Dynamic Programming
LP Final Project

Raina & Noura

University of Colorado Denver

Motivation (Brute Force)

Many well-known NP-Hard problems such as 0/1 Knapsack and TSP have no known polynomial-time solution. Using brute force means checking every possible combination, which leads to exponential time:

- Knapsack: $O(2^n)$
- TSP: $O(n!)$

This extreme growth is why we need smarter techniques than brute force.

What is Dynamic Programming?

- An algorithmic technique used to reduce the time complexity of hard problems.
- Developed by Richard Bellman in the 1950s.
- Solves a complex problem by breaking it into smaller subproblems and storing their results to avoid repeated computations.
- We can apply Dynamic Programming only when the problem satisfies certain structural properties.

A problems must have

1. ***Optimal Substructure***
   The optimal solution can be built from optimal solutions of smaller subproblems
2. ***Overlapping Sub-Problems***
   The recursive algorithm solves the same subproblems over and over.

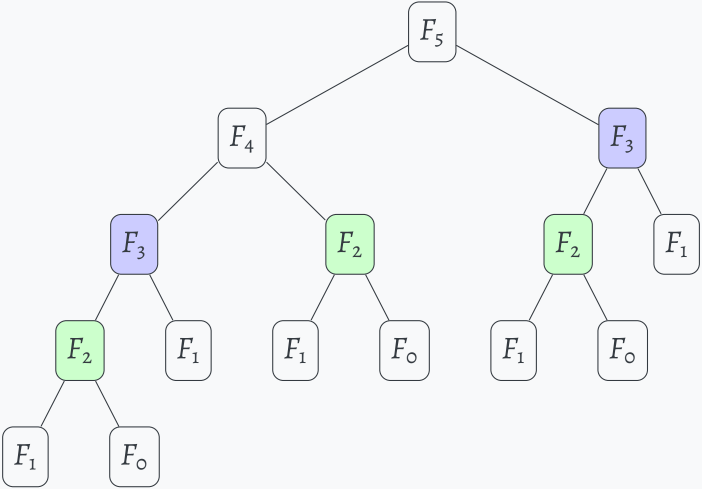
Fibonacci Series

Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

When $n \leqslant 1$, $\begin{cases} \mathrm{Fib}(0) = 0, \\ \mathrm{Fib}(1) = 1, \end{cases}$

Otherwise $\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$

Fibonacci Recursion Tree



Handling Overlapping Subproblems

Memoization: Top-Down

- Uses recursion.
- Store the result when a subproblem is first solved.
- Look up the stored value before computing.
- Avoids recomputation.

Tabulation: Bottom-Up

- Iterative approach.
- Fill a DP table from smaller to larger subproblems.
- No recursion.
- Read the final answer from the table.

A Framework for Solving DP Problems

1. Define the Subproblems
2. Define the Recurrence Relation
3. Identify the Base Case
4. Build the Algorithm
5. Analyze the Time and Space Complexity

The *Knapsack* problem

Brute Force: $O(2^n)$

$$\max \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leqslant W$$

$$x \in \{0, 1\}$$

The Knapsack Problem

1/5: Define the Subproblems

	Weight capacity				
Item Number	0	1	2	...	W
0	$x^*_{0,0}$	$x^*_{0,1}$	$x^*_{0,2}$	$\dots$	$x^*_{0,W}$
1	$x^*_{1,0}$	$x^*_{1,1}$	$x^*_{1,2}$	$\dots$	$x^*_{1,W}$
2	$x^*_{2,0}$	$x^*_{2,1}$	$x^*_{2,2}$	$\dots$	$x^*_{2,W}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	$x^*_{n,0}$	$x^*_{n,1}$	$x^*_{n,2}$	$\dots$	$x^*_{n,W}$

The Knapsack Problem

2/5: Define the Recurrence Relation

$$x^*_{i,\Omega}$$

Take item i

$$v_i + x^*_{i-1,\Omega-w_i}$$

Skip item i

$$x^*_{i-1,\Omega}$$

$$x^*_{i,\Omega} = \max\{(v_i + x^*_{i-1,\Omega-w_i}), (x^*_{i-1,\Omega})\}$$

The Knapsack Problem

3/5: Identify the Base Case

	Weight capacity				
Item Number	0	1	2	...	W
0	0	0	0	...	0
1	0	$x^*_{1,1}$	$x^*_{1,2}$	...	$x^*_{1,W}$
2	0	$x^*_{2,1}$	$x^*_{2,2}$	...	$x^*_{2,W}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	0	$x^*_{n,1}$	$x^*_{n,2}$	...	$x^*_{n,W}$

The Knapsack Problem

4/5: Build the algorithm.

	Weight capacity				
Item Number	0	1	2	...	W
0	$x^*_{0,0}$	$x^*_{0,1}$	$x^*_{0,2}$	...	$x^*_{0,W}$
1	$x^*_{1,0}$	$x^*_{1,1}$	$x^*_{1,2}$	...	$x^*_{1,W}$
2	$x^*_{2,0}$	$x^*_{2,1}$	$x^*_{2,2}$	...	$x^*_{2,W}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	$x^*_{n,0}$	$x^*_{n,1}$	$x^*_{n,2}$	...	$x^*_{n,W}$

for $i = 1, \dots, n$
    for $\Omega = 1, \dots W$

$$Opt[i][\Omega] = \begin{cases} Opt[i-1][\Omega], & w_i > \Omega \\ \max(Opt[i-1][\Omega],\ v_i + Opt[i-1][\Omega - w_i]), & \text{otherwise} \end{cases}$$

return $Opt[n][W]$

The Knapsack Problem

5/5: Analyze the time and space complexity.

Brute Force	**Dynammic Programming**
Time: $O(2^n)$	Time: $O(nW)$
Space: $O(n)$	Space: $O(nW)$

Other Applications

- The Traveling Salesman Problem (Held-Karp Algorithm)

$$O(n!) \quad \rightarrow \quad O(n^2 2^n)$$

- Sequence Alignment (Needleman–Wunsch algorithm)

$$O(2^{m+n}) \quad \rightarrow \quad O(mn)$$

- Egg-Dropping Problem

$$O(2^F) \quad \rightarrow \quad O(EF^2)$$