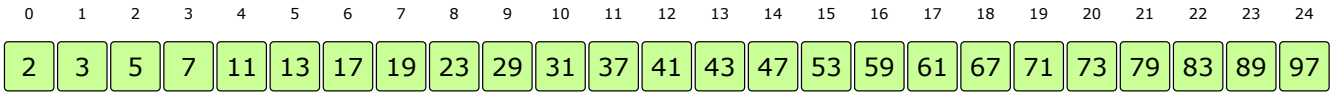
Implementing Binary Search of an Array

Let's think about binary search on a sorted array. Many programming languages already provide methods for determining whether a given element is in an array and, if it is, its location. But we want to implement it ourselves, to understand how you can implement such methods. Here's an array of the first 25 prime numbers, in order:



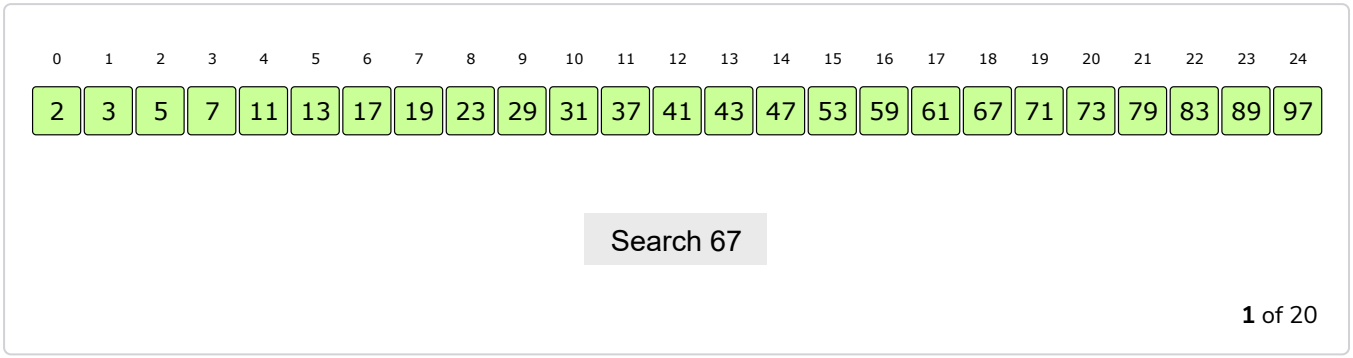
Suppose we want to know whether the number 67 is prime. If 67 is in the array, then it's prime.

We might also want to know how many primes are smaller than 67. If we find the position of the number 67 in the array, we can use the position to figure out how many smaller primes exist.

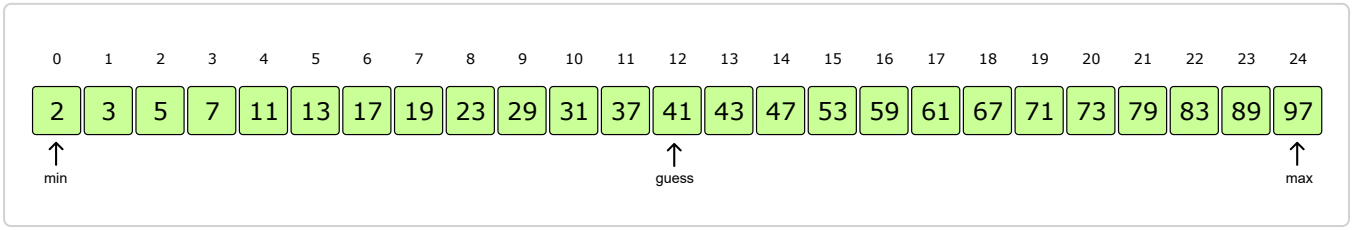
The position of an element in an array is known as its index. Array indices start at 0 and count upwards. If an element is at index 0 then it is the first element in the array. If an element is at index 3, then it has 3 elements which come before it in the array.

Looking at the example below, we can read the array of prime numbers from left to right, one at a time, until we find the number 67—in the green box—and see that it is at array index 18. Looking through the numbers in order like this is a **linear search**.

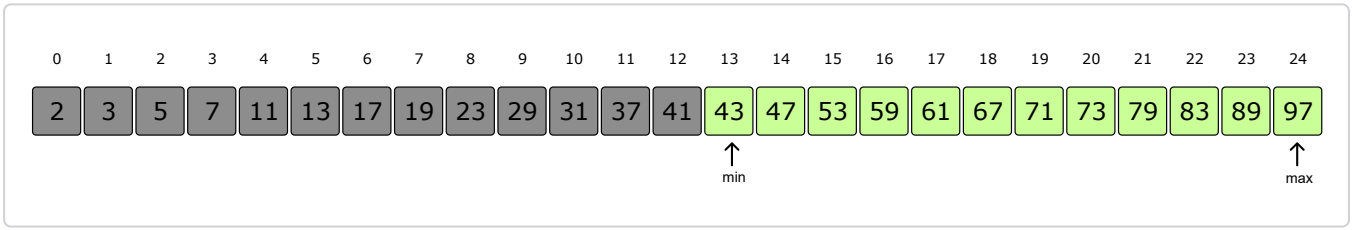
Once we know that the prime number 67 is at index 18, we can identify that it is a prime. We can also quickly identify that there are 18 elements which come before 67 in the array, meaning that there are 18 prime numbers smaller than 67.



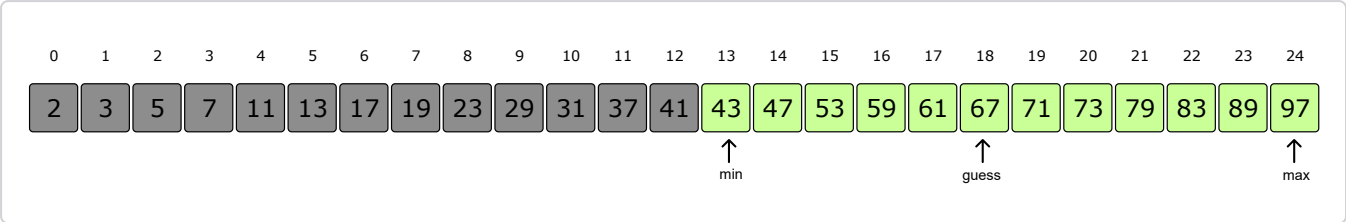
Did you see how many steps that took? A binary search might be more efficient. Because the array primesprimes contains 25 numbers, the indices into the array range from 0 to 24. Using our pseudocode from before, we start by letting **min** = **0** and **max**=**24**. The first guess in the binary search would therefore be at index 12, which is **(0** + **24)** / **2**. Is **primes[12]** equal to 67? No, **primes[12]** is **41**.



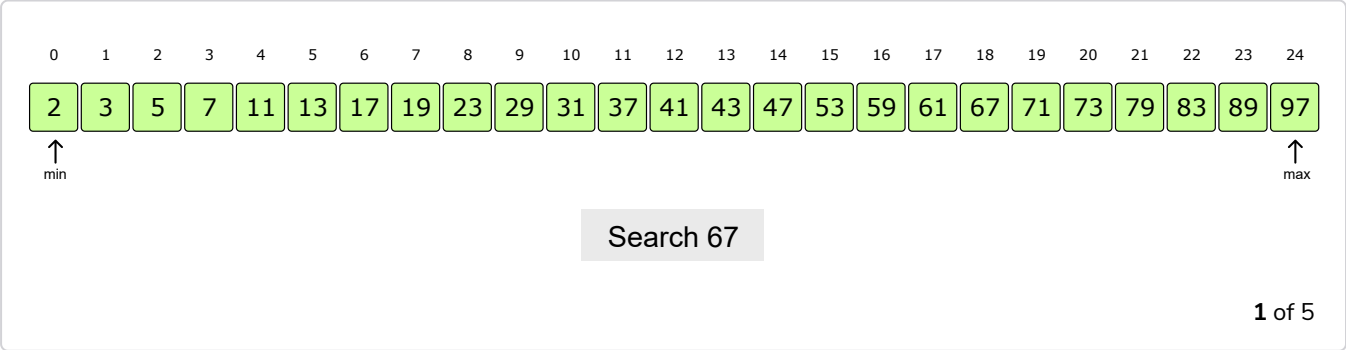
Is the index we are looking for higher or lower than 12? Since the values in the array are in increasing order, and **41** < **67,** the value 67 should be to the right of index 12. In other words, the index we are trying to guess should be greater than 12. We update the value of **min** to **12** + **1**, or 13, and we leave **max** unchanged at 24.



What's the next index to guess? The average of 13 and 24 is 18.5, which we round down to 18, since an index into an array must be an integer. We find that **primes[18]** is **67**.



The binary search algorithm stops at this point, since it has found the answer. It took only two guesses, instead of the 19 guesses that linear search would have taken. You can step through that again in the visualization below:



Pseudocode

We just described the binary search algorithm in English, stepping through one example. That's one way to do it, but a human language explanation can vary in quality. It can be too short or too long, and—most importantly—it's not always as precise as it should be. We could jump to showing you binary search in a programming language like JavaScript or Python, but programs contain lots of details due to requirements imposed by the programming language and because programs have to handle errors caused by bad data, user error, or system faults. These details can make it hard to understand the underlying algorithm from studying just the code. That's why we prefer to describe algorithms in something called **pseudocode**, which mixes English with features that you see in programming languages.

Below is the pseudocode for binary search, modified for searching in an array. The inputs are the array, which we call arrayarray; the number **n** of elements in **array**; and **target**, the number being searched for. The output is the index in **array of target**:

1. Let **min** = **0** and **max=n−1**.

2. Compute **guess** as the average of **max** and **min**, rounded down so that it is an integer.
3. If **array[guess]** equals **target**, then stop. You found it! Return **guess**.
4. If the guess was too low, that is, array[guess] < target, then set **min** = **guess** + **1**.
5. Otherwise, the guess was too high. Set **max** = **guess - 1**.
6. Go back to step two.

Implementing Pseudocode

We'll alternate between English, pseudocode, and an actual programming language like Javascript or Python in these tutorials, depending on the situation. As a programmer, you should learn to understand pseudocode and be able to turn it into your language of choice. Even though we're using JavaScript here, it should be straightforward for you to implement pseudocode using other languages.

How would we turn the pseudocode in the example above into a JavaScript/Python program? We should create a function because we're writing code that accepts an input and returns an output, and we want that code to be reusable for different inputs. The parameters to the function—let's call it **binarySearch**—will be the array and target value, and the return value of the function will be the index of the location where the target value was found.

Now let's go into the body of the function and decide how to implement our plan. Step six says to go back to step two. That sounds like a loop. Should it be a for-loop or a while-loop? If you really wanted to use a for-loop, you could, but the indices guessed by binary search don't go in the sequential order that would make a for-loop convenient. We might first guess the index 12 and then 18, based on some computations. Because of this, a while-loop is the better choice.

There's also an important step missing in the pseudocode that doesn't matter for the guessing game, but does matter for the binary search of an array. What would happen if the number you are looking for is not in the array? Let's start

by figuring out what index the **binarySearch** function should return in this case. It should be a number that cannot be a legal index into the array. We'll use -1, since that cannot be a legal index into any array—actually, any negative number would do.

The target number isn't in the array if there are no possible guesses left. In our example, suppose that we're searching for the target number 10 in the **primes** array. If it were there, 10 would be between the values 7 and 11, which are at indices 3 and 4. If you trace out the index values for **min** and **max** as the **binarySearch** function executes, you would find that they eventually get to the point where **min** = **3** and **max**=**4**. The guess is then index 3—since **(3** + **4)** / **2** = **3.5** and we round down—and **primes[3]** is less than 10, so that **min** becomes 4. With both **min** and **max** equaling 4, the guess must be index 4, and **primes[4]** is greater than 10. Now **max** becomes 3. What does it mean for **min** to equal 4 and **max** to equal 3? It means that the only possible guesses are at least 4 and at most 3. There are no such numbers! At this point, we can conclude that the target number, 10, is not in the **primes** array, and the **binarySearch** function would return -1. In general, once **max** becomes strictly less than **min**, we know that the target number is not in the sorted array.

Here is modified pseudocode for binary search that handles the case in which the target number is not present:

1. Let **min** = **0** and **max**=**n−1**.
2. If **max**<**min**, then stop; **target** is not present in **array**. Return -1.
3. Compute **guess** as the average of **max** and **min**, rounded down so that it is an integer.
4. If **array[guess]** equals **target**, then stop. You found it! Return **guess**.
5. If the guess was too low, that is, **array[guess]** < **target**, then set **min** = **guess** + **1**.
6. Otherwise, the guess was too high. Set **max** = **guess - 1**.
7. Go back to step two.

Now that we've thought through the pseudocode together, try implementing binary search yourself. It's fine to look back at the pseudocode. In fact, it's a good thing because then you'll have a better grasp of what it means to convert pseudocode into a program.