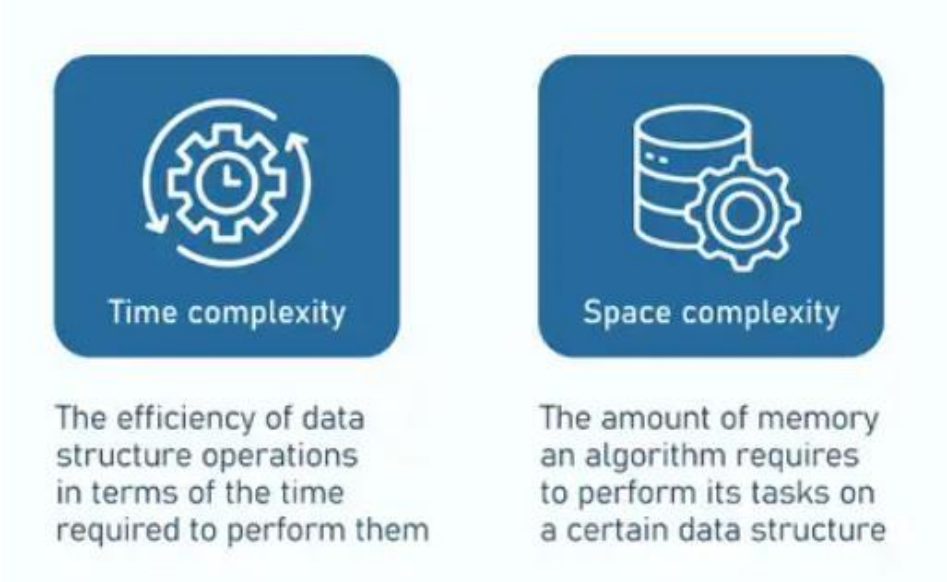
Humans use few basic tricks to reduce the execution time of codes. For example, we avoid loops (FOR) whenever possible, and use list comprehension (like in python) or numpy arrays for iteration. List comprehension offers a more compact, readable, and efficient alternative to loops. We cache parts of the code that compute the same thing several times (recursion), we store and reuse those instead of recomputing. We put checks (if-else) for parts of the code that need to run only some of the time.

By the way, numba is a compiler that makes the runtime of python code fast.

Also, we choose the right data structures which saves us from costly computation.

Data structure is data organized with an aim. It is a collection of data values and their relationships. Data structures serve as frameworks to arrange and store data for specific objectives. The critical characteristics of data structures are time complexity and space complexity.

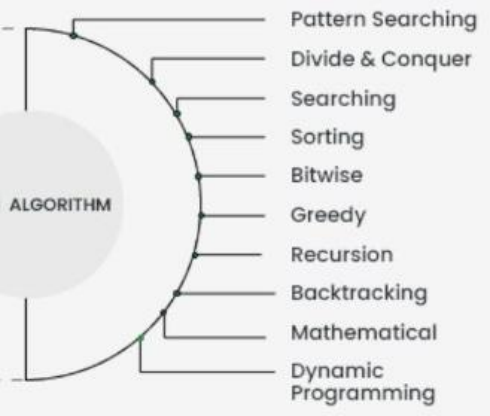


Time complexity helps us understand how the runtime of an operation grows as the input data-size increases.

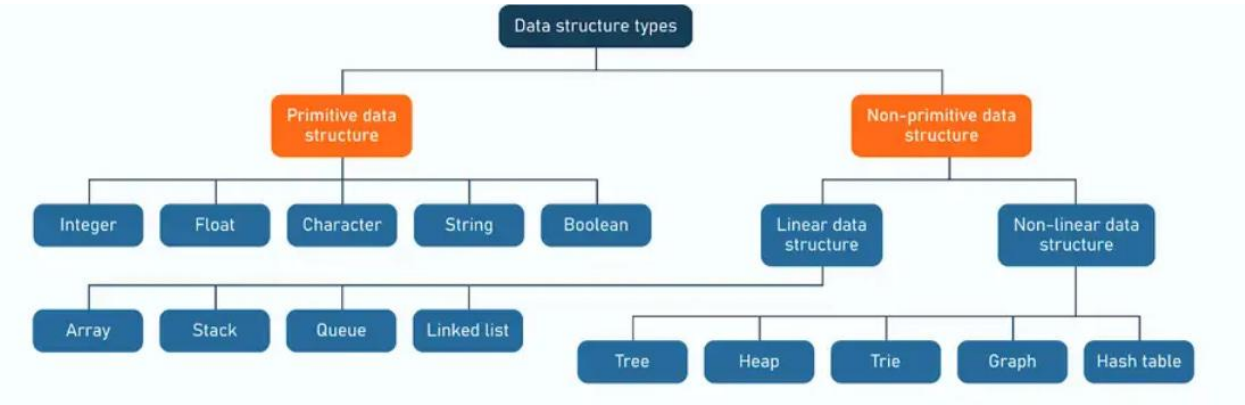
Space complexity helps us understand the memory usage of an algorithm which is particularly important in resource-constrained environments.

Additional space (memory) needs to be allocated for accommodating additional input. Most of the times, there's a trade-off between saving time and saving space depending on the application, for example an app built for web versus mobile, few users versus million users. Another example is, higher tokens in the input fed to a transformer architecture means more memory usage. LLMs using higher number of tokens (higher memory allocation) have higher space complexity and slower processing.

Also, there are different kinds of algorithms. Time complexity is a measure of how the execution time of an algorithm scale with the size of the input data. It is typically expressed using Big O notation.



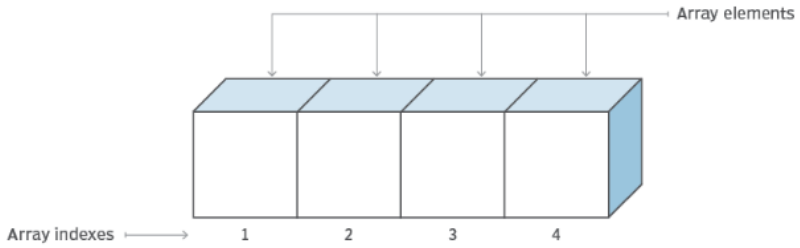
And there are types of data structures meant to serve different purposes. **Linear data structures** as the name suggests are straight-forward to implement, the **nonlinear data structures** are not, but their memory usage is much efficient.



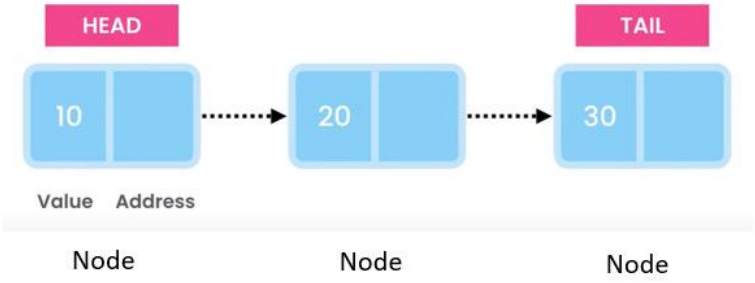
With simple linear data structures like array and linked list, comprehension of the big O notation becomes easier especially in sorting and searching algorithms.

Linear data structures	Non-linear data structures
· Elements are arranged sequentially.	· Elements are arranged in a hierarchical or arbitrary manner.
· An element has one predecessor and successor (except for the first & last).	· Elements can have multiple predecessors or successors.
· Elements are placed at a single level.	· Can have multiple levels.
· Can be traversed in a single run.	· Cannot be traversed in a single run.
· Longer traversal times due to the sequential arrangement of elements.	· Quick traversal due to the hierarchical arrangement of elements.
· Memory usage is less efficient.	· Memory usage is more efficient.

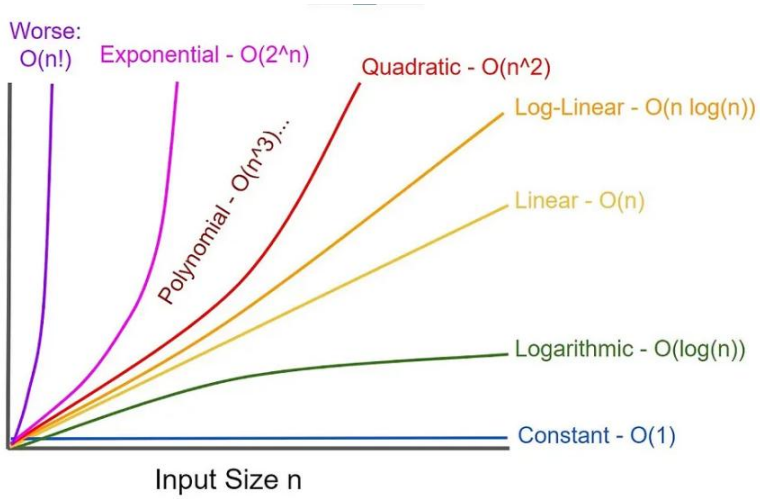
An **array** stores a list of items (e.g. strings, numbers) sequentially. Looking up an array item by its index is fast, but as the array size increases look-up gets costly particularly if the item is at the end of the array. Resizing an array can be costly.



A **linked list** is used when items need to be inserted or removed at a later point in time, as arrays are static (fixed-size memory). Linked list is a dynamic-size data structure. It stores a list of items in sequence and unlike an array, a linked list can grow and shrink automatically. Each node in a linked list points to or references the next node but nodes can be all over the place in memory. Accessing the element of a linked list by its index or value is slow, more so if it's the tail node.



Algorithms use data structures, and the performance of an algorithm depends on the type of data structure used. **O(n) is the run-time complexity of an algorithm, where n is input data-size**. It has a linear or nonlinear growth and tells how scalable the algorithm is.



A single operation like printing a number from an array runs in constant time, that is the complexity is O(1) wherein the array size does not matter. Inserting an item in a linked list has runtime complexity O(1), as it is about creating a node and linking to either the head node or the tail node. Deleting an item from a linked list if it's at the start has complexity O(1) but if it's at the end, the complexity is O(n) as the operation has to traverse the entire list length.

When the method in a single loop iterates over all items in an array and prints them, the algorithm cost increases with the number of operations, in other words increases linearly with the array size. The runtime complexity becomes linear O(n). Both inserting an item in an array and deleting an item from an array have runtime complexity O(n).

For nested loops, the algorithm runs in quadratic time as O(n) for inner loop multiplied by O(n) for outer loop, that is the complexity is O(n^2). Such algorithms are slower than those running in linear time, we see the difference only when the input size 'n' grows larger and larger. If we put another loop inside, the runtime complexity becomes O(n^3) making the method even slower.

An algorithm that runs in logarithmic time O(log n) is more efficient and scalable than an algorithm that runs in linear time, O(n). In linear search algorithms like finding an element from an array, the complexity is linear while binary search algorithms (through sorted array) have logarithmic complexity. Binary searches are faster as the search space is narrowed down by putting a condition/check. In algorithms where we reduce our work by half in every step, the runtime complexity is O(log n).

An algorithm grows faster in exponential time than in quadratic time, as the input size grows, Hence, an algorithm running in exponential time O(2^n) is not scalable. This is the opposite of logarithmic complexity. An algorithm with exponential complexity becomes very slow very soon. The most non-scalable algorithm among all therefore is the one with O(2^n).

The factors that influence time complexity are as follows:

1. Data-size, that is the number of rows (n) in the dataset
2. Data volume, that is the dimensions or number of columns (d) in the dataset
3. Algorithm-specific parameters, like number of iterations

Parametric models (linear regression, naive Bayes) often have high training costs but have low latency or fast inference time. These make them suitable for deployment in real-time.

Non-parametric models like kNN have a costly prediction (inference) phase, as they require comparing new data points to the entire training dataset, on the contrary decision trees can have relatively fast inference phase. Once the tree (or ensemble) is built, making a prediction involves traversing the tree from the root to a leaf node, an operation often with a complexity of O(log n).

Example 1

The time complexity of the kMeans algorithm is typically expressed as O(n*d*k*i), where k is the number of clusters and i is the number of iterations required for the algorithm to converge.

In each iteration, the kMeans algorithm calculates its distance to all k cluster-centroids for every data point. Calculating the distance between a data point and a centroid involves operations across d dimensions. Therefore, assigning all n data points to their closest centroids in a single iteration takes

O(n*d*k) time. The algorithm repeats this operation and centroid update steps for i iterations, until convergence (or a specified maximum number of iterations is reached). That explains O(n*k*d*i).

Example 2

The self-attention mechanism in transformer architecture allows each word to attend to every other word in the sequence, weighing the importance for the current token. If there are 'n' words in the input sequence, LLMs analyse relationships between tokens with an operation that has $O(n^2)$ time complexity. So, the efficiency of self-attention operation is killed with higher number of tokens.