Big-θ (Big-Theta) notation

Let's look at a simple implementation of linear search (in the language of your choice):

JS C++ Python

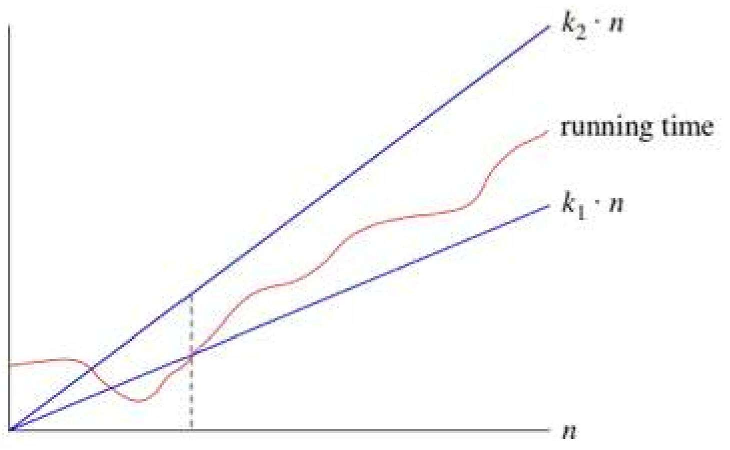
```
var doLinearSearch = function(array, targetValue) {
  for (var guess = 0; guess < array.length; guess++) {
    if (array[guess] === targetValue) {
        return guess;  // found it!
    }
  }

  return -1;  // didn't find it
};
```

Let's denote the size of the array by **n**. The maximum number of times that the for-loop can run is **n**, and this worst case occurs when the value being searched for is not present in the array. Each time the for-loop iterates, it has to do several things: compare guess with **n**; compare **array[guess]** with **targetValue;** possibly return the value of **guess**; and increment **guess**. Each of these little computations takes a constant amount of time each time it executes. If the for-loop iterates **n** times, then the time for all **n** iterations is $c_1 \cdot n$, where c_1 is the sum of the times for the computations in one loop iteration. Now, we cannot say here what the value of c_1 is, because it depends on the speed of the computer, the programming language used, the compiler or interpreter that translates the source program into runnable code, and other factors. This code has a little bit of extra overhead, for setting up the for-loop (including initializing guess to **0**) and possibly returning **-1** at the end. Let's call the time for this overhead c_2, which is also a constant. Therefore, the total time for linear search in the worst case is $c_1 \cdot n + c_2$.

As we've argued, the constant factor c_1 and the low-order term c_2 don't tell us about the rate of growth of the running time. What's significant is that the worst-case running time of linear search grows like the array size **n**. The

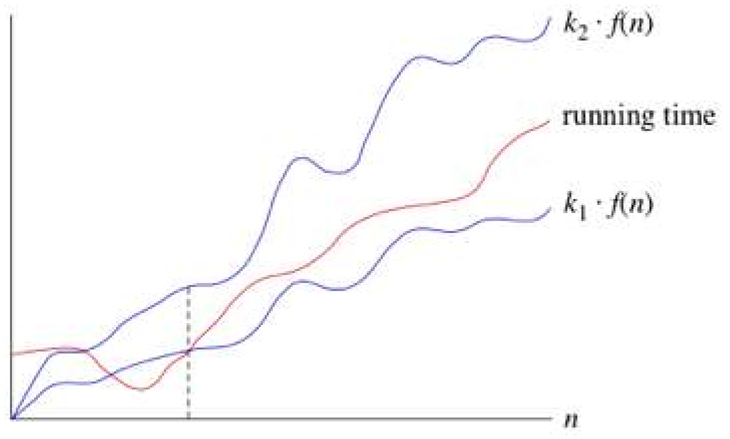
notation we use for this running time is **Θ(n)**. That's the Greek letter "**theta**," and we say "**big-Theta of n**" or just "**Theta of n**."

When we say that a particular running time is **Θ(n)**, we're saying that once **n** gets large enough, the running time is at least **$k_1 \cdot n$** and at most **$k_2 \cdot n$** for some constants **k_1** and **k_2**. Here's how to think of **Θ(n)**:



For small values of n, we don't care how the running time compares with **$k_1 \cdot n$** or **$k_2 \cdot n$**. But once **n** gets large enough—on or to the right of the dashed line—the running time must be sandwiched between **$k_1 \cdot n$** and **$k_2 \cdot n$**. As long as these constants **k_1** and **k_2** exist, we say that the running time is **Θ(n)**.

We are not restricted to just **n** in **big-Θ** notation. We can use any function, such as **n^2**, **n lgn**, or any other function of **n**. Here's how to think of a running time that is **Θ(f(n))** for some function **f(n)**:



Once **n** gets large enough, the running time is between **$k_1 \cdot f(n)$** and **$k_2 \cdot f(n)$**.

In practice, we just drop constant factors and low-order terms. Another advantage of using **big-Θ** notation is that we don't have to worry about which time units we're using. For example, suppose that you calculate that a running time is **$6n^2$** + **100n** + **3006** microseconds. Or maybe it's milliseconds. When you use **big-Θ** notation, you don't say. You also drop the factor **6** and the low-order terms **100n** + **300**, and you just say that the running time is **$\Theta(n^2)$**.

When we use **big-Θ** notation, we're saying that we have an asymptotically tight bound on the running time. "Asymptotically" because it matters for only large values of **n**. "Tight bound" because we've nailed the running time to within a constant factor above and below.