📚 Big O Notation: Measuring Algorithmic Complexity

LESSON PROGRESS

50% Complete

Big O notation is a mathematical concept that characterises limiting behaviour, i.e. the value a function approaches as its input approaches a certain value or infinity. It belongs to a family of notations pioneered by German mathematicians Paul Bachmann, Edmund Landau, and others, commonly referred to as Bachmann–Landau or **asymptotic notation.** The letter “O” was selected by Bachmann to denote “Ordnung,” which translates to “order” in English and reflects the notion of an order of approximation.

In computer science, when discussing algorithms, we often measure their **efficiency** using **Big O notation**. Big O notation gives us a **high-level** understanding of how an algorithm's execution time and space (memory allocation) grows as the size of the input data increases.

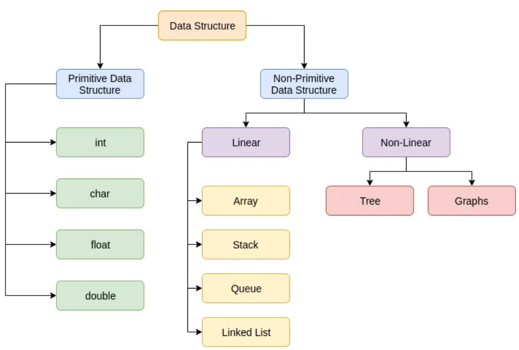
Asympto...what?

An **asymptote** of a curve is a line such as the distance between the curve and the line approaches zero as one or both of the x or y coordinates in a plane tend to infinity.

Take this simple function:

$$f(x) = frac1x$$

For positive values of x only we get this graph:



The graph contains the points $(1, 1)$, $(2, 0.5)$, $(4, 0.25)$ and $(10, 0.1)$. As the value of x tends to infinity the output of $f(x)$ will

approach the x axis but, crucially, will never touch it. For example, $f(100) = .01$, $f(1000) = .001$ and $f(1000) = .0001$ but **never** 0

. $f(1000000)$? well, it's 0.000001, still, never 0.

It can be said then that the x axis is an asymptote to the curve $f(x) = frac1x$

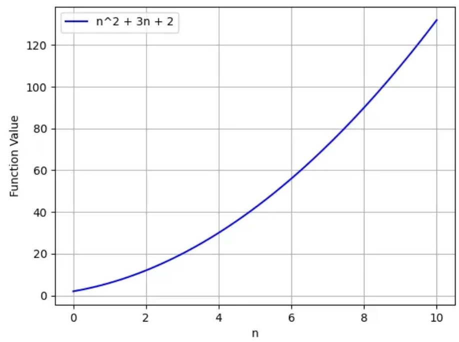
Asymptotical analysis

In asymptotical analysis we are interested in how a function behaves with very large input values. By understanding a function's limiting behaviour we can find its asymptotically equivalent.

Take this function:

$$f(n) = n^2 + 3n + 2$$

If we evaluate $f(n)$ with only positive values of n we get the following graph:



What would happen if we plug a very large number? Well, the function tends to inifinity when the input tends to inifity, we know as much. But let's zoom in a bit.

When n becomes very large $3n$ becomes insignificant compared to n^2 . For example, with $n = 1000000$, $n^2 = 1000000000000$ and $3n = 3000000$. We can of course completely disregard the constant portion of our function, at this scale that 2 really doesn't change anything.

Two simplification rules can be applied here:

- If $f(n)$ is a sum of several terms, if there is one with largest growth rate, it can be kept, and all others omitted.
- If $f(n)$ is a product of several factors, any constants (factors in the product that do not depend on n) can be omitted.

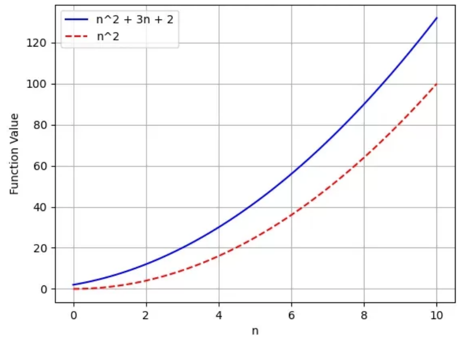
Let's apply this to our function:

Since our function is a sum of several terms and the one with the largest growth rate is n^2 , we can disregard the others. Our new function is just:

$$g(n) = n^2$$

There are no factors in our function that do not depend on n so no further simplifications apply! Let's graph our functions together:

Graph of n^2 + 3n + 2 and n^2



We can say that $f(n)$ is asymptotically equivalent to n^2 (for us, $g(n)$) as n approaches infinity. This relation is formally written as:

$$f(n)simg(n)quadtextasquadntoinfty$$

It reads: "$f(n)$ is asymptotically equivalent to $g(n)$ when n approaches infinity"

This is only true when the value of $fracf(n)g(n)$ when n tends to infinity is equal to 1, or:

$$lim_{ntoinfty}fracf(n)g(n) = 1$$

In our case, we know both $f(n)$ and $g(n)$ tend to infinity with very large values of n. Trivially, $fracinftyinfty = 1$

Why was all of that important?

In practice, Big O notation is asymptotical. It's concerned with very large inputs to analyse limiting behaviour. We can then say:

$$f(n) = n^2 + 3n + 2 = O(n^2)$$

In computer science, when analysing the complexity of algorithms, we focus on the operations performed by said algorithm, e.g. inserting, searching or deleting in the context of a particular data structure, to determine the overall complexity.

For instance, if within the same algorithm we access elements from an array by their index ($O(1)$) but then we compare elements of a list against all other elements of the same list in a nested loop ($O(n^2)$), the overall complexity of the algorithm will be $O(n^2)$

Throughout the course we will learn about various data structures, different operations and algorithms that apply to them. Wherever it applies, we will present the operation and algorithm's complexity given in Big O notation.

Common Big O Complexities

Time Complexity	Name	Description	Example Algorithm/Use Case

$O(1)$	**Constant Time**	The number of operations does not increase with the size of the input. This means that no matter how large the input grows, the execution time remains the same.	– **Array Indexing**: Accessing the 10th element in an array takes the same time as accessing the 1st element.
$O(log n)$	**Logarithmic Time**	The execution time grows logarithmically in	– **Binary Search**: Searching for an

		ally in proportion to the input size. Algorithms with this complexity typically divide the problem in half each step, leading to very efficient performance for large datasets.	an element in a sorted array by repeatedly dividing the search interval in half.
$O(n)$	**Linear Time**	The execution time grows linearly with the input	– **Linear Search**: Scanning through each

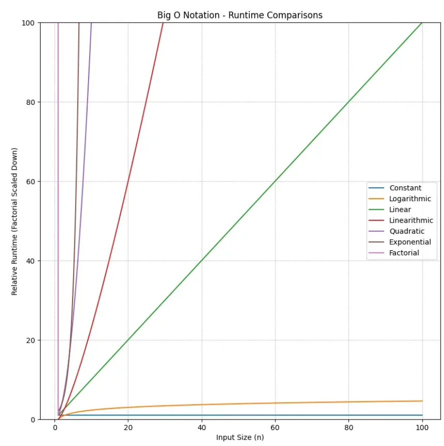
		size. Each additional element in the input requires a constant amount of additional time.	element in an unsorted list to find a target value.
$O(nlogn)$	**Linearithmic Time**	Combines linear and logarithmic growth rates. This is common in efficient sorting algorithms that divide the data into small	– **Merge Sort**: Divides the array into halves, recursively sorts each half, and then merges them.

		er chunks, sort them, and then merge the sorted chunks.	– **Quick Sort**: Partitions the array around a pivot and recursively sorts the partitions (average case).
$O(n^2)$	**Quadratic Time**	The execution time grows proportionally to the square of the input size. Typic	– **Bubble Sort**: Repeatedly steps through the list, compares

		ally arises from algorithms with nested loops, where each element is compared with every other element.	adjacent elements, and swaps them if they are in the wrong order. – **Checking All Pairs**: Evaluating every possible pair in a list for a specific condition.
$O(2^n)$ and $O(n!)$	**Exponential** and **Fact**	The execution time grow	– **Traveling Sale**

	Factorial Time	grows exponentially or factorially with the input size. Such algorithms become impractical even for moderately large inputs due to their rapid growth rate.	**Salesman Problem (Brute Force)**: Evaluating all possible permutations to find the shortest possible route. – **Recursive Fibonacci**: Calculating Fibonacci numbers using

			g naive recursion without memoization.



In simple terms: we are after the least amount of operations and memory for the largest possible input.

A simple example

Imagine you have a list of numbers and you want to find the largest number in the list. The following snippet will do the trick:

```
def find_largest(arr
    # Get the length
    n = len(arr)
    # Outer loop to
```

```
    for i in range(n
        # Assume the
        is_largest =
        # Inner loop
        for j in ran
            # If any
            if arr[j
                is_l
                brea
        # If the cur
        if is_larges
            return a

numbers = [3, 1, 4,
largest = find_large
print("Largest numbe
```

However, by nesting a second loop, we are effectively increasing the number of operations by a factor of n^2. For a list of 1 million elements, the function could perform up 1000000000000 comparisons. Of course this can be denoted as $O(n^2)$

Complexity

- **Worst-case Time Complexity**: $O(n^2)$. We potentially compare each element with all others.
- **Space Complexity**: $O(1)$. No extra memory allocation.

Let's take a look at the next snippet, an optimised way of achieving the same thing:

```
def find_largest(arr
    # Initialize the
    max_num = arr[0]
    # Loop through t
    for i in range(1
        # If the cur
        if arr[i] >
            max_num
```