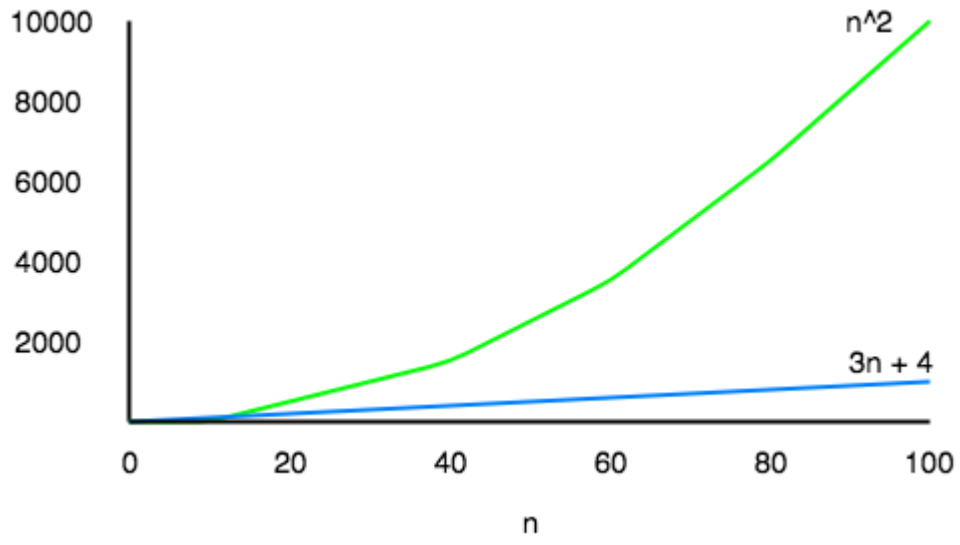
Asymptotic Notations

When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.

When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc. This formula often contains unimportant details that don't really tell us anything about the running time.

Let us take an example, if some algorithm has a time complexity of $T(n) = (n^2 + 3n + 4)$, which is a quadratic equation. For large values of `n`, the `3n + 4` part will become insignificant compared to the `n^2` part.



For n = 1000, `n^2` will be `1000000` while `3n + 4` will be `3004`.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of `200n^2` will be faster than some other algorithm that takes `n^3` time, for any value of `n` larger than `200`. Since we're only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored too.

What is Asymptotic Behaviour

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Remember studying about **Limits** in High School, this is the same.

The only difference being, here we do not have to find the value of any expression where `n` is approaching any finite number or infinity, but in case of Asymptotic notations, we use the same model to ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

Let's take an example to understand this:

If we have two algorithms with the following expressions representing the time required by them for execution, then:

Expression 1: $(20n^2 + 3n - 4)$

Expression 2: $(n^3 + 100n - 2)$

Now, as per asymptotic notations, we should just worry about how the function will grow as the value of `n` (input) will grow, and that will entirely depend on n^2 for the Expression 1, and on n^3 for Expression 2. Hence, we can clearly say that the algorithm for which running time is represented by the Expression 2, will grow faster than the other one, simply by analysing the highest power coeeficient and ignoring the other constants(20 in $20n^2$) and insignificant parts of the expression(`3n - 4` and `100n - 2`).

The main idea behind casting aside the less important part is to make things **manageable**.

All we need to do is, first analyse the algorithm to find out an expression to define it's time requirements and then analyse how that expression will grow as the input(n) will grow.

Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

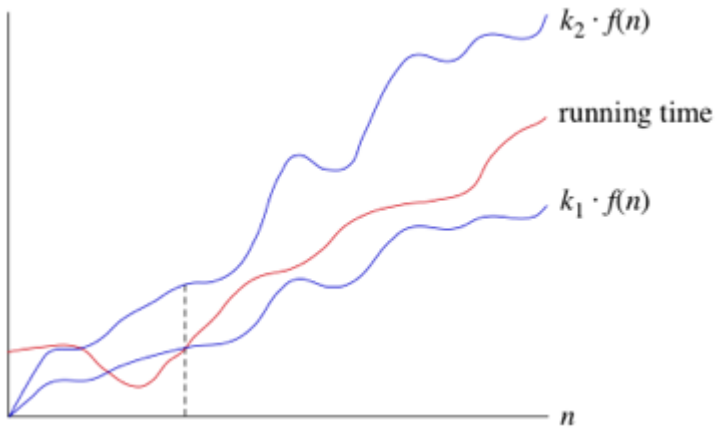
1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

Tight Bounds: Theta

When we say tight bounds, we mean that the time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of $\Theta(n^2)$ means, that the avaerage time for any input `n` will remain in between, `k1 * n^2` and `k2 * n^2`, where k1, k2 are two constants, therby tightly binding the expression rpresenting the growth of the algorithm.



Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input `n`.

The question is why we need this representation when we already have the big-Θ notation, which represents the tightly bound running time for any algorithm. Let's take a small example to understand this.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In **Worst case**, starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of `n`, where `n` represents the number of total elements.

But it can happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be `1`.

Now in this case, saying that the big-Θ or tight bound time complexity for Linear search is Θ(n), will mean that the time required will always be related to `n`, as this is the right way to represent the average time complexity, but when we use the big-O notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed `n`, defining the upper bound, hence saying that it can be less than or equal to `n`, which is the correct representation.

This is the reason, most of the time you will see Big-O notation being used to represent the time complexity of any algorithm, because it makes more sense.

Lower Bounds: Omega

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take atleast this much time to cmplete it's execution. It can definitely take more time than this too.
