## What is an Algorithm?

An **algorithm** is a finite, step-by-step set of instructions or rules designed to solve a specific problem. Think of it as a recipe 🧑‍🍳:

* **Problem:** You want to bake a cake.
* **Inputs:** Ingredients (flour, eggs, sugar).
* **Algorithm:** The recipe's instructions (mix flour and eggs, preheat oven, bake for 30 minutes).
* **Output:** The finished cake.

In computer science, an algorithm takes some input, follows a set of defined steps, and produces an output that solves the problem.

## 1. Algorithm Design

**Algorithm Design** is the creative process of developing this step-by-step procedure. It's about finding a *correct* and *efficient* way to solve the problem. There are several standard "paradigms" or techniques for designing algorithms:

### Common Design Paradigms

* **Divide and Conquer:**
  + **Idea:** Break a large problem into smaller, similar subproblems. Solve the subproblems (often recursively), and then combine their solutions to get the final answer.
  + **Example:** **Merge Sort**. To sort a list, you divide it in half, sort each half, and then merge the two sorted halves back together.
* **Greedy Algorithms:**
  + **Idea:** Make the choice that looks best *at the current moment* in the hope that these local "best" choices will lead to a global "best" solution.
  + **Example:** **Dijkstra's Algorithm** for finding the shortest path. At each step, it greedily picks the next unvisited node that is closest to the start.
* **Dynamic Programming (DP):**
  + **Idea:** Break a complex problem into simpler, overlapping subproblems. Solve each subproblem *only once* and store its solution (a process called "memoization"). When the same subproblem arises again, you just look up the saved answer instead of re-calculating it.
  + **Example:** Calculating the **Fibonacci sequence**. To find $F(5)$, you need $F(4)$ and $F(3)$. To find $F(4)$, you need $F(3)$ and $F(2)$. Instead of re-calculating $F(3)$ twice, you calculate it once, save it, and reuse the answer.
* **Backtracking:**
  + **Idea:** Build a solution incrementally, one piece at a time. If you hit a point where the current path can't lead to a valid solution, you "backtrack" (undo the last choice) and try a different path.
  + **Example:** Solving a **Sudoku puzzle**. You place a number in a cell, then move to the next. If you get stuck, you go back to the previous cell and try a different number.
* **Brute Force:**
  + **Idea:** The simplest approach. Try every single possible solution until you find the right one.
  + **Example:** Trying to guess a 4-digit PIN. You would try 0000, 0001, 0002, and so on. It's guaranteed to work but is often extremely slow.

## 2. Algorithm Analysis

**Algorithm Analysis** is the process of measuring an algorithm's performance. It's not about how "smart" the algorithm is, but about how *efficient* it is. We primarily measure two things:

1. **Time Complexity:** How much time does the algorithm take to run?
2. **Space Complexity:** How much memory (RAM) does the algorithm need?

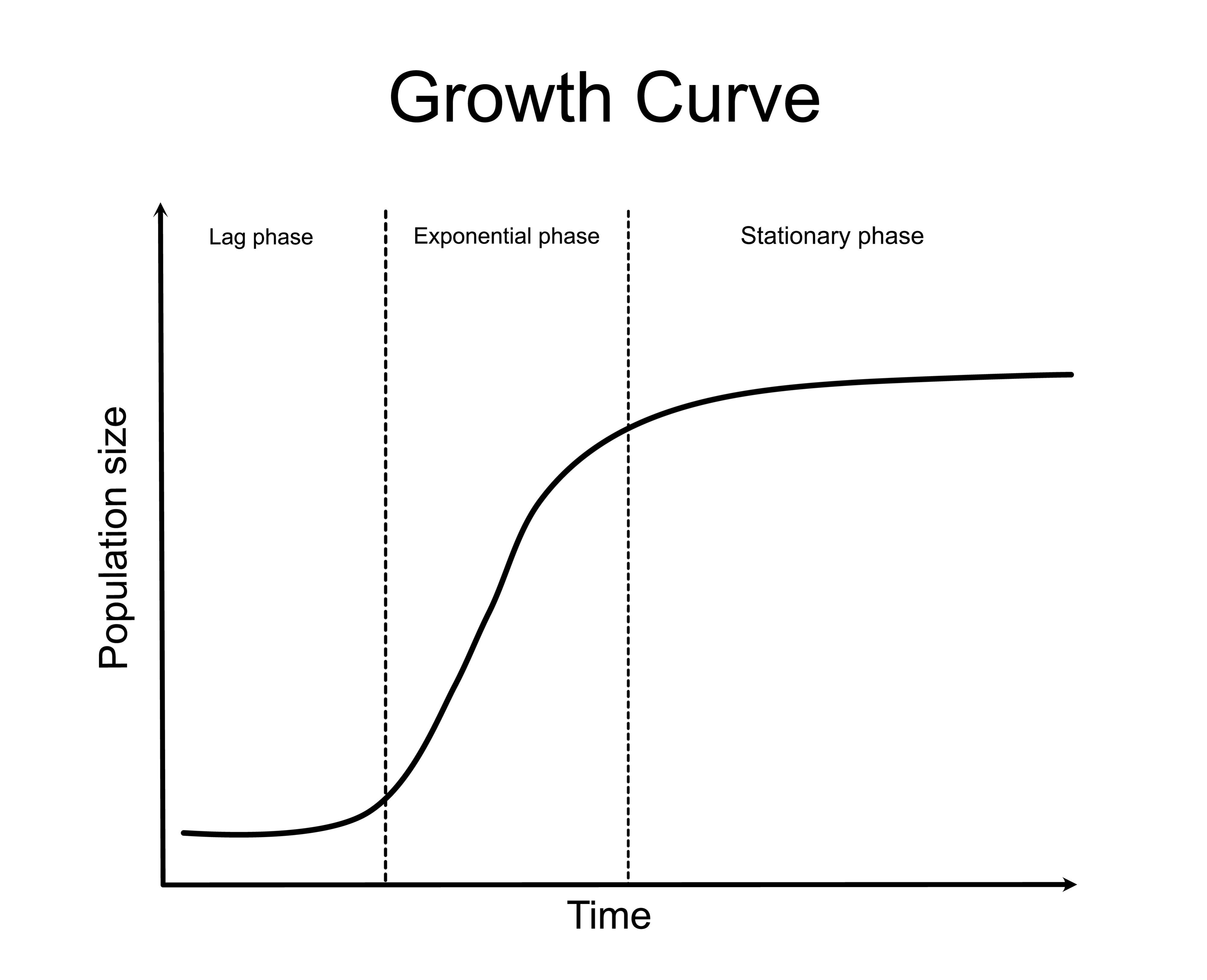
We don't measure time in seconds, because that depends on the computer's speed. Instead, we measure it in terms of **how the runtime grows as the input size ($n$) increases.**

### Asymptotic Notation (Big O)

We use **Big O notation ($O$)** to describe an algorithm's **worst-case** time complexity. It tells us the upper bound of its growth rate.

Here are the most common complexities, from fastest to slowest:

* **$O(1)$ — Constant Time:**
  + **Meaning:** The algorithm takes the same amount of time regardless of the input size.
  + **Example:** Accessing an element in an array by its index (e.g., my\_list[5]).
* **$O(\log n)$ — Logarithmic Time:**
  + **Meaning:** The time increases very slowly as the input size grows. Doubling the input size adds only one "step."
  + **Example:** **Binary Search**.



Shutterstock

* **$O(n)$ — Linear Time:**
  + **Meaning:** The runtime grows in direct proportion to the input size. Doubling the input size doubles the time.
  + **Example:** Finding the largest number in an unsorted list (you have to look at every element).
* **$O(n \log n)$ — Log-Linear Time:**
  + **Meaning:** A very efficient "sweet spot" for sorting algorithms.
  + **Example:** **Merge Sort** or **Quick Sort** (on average).
* **$O(n^2)$ — Quadratic Time:**
  + **Meaning:** The runtime grows quadratically. If the input size is 10, the time is ~100. If the input is 100, the time is ~10,000.
  + **Example:** A simple sorting algorithm like **Bubble Sort** (which uses nested loops).
* **$O(2^n)$ — Exponential Time:**
  + **Meaning:** The runtime doubles with *each new element* added to the input. This is extremely slow and only practical for very small input sizes.
  + **Example:** Brute-force solutions to problems like the Traveling Salesman.

## Why Does This Matter?

The design and analysis of algorithms are at the heart of computer science.

Imagine you have a dataset with 1 million items to sort.

* An $O(n^2)$ algorithm (like Bubble Sort) would take roughly $1,000,000^2 = 1,000,000,000,000$ operations. This could take **days**.
* An $O(n \log n)$ algorithm (like Merge Sort) would take roughly $1,000,000 \times 20 = 20,000,000$ operations. This would take **less than a second**.

Choosing the right algorithm is the difference between an application that is fast and scalable and one that is completely unusable.