Dynamic Programming – Part 1

By: **Aaryan Vinod Kumar**

Introduction

Dynamic Programming (DP) is an algorithmic paradigm used to solve problems by breaking them into smaller overlapping subproblems, solving each subproblem only once, and reusing the results to build the final solution.

DP is not a specific algorithm.
It is a problem-solving technique that optimizes recursive solutions by avoiding repeated computation.

In simple terms:

> **"If you are solving the same subproblem again and again, store its answer and reuse it."**

Why Dynamic Programming Exists

Many problems are naturally solved using **recursion**.
However, naive recursion often leads to:

- Repeated computation of the same subproblems
- Exponential time complexity
- Poor performance for large inputs

Dynamic Programming exists to **fix this inefficiency**.

Core Idea of Dynamic Programming

Dynamic Programming is based on **two essential properties**:

1. Overlapping Subproblems

A problem has overlapping subproblems if the same smaller problem is solved **multiple times** during recursion.

Instead of recomputing it every time, DP **stores the result** and reuses it.

2. Optimal Substructure

A problem has optimal substructure if its **optimal solution can be constructed from optimal solutions of its subproblems**.

This means:

- Solving smaller parts optimally helps solve the bigger problem optimally.

Dynamic Programming in Simple Terms

Imagine climbing a staircase where you can take 1 or 2 steps at a time.

To reach step `n`, you must come from:

- step `n-1`, or
- step `n-2`

So:

- Ways(n) = Ways(n-1) + Ways(n-2)

A recursive solution keeps recomputing `Ways(n-1)` and `Ways(n-2)` again and again.

Dynamic Programming simply says:

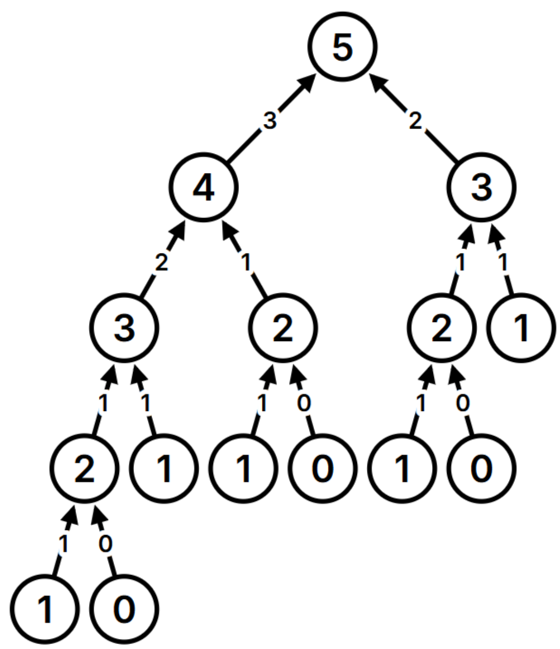
> “Once you compute `Ways(k)`, remember it.”

Why Plain Recursion Fails

Let’s consider the classic **Fibonacci sequence**:

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

Recursive Call Tree (credit to BrunoPapa on GitHub)



Observations:

- `F(3)` is computed **twice**
- `F(2)` is computed **three times**
- As `n` grows, repetition explodes

Result:

- Time Complexity → **$O(2^n)$**
- Extremely inefficient

Dynamic Programming eliminates this repetition.

Dynamic Programming vs Other Paradigms

DP vs Divide and Conquer

Divide & Conquer	Dynamic Programming
Subproblems are independent	Subproblems overlap
No reuse of results	Results are reused
Example: Merge Sort	Example: Fibonacci

DP vs Greedy

Greedy	Dynamic Programming
Makes locally optimal choices	Considers all states
No backtracking	Explores all possibilities
Fast but risky	Slower but guaranteed optimal

DP vs Branch and Bound

Branch and Bound	Dynamic Programming
Prunes bad paths	Reuses solved states
Still explores search tree	Builds from subproblems
Good for optimization	Good for structured recurrence

The DP Mindset

Dynamic Programming requires a shift in thinking.

Instead of asking:

> "How do I solve this problem directly?"

You ask:

1. **What is the smallest version of this problem?**
2. **How does a larger solution depend on smaller ones?**
3. **Can I store and reuse answers?**

This leads to **state-based thinking**.

What is a DP State?

A **state** represents:

> The minimum information needed to describe a subproblem.

Examples:

- Fibonacci → `dp[n]` = value of `F(n)`
- Staircase → `dp[n]` = number of ways to reach step `n`
- Strings → `dp[i][j]` = answer for prefixes of length `i` and `j`

State definition is the **hardest and most important** part of DP.

General DP Problem-Solving Framework

Even though implementations differ, **every DP problem follows this structure**:

1. **Define the State**
   - What does `dp[x]` represent?
2. **Define the Transition**
   - How do we move from smaller states to bigger ones?
3. **Identify Base Cases**
   - Smallest problems with known answers
4. **Compute the Final Answer**
   - Usually `dp[n]`, `dp[m][n]`, etc.

This framework remains constant across all DP problems.

When Dynamic Programming Should NOT Be Used

DP is powerful, but not universal.

Do **NOT** use DP when:

- There are **no overlapping subproblems**
- Greedy already gives an optimal solution
- Problem size is too small to justify overhead
- State space is too large to store

Blindly applying DP leads to:

- High memory usage
- Overcomplicated solutions

Advantages of Dynamic Programming

- **Drastically improves performance**
- Converts exponential solutions to polynomial
- Guarantees optimal solutions
- Applicable to a wide range of problems

Limitations

- Requires careful state design
- Can consume large amounts of memory
- Difficult to debug if state or transition is wrong
- Not always intuitive initially

Credits for the Visualiser

Thank you to BrunoPapa for making a recursive tree visualiser, you can check their project out on GitHub.

