BIG O: -

This asymptotic notation means that "the running time grows at most this much, but it could grow more slowly."

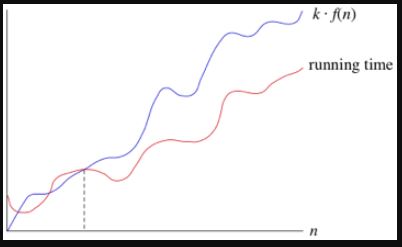


If a running time is

then for large enough n, the running time is at most 

for some constant k.

Here's how to think of a running time that is



We say that the running time is "big-O of f(n) " or just "O of f(n)" We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

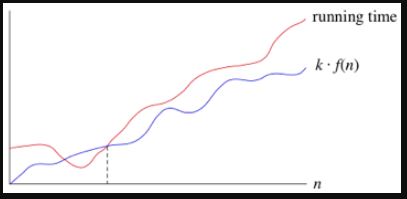
It always provides us with the worst-case scenario of time taken by an algorithm with respect to the input provided.

Suppose you have 10 dollars in your pocket. You go up to your friend and say, "I have an amount of money in my pocket, and I guarantee that it's no more than one million dollars." Your statement is absolutely true, though not terribly precise.

Big-Omega(Ω):

This says that an algorithm takes at least a certain amount of time, without providing an upper bound. We use big-Ω notation; that's the Greek letter "omega."

If a running time is Ω(f(n)) , then for large enough n , the running time is at least k⋅f(n) , for some constant k. Here's how to think of a running time that is Ω(f(n)):

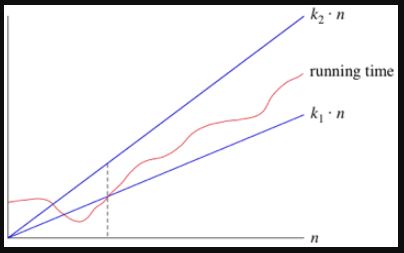


We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

This basically represents the best time for an algorithm with respect to the inputs provided.

Big Theta(Θ): -

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 k1⋅n and at most k2⋅n for some constants k1 and k2. Here's how to think of Θ(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 n. "Tight bound" because we've nailed the running time to within a constant factor above and below.