We want to know how long our algorithms will take to complete their task in because we want to be efficient. In binary search, we make guesses and checks, but there’s more to that. The time it will take the computer to process the algorithm does not only depend on the number of steps (checks and guesses), but also on *how the code is written*.

* First, we think about the running time of the algorithm as a function of the size of its input. (If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right?)
* Second, The second is how fast this function grows with the size of the input(it’s called the **rate of growth** of the running time). Binary search complexity function has a logarithmic progression, whereas linear search function has a linear progression. Sinc log grows slower than linear, we conclude that binary is here better than linear.

So, depending on the size of the input we want to analyze, we are gonna chose different algorithms. (Usually the input is really large)

If we compare n² and bn+c, there will always be a moment when the square function surpasses the linear one, and from that point on it’s forever, no matter the coefficients. That’s because an²-bn-c is croissant and linéaire, so it will pass by 0 (bijection) and continue above : there is a value of n where an² starts getting bigger than bn+c.

**Mesuring the rate of growth of a function**

**Big Theta (Big-θ) notation**

Let array.length = n;

The complexity of a linear search algorithm is n times the number of steps in one loop operation (n\*c1).

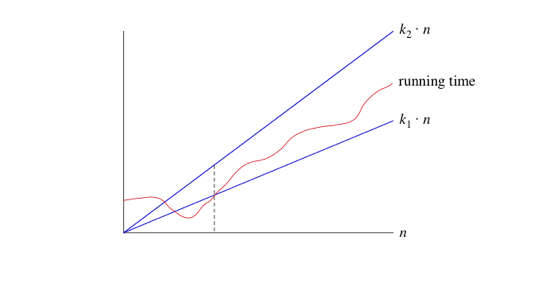
It also has to set up the for loop and possibly return -1 if not in array (so one extra step here), and that is a certain number of extra steps we’ll call c2.

So the worst case growth function is : n\*c1+c2.

But we don’t want to mesure the rate of growth with the worst case scenario, so we are not gonna use these coefficients. We are gonna try and predict the *zone of probability* where the growth function will be.

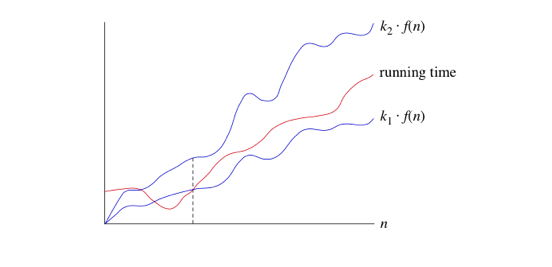
We want to basically be able to take two functions and say “the running time growth will be somewhere between these two functions”.

Thêta(n) is how to expect the growth function to behave as the array size grows. If n becomes really big, we are saying that the running time will be somewhere between k1\*n and k2\*n:



to the right of the dashed line—the running time must be sandwiched between k1⋅n and k2⋅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 n and can apply theta to non-linear functions:

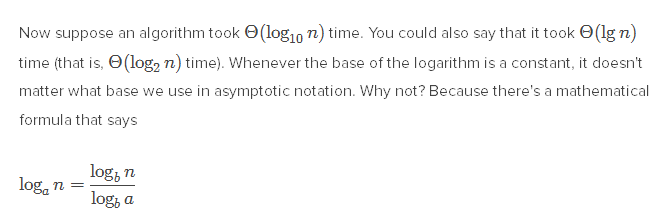


Where f(n) = log(n)\*n

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 nnn. "Tight bound" because we've nailed the running time to within a constant factor above and below.

When we use asymptotic notation to express the rate of growth of an algorithm's running time in terms of the input size n, it's good to bear a few things in mind.

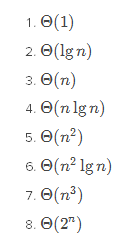
Let's start with something easy. Suppose that an algorithm took a constant amount of time, regardless of the input size. For example, if you were given an array that is already sorted into increasing order and you had to find the minimum element, it would take constant time, since the minimum element must be at index 0. Since we like to use a function of nnn in asymptotic notation, you could say that this algorithm runs in Θ(n0)\ time. Why? Because n^0 = 1, and the algorithm's running time is within some constant factor of 1. In practice, we don't write Θ(n0however; we write Θ(1)



That is because blogb a \* loga n = blogb n = n for a, b >0

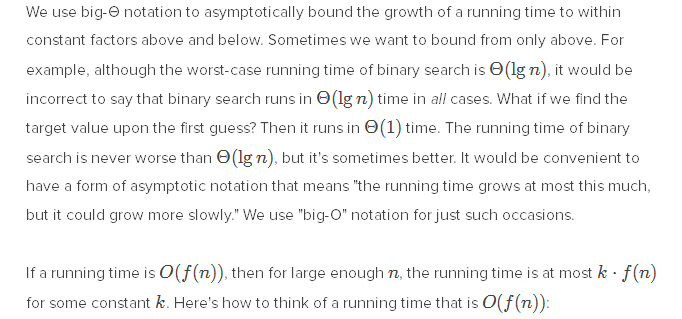
That’s why we will usually use log2. However, it still means that a log6 function will grow slower than a log2 function (works if we replace a and b with 6 and 2)

Here is a list of growth functions in ascending order:



f(n) and theta(f(n)) are equivalent. if f(n) is polynomial, so will be theta.

**Big-O notation:**



Big-O targets the worst possible outcome (we say it gives an asymptotic upper bound).

So it is correct to say that a binary search runs in O(n) time, although it should be lg(n). that’s because O(n) is the upper bound given a possible outcome. It’s like saying “I will complete my homework in less than 10 000 hours.”

nk is O(cn) (means that O(cn) is an upper bound of nk. Very misleading.)

If f(n) is O(g(n)) then, informally, f(n) is within a constant factor of g(n).

**Big-Omega notation**

You guessed it, the big-omega is the other side of the coin, it’s the slowest possible growth for a given outcome, the lower bound (still and always equal to k\*f(n)).

