move on to the implementation. On the left, we have the find_max function, which identifies the maximum value in a list. On the right, we begin writing the linear_search function. Just like in find_max, we start by defining the function with its input parameters. In linear_search, these parameters are the list of numbers and the target value we're searching for.**[Next]**
As we move on, both functions use a loop to iterate through the list.
In find_max, the loop checks if each number is greater than the current maximum. In contrast, linear_search checks if each number matches the target value. **[Next]**
If a match is found, the function returns the result immediately. **[Next]**
Notice how this mirrors the comparison step in find_max, but instead of updating a maximum value, linear_search is focused on identifying and returning the position of the target. The question mark you see is what we will shortly handle to return the position or index.**[Next]**
If the loop completes without finding a match, linear_search returns -1, indicating the target isn't in the list. **[Next]**
Finally, to demonstrate the function, we use a sample list and a target value,**[Next]**
printing out the result to show where the target is found.
[Next]

→Slide6:

Now, let's refine our linear_search function to make it easier to return the position of the target. **[Next]**
Instead of using a simple for loop to iterate through the numbers, we'll switch to a range-based loop. This loop allows us to work with the index directly, which is crucial for

returning the position of the target in the list. **[Next]**

The range(len(numbers)) expression generates a sequence of indices from 0 to the length of the list minus one. By iterating over these indices, we can access both the index and the corresponding element in the list. This is different from our previous loop, where we only accessed the elements directly without knowing their positions. **[Next]**

Let's break down how this refined loop works. First, for each index i in the range of the list's length, we check whether the element at that index equals the target. If a match is found, the function immediately returns the index i, **[Next]**

which gives us the exact position of the target in the list. **[Next]**

If the loop completes without finding the target, we return -1, indicating that the target is not present in the list. **[Next]**

By using the range-based loop, we enhance the linear_search function's ability to not only determine if the target exists but also to provide the specific location of the target within the list. This refinement is particularly useful when the position of the element is as important as finding the element itself

[Next]

→Slide7:

Now, let's further refine our linear_search function to make the output more informative and user-friendly. So far, we directly print the result of the linear_search function, which shows the position if the target is found or a -1 if it isn't. **[Next]**

While this worked, it doesn't clearly communicate whether the target is found or not in a way that's easy to understand. **[Next]**

To improve this, we'll first store the result of the linear_search function in a variable called index. This allows us to check the result before deciding what to print. **[Next]**

If index equals -1, it means the target wasn't found in the list, so we print a message saying, "The target is not found!" This makes it clear to the user that the search was unsuccessful.

[Next]

On the other hand, if index is not -1, it means the target was found, and we can print a message specifying the exact location, like this: "The target is found at location: index." This approach makes the output much more informative and user-friendly, clearly distinguishing between a successful and an unsuccessful search. **[Next]**

In our final refinement, we'll improve the flexibility of our linear_search function by allowing the user to specify the target value rather than hardcoding it in the code. Previously, we set the target value directly using a statement like target = 1. While this works, it's limited

because the target is fixed within the code. **[Next]**

To make our function more dynamic and interactive, we'll replace this with a statement that prompts the user to enter the number they want to search for. We use the input function, which asks the user to type a number. This input is then converted into an integer by wrapping the input("Enter the number to search: ") into int(), ensuring that it can be compared with the numbers in our list. **[Next]**

This small change makes our program much more versatile. Now, the user can search for any number they choose, without having to modify the code itself. This refinement is particularly useful in real-world applications, where the data we're working with can vary each time the program runs. It's a great example of how we can take a simple algorithm and adapt it to be more user-friendly and applicable in different scenarios.

Let's see linear search in action

[Next] **STOP**

→Slide9:

Let's analyze the complexity of the linear_search function by counting the steps involved. We start with the first line, where the function is defined. **[Next]**

This line does not depend on the input size, so it takes a constant 1 step.

Next, we move to the loop. The loop iterates over the list, checking each element to see if it matches the target. **[Next]**

Since the loop runs once for each element in the list, it can take up to n steps, where n is the number of elements in the list. Now, this is true only in what we call the worst case.

[Next]

The worst case occurs when the target is either at the very end of the list or not in the list at all. In these situations, the loop has to check every single element before finishing.

On the other hand, there's also a best case. **[Next]**

The best case happens when the target is found at the very first position in the list. In this scenario, the loop only needs to run once, taking just 1 step. In the exercise, you will be able to play with these cases.

Inside the loop, there's an if statement that checks if the current element matches the target.

[Next]

This condition is checked every time the loop runs. If a match is found, the function immediately returns the index of that element. **[Next]**

No matter, when this return statement is executed, it will execute only once because the function does not continue once the return statement is executed anywhere inside the

function. In the best case, this return happens after just one step, when the target is at the beginning of the list. In the worst case, the return happens after n steps, either because the target is at the end of the list or it isn't found at all, leading to the final return statement outside the loop.

So, after the loop, if the target wasn't found, the function returns -1. **[Next]**

This final check also takes a constant 1 step. When we sum up the steps, **[Next]**

in the worst case, the total number of steps can be represented as 2n + 3. **[Next]**

As we did for our find_max algorithm, in Big-O notation, this simplifies to O(n). **[Next]**

Big-O simply ignores all the constants and coefficients. **[Next]**

This means the time complexity of the linear_search function is linear in the worst case, just like the find_max algorithm.

In the best case, however, **[Next]**

the time complexity is O(1) because the search completes after a single comparison.

In general, as the input size increases, the time it takes to complete the search grows proportionally in the worst case. **[Next]**

This means that for every additional element in the input, the number of operations required by the algorithm increases by one. As the input size grows, the algorithm's time complexity increases at a constant rate, resulting in linear growth.

You will have more to explore about the worst, and best cases in the exercise associated with this video. You will also find a third case, the average case.

Well! A lot of information. Don't worry, we shall unroll these concepts again in our next video.

[Next]

Slide10:

Let's recap what we covered today: **[Next]**

1. We analyzed the step count or complexity of our find_max algorithm. **[Next]**
2. We also briefly defined the big oh notation and hope to solidify the concept as we move forward. **[Next]**
3. We then took a deep dive into the Linear Search algorithm comparing it to the find_max algorithm to highlight their similarities. From there, we refined our Linear Search code, making it more robust and user-friendly. **[Next]**
4. To wrap up, we conducted a complexity analysis, exploring how Linear Search performs in best case, and worst case scenarios. **[Next]**

With all this knowledge at your fingertips, it's time to apply what you've learned. We've

prepared an exercise for you on GitHub, linked in the video description. **[Next]**
This exercise will challenge you to explore the step counts of Linear Search in different scenarios, from the best case to the worst case. By working through it, you'll solidify your understanding and gain practical insights into how the algorithm behaves in various situations. So, make sure to check it out and put your skills to the test! **[Next]**
Coming up next, we'll explore Binary Search—an algorithm that takes search efficiency to a whole new level. While Linear Search checks elements one by one, Binary Search is much faster, but there's a catch: it only works on sorted lists. **[Next]**
By continually halving the search range, Binary Search quickly zeroes in on the target, making it a vital tool in algorithm design. Don't miss our next video, where we'll break down how Binary Search works and why it's such a game changer!

[Next]

Slide11:

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 Binary Search. Happy coding!

[Next]

****************DEMO SLIDE*****************

→Slide8:

We are here on PythonTutor website and here is our code:
Let's walk through the execution of our code step-by-step using PythonTutor.
The first line defines the linear_search function, but the control does not enter the function yet. Instead, the function is stored in memory, ready to be called later.

[Next]

Next, control moves to line 7, where we define sample_list, a list containing the numbers [5, 3, 9, 1, 6]. This list will be the data we search through.

[Next]

Control then moves to line 8, where the program prompts the user to enter a number to search for. Let's inputs 1

[Next]

On line 9, the linear_search function is called with sample_list and target as arguments.

[Next]

Control now enters the function starting at line 2, where a loop is initiated using

range(len(numbers)). This loop will iterate over the indices of the list, allowing us to check each element.

[Next]

On the first iteration of the loop i is 0, and control moves to line 3. The function checks if the element at index 0, which is 5, equals the target 1. Since it doesn't, control moves back to line 2 for the next iteration.

[Next]

On the second iteration i is 1, and the element at index 1, which is 3, is checked against the target. Again, it's not a match, so control loops back to line 2.

[Next]

In the third iteration i is 2, and the element at index 2 is 9. Since this isn't equal to 1, the loop continues.

[Next]

In the fourth iteration i is 3, and the element at index 3 is 1, which matches the target. Since a match is found, control moves to line 4 and the function returns the index 3.

[Next]

Control exits the loop and the function, returning to line 9.

[Next]

Back at line 10, the program checks if index is -1, which would indicate the target wasn't found. Since index is not -1 and is 3 instead, control moves to line 12.

[Next]

Finally, at line 13, the program prints The target is found at location: 3, confirming that the target 1 was located at index 3 in the list.

[Next]

Transcripts for the Video Explaining the Solution to Exercise

Show find_max_exercise.py:

Hi everyone! In this video, we're going to tackle the exercise that builds on our 'Find Max' algorithm. We'll develop the solution step by step and understand the concept of algorithm complexity in a simple and accessible way.

Let's start with the exercise. We have a function `find_max` that needs to find the maximum number in a list and count the number of steps it takes to do so. We will focus on the steps that are influenced by the size of the input. Let's go to PythonTutor and see the `find_max` function to understand which lines are influenced by the input size and which are not. **[Ctrl+P]**

[Ctrl+P] In PythonTutor find_max_exercise.py

- Line 1 defines the function and is not influenced by the input size. It executes only once when the function is called regardless of the size of the input list.
- Line 3 initializes the `max_number` variable with the first element of the list. It is a single operation and executes only once no matter what the size of the input list is.
- Line 4 starts a loop that iterates through each element in the list. The number of iterations is directly influenced by the size of the input list. For a list of size 10, the loop will run 10 times, and for a list of size 20, it will run 20 times. In general, for a list of size `n`, the loop will run `n` times.
- Lines 6 and 7 are also influenced by the length of the list because they are part of the loop. Therefore, this loop is the part of code where we need to count the steps.

[Ctrl+P]

[Ctrl+P] Editing Code

Let's first define and initialize a variable `steps` to 0 at the beginning of our function:

```
steps = 0
```

Now, we increment this variable inside our loop for each element checked.

```
steps += 1
```

The comments clearly explain these TODOs.

The last line returns the `max_number` found. It is a single operation and is not influenced by any change in the size of the input list."

Since `return` is the last statement executed in the function, we print the number of steps taken to find the maximum number before it here. **[Ctrl+P]**

Paste line `print("It took", steps, "steps to find the maximum number from a list of", len(numbers), "numbers")`

[Ctrl+P] Running the Function with Different Lists

To demonstrate, we'll use different lists as input and observe the change in the count of `steps` with changing input size." "Let's use the first list with 5 numbers as input and count the steps:

[uncomment]sample_numbers = [5, 3, 9, 1, 6]

We press `visualize execution` … and we now press `last` to directly see the final output.

Read the output

Lets go back to the code and use the second list with 10 numbers as input to `find_max`

[uncomment] sample_numbers = [8, 2, 10, 4, 7, 1, 12, 14, 3, 6]

We press `visualize execution` … and we now press `last` to directly see the final output.

Read the output

Let's go back to edit the code and execute it for the third and last list that has 22 numbers

[uncomment] sample_numbers = [15, 3, 9, 8, 22, 4, 13, 2, 11, 17, 1, 19, 6, 18, 7, 20, 5, 14, 10, 21, 16, 12]

Press Visualize … and Press last Passing this list to `find_max`

Read the output

Explaining the Step Counts

In general, for each list, the number of steps taken by the algorithm is equal to the number of elements in the list. This is because the algorithm iterates through each element once to find the maximum number. Steps like initializing or returning the `max_number` are not influenced by the input size, but the looping and comparisons are."

This linear relationship between the input size and the number of steps taken is what we refer to as linear complexity, often denoted as O(n).

While this may seem intuitive, you might have questions like:

- Why do we use the notation O(n) and not just `n`?
- Why are we talking about input size only and not other factors like processing and memory speeds?
- Why did we count the step inside the loop as 1, ignoring the count for the `if` statement and updating the `max_number`?

Don't worry about these details; we shall slowly but surely answer these questions too.

Closing

That's all for now. Thank you for watching and joining me on this journey through algorithms. Don't forget to try this exercise with different lists and observe the step counts yourself. Share your thoughts and questions in the comments. See you next time for our next lesson on linear search. Happy coding!