

Dynamic Programming

Linear Programming Final Project

Raina Burton & Noura Allugmani

University of Colorado Denver
December 10, 2025

1 Motivation

Many classical NP-hard problems, such as the 0/1 Knapsack Problem and the Traveling Salesman Problem (TSP), cannot be solved in polynomial time using any known algorithm. A brute-force method must examine every possible combination of choices, which leads to exponential time complexity.

For example, a brute-force approach to the Knapsack problem requires checking all 2^n subsets of items, giving a complexity of approximately $O(2^n)$. Likewise, solving the Traveling Salesman Problem by enumerating all possible tours yields $O(n!)$. These growth rates quickly become infeasible as n increases.

Approximation algorithms such as greedy techniques or 2-approximation algorithms can significantly reduce computation time, but they sacrifice exactness. They can be much faster, but they cannot ensure the optimal answer. Dynamic Programming (DP) offers a way to reduce computation time while still guaranteeing correctness. For example, DP reduces Knapsack from $O(2^n)$ to $O(nW)$ when weights are integral, and reduces TSP from $O(n!)$ to $O(2^n n^2)$ with the Held-Karp algorithm.

Beyond integer programming, DP appears in many other areas: computing Fibonacci numbers, matrix chain multiplication, sequence alignment in bioinformatics, and even puzzles such as the Tower of Hanoi.

To see why DP is useful, consider a simple example: the Fibonacci sequence. It demonstrates how a naive recursive approach can become inefficient very quickly.

Using the recurrence

$$F_0 = 1, \quad F_1 = 1, \quad F_N = F_{N-1} + F_{N-2},$$

the recursive method keeps recomputing the same values many times. For example, F_{N-2} , F_{N-3} , and smaller terms appear repeatedly in different branch of the recursion tree. This repetition makes the algorithm exponential, around $O(2^N)$.

2 Dynamic Programming DP

Dynamic Programming a programming technique used to reduce time complexity of difficult problems. In fact, a large class of exponential problems has a polynomial solution via Dynamic Programming[1]. Dynamic programming works by breaking down complex problems into their sub-problems, an algorithmic approach that was developed by Richard Bellman in the 1950s. The technique relies on optimal substructure (or the principle of optimality) and overlapping subproblems [4]. Let's begin discussing optimal substructure.

Optimal substructure A problem has optimal substructure if the best (optimal) solution can be built by combining the best solutions to its smaller parts (subproblems) [3]. This is what makes dynamic programming (DP) effective: instead of keeping all possible solutions for subproblems, by DP, we only need to store the optimal ones to build the final answer. DP cannot be used to solve a problem if it does not have this substructure.

Overlapping subproblems A problem has overlapping subproblems if the recursive algorithm solves the same subproblems over and over, rather than always generating different subproblems [3]. Many problems can be broken down into smaller subproblems that repeat. Without DP, these problems will be solved multiple times unnecessarily, leading to an unnecessarily high time complexity.

Let's recall the Fibonacci problem we mentioned above. For example, to compute the 5th Fibonacci number $F(5)$, it can be broken down into the subproblems of computing $F(4)$ and $F(3)$. Then, the subproblem $F(4)$ can itself be broken down further into computing $F(3)$, which means the computation of $F(3)$ is reused. This shows that the Fibonacci sequence exhibits overlapping subproblems.

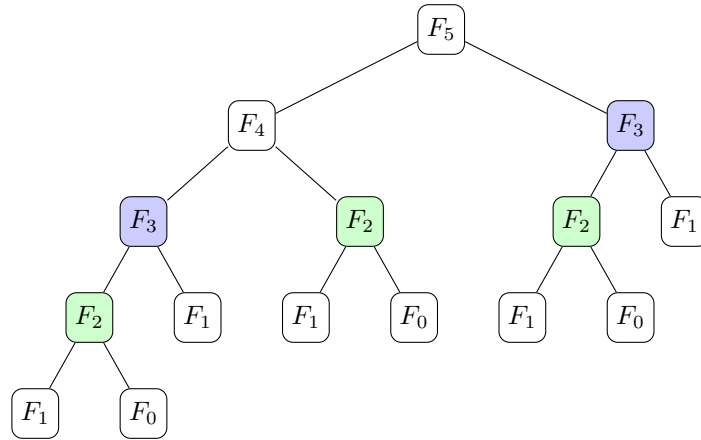


Figure 1: Recursive computation tree for F_5 , highlighting how the recursive algorithm repeatedly solves the same subproblems F_3 and F_2 .

Dynamic programming is powerful because it solves problems that have two key properties: overlapping subproblems and optimal substructure. When a problem has overlapping subproblems, the number of computations can grow exponentially if we use a naive recursive approach. Dynamic programming avoids this by solving each subproblem only once and storing the result. This makes many DP algorithms run in polynomial time, which is much faster and more efficient. A dynamic programming approach breaks a problem into smaller subproblems, solves each one once, stores the solution, and then combines these solutions to achieve the optimal result. This is particularly useful when the number of repeated subproblems grows exponentially with the input size. There are two implementation approaches for solving dynamic programming problems: the top-down (memoization) approach and the bottom-up (tabulation) approach, and we will consider both of them in the following.

2.1 Top-Down Approach (Memoization):

Memoization is a programming technique used to reduce expensive calls to functions. While running, a program using a memoization technique will first check whether the value has been previously computed, and only if it hasn't, will it continue to compute.

2.2 Bottom-Up Approach (Tabulation):

Tabulation, like memoization, is a programming technique used to reduce expensive calls to functions. However, tabulation involves starting from the bottom of the recursion and computing every possible value so that the Dynamic Program can quickly reference and move on.

2.3 A Framework for Solving DP Problems

Dynamic programming goes through five steps to solve problems:

1. Define the subproblems.
2. Define the recurrence relation.
3. Identify the base case.
4. Build the algorithm (DP approach).
5. Analyze the time and space complexity.

3 The Traveling Salesman Problem: TSP

Let's see how dynamic programming solves TSP. We consider the IP formulation of the TSP problem presented below. Given a set of n cities and travel costs c_{ij} between city i and city j , we model the problem on a graph $G = (V, E)$, where $V = 1, \dots, n$ is the set of vertices (cities) and E is the set of edges.

- $\delta(i)$: set of edges incident to vertex i
- $\delta(S)$: set of edges with exactly one vertex in $S \subset V$
- C_e : cost of edge e
- variables: $x_e \in \{0, 1\}$ indicates whether edge e is used

$$\begin{aligned} \min \sum_{e \in E} C_e x_e \\ \sum_{e \in \delta(i)} x_e &= 2 \quad \forall i \in V \\ \sum_{e \in \delta(S)} x_e &\geq 2 \quad \forall S \subset V, S \neq \emptyset, V \\ x_e &\in \{0, 1\} \quad \forall e \in E \end{aligned}$$

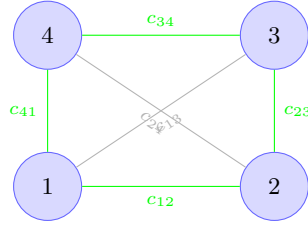


Figure 2: TSP instance with 4 cities, represented as a complete weighted graph $G = (V, E)$, with one possible tour.

The complexity of this problem grows exponentially with $O(n!)$ in brute-force recursion which means it is NP-hard problem. In 1962, Bellman introduced the Held–Karp algorithm as a dynamic programming approach for solving the TSP, looking for go through each cities i with minimum cost. The TSP has an optimal substructure and overlapping subproblems, so we can apply dynamic programming to solve it in five steps:

3.1 Define the subproblems depending on Held–Karp approaches.[2]

Fix city 1 as the starting point, then the tour can only end at one of the remaining $n - 1$ cities. For each possible ending city, the set of visited cities must include city 1 and that ending city, while the other $n - 2$ cities can be either included or not. Each of these cities has two choices, so this gives 2^{n-2} possible subsets for every ending city. Putting this together, the number of subsets we need to consider becomes $(n - 1)2^{n-2}$.

3.2 Define the recurrence relation

In this step, we compute the minimum cost for each subset by using the values of smaller subsets. For every group of cities, we look at the smaller group that comes before it and add the travel cost to the next city. Then we choose the smallest among all possible previous choices. By building from the smallest subsets upward, we gradually obtain the optimal cost for larger and larger sets of cities.

Fix city 1 as the start. The Held-Karp idea is to take a subset $S \subseteq \{2, \dots, n\}$ and a city $e \neq 1$ that we want to end at. We are interested in the shortest path that starts at city 1, visits every city in S in any order, and finishes at e . We denote this value by $g(S, e)$, and let $d(u, v)$ be the direct travel cost between cities u and v .

The algorithm computes $g(S, e)$ beginning with the smallest subsets.

- The smallest possible subproblem (the empty set): when there are no cities left to visit, the only option is to go directly from city 1 to the ending city e , so the cost becomes simply $d(1, e)$. Then

$$g(\emptyset, e) = d(1, e)$$

- For larger subsets, we remove the last city e and choose the previous city k that gives the minimum total cost:

$$g(S, e) = \min_{k \in S} (g(S \setminus \{e\}, k) + d(k, e))$$

When the subset S contains only one or two cities, computing $g(S, e)$ is simple because there are only a few possible paths to check. As the subsets grow, the algorithm always follows the same principle: combine the best solutions from smaller subsets to find the minimum cost for the larger ones.

3.3 Identify the base case

The base case happens when the subset of cities is empty. This is the smallest version of the problem because there are no cities left to visit. In this case, the only option is to go directly from city 1 to the ending city e , so the cost becomes simply

$$g(\emptyset, e) = d(1, e)$$

This base value is the starting point that the algorithm uses to build all larger subsets.

3.4 Build the algorithm (DP approach)

In this step, we put everything together and build the dynamic programming table using the base case and the recurrence. We start with the smallest subsets and compute their values first, since these are already known. Then, for each larger subset of cities, we apply the recurrence to combine the results from the smaller subsets. This bottom-up process continues until all subsets are filled. For each subset and each possible ending city, we compute the minimum cost by choosing the best previous city and adding its travel cost. After all subsets have been processed, we obtain the final tour cost by completing the cycle and returning to city 1:

$$\text{OPT} = \min_{e \neq 1} (g(\{2, \dots, n\} \setminus \{e\}, e) + d(e, 1)).$$

This gives the minimum cost of a Hamiltonian cycle that visits every city exactly once.

3.5 Analyze the time and space complexity

In the last step, we look at how much work the dynamic programming approach needs. Since we fix city 1 as the starting point, we only consider the remaining $n - 1$ cities as possible ending points. For each ending city, there are 2^{n-2} different subsets to check, so the algorithm ends up filling a table of size $O(n2^n)$. When we apply the recurrence to each entry, the total running time becomes $O(n^2 2^n)$. This is still exponential, but it is much faster than brute force, which needs $O(n!)$ time. The space requirement is also exponential because we store a value for every subset. Even with this cost, the Held-Karp method is a major improvement and is one of the classic exact approaches for solving the TSP.

4 The Knapsack Problem

Given a set of n items numbered from 1 up to n , each with a weight w_i and a value v_i , along with a maximum weight capacity W , the goal of the Knapsack Problem is to

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^n v_i x_i \\ & \text{subject to} && \sum_{i=1}^n w_i x_i \leq W \quad \text{and} \quad x \in \{0, 1\} \\ & && x \in \{0, 1\}. \end{aligned}$$

where $x_i = 1$ if item i is taken and $x_i = 0$ if not. Because there are 2^n possible subsets of the n items, a brute-force approach to the knapsack problem has complexity $O(2^n)$.

The Knapsack Problem has both optimal substructure and overlapping sub-problems (as we'll see shortly,) so we can use our framework for solving Dynamic Programming problems.

4.1 Define the subproblems.

The optimal solution to the Knapsack problem has two main constraints: the weight capacity of the bag and the available items. Let's examine each constraint separately.

- **Weight Capacity:** We are given some weight capacity W . We'll solve the IP for all weight capacities $\Omega = 1, \dots, W$.
- **Available Items:** We're given n items with associated weights w and values v . We order the items in ascending order by weight and solve the integer program for the first i items for all $i = 1, \dots, n$.

For each $\Omega = 1, \dots, W$, we solve the IP for all $i = 1, \dots, n$ which means we solve nW integer programs.

	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	$x_{n,0}^*$	$x_{n,1}^*$	$x_{n,2}^*$...	$x_{n,W}^*$

These nW integer programs are the subproblems.

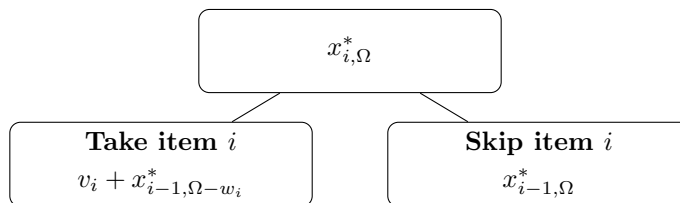
4.2 Define the recurrence relation.

Suppose we are solving the sub-IP for some weight capacity Ω and the first i available items. Let w_i be the weight of the i^{th} item and suppose that $w_i \leq \Omega$, the optimal solution, x_i^* to the subproblem given i and Ω is

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

In other words, given that we've solved the IP for Ω and $i-1$, to solve the IP for Ω and i , we only have to consider whether it's optimal to take the i^{th} item. If we take the i^{th} item, we can only hold $\Omega - w_i$ more weight in our bag, and only have the first $i-1$ items available still, so we examine the optimal solution for $\Omega - w_i$ and $i-1$.

We compare the optimal solution given that we take item i with the optimal solution given that we don't take item i and the maximum value is the optimal solution for Ω and i .



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

4.3 Identify the base case.

The base case is the integer program with $\Omega = 0$ and $i = 0$ (the IP with no weight capacity and no items to choose from.)

4.4 Build the algorithm (DP approach). [5]

```

for sub_item in range(1, len(i_v_w) + 1):
    val, wt = i_v_w[sub_item - 1]
    for sub_weight in range(max_weight + 1):

```

	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	$x_{n,1}^*$	$x_{n,2}^*$...	$x_{n,W}^*$

```

if wt > sub_weight:
    opt_sol[sub_item][sub_weight] = opt_sol[sub_item - 1][sub_weight]
else:
    opt_sol[sub_item][sub_weight] = max(
        opt_sol[sub_item - 1][sub_weight],
        val + opt_sol[sub_item - 1][sub_weight - wt]
    )

```

4.5 Analyze the time and space complexity.

We find an optimal solution for each number of items up to n and weight capacities up to W , meaning that we solve nW integer programs. Fortunately, by dynamic programming methods, we don't solve each sub-IP by brute force. Rather, we use tabulation and memoization to ensure that the only computations to do nW times is to

- Find the optimal solution in the case that we take item i (by referencing optimal solutions for smaller weights and numbers of items)
- Compare this with the optimal solution for Ω and $i - 1$.

By DP methods, computing the nW optimal solutions only has time complexity of $O(nW)$, while storing the optimal solutions to the nW sub-problems gives a space complexity of nW .

References

- [1] Binu Ayyappan and Santhoshkumar Gopalan. A performance and power characterization study of memoization and tabulation methods in graph neural networks by assessing dynamic programming workloads. In *2022 6th international conference on information technology, information systems and electrical engineering (ICITISEE)*, pages 1–6. IEEE, 2022.
- [2] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.
- [3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- [4] Jeff Erickson. *Algorithms*. 2019. Available online at algorithms.wtf.
- [5] Ameen Shaheen and Azzam Sleit. Comparing between different approaches to solve the 0/1 knapsack problem. *International Journal of Computer Science and Network Security (IJCSNS)*, 16(7):1, 2016.

5 Other References

- Dynamic Programming Lectures, MIT OpenCourseWare.
- Held–Karp algorithm, Wikipedia
- Knapsack Problem, Wikipedia