

8 Algorithm III: Dynamic Programming

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Understand the concept and principles of dynamic programming.
- Learn how to apply dynamic programming techniques to solve practical problems.
- Understand how memoization can optimize dynamic programming solutions.

8.1 Dynamic Programming

Dynamic programming (DP) is a powerful algorithmic paradigm used to solve optimization problems by breaking them down into simpler, overlapping subproblems. It is particularly effective when the problem exhibits an *optimal substructure*, i.e., the optimal solution can be constructed from the optimal solutions of its subproblems.

As an example, consider the Fibonacci sequence, which perfectly illustrates these characteristics. The n -th Fibonacci number is defined as:

$$F(n) = F(n-1) + F(n-2), \quad \text{when } n \geq 2.$$

Code 8.1 implements the function recursively. The recursion terminates when n equals 0 or 1. However, this naive approach is inefficient because it performs many redundant calculations.

```
from functools import lru_cache

@lru_cache()
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n-1) + fib(n-2)
```

Code 8.1: Naive recursive implementation of the n -th Fibonacci number

Figure 8.1 expands the function calls when computing the 6-th Fibonacci number with Code 8.1. The number of recursive calls for n can be expressed as

$$T(n) = T(n-1) + T(n-2) + 1,$$

where $T(0) = 1$ and $T(1) = 1$. This recurrence leads to an exponential time complexity. As a result, the function may take several hours even for $n > 50$.

In practice, performance can be greatly improved using a memoization-based approach. As shown in Code 8.2, we store previously computed results in a dictionary called `memo`. Before performing a recursive call, the function first checks whether the result already exists in `memo`. If so, it retrieves the value directly, avoiding redundant computation. This approach trades space for time efficiency. Python conveniently provides the `@lru_cache()` decorator, which automates this memoization process. As shown

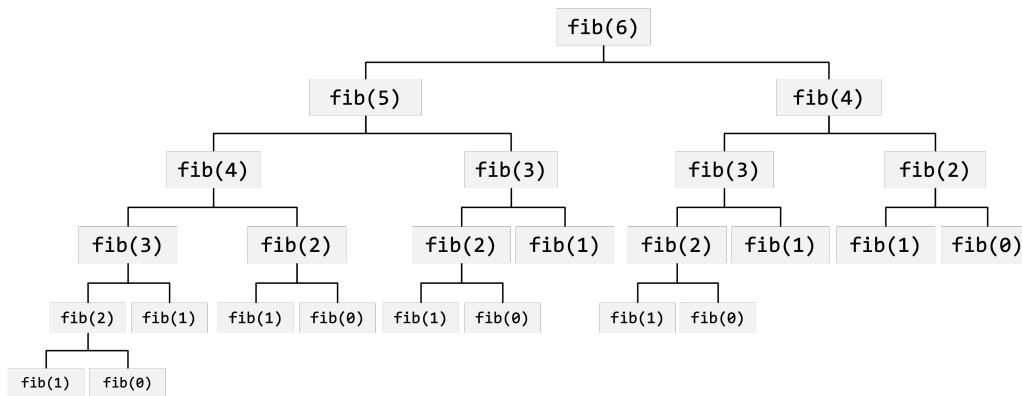


Figure 8.1: Recursive function call tree when executing `fib(6)` in Code 8.1.

in Code 8.1, we can annotate the `fib()` function with the decorator to achieve performance comparable to that of Code 8.2.

```
def fib_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n == 0 or n == 1:
        return n
    memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
    return memo[n]
```

Code 8.2: Fibonacci computation with explicit memoization

Alternatively, we can use an iterative implementation to eliminate recursion altogether. Code 8.3 shows an iterative Fibonacci algorithm whose performance is comparable to the memoized version. It also follows the dynamic programming principle by iteratively accumulating results from smaller subproblems.

```
def fib_optimized(n):
    if n == 0:
        return 0
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Code 8.3: Iterative implementation of the Fibonacci function

In summary, the general strategy for solving dynamic programming problems can be outlined as follows:

- 1) Characterize the structure of the optimal solution.
Example: The n th Fibonacci number depends on the $(n - 1)$ th and $(n - 2)$ th numbers.
- 2) Define the DP state (`dp[i]`): what does it represent?
Example: Let `dp[i]` represent the i -th Fibonacci number.
- 3) Write a recurrence relation between states.
Example: `dp[i] = dp[i-1] + dp[i-2]`.
- 4) Initialize the base cases.
Example: `dp[0] = 0, dp[1] = 1`.

5) Compute the solution iteratively (bottom-up) or recursively with memoization (top-down).

Example: Compute `dp[i]` for $i = 2, 3, \dots, n$ to obtain `dp[n]`.

8.2 Solving Problems with DP

In this section, we introduce two famous problems that can be solved by DP.

8.2.1 Coin Problem

Given coin denominations $[a, b, c]$, where $a < b < c$, and a target amount n , determine the minimum number of coins required to make up the amount. If it is impossible to form n using the given coins, return -1 .

The problem is straightforward when the coin system is canonical: a greedy strategy that repeatedly takes the largest coin not exceeding the remaining amount always produces an optimal solution (for example, $[1, 2, 5]$). However, for *non-canonical* coin systems, such as $[1, 2, 4]$, the greedy algorithm does not necessarily yield an optimal solution; a dynamic programming approach is required to ensure optimality.

Problem formulation. We can formulate the general coin problem in a top-down recursive manner.

$$dp(x) = \begin{cases} 0, & \text{if } x = 0, \\ \infty, & \text{if } x < 0, \\ \min_{c \in \text{coins}} (dp(x - c) + 1), & \text{otherwise.} \end{cases}$$

Top-down recursive implementation. Code 8.4 implements the program. This approach starts from the target amount n and recursively explores all possible combinations of coins. It avoids redundant subproblems via the built-in `@lru_cache()` decorator for memoization.

```
from functools import lru_cache

@lru_cache()
def dp_coin(coins, amount):
    if amount == 0:
        return 0
    if amount < 0:
        return float('inf')

    best = float('inf')
    for coin in coins:
        res = dp_coin(tuple(coins), amount - coin)
        if res != float('inf'):
            best = min(best, res + 1)

    return best if best != float('inf') else -1
```

Code 8.4: Top-down recursive DP solution for the coin problem

Bottom-up iterative implementation. We can also implement the coin problem in an iterative way. In this approach, computation starts from amount 0 and builds up to n . The state array `dp[i]` represents

the minimum number of coins required to make up the amount i , and each state is computed by iterating over all coin denominations.

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0 # base case

    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1
```

Code 8.5: Bottom-up iterative DP solution for the coin problem

Table 8.1 demonstrates the execution steps of the bottom-up algorithm when `coins = [1, 3, 4]` and `amount = 6`. Each cell records the minimum number of coins needed to form that amount after considering a specific coin.

Table 8.1: DP table evolution for `coins = [1, 3, 4]` and `amount = 6`.

	0	1	2	3	4	5	6
Initial	0	∞	∞	∞	∞	∞	∞
Coin 1	0	1	2	3	4	5	6
Coin 3	0	1	2	1	2	3	2
Coin 4	0	1	2	1	1	2	2

As shown in Table 8.1, when all coins are considered, the minimum number of coins required to make up the amount 6 is 2 (using coins 3 + 3 or 4 + 2). The time complexity of this algorithm is $O(n \times k)$, where n is the target amount and k is the number of coin types.

8.2.2 Longest Common Subsequence

Given two strings `text1` and `text2`, find the length of their *longest common subsequence (LCS)*. A subsequence is a sequence that appears in the same relative order in both strings, but not necessarily contiguously. For example, the longest common subsequence between “fundamental” and “fudan” is “fudan”.

Problem formulation. Let $dp(i, j)$ denote the length of the LCS between the prefixes `text1[0..i]` and `text2[0..j]`. We can define the recurrence relation as follows:

$$dp(i, j) = \begin{cases} 0, & \text{if } i < 0 \text{ or } j < 0, \\ dp(i - 1, j - 1) + 1, & \text{if } \text{text1}[i] = \text{text2}[j], \\ \max(dp(i - 1, j), dp(i, j - 1)), & \text{otherwise.} \end{cases}$$

Recursive implementation. The following recursive implementation directly follows the recurrence, using Python’s `@lru_cache()` decorator to avoid redundant recomputation.

```

from functools import lru_cache

@lru_cache()
def lcs(text1, text2, i=None, j=None):
    if i is None: i = len(text1) - 1
    if j is None: j = len(text2) - 1
    if i < 0 or j < 0:
        return 0

    if text1[i] == text2[j]:
        return lcs(text1, text2, i - 1, j - 1) + 1
    else:
        return max(
            lcs(text1, text2, i - 1, j),
            lcs(text1, text2, i, j - 1)
        )

```

Code 8.6: Top-down recursive DP for LCS

Iterative implementation. Alternatively, we can construct the dp table iteratively, filling it row by row.

```

def lcs_iter(text1, text2):
    n, m = len(text1), len(text2)
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[n][m]

```

Code 8.7: Iterative bottom-up DP for LCS

Table 8.2: Computation of the LCS length for “fudan” and “fundamental”.

	f	u	n	d	a	m	e	n	t	a	l
f	0	0	0	0	0	0	0	0	0	0	0
u	0	1	1	1	1	1	1	1	1	1	1
d	0	1	2	2	3	3	3	3	3	3	3
a	0	1	2	2	3	4	4	4	4	4	4
n	0	1	2	3	3	4	4	4	5	5	5

Table 8.2 shows the filling process of the dynamic programming table given $\text{text1} = \text{"fundamental"}$ and $\text{text2} = \text{"fudan"}$. The columns correspond to the characters of $\text{text1} = \text{"fundamental"}$, and the

rows correspond to the characters of `text2 = "fudan"`. Each cell (i, j) represents the length of the longest common subsequence (LCS) between the prefixes `text2[0..i]` and `text1[0..j]`. The first row and first column are initialized to 0, representing the LCS length when one of the strings is empty. For each cell, if the characters match (`text2[i] = text1[j]`), the LCS length is calculated as $dp(i-1, j-1) + 1$; otherwise, it is the maximum of the cell above and the cell to the left, $dp(i-1, j)$ or $dp(i, j-1)$. The final value at the bottom-right cell, $dp[5][11] = 5$, represents the length of the LCS, which is “fudan”.

8.3 Harder Problems

Not all problems that can be solved via dynamic programming can be efficiently optimized via simple memoization or iterative tabulation. This section discusses such problems.

8.3.1 Hamiltonian Path

A Hamiltonian path in a graph is a path that visits each vertex exactly once. Taking Figure 8.2 as an example, one possible Hamiltonian path in the graph is $A - C - B - D$. If the path is required to start and end at the same vertex, it is called a *Hamiltonian cycle*. Finding such a path is a classical NP-hard problem, meaning that no polynomial-time algorithm is known for solving it in general. Compared to other dynamic programming examples such as Fibonacci or coin change, the Hamiltonian path problem is significantly more challenging.

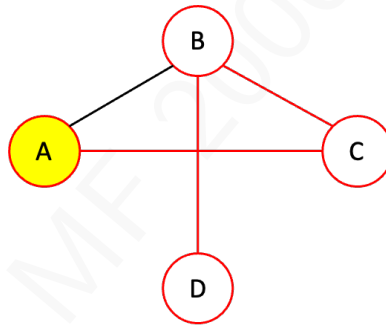


Figure 8.2: A sample graph.

Problem formulation. Let $dp(S, v)$ denote whether there exists a Hamiltonian path that *ends* at vertex v and visits exactly the set of vertices S . The recurrence is defined as follows:

$$dp(S, v) = \bigvee_{u \in S, u \in \text{neighbors}(v)} dp(S \setminus \{v\}, u)$$

The base case occurs when S contains only a single vertex v , in which case

$$dp(\{v\}, v) = \text{True}.$$

Recursive implementation. In practice, we can represent the set S as a bitmask to efficiently encode which vertices have been visited. Code 8.8 implements the Hamiltonian path search recursively.

```

import math
from functools import lru_cache
def hamiltonian_path(graph):
    n = len(graph)

```

```

ALL_VISITED = (1 << n) - 1 # bitmask with all nodes visited
@lru_cache(None)
def dfs(mask, v):
    if mask == (1 << v): # Base case: only u is visited
        return True
    # Try to reach u from any previous vertex v
    for u in range(n):
        if mask & (1 << u) and u != v and graph[u][v]:
            prev_mask = mask ^ (1 << v)
            if dfs(prev_mask, u):
                return True
    return False

for end in range(n): # Try all possible end vertices
    if dfs(ALL_VISITED, end):
        return True
return False

```

Code 8.8: Recursive bitmask DP for Hamiltonian path

Iterative implementation. The same problem can be solved iteratively using DP over subsets of vertices. Each state is a pair $(mask, u)$ indicating whether a path exists ending at vertex u and visiting all vertices in $mask$.

```

n = len(graph)
vertices = list(graph.keys())
index = {v:i for i,v in enumerate(vertices)}

dp = [[False]*n for _ in range(1<<n)]
parent = [[-1]*n for _ in range(1<<n)]

# Initialize starting positions
for i in range(n):
    dp[1<<i][i] = True

for mask in range(1<<n):
    for u in range(n):
        if not dp[mask][u]:
            continue
        for v in range(n):
            if not (mask & (1<<v)) and vertices[v] in graph[vertices[u]]:
                dp[mask | (1<<v)][v] = True
                parent[mask | (1<<v)][v] = u

```

Code 8.9: Bottom-up DP for Hamiltonian Path

8.3.2 Travels Salesman Problem

The Travelling Salesman Problem (TSP) is a weighted version of the Hamiltonian cycle problem. Given a complete weighted graph, the goal is to find the shortest possible tour that visits each vertex

exactly once and returns to the starting vertex. A dynamic programming formulation defines the subproblem as follows: let $dp[S][v]$ be the minimum cost to start from a fixed origin, visit all vertices in the subset $S \subseteq V$, and end at vertex v .

A dynamic programming formulation defines the subproblem as follows: let $dp(S, v)$ denote the minimum cost to start from a fixed origin, visit all vertices in the subset $S \subseteq V$, and end at vertex v . The recurrence relation is:

$$dp(S, v) = \min_{u \in S, u \neq v} (dp(S \setminus \{v\}, u) + w(u, v)),$$

where $w(u, v)$ is the weight of the edge from u to v . The optimal tour cost is then

$$\min (dp(V, v) + w(v, 0)).$$

For both the Hamiltonian path problem and the TSP, the DP formulation has 2^n possible subsets S and n choices for the ending vertex v , giving $O(n \cdot 2^n)$ states. In the TSP, each state requires iterating over up to n vertices to compute the minimum, resulting in an overall time complexity of $O(n^2 \cdot 2^n)$. The space complexity is $O(n \cdot 2^n)$ to store the DP table.