Dynamic Programming: The Art of Breaking and Building

Joshua Arrazola • 13 Jul 2025

A Historical and Mathematical Motivation

The origins of *dynamic programming* date back to the 1950s, introduced by Richard Bellman in the context of decision processes. Contrary to what the term might suggest, it has little to do with programming as understood today; the "programming" in dynamic programming refers to the methodical planning of decisions. Bellman's primary concern was to decompose complex multistage decision problems into simpler sub-decisions that could be solved recursively.

At its core, dynamic programming provides a principled approach to solving problems that exhibit two key properties:

1. **Optimal substructure**: An optimal solution to the problem can be constructed from optimal solutions of its subproblems.
2. **Overlapping subproblems**: The problem contains subproblems that are solved multiple times in naive recursion.

To illustrate why simple recursion may fail, both conceptually and computationally, consider the Fibonacci sequence, a classic example often underestimated in complexity.

Recursive Explosion in the Fibonacci Sequence

Let $F(n)$ denote the n-th Fibonacci number, defined recursively as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases}$$

This definition is elegant, but computationally catastrophic. A naive recursive implementation performs a massive number of redundant calculations. To see this, let us model the number of recursive calls $T(n)$ as:

$$T(n) = T(n-1) + T(n-2) + 1, \quad T(0) = T(1) = 1.$$

Solving this recurrence reveals that $T(n) = \Theta(2^n)$. In other words, computing $F(50)$ would take on the order of 10^{15} function calls prohibitively expensive.

The inefficiency stems from solving the same subproblem repeatedly. The call tree of $F(5)$ already computes $F(2)$ three times, and this redundancy compounds exponentially with n. Hence, although the recurrence is well defined mathematically, its direct computation lacks structure and is practically unusable for large n.

Memoization: How to avoid repetition

To address this, we can store previously computed results in a lookup table—a method called *memoization*. Define an array $\text{fib}[\,]$ such that:

$$\text{fib}[n] = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{fib}[n-1] + \text{fib}[n-2] & \text{otherwise, once computed.} \end{cases}$$

By caching results, we reduce the time complexity to $\Theta(n)$, as each subproblem is solved only once. This transformation (from exponential to linear) demonstrates the power of dynamic programming: we do not change the recurrence, we change how it is computed, meaning that we don't have to change our mindset about a different way of solving problems, we just need to add an additional layer to it.

Recursive Trees vs Dependency Graphs

Conceptually, naive recursion constructs a *recursion tree*, whose size reflects exponential growth. Dynamic programming, instead, constructs a *directed acyclic graph (DAG)* of dependencies, which can be solved via topological ordering (bottom-up) or through caching (top-down).

The Principle of Optimality

Dynamic programming is a direct consequence of how some problems are naturally structured. At the heart of this structure lies **Bellman's Principle of Optimality**, which provides the theoretical foundation for decomposing problems into subproblems.

> **Bellman's Principle of Optimality**: *An optimal policy has the property that, whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

While originally formulated in the context of control theory, this principle generalizes to any setting where a solution can be incrementally constructed from smaller components.

Formal Definition

Let us consider a problem defined over a set of **states** S, with a goal of finding a function $\mathrm{OPT} : S \to \mathbb{R}$ that gives the optimal cost (or value) associated with each state.

We define a transition model via:

- A set of actions $A(s)$ available in state s,
- A cost function $C(s, a)$ for applying action $a \in A(s)$,
- A transition function $T(s, a) \in S$ yielding the next state.

Then the value function satisfies the recurrence:

$$\mathrm{OPT}(s) = \begin{cases} 0 & \text{if } s \text{ is terminal,} \\ \min\limits_{a \in A(s)} \{C(s, a) + \mathrm{OPT}(T(s, a))\} & \text{otherwise.} \end{cases}$$

This equation expresses that the best outcome from state s is obtained by choosing the action a that minimizes the immediate cost plus the cost of solving the resulting subproblem.

From Recurrence to Algorithm

To use this recurrence computationally, we require:

1. A definition of the **state** $s \in S$ in terms of input parameters,
2. A **base case** for terminal states,
3. A **transition rule** for moving between states.

Let us walk through a concrete example to see how the abstract recurrence becomes a dynamic programming solution.

Example: The 0/1 Knapsack Problem

Problem Statement: Given n items with weights $w_1, w_2, \ldots, w_n \in \mathbb{N}$ and values $v_1, v_2, \ldots, v_n \in \mathbb{R}$, and a knapsack of capacity W, choose a subset of items whose total weight does not exceed W and whose total value is maximized.

State Definition: Let $\mathrm{DP}(i, c)$ denote the maximum value achievable using the first i items and remaining capacity c.

Recurrence Relation:

$$\mathrm{DP}(i,c) = \begin{cases} 0 & \text{if } i = 0 \text{ or } c = 0, \\ \mathrm{DP}(i-1,c) & \text{if } w_i > c, \\ \max\left\{\mathrm{DP}(i-1,c),\ v_i + \mathrm{DP}(i-1,c-w_i)\right\} & \text{otherwise.} \end{cases}$$

Interpretation:

- If we cannot include item i, we carry over the previous solution.
- If we can include it, we compare the benefit of including it versus skipping it.

This recurrence obeys Bellman's principle: the optimal value of the full problem depends solely on optimal values of smaller subproblems. No global re-evaluation is needed, only local reasoning based on subproblem outcomes.

Computational Implementation

This recurrence leads directly to two algorithmic strategies:

- **Top-down** (memoized recursion): Start from $\mathrm{DP}(n, W)$, recursively call subproblems, and cache results.
- **Bottom-up** (tabulation): Fill a table from $\mathrm{DP}(0, 0)$ upward using iterative loops.

In either case, the total number of subproblems is $\mathcal{O}(nW)$, and each subproblem takes $\mathcal{O}(1)$ time to compute. Thus, the overall time complexity is $\mathcal{O}(nW)$, a dramatic improvement over the $\mathcal{O}(2^n)$ brute-force approach.

With this formal foundation, we can now recognize a dynamic programming problem by identifying:

- A state space defined by parameters,
- A recurrence that expresses optimality via subproblems,
- Base cases, and
- A strategy (memoized or iterative) for evaluation.

In the next and final section, we will explore how dynamic programming serves not just as an algorithmic trick, but as a mathematical lens through which we analyze and decompose complex structures.

Dynamic Programming as a Mathematical tool

Programming dynamic solutions is often taught as a technique: identify overlapping subproblems, find a recurrence, fill a table. But such a perspective is incomplete. Dynamic programming is not merely a computational shortcut: it is a framework for reasoning about problems whose structure is inherently recursive and whose solutions are inherently compositional (with compositional i'm saying that every sub-answer is reachable from other sub-problems).

The Mindset Shift

At the heart of dynamic programming lies a simple but transformative idea: to solve a large problem, analyze its structure and identify the minimal information needed to describe any intermediate state. The art is in choosing the right representation of state.

Consider this: given a problem, what are its parameters of progress? What defines a "subproblem"? What determines that one state is closer to completion than another?

These are mathematical questions, and at the moment we answer them correctly, the problem begins to reveal a network of smaller, solvable parts.

Examples Beyond the Classical Paradigm

The beauty of dynamic programming is best appreciated when applied to non-obvious domains. Let us consider a few generalizations that showcase its breadth.

Tree DP

Let T be a tree and each node u have some cost or weight. Suppose we wish to compute a function $f(u)$ over the subtree rooted at u, where f depends recursively on values of f at u's children. Since trees are acyclic, we can traverse from leaves to root and compute $f(u)$ in linear time using dynamic programming on the structure of the tree.

This is not a trick, it is the natural way to think about recursive structure over non-linear domains.

Digit DP and Bitmasking

In counting problems, one often needs to enumerate integers satisfying digit level constraints (e.g., no two adjacent digits are equal). Representing the number by its digits and defining states like $\mathrm{DP}(i, \mathrm{tight}, \mathrm{mask})$ leads to what is called *Digit DP*, a powerful method in combinatorics and number theory.

Similarly, bitmask DP exploits the idea that subsets of a finite set can be represented as bit vectors. This allows exponential state spaces to be handled efficiently via memoization when transitions are sparse.

Optimization Tricks

In some problems, even standard DP is too slow. Fortunately, under certain mathematical conditions, DP can be optimized further:

- **Convex Hull Trick**: Optimizes DP recurrences of the form $\mathrm{DP}(i) = \min_j\{m_j x_i + b_j\}$, assuming the slopes m_j are monotonic.
- **Knuth Optimization**: Applies to certain cost functions satisfying the quadrangle inequality, reducing $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.
- **Divide and Conquer DP**: Reduces complexity when the recurrence has a monotonic decision boundary.

This are mathematical properties of convexity, monotonicity, and structure are given by the nature of the *ad hoc* problem we are trying to solve.

The True Value of Dynamic Programming

Dynamic programming forces us to ask deeper questions:

- What is the structure of this problem?
- How can it be decomposed?
- What information must be preserved at each step?

These are not programming questions, they are modeling questions. And their answers often lead to formulations that transcend the original problem.

Indeed, many hard problems in algorithm design reduce to discovering the right DP formulation, choosing the correct state space and recurrence. This is less about programming and more about insight.

-J