**Search in an Arrays**

Finally, we're going to look at the most important operation of all. More often than not, it comes down to the speed of searching for an element in a data structure that helps programmers make design choices for their codebases.

There's more than one way of searching an Array, but for now, we're going to focus on the simplest way. Searching means to find an occurrence of a particular element in the Array and return its position. We might need to search an Array to find out whether or not an element is present in the Array. We might also want to search an Array that is arranged in a specific fashion to determine which index to insert a new element at.

If we know the index in the Array that *may* contain the element we're looking for, then the search becomes a constant time operation—we simply go to the given index and check whether or not the element is there.

Linear Search

If the index is not known, which is the case most of the time, then we can check every element in the Array. We continue checking elements until we find the element we're looking for, or we reach the end of the Array. This technique for finding an element by checking through all elements one by one is known as the **linear search** algorithm. In the worst case, a linear search ends up checking the entire Array. Therefore, the time complexity for a linear search is O(N)*O*(*N*).

A diagram of a row of squares

AI-generated content may be incorrect.

Let's see the linear search algorithm in action, with all the edge cases handled properly. When we say *edge cases*, we basically mean scenarios that you wouldn't expect to encounter. For example, the element you're searching for might not even exist in the Array. Or, an even rarer, but possible, scenario is that the input Array doesn't contain any elements at all, or perhaps it is null. It's important to handle all of these edge cases within the code.

public static boolean linearSearch(int[] array, int length, int element) {

// Check for edge cases. Is the array null or empty?

// If it is, then we return false because the element we're

// searching for couldn't possibly be in it.

if (array == null || length == 0) {

return false;

}

// Carry out the linear search by checking each element,

// starting from the first one.

for (int i = 0; i < length; i++) {

// We found the element at index i.

// So we return true to say it exists.

if (array[i] == element) {

return true;

}

}

// We didn't find the element in the array.

return false;

}

That's the function we can call to determine whether or not a particular element is in an Array. Notice how we take care of the edge cases before proceeding with the actual search, and that we don't check the rest of the elements once we'd found the element we were looking for.

There are many variations to this algorithm, such as returning the first location, last location, or all the locations (an element could be in the Array more than once). Let's see what happens when we call the linearSearch function.

public class ArraySearch {

public static void main(String args[]) {

// Declare a new array of 6 elements

int[] array = new int[6];

// Variable to keep track of the length of the array

int length = 0;

// Iterate through the 6 indexes of the Array.

for (int i = 0; i < 6; i++) {

// Add a new element and increment the length as well

array[length++] = i;

}

// Print out the results of searching for 4 and 30.

System.out.println("Does the array contain the element 4? - " + ArraySearch.linearSearch(array, length, 4));

System.out.println("Does the array contain the element 30? - " + ArraySearch.linearSearch(array, length, 30));

// Does the array contain the element 4? - true

// Does the array contain the element 30? - false

}

public static boolean linearSearch(int[] array, int length, int element) {

// Check for edge cases

if (array == null || length == 0) {

return false;

}

// Check each element starting from the first one

for (int i = 0; i < length; i++) {

// We found the element at index i, so return true.

if (array[i] == element) {

return true;

}

}

// We didn't find the element in the array.

return false;

}

}

As expected, we're able to find the element 4 in the Array, but not 30.

Binary Search

*This section is optional. It briefly introduces a more advanced searching algorithm that you will learn more about in a later Explore Card.*

There is another way of searching an Array. If the elements in the Array are in *sorted order*, then we can use binary search. Binary search is where we repeatedly look at the middle element in the Array, and determine whether the element we're looking for must be to the left, or to the right. Each time we do this, we're able to halve the number of elements we still need to search, making binary search a lot faster than linear search!

The downside of binary search though is that it only works if the data is sorted. If we only need to perform a single search, then it's faster to just do a linear search, as it takes longer to sort than to linear search. If we're going to be performing a lot of searches, it is often worth sorting the data first so that we can use binary search for the repeated searches.

You can find out more about binary search on our [Binary Search Explore Card](https://leetcode.com/explore/learn/card/binary-search/). For Arrays 101, it is okay for you to use either linear search or binary search.

Hopefully, the three basic Array operations are clear now! Like always, there are a couple of problems for you to try for yourself now.

After that, we'll be having a look at In-Place Array Operations. What are those, you might be asking? Let's not get ahead of ourselves though—you'll find out soon!