Section 2: DECOMPOSING the Fibonacci PROBLEM INTO COMPUTATIONAL SUBPROBLEMS

🔍 2.1. WHY DECOMPOSE THE PROBLEM?

When confronted with computational challenges of any significant complexity—particularly one as elegantly structured as the Fibonacci sequence—it is tremendously advantageous to employ the technique of **problem decomposition**. This foundational approach in computer science involves breaking a substantial problem into smaller, more manageable components or subproblems.

For the Fibonacci sequence, decomposition is particularly illuminating because it reveals the inherent recursive structure that defines the sequence itself. By systematically dissecting the problem, students gain invaluable insights into:

- **Algorithmic thinking**: How complex computations can be approached in a methodical, step-by-step manner rather than attempting to solve everything at once.
- **Solution reusability**: How results from smaller calculations can be stored and reused, avoiding redundant work—a principle that underpins many advanced optimisation techniques.
- **Natural algorithm emergence**: How the mathematical definition of Fibonacci naturally suggests either recursive or iterative computational approaches.
- **Computational efficiency**: How naïve implementations can quickly become prohibitively expensive in terms of processing time and memory usage.

The process of decomposition serves as an excellent pedagogical tool, providing a concrete example of how theoretical mathematical concepts translate into practical computational solutions. This approach lays crucial groundwork for understanding more sophisticated techniques such as **dynamic programming**, **memoisation**, and **efficient algorithm design**.

🧩 2.2. UNDERSTANDING THE STRUCTURE OF FIBONACCI

The Fibonacci sequence is defined by its recurrence relation:

$$F(n) = F(n-1) + F(n-2)$$

This mathematical definition reveals something profoundly important: calculating any Fibonacci number (beyond the base cases) inherently requires knowing the values of two previous Fibonacci numbers. Let us explore this dependency structure more thoroughly.

Consider the task of computing F(5). According to our recurrence relation:

$$F(5) = F(4) + F(3)$$

But each of these terms must themselves be calculated using the same relation:

$$F(4) = F(3) + F(2) \quad F(3) = F(2) + F(1)$$

Continuing this pattern:

$$F(2) = F(1) + F(0)$$

This naturally creates what computer scientists refer to as a **dependency tree**—a hierarchical structure showing which values must be calculated before others can be determined. For F(5), this tree would look like:

```
F(5)
├── F(4)
│   ├── F(3)
│   │   ├── F(2)
│   │   │   ├── F(1) [Base case: 1]
│   │   │   └── F(0) [Base case: 0]
│   │   └── F(1) [Base case: 1]
│   └── F(2)
│       ├── F(1) [Base case: 1]
│       └── F(0) [Base case: 0]
└── F(3)
    ├── F(2)
    │   ├── F(1) [Base case: 1]
    │   └── F(0) [Base case: 0]
    └── F(1) [Base case: 1]
```

Upon examining this tree structure, one immediately notices a critical inefficiency: certain calculations appear multiple times:

- F(3) must be calculated twice
- F(2) appears three times
- F(1) appears five times
- F(0) appears three times

This redundancy becomes increasingly problematic as n grows larger. For instance, computing F(30) using this naïve approach would involve billions of redundant calculations—a classic example of combinatorial explosion.

This observation is tremendously valuable pedagogically, as it motivates the need for more efficient algorithmic approaches, particularly **memoisation**—storing previously calculated results to avoid redundant computation.

2.3. COMPUTATIONAL THINKING: DEFINING SUBPROBLEMS

When approaching the Fibonacci problem computationally, we must precisely define our subproblems. This is a crucial exercise in computational thinking—the process of formulating problems in ways that enable computational solutions.

For the Fibonacci sequence, we identify two distinct types of subproblems:

- **Base cases**: These are the foundational values upon which all other calculations depend. For Fibonacci, they are:
  - F(0) = 0
  - F(1) = 1
- **Recursive case**: This defines how to calculate any non-base case Fibonacci number:
  - $F(n) = F(n-1) + F(n-2)$, for $n \geq 2$

This formulation reveals that we are effectively breaking down the problem of computing F(n) into two smaller subproblems:

1. Computing F(n-1) — a problem of size n-1
2. Computing F(n-2) — a problem of size n-2

We then continue this process recursively until we reach our base cases, at which point we can begin combining results back up the chain to obtain our final answer.

This pattern of problem reduction—breaking down a problem into smaller instances of the same problem—is characteristic of **recursive algorithms** and forms the basis for many powerful computational techniques, including divide-and-conquer strategies.

2.4. THINKING LIKE A COMPUTER

To truly understand how computational solutions to the Fibonacci problem work, it's helpful to simulate the execution process mentally—essentially "thinking like a computer."

Let's trace through the computation of F(6) step by step, following the recursive definition:

1. To compute F(6), I need F(5) and F(4)
2. To compute F(5), I need F(4) and F(3)
3. To compute F(4), I need F(3) and F(2)

4. To compute F(3), I need F(2) and F(1)
5. To compute F(2), I need F(1) and F(0)
6. F(1) is a base case, value is 1
7. F(0) is a base case, value is 0
8. Now I can compute F(2) = F(1) + F(0) = 1 + 0 = 1
9. Now I can compute F(3) = F(2) + F(1) = 1 + 1 = 2
10. Now I can compute F(4) = F(3) + F(2) = 2 + 1 = 3
11. Now I can compute F(5) = F(4) + F(3) = 3 + 2 = 5
12. Finally, I can compute F(6) = F(5) + F(4) = 5 + 3 = 8

This detailed tracing reveals two important computational characteristics:

- The recursive approach requires maintaining a **call stack** that keeps track of unfinished computations.
- Many subproblems (like F(3) and F(2)) are recalculated multiple times.

Understanding this execution flow helps students appreciate why naïve recursive implementations become unacceptably slow for larger values of n and motivates the exploration of optimisation techniques.

2.5. VISUALISING SUBPROBLEM CONSTRUCTION

Analogies often provide powerful mental models for understanding computational concepts. One particularly illuminating metaphor for Fibonacci computation is that of **building a wall** brick by brick.

Imagine you are tasked with constructing a wall where each brick is a Fibonacci number. The unique property of this wall is that each new brick's size must be equal to the sum of the two preceding bricks.

- You start with two foundation bricks labelled 0 and 1 (your base cases).
- The third brick must be 0 + 1 = 1 units long.
- The fourth brick must be 1 + 1 = 2 units long.
- The fifth brick must be 1 + 2 = 3 units long.

And so on, with each new brick's size determined by adding the sizes of the two bricks placed immediately before it.

This visualisation helps students understand that:

1. You must build the wall sequentially, from left to right (or bottom to top).
2. Each new brick depends directly on only the two previous bricks.
3. Once a brick is placed, its value is fixed and can be reused for future calculations.

This mental model naturally leads to the realisation that we don't need to rebuild the entire wall from scratch each time we want to add a new brick—we only need to remember the sizes of the last two bricks we placed. This insight is the conceptual foundation for both memoisation and iterative approaches.

2.6. RECURSION vs ITERATION AS A RESULT OF DECOMPOSITION

The way we decompose the Fibonacci problem directly suggests two fundamental implementation approaches:

Recursive strategy (top-down approach):

- Start with the desired Fibonacci number F(n).
- Break it down into subproblems F(n-1) and F(n-2).
- Continue breaking down until reaching base cases.
- Combine results back up the chain.

Iterative strategy (bottom-up approach):

- Start with the base cases F(0) and F(1).
- Systematically build up, computing F(2), then F(3), and so on.

- Continue until reaching F(n).

Both approaches compute the same mathematical sequence, but they differ significantly in their computational characteristics:

- The recursive approach is elegant and mirrors the mathematical definition directly, but suffers from redundant calculations unless optimised.
- The iterative approach requires tracking fewer variables and naturally avoids redundant calculations, but may appear less intuitive to those accustomed to the mathematical definition.

This duality illustrates a common pattern in algorithm design: problems with recursive mathematical definitions can often be implemented either recursively or iteratively, with different performance characteristics.

2.7. USING A TABLE TO TRACK SUBPROBLEMS

A practical technique for visualising the computational structure of Fibonacci calculations is to use a table that tracks the dependencies and calculations for each term:

n	F(n) = F(n-1) + F(n-2)	Computation required
0	-	0 (base case)
1	-	1 (base case)
2	1 + 0	F(1) + F(0) = 1
3	2 + 1	F(2) + F(1) = 2
4	3 + 2	F(3) + F(2) = 3
5	5 + 3	F(4) + F(3) = 5
6	8 + 5	F(5) + F(4) = 8

This tabular representation highlights an important insight: if we store the results of each calculation (a technique known as **memoisation**), we can avoid redundant computation. Once F(3) is calculated, for example, we can reuse that result whenever F(3) appears in subsequent calculations.

This observation naturally leads to more efficient implementation strategies, such as:

- Using an array or dictionary to store previously calculated Fibonacci numbers
- Using an iterative approach that naturally builds up the sequence from the bottom

SUMMARY OF SECTION 2

- Decomposing the Fibonacci problem reveals its naturally recursive structure, where each number depends on the two previous numbers in the sequence.
- This decomposition creates a tree of dependencies where the same subproblems appear multiple times, highlighting the need for optimisation strategies.
- The problem naturally breaks down into base cases (F(0) and F(1)) and a recursive case (F(n) = F(n-1) + F(n-2) for $n \geq 2$).
- "Thinking like a computer" when calculating Fibonacci numbers helps us understand the inefficiencies in naïve approaches and motivates optimisation.
- Visual metaphors such as the "wall of bricks" provide intuitive understanding of how Fibonacci numbers build upon previous results.
- Problem decomposition suggests both recursive (top-down) and iterative (bottom-up) implementation strategies, each with different performance characteristics.
- Using tables to track calculations reveals opportunities to store and reuse intermediate results, leading naturally to techniques like memoisation and dynamic programming.

This foundational understanding of problem decomposition will serve as crucial scaffolding as we explore specific implementation strategies in subsequent sections, from basic recursive approaches to highly optimised algorithms.