Data Structures and Algorithms

Foundations of Data Structures and Algorithms

A Day in the Life of a MERN Stack Developer

Joe is a software developer at XYZ PVT Ltd. He has been given the task of optimizing the performance of the company's web application. As Joe digs into the project, he realizes the importance of understanding data structures.

In this lesson, we'll follow Joe's journey as he explores the fundamental concepts of data structures, differentiates between data structures and data types, and explains why a solid grasp of data structures is crucial for writing efficient code. Joe will begin by distinguishing between primitive and non-primitive data types, understanding the implications of linear and non-linear structures, and grasping the significance of time and space complexity in algorithm design.

This knowledge will set the stage for Joe's success in tackling future software challenges.

Learning Objectives

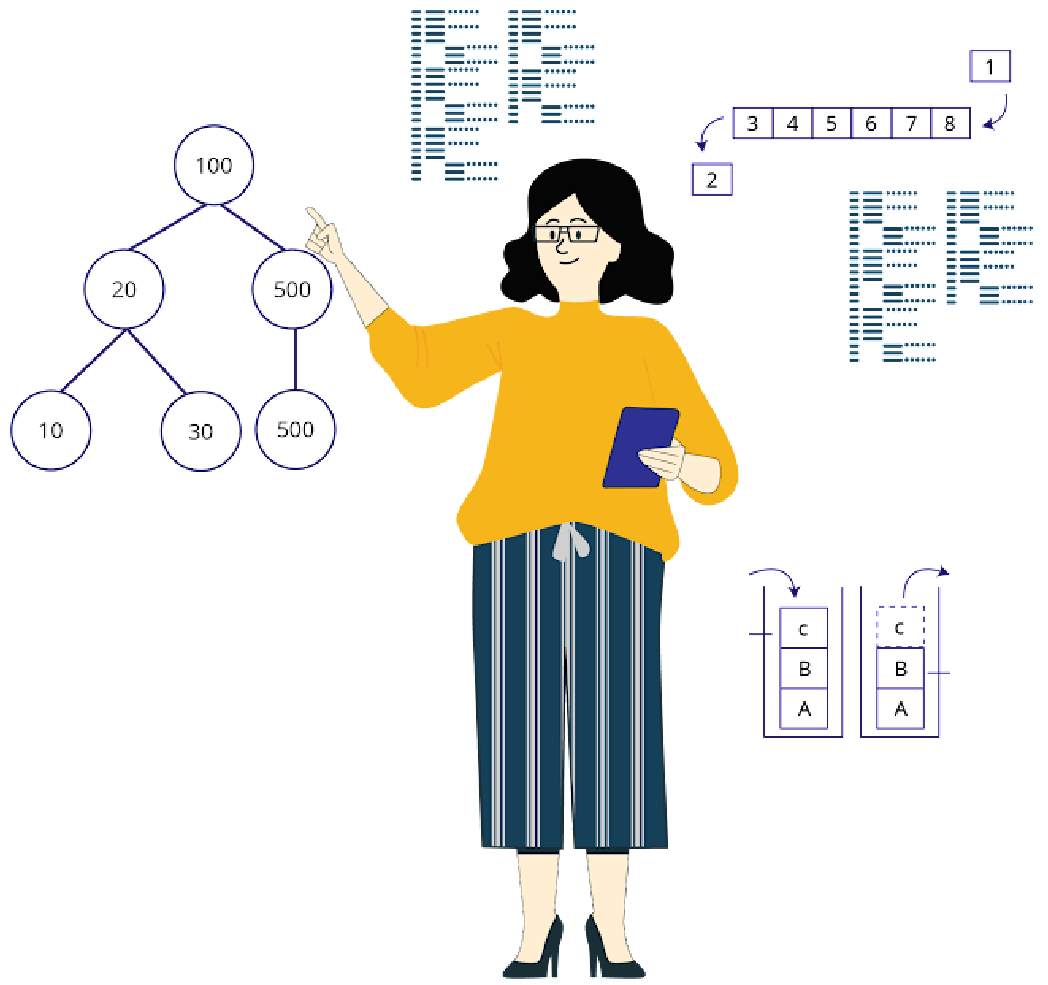
By the end of this lesson, you will be able to:

- Define data structures and data types to manipulate data efficiently
- Understand the practical applications of algorithms to equip them for problem solving
- Illustrate the categories of data structures and data types to facilitate clear decision-making in programming
- Interpret the time and space complexity of different algorithms to assess their performance

Basics of Data Structures

What Are Data Structures?

Data structures are specialized formats for organizing and storing data to perform operations efficiently.



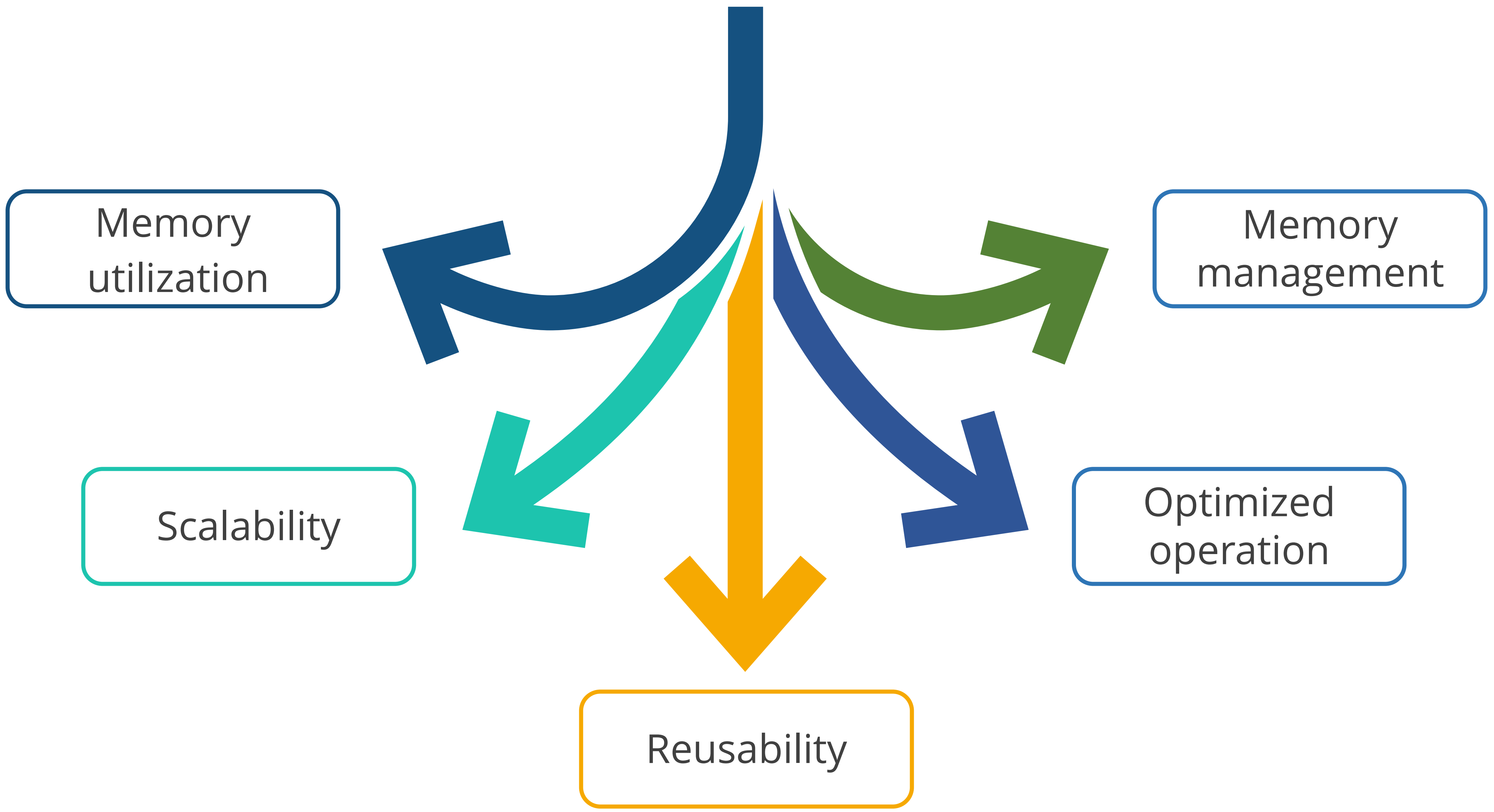
They provide a way to manage and organize data in a computer's memory in such a way that it can be accessed and modified quickly and effectively.

Data Structure vs. Data Type

Data structure	Data type
A data structure is a collection of different kinds of data.	A data type is the kind of variable that is being used throughout the program.
Practical implementation through data structures is called concrete implementation.	Practical implementation through data types is called abstract implementation.
It can hold different kinds and types of data within one single object.	It can hold values and not data, so it is data-less.
Examples: stacks, queues, and tree	Examples: Int, float, and double

Need of Data Structures

Here are some key reasons highlighting the need of data structures:



Need of Data Structures

Memory utilization

Allocation and deallocation of memory minimizes waste and maximizes resources.

Optimized operation

Different data structures are designed to optimize specific operations.

Reusability

Improve development efficiency, maintainability, and code quality.

Memory management

Crucial for efficient use of resorces and avoiding issues like memory leaks.

Scalability

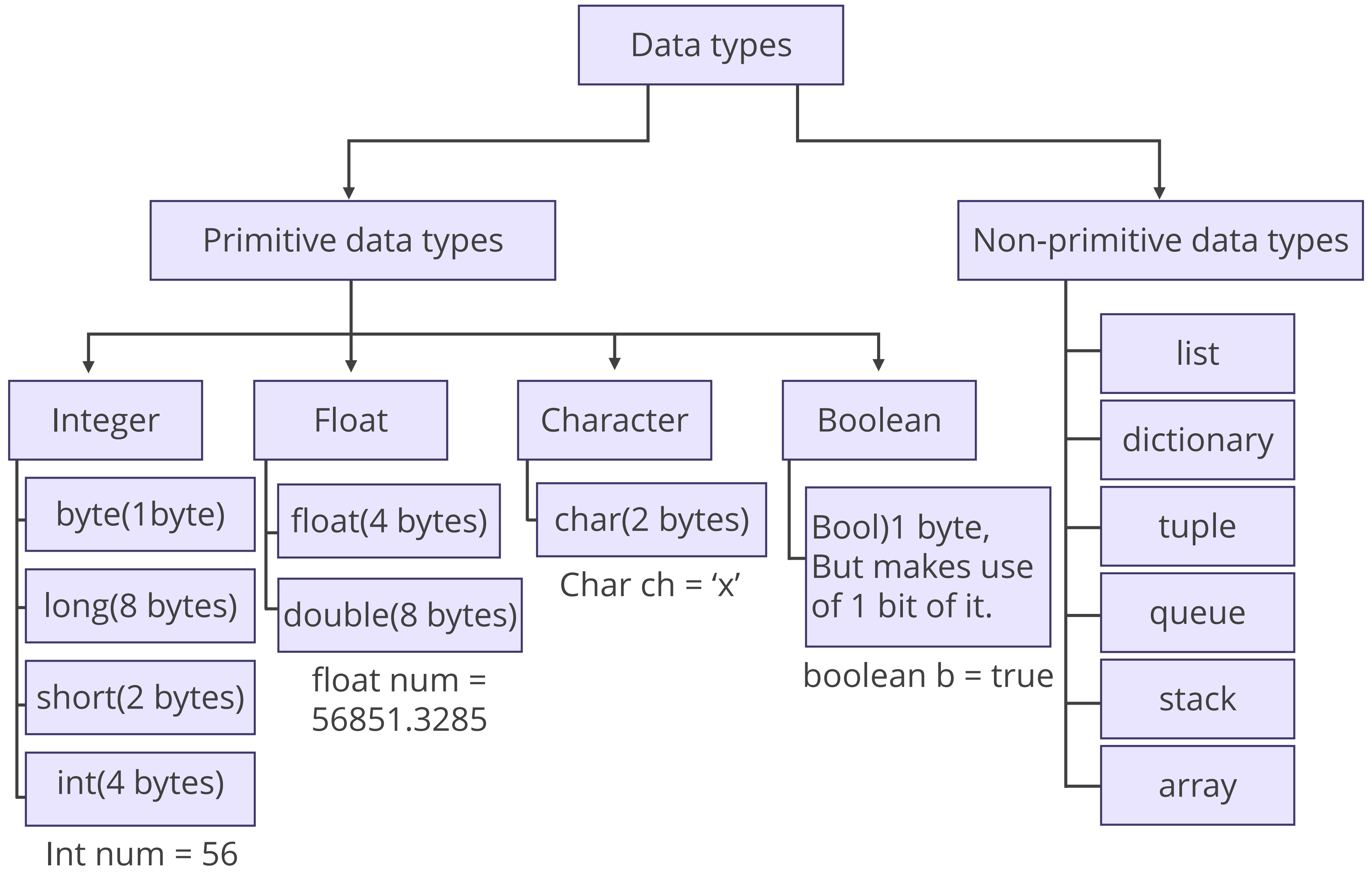
Scalable data structures help in managing and processing large data sets efficiently.

Classification of Data Types

Primitive and Non-Primitive

Primitive	Non-primitive
• They are the most basic data types directly supported by programing languages. • It allows to store only one single data type values. • Size depends on the type of data structure. • Example: Boolean, character, integer, float	• They are more complex and composed of primitive data types. • It allows to store multiple data types and values. • Size is not fixed in non-primitive data structure. • Example: Array, list, stack, and queue

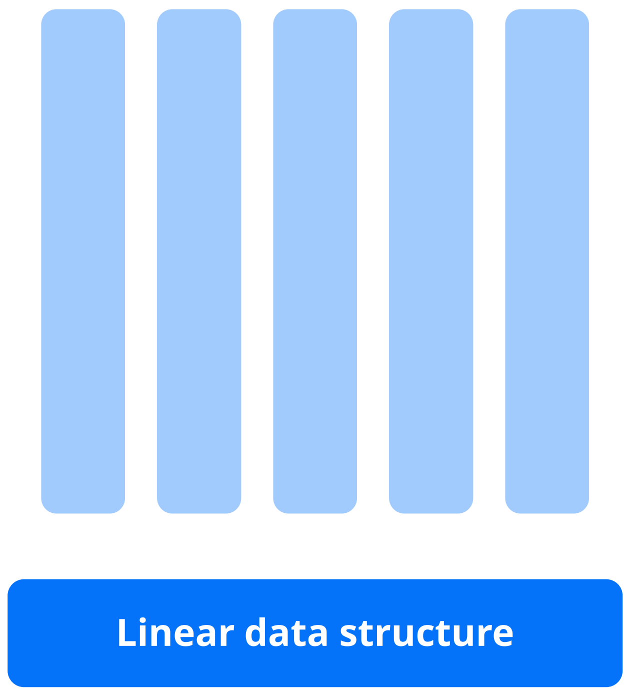
Data Types: Categorization



Classification of Data Structures

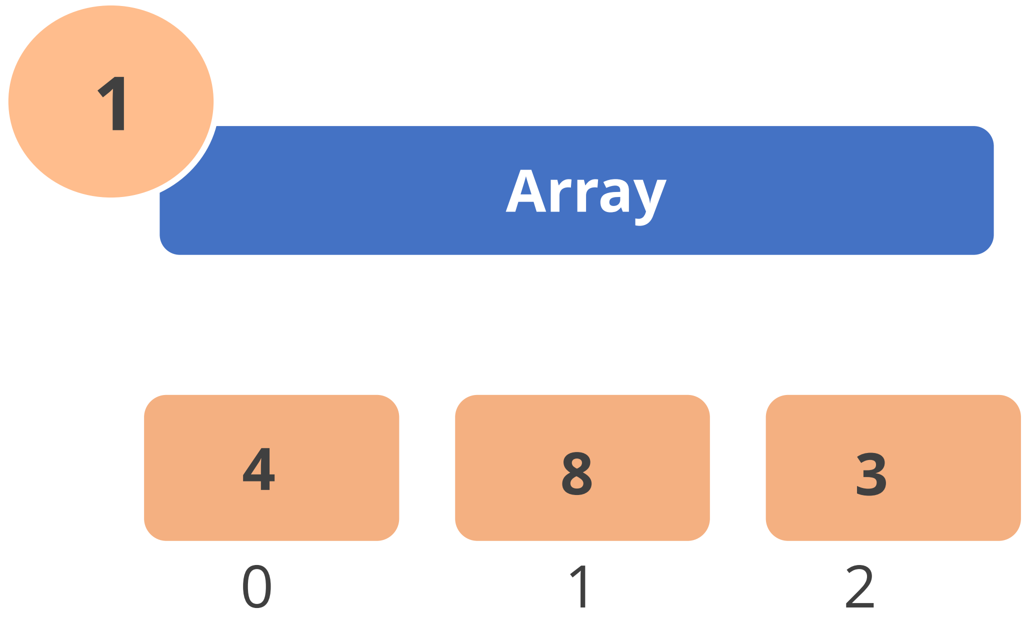
Linear Data Structure

A linear data structure is a sequence of elements where each element is connected to its previous and next adjacent elements. Operations in linear data structures are generally performed in a sequential manner.



Examples: Arrays, linked list, stacks, and queue

Linear Data Structure

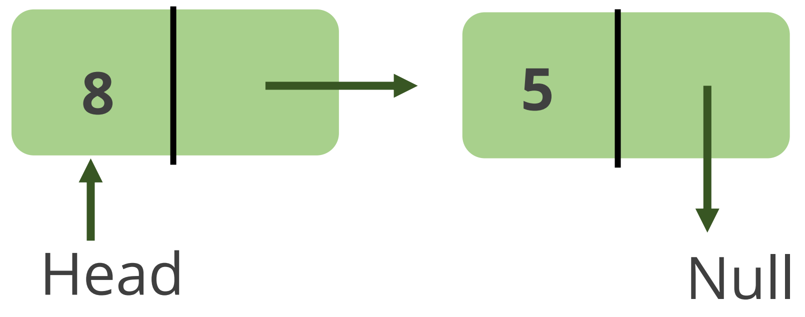


An array is a fixed-size data structure that stores elements of the same data type at contiguous memory locations, allowing random access to elements using indices.

Linear Data Structure

2

Linked list

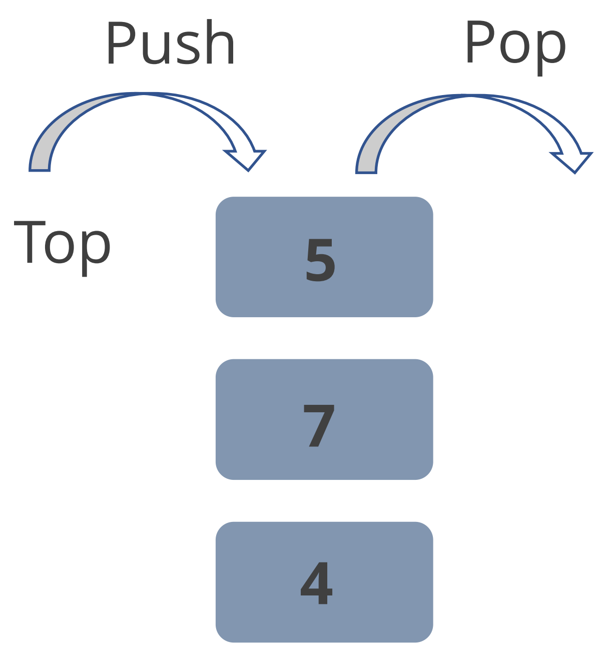


A linked list is a dynamic data structure consisting of nodes, each containing data and a reference to the next node, allowing for sequential access to its elements.

Linear Data Structure

3

Stack

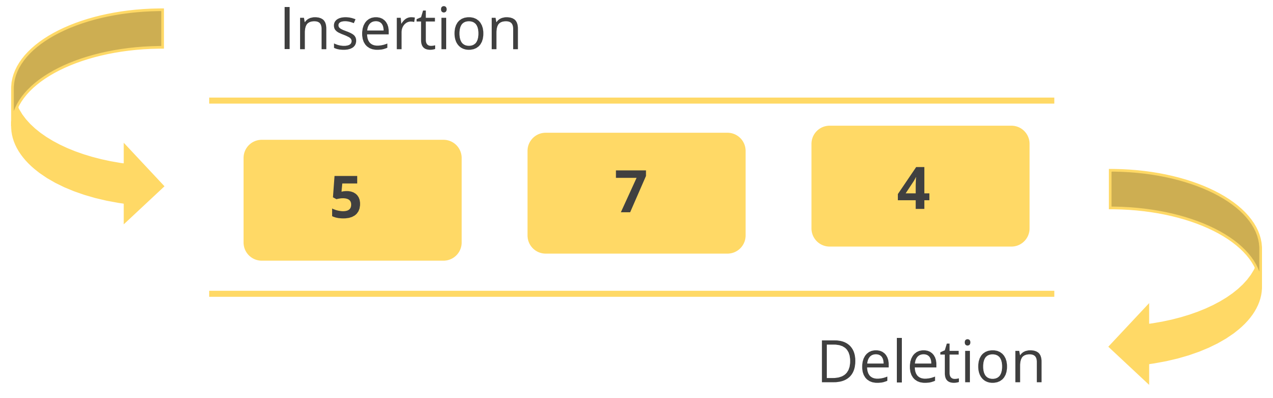


A stack is a linear data structure that follows the LIFO (last-in-first-out) principle, where elements are added and removed from the top.

Linear Data Structure

4

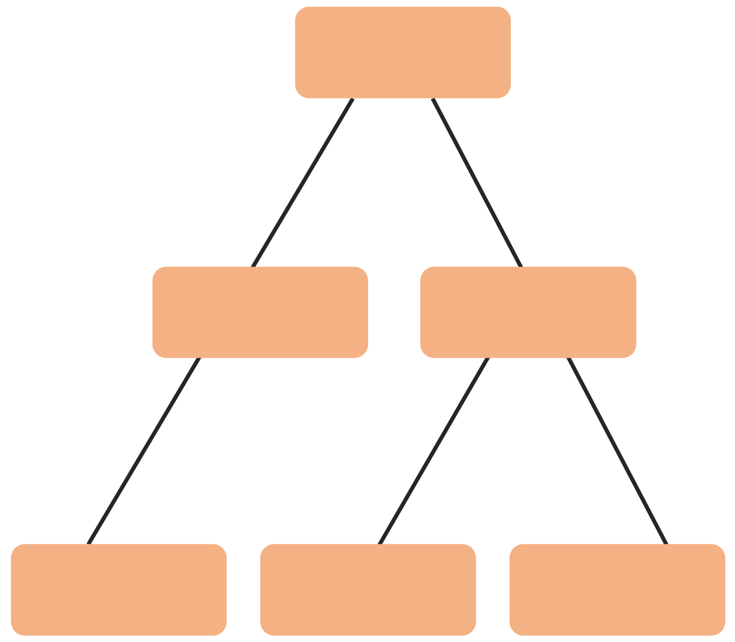
Queue



A queue is a linear data structure that follows the FIFO (first-in-first-out) principle, with elements added to the back and removed from the front.

Non-Linear Data Structure

Non-linear data structures are those in which data elements are not arranged sequentially. Instead, they are organized hierarchically or in a multi-level manner, allowing one element to be connected to one or more other elements.



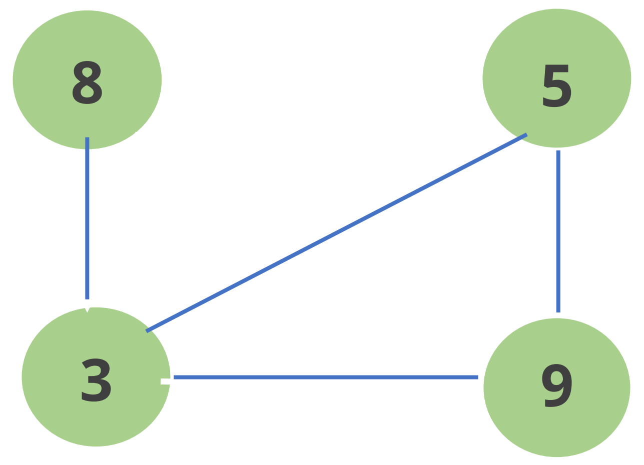
Non-linear data structure

Examples: Trees and graphs

Non-Linear Data Structure

1

Graph

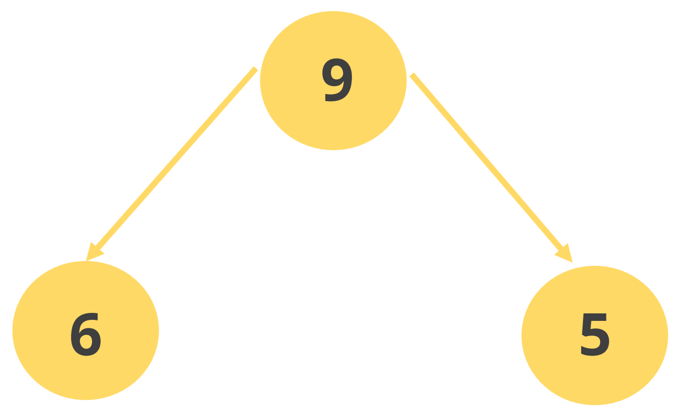


A graph is a non-linear data structure composed of nodes and edges. The nodes, also known as vertices, are connected by edges, which are lines or arcs.

Non-Linear Data Structure

2

Tree



A tree, like a graph, is a collection of vertices and edges. However, in a tree data structure, there can only be one edge between two vertices.

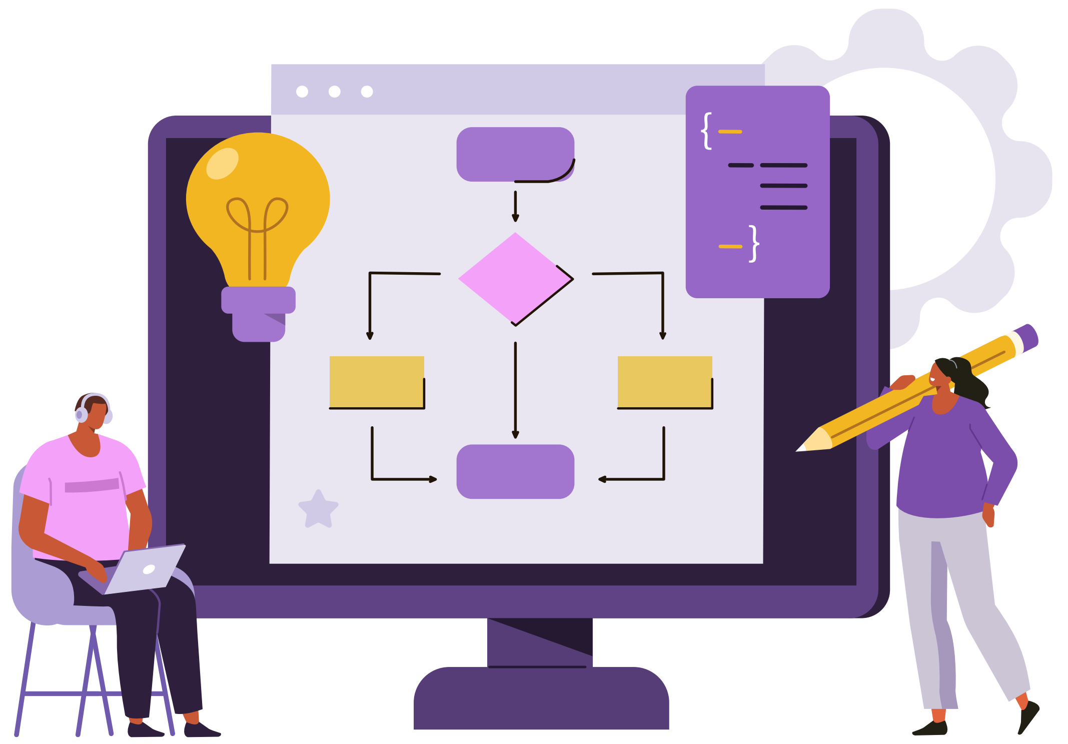
Comparison of Linear and Non-Linear

	Linear	Non-linear
Arrangement	Data elements are arranged sequentially	Data elements have a hierarchal structure
Traversal	Can traverse it in a single run	Need multiple runs to traverse
Complexity of time	Simpler in structure and organization	Can represent more complex relationships and dependencies
Application	Suitable for scenarios where elements follow a specific order or priority	Suitable for scenarios where relationships and connections between elements are crucial

Introduction to Algorithms

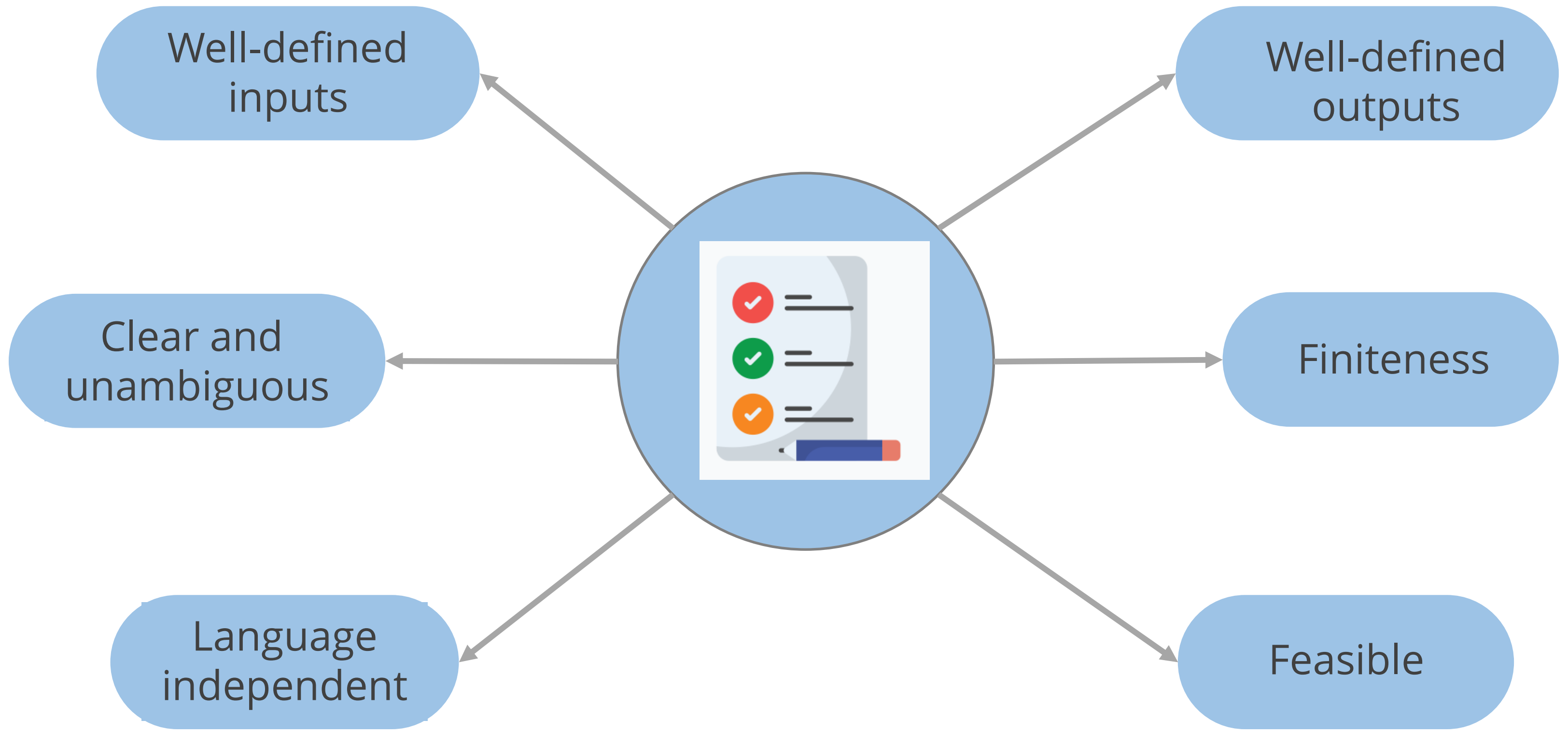
What Is an Algorithm?

It is a set of well-defined instructions or a step-by-step procedure designed to perform a specific task or solve a particular problem.



Characteristics of Algorithms

Algorithms are fundamental qualities that define their functionality and effectiveness. Here are the key characteristics:



JavaScript algorithms integrate standard algorithmic design principles, adapted to align with the distinct characteristics and application contexts of the JavaScript language.

Importance of an Algorithm

Their importance can be highlighted in several key areas, such as:

01 Problem solving

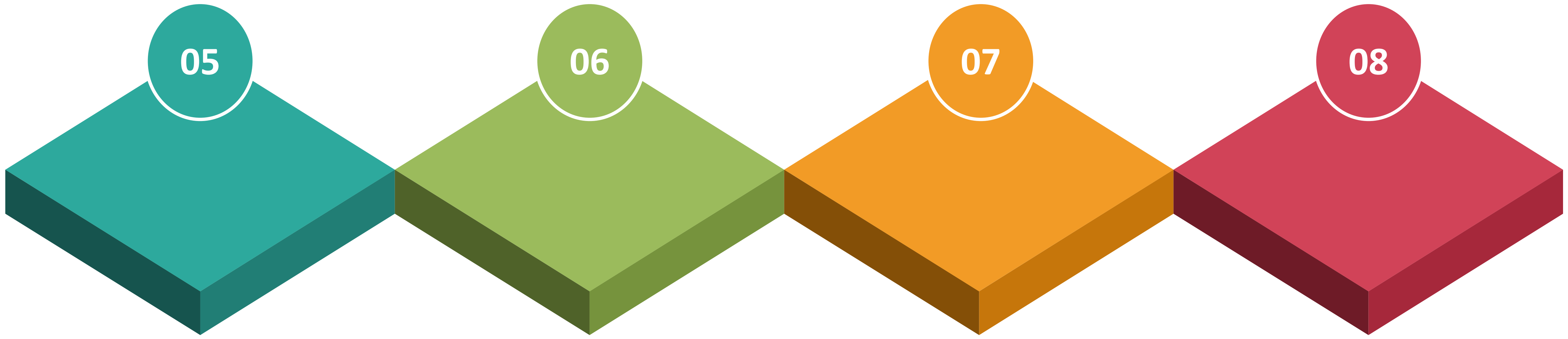
02 Efficiency and performance

03 Consistency and reliability

04 Automation

Importance of an Algorithm

It plays a critical role in various aspects of computing and problem-solving. Their importance can be highlighted in several key areas:



Data process

Foundation for software development

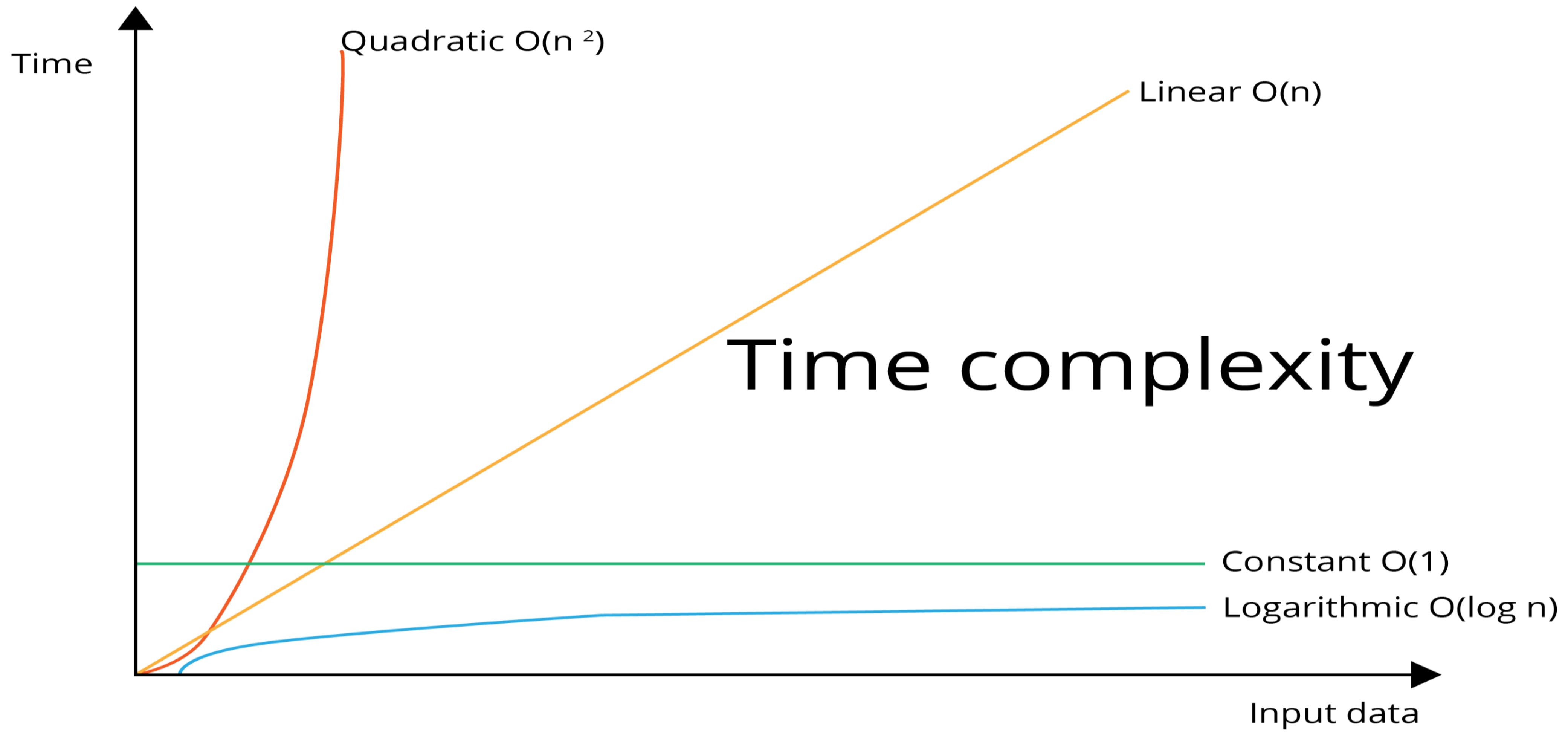
Technological innovation

Scalability

Algorithms are crucial in the digital domain, forming the core components of computer science, information technology, and the progress of digital advancements.

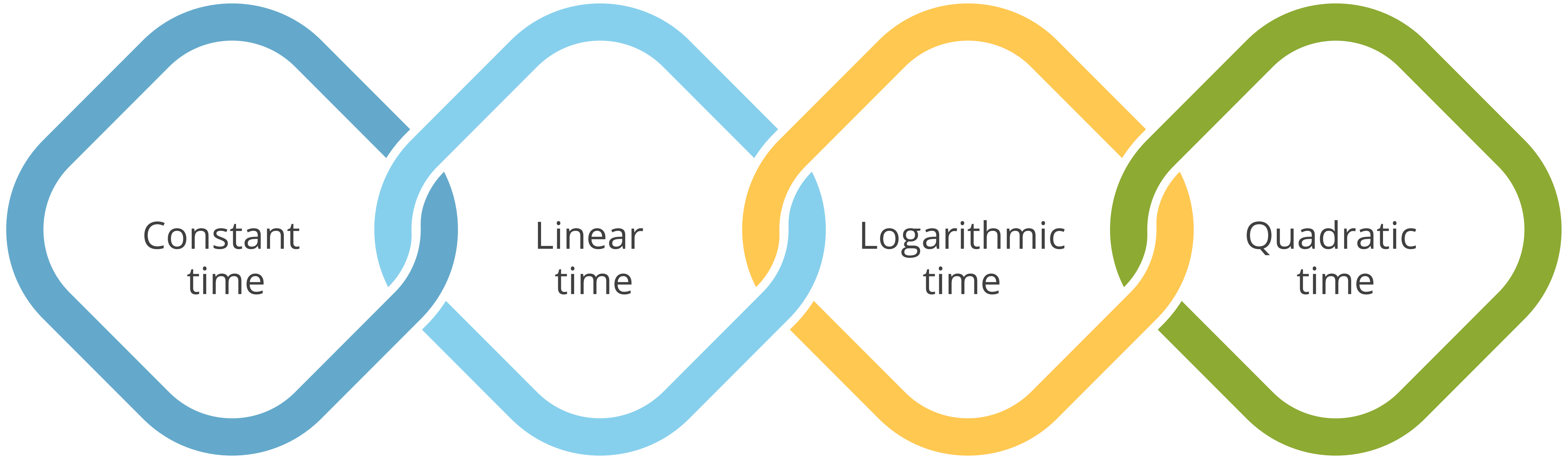
Time Complexity of an Algorithm

What Is Time Complexity?



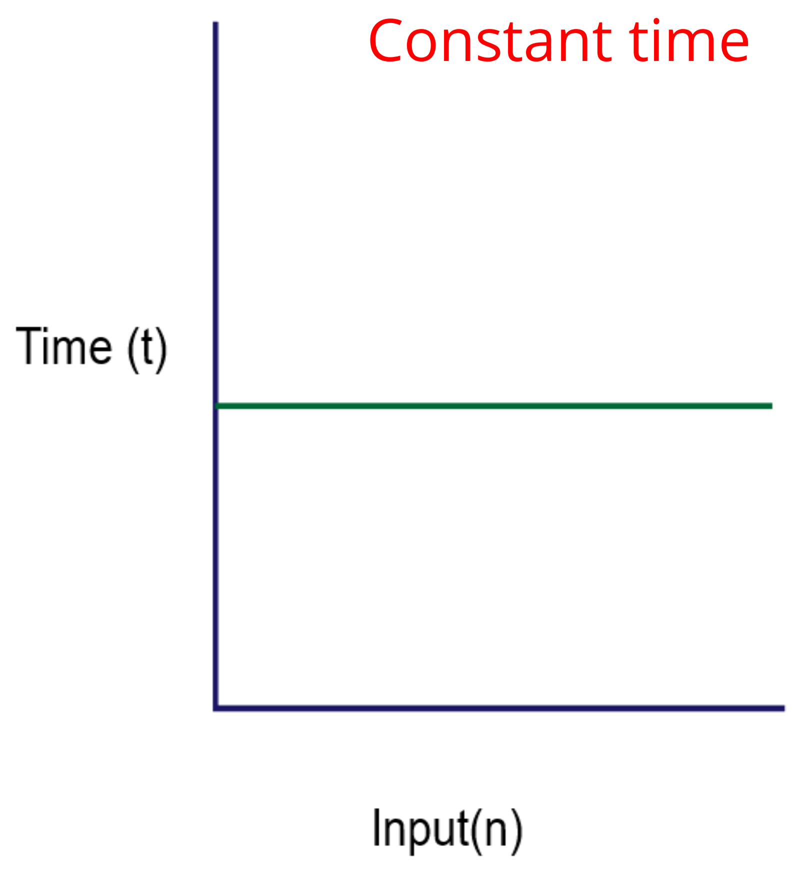
Time complexity of an algorithm is the amount of time it takes to run as a function of input. It measures the time taken to execute each statement of code in an algorithm.

Types of Time Complexity



O(1): Constant Time Complexity

Constant time complexity, often denoted as O(1), means that the execution time of an algorithm or operation remains constant, regardless of the size of the input data.



It is desirable for operations with a fixed number of steps.

As the input size increases, the time taken to perform the operation remains the same.

Calculation: Constant Time Complexity

Some examples of constant time complexity in JavaScript:

Accessing elements in arrays

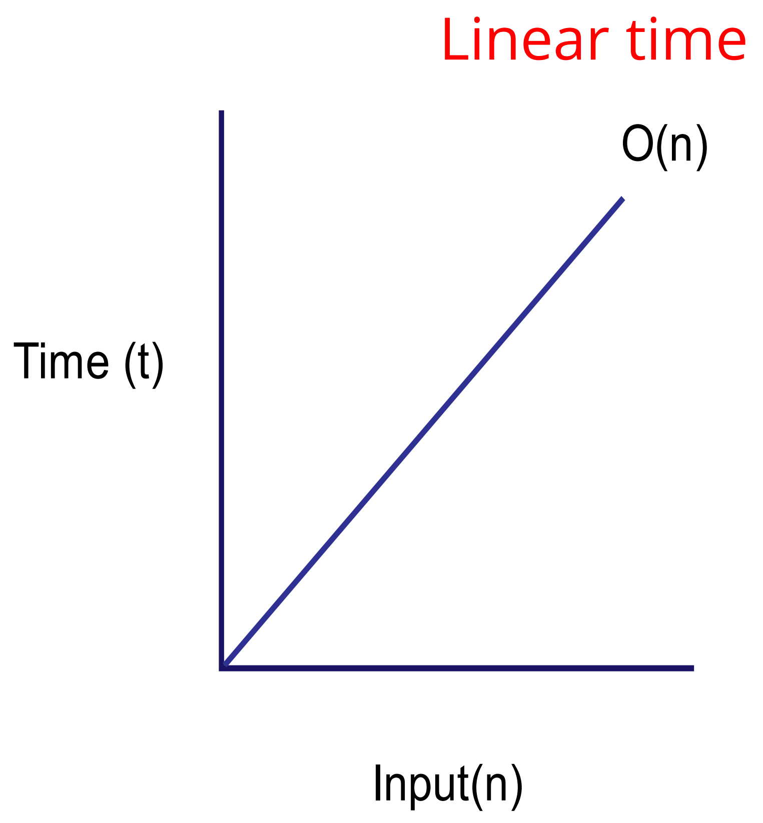
Addition of two elements

```
const myArray  [1, 2, 3, 4, 5];
const element = myArray[2];
// Accessing element at index 2
```

```
function exampleDataStructure(n) {
  let stack = []; // O(1) space for the
stack reference
  for (let i = 0; i < n; i++) {
    stack.push(i); // O(n) space for the
elements in the stack
  }
}
```

O(N): Linear Time Complexity

Linear time complexity (O(n)) in JavaScript refers to algorithms whose execution time grows linearly with the size of the input data.



It is efficient for sorting algorithms.

As the input size (n) increases, the time taken to perform the operation increases linearly.

Calculation: Linear Time Complexity

Some examples to linear time complexity in JavaScript:

Iteration of array

Linear search

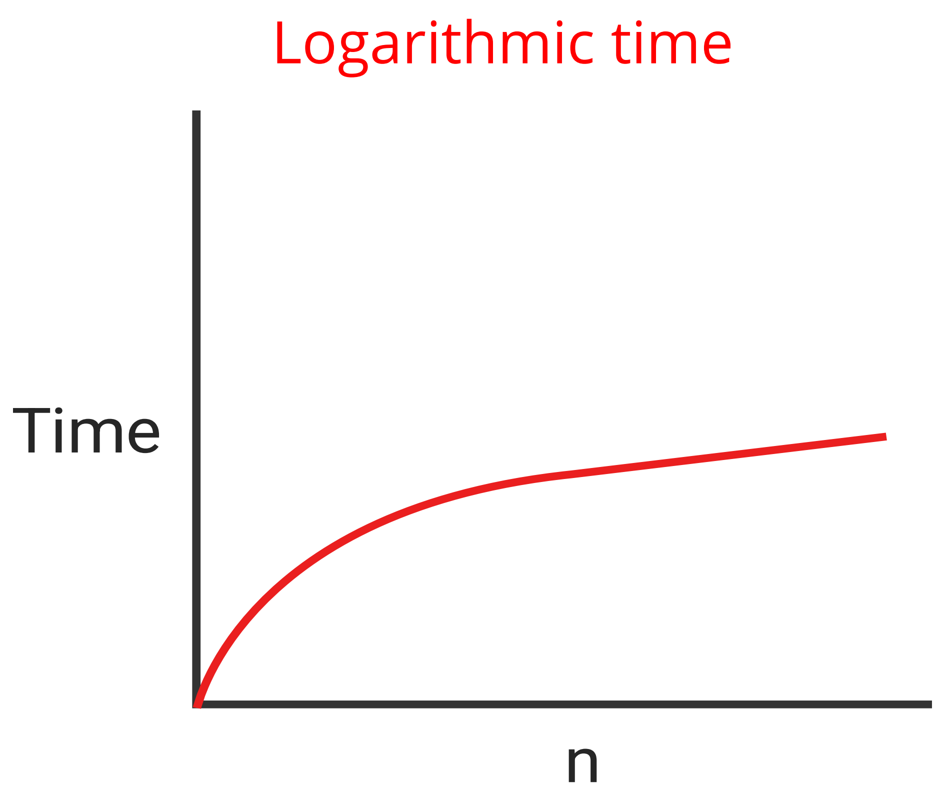
```
const myArray = [1, 2, 3, 4, 5];

for (let i = 0; i < myArray.length;
i++) {
  // Linear time complexity operation
for each element
  console.log(myArray[i]);
}
```

```
function linearSearch(array, target) {
  for (let i = 0; i < array.length; i++)
{
    if (array[i] === target) {
      return i; // Element found
    }
  }
  return -1; // Element not found
}
```

O(log N): Logarithmic Time Complexity

Logarithmic time complexity (O(log n)) often involves algorithms or operations that efficiently reduce the size of the problem by a constant factor with each step.



It is efficient for large datasets.

As the input size (n) increases, the time taken to perform the operation increases logarithmically.

Calculation: Logarithmic Time Complexity

Logarithmic time complexity using binary search:

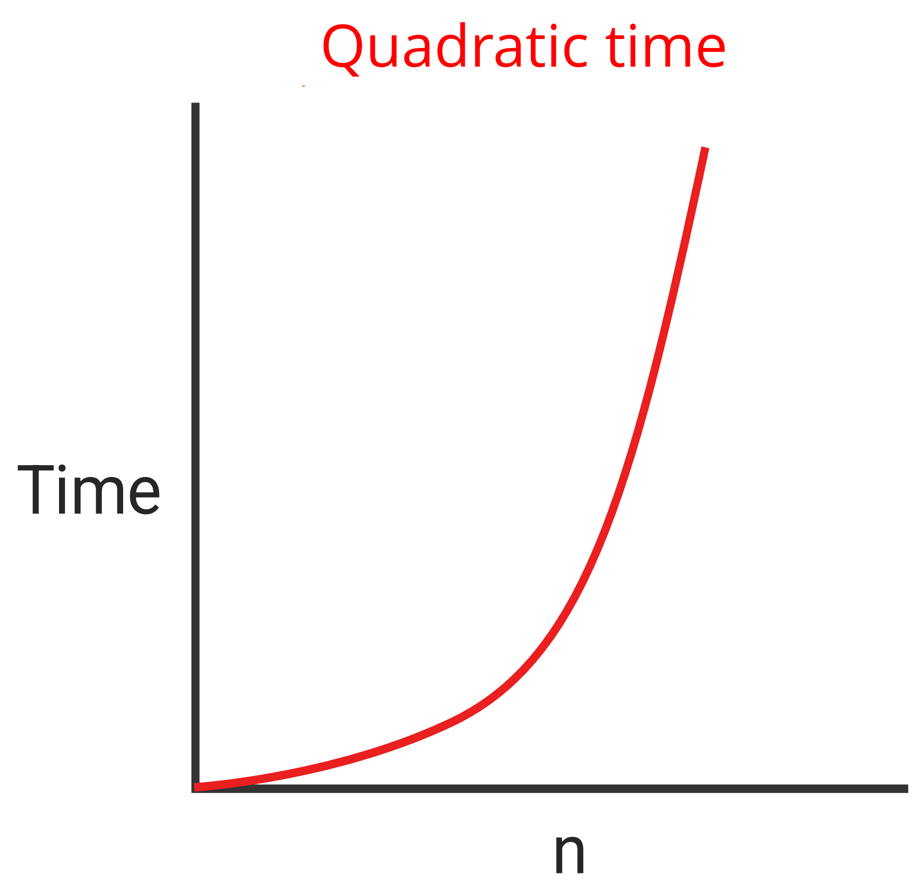
```
function binarySearch(sortedArray, target) {
  let low = 0;
  let high = sortedArray.length - 1;

  while (low <= high) {
    const mid = Math.floor((low + high) / 2);
    if (sortedArray[mid] === target) {
      return mid; // Element found
    } else if (sortedArray[mid] < target) {
      low = mid + 1; // Search in the right half
    } else {
      high = mid - 1; // Search in the left half
    }
  }

  return -1; // Element not found
```

O(N^2): Quadratic Time Complexity

Quadratic time complexity (O(n^2)) in JavaScript refers to algorithms or operations whose execution time grows quadratically with the size of the input data.



It is efficient for small datasets.

As the input size (n) increases, the time taken to perform the operation increases quadratically.

Calculation: Quadratic Time Complexity

Quadratic time complexity using bubble sort:

```
function bubbleSort(array) {
  const n = array.length;

  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - 1 - i; j++) {
      if (array[j] > array[j + 1]) {
   // Swap elements if they are in the wrong order
        const temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
    }
  }
```

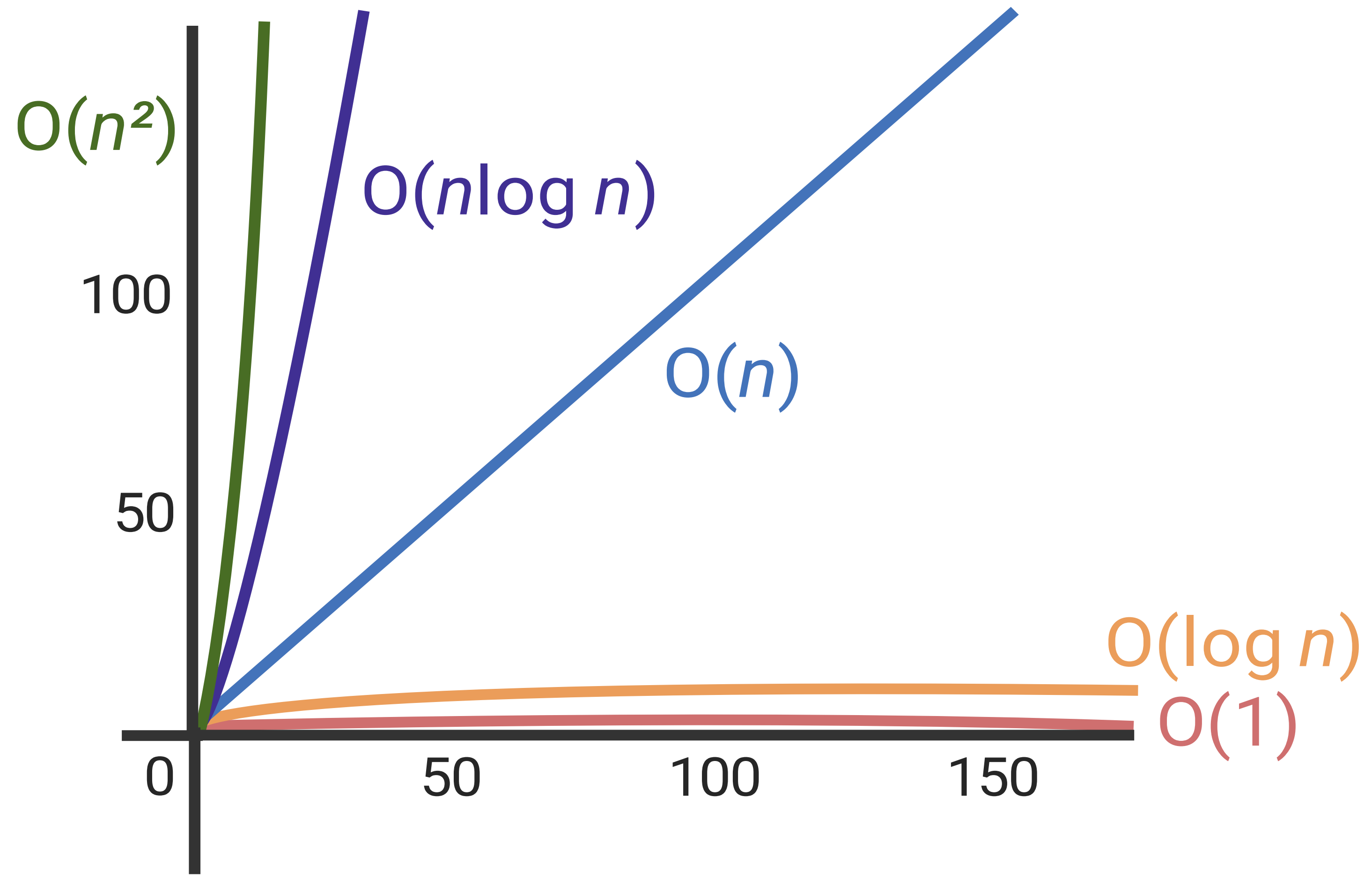
Calculation: Quadratic Time Complexity

Quadratic time complexity using nested loop:

```
function quadraticExample(array) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      // Quadratic time complexity operation
      console.log(array[i], array[j]);
    }
  }
}
```

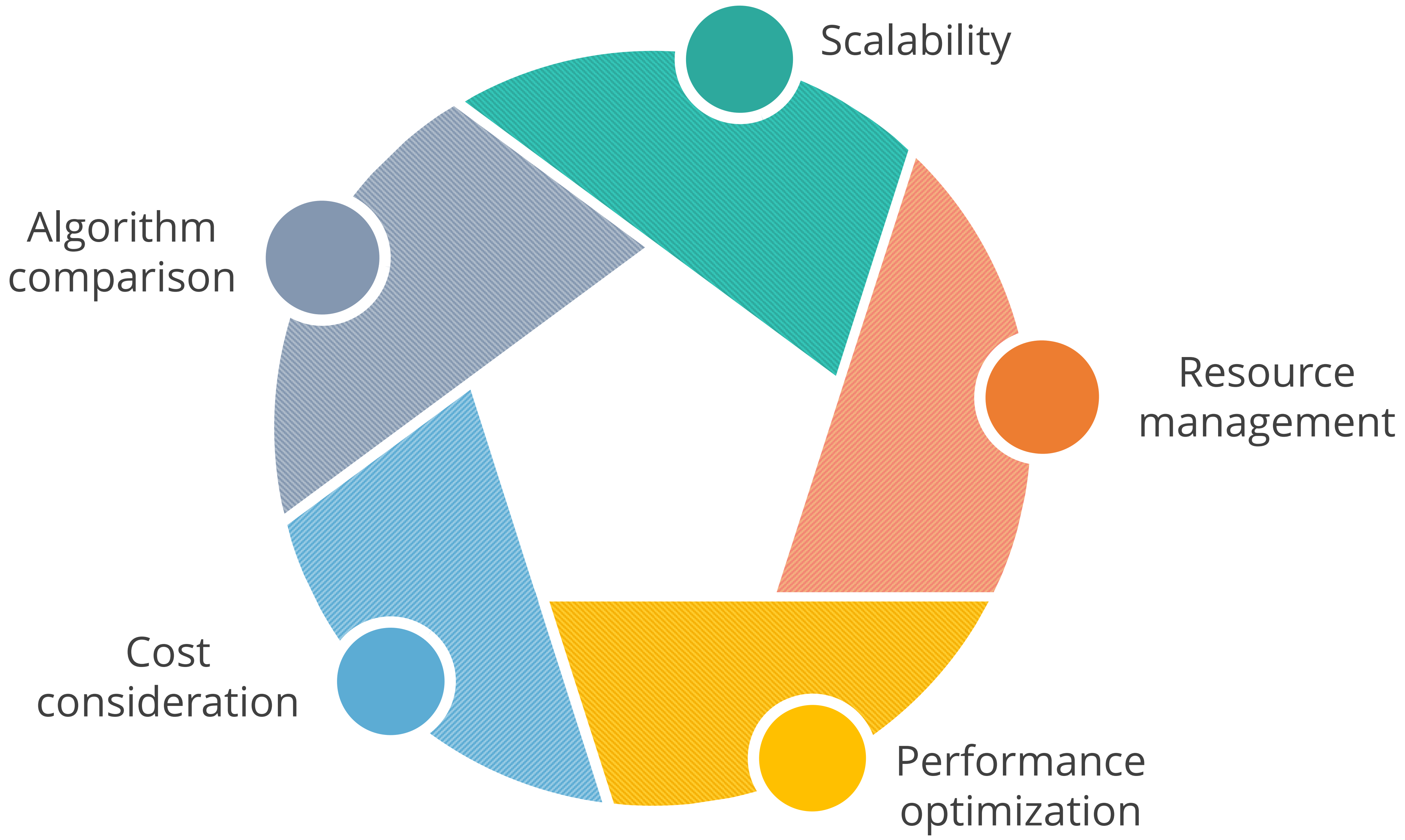
Space Complexity of an Algorithm

What Is Space Complexity?



Space complexity in data structures refers to the amount of memory an algorithm uses to solve a problem. It measures the memory space required to store the data and structures used by the algorithm.

Need of Space Complexity



Types of Space Complexity

Constant space O(1)	Algorithms with constant space complexity use a fixed amount of memory, regardless of the input size.
Linear space O(n)	It indicates that the amount of memory used by the algorithm grows linearly with the size of the input.
Logarithmic space O(log n)	Logarithmic space complexity increase their memory usage by a logarithmic factor as the input size grows.

Types of Space Complexity

Polynomial space O(n^k)	Memory requirements of an algorithm grow as a polynomial function of the input size.
Exponential space O(2^n)	Memory used by the algorithm doubles (or grows exponentially) with each additional element in the input.
Quasilinear space O(n log n)	Quasilinear space complexity is a combination of linear and logarithmic growth.

Calculation: Space Complexity

Some examples to calculate space complexity in JavaScript:

Arrays and objects

```
function exampleArray(n) {
  let arr = []; // O(1) space for the
array reference
  for (let i = 0; i < n; i++) {
    arr.push(i); // O(n) space for the
elements in the array
  }
}
```

Data structures

```
function exampleDataStructure(n) {
  let stack = []; // O(1) space for the
stack reference
  for (let i = 0; i < n; i++) {
    stack.push(i); // O(n) space for the
elements in the stack
  }
}
```

Key Takeaways

- A data structure is a way of organizing and storing data to perform operations efficiently.
- Data types are broadly categorized into primitive and non-primitive.
- Time complexity is a measure of the amount of time an algorithm takes to complete based on the input size.
- Space complexity is a measure of the amount of memory an algorithm requires relative to the input size.

Thank You