Dynamic Programming

Optimize Recursive Solutions by Reusing Subproblem Results

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction

What is Dynamic Programming?

Dynamic Programming (DP): An optimization technique for recursive problems

Core Idea:

- Break problem into overlapping subproblems
- Solve each subproblem once
- Store results to avoid recomputation
- Build solution from cached subproblem results

Key Difference from Divide-and-Conquer:

- **D&C**: Subproblems are independent (merge sort, quicksort)
- **DP**: Subproblems overlap (fibonacci, shortest path)

When to Use Dynamic Programming

Two Required Properties:

1. **Overlapping Subproblems**
   - Same subproblems solved multiple times in naive recursion
   - Caching results provides significant speedup
2. **Optimal Substructure**
   - Optimal solution contains optimal solutions to subproblems
   - Can build global optimum from local optima

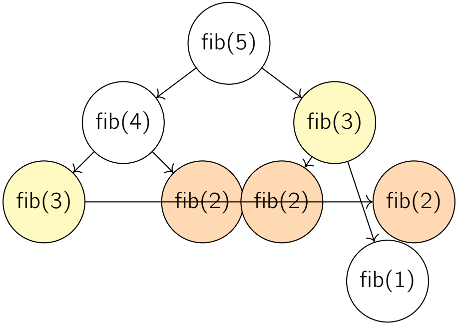
Additional Requirement:

- Must be able to define recursive relation between states

Overlapping Subproblems and Optimal Substructure

Overlapping Subproblems: Fibonacci Example

Problem: Compute n-th Fibonacci number



Observation:

- fib(3) computed 2 times (yellow)
- fib(2) computed 3 times (orange)
- **Exponential time**: $O(2^n)$ without caching

Naive Recursion vs Memoization

Naive Recursion: $O(2^n)$

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

# fib(40) takes seconds!
```

With Memoization: $O(n)$

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

# fib_memo(40) instant!
```

Key Insight: Cache results to avoid recomputation

- Each subproblem solved exactly once
- Lookup takes $O(1)$ time
- Total time: $O(n)$ instead of $O(2^n)$

Optimal Substructure

Definition: Optimal solution contains optimal solutions to subproblems

Example 1: Shortest Path ✓

- If shortest path $A \to C$ goes through B
- Then $A \to B$ and $B \to C$ must also be shortest paths
- Can build optimal solution from optimal subproblems

Counter-example: Longest Simple Path $\times$

- Longest path $A \to C$ through B
- Does **NOT** guarantee $A \to B$ and $B \to C$ are longest
- Why? Cannot revisit nodes (constraint breaks substructure)
- This problem is NP-hard!

Problem Classification

Problem	Overlapping?	Optimal Substructure?	DP?
Fibonacci	Yes	Yes	✓
Shortest path	Yes	Yes	✓
LIS	Yes	Yes	✓
Knapsack	Yes	Yes	✓
Merge sort	No	Yes	× (D&C)
Longest simple path	Yes	No	× (NP-hard)

When to use DP:

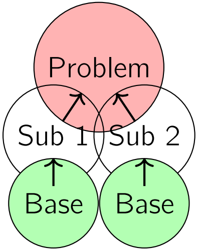
- ✓ Problem has overlapping subproblems
- ✓ Problem has optimal substructure
- ✓ Can define recursive relation
- × Subproblems independent → use divide-and-conquer
- × No optimal substructure → greedy or other approach

Top-down (Memoization) vs Bottom-up

Two Approaches to Dynamic Programming

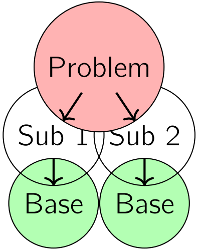
Top-down (Memoization)

- Start from original problem
- Recurse down to base cases
- Cache results along the way
- Natural recursive thinking



Bottom-up (Tabulation)

- Start from base cases
- Build up iteratively
- Fill table in correct order
- No recursion overhead



Example: Climbing Stairs - Top-down

Problem: n stairs, can climb 1 or 2 steps at a time. How many ways to reach top?

Top-down with Memoization:

```python
def climb_stairs_memo(n, memo={}):
    # Base cases
    if n <= 2:
        return n

    # Check cache
    if n in memo:
        return memo[n]

    # Recursive relation: ways(n) = ways(n-1) + ways(n-2)
    memo[n] = climb_stairs_memo(n-1, memo) + \
              climb_stairs_memo(n-2, memo)
    return memo[n]
```

Time: $O(n)$ **Space**: $O(n)$ + recursion stack

Example: Climbing Stairs - Bottom-up

Bottom-up with Tabulation:

```python
def climb_stairs_dp(n):
    if n <= 2:
        return n

    # DP table
    dp = [0] * (n + 1)
    dp[1] = 1
    dp[2] = 2

    # Fill table bottom-up
    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

Time: $O(n)$ **Space**: $O(n)$

No recursion overhead, better cache locality

Comparison: Top-down vs Bottom-up

Aspect	Top-down	Bottom-up
Direction	Problem $\rightarrow$ base cases	Base cases $\rightarrow$ problem
Implementation	Recursion + cache	Iteration + table
Subproblems	Only needed	All
Space	$O(n)$ + stack	$O(n)$
Time overhead	Function calls	None
Intuition	Natural	Requires planning
Space optimization	Harder	Easier

When to choose:

- **Top-down**: Recursive solution natural, not all subproblems needed
- **Bottom-up**: Want best performance, need space optimization

State Definition and Transitions

Designing a DP Solution

Core of DP: Properly defining states and transitions

5-Step Process:

1. **Identify what varies**: What parameters change between subproblems?
2. **Define DP array**: `dp[i]`, `dp[i][j]`, etc.
3. **Specify meaning**: What does `dp[i]` represent?
4. **Find recurrence**: How to compute `dp[i]` from smaller states?
5. **Set base cases**: Initial values for smallest subproblems

State: A unique subproblem characterized by parameters
Good state: Captures all information needed to solve subproblem

Example 1: Longest Increasing Subsequence

Problem: Find length of longest increasing subsequence in array

State definition: `dp[i]` = length of LIS ending at index i

```
def length_of_LIS(nums):
    n = len(nums)
    dp = [1] * n  # Base case: each element is LIS of length 1

    # Transition: for each i, check all j < i
    for i in range(1, n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)  # Answer: maximum among all dp[i]

# Example: [10, 9, 2, 5, 3, 7, 101, 18]
# dp =     [1,  1, 1, 2, 2, 3, 4,   4]
#                        ^        ^
#                    [2,5,7,101] or [2,5,7,18]
```

Time: $O(n^2)$ **Space**: $O(n)$

Example 2: 0/1 Knapsack

Problem: n items with weights and values, capacity W. Maximize value.

State: `dp[i][w]` = max value using first i items with capacity w

```python
def knapsack(weights, values, W):
    n = len(weights)
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    # Transition
    for i in range(1, n + 1):
        for w in range(W + 1):
            # Don't take item i-1
            dp[i][w] = dp[i-1][w]

            # Take item i-1 (if it fits)
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i][w],
                               dp[i-1][w - weights[i-1]] + values[i-1])

    return dp[n][W]
```

Recurrence: `dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i-1]] + value[i-1])`

Example 3: Edit Distance

Problem: Min operations to convert word1 to word2 (insert, delete, replace)

State: `dp[i][j]` = min ops to convert `word1[0..i-1]` to `word2[0..j-1]`

```python
def min_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Base cases
    for i in range(m + 1):
        dp[i][0] = i  # Delete all characters
    for j in range(n + 1):
        dp[0][j] = j  # Insert all characters

    # Transition
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i-1] == word2[j-1]:
                dp[i][j] = dp[i-1][j-1]  # No operation needed
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],      # Delete from word1
                    dp[i][j-1],      # Insert to word1
                    dp[i-1][j-1]     # Replace
                )
    return dp[m][n]
```

Common State Patterns

Pattern	State	Example Problems
Linear	`dp[i]`	Fibonacci, climbing stairs
2D grid	`dp[i][j]`	Unique paths, edit distance
Subsequence	`dp[i]` ending at i	LIS, max subarray
Knapsack	`dp[i][w]`	0/1 knapsack, coin change
Interval	`dp[i][j]` for [i,j]	Matrix chain mult
State machine	`dp[i][state]`	Stock trading

Key Insight: Choose state that:

- Uniquely identifies each subproblem
- Contains all necessary information
- Allows expressing recurrence relation
- Leads to polynomial time/space complexity

1D/2D DP and Space Optimization

Space Optimization: Fibonacci

Observation: Only need previous 2 values

1D DP: $O(n)$ **space**

```python
def fib_1d(n):
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Optimized: $O(1)$ **space**

```python
def fib_optimized(n):
    if n <= 1:
        return n
    prev2, prev1 = 0, 1
    for i in range(2, n + 1):
        curr = prev1 + prev2
        prev2, prev1 = prev1, curr
    return prev1
```

Key Idea: Keep only what you need

- Identify dependencies in recurrence
- Store only necessary previous values
- Update in correct order

2D DP Space Optimization: Unique Paths

Problem: $m \times n$ grid, count paths from top-left to bottom-right

2D DP: $O(m \times n)$

```python
def unique_paths_2d(m, n):
    dp = [[0] * n for _ in range(m)]

    # Base cases
    for i in range(m):
        dp[i][0] = 1
    for j in range(n):
        dp[0][j] = 1

    # Fill table
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + \
                       dp[i][j-1]

    return dp[m-1][n-1]
```

Optimized: $O(n)$

```python
def unique_paths_1d(m, n):
    dp = [1] * n  # Only current row

    for i in range(1, m):
        for j in range(1, n):
            dp[j] = dp[j] + dp[j-1]
            # dp[j]: previous row
            # dp[j-1]: current row

    return dp[n-1]
```

Observation: Each row only depends on previous row

Rolling Array Technique

Idea: Use modulo to reuse array space

```
def optimized_2d(m, n):
    # Only keep 2 rows in memory
    dp = [[0] * n for _ in range(2)]

    for i in range(m):
        for j in range(n):
            if i == 0 or j == 0:
                dp[i % 2][j] = 1
            else:
                dp[i % 2][j] = dp[(i-1) % 2][j] + dp[i % 2][j-1]

    return dp[(m-1) % 2][n-1]
```

When to use:

- State `dp[i][j]` only depends on previous row `dp[i-1][...]`
- Reduces space from $O(m \times n)$ to $O(2 \times n)$ or $O(n)$

State Compression with Bitmasks

Use case: Small state space (e.g., subsets of n items)

Example: Traveling Salesman Problem

```python
def tsp(dist):
    n = len(dist)
    # dp[mask][i] = min cost to visit cities in mask, ending at i
    # mask is bitmask representing visited cities
    dp = [[float('inf')] * n for _ in range(1 << n)]
    dp[1][0] = 0  # Start at city 0

    for mask in range(1 << n):
        for u in range(n):
            if dp[mask][u] == float('inf'):
                continue
            for v in range(n):
                if mask & (1 << v):  # Already visited
                    continue
                new_mask = mask | (1 << v)
                dp[new_mask][v] = min(dp[new_mask][v],
                                      dp[mask][u] + dist[u][v])

    return min(dp[(1<<n)-1][i] + dist[i][0] for i in range(1, n))
```

Space: $O(2^n \times n)$ instead of exponential states

Space Optimization Checklist

Questions to ask:

1. Can I use only $O(1)$ variables instead of array?
   - Example: Fibonacci needs only 2 variables
2. Do I only need the previous row/column?
   - Example: Unique paths, knapsack
3. Can I update in-place without affecting future computations?
   - Example: Coin change with forward iteration
4. Is the state space small enough for bitmask?
   - Example: TSP with $n \leq 20$ cities

Trade-off: Space optimization may reduce code clarity

Combining DP with Data Structures

DP + Hash Map

Use case: Fast lookup of DP states with large or sparse index space

```
# Problem: Count subsequences with sum k
def count_pairs_with_sum(nums, k):
    dp = {}  # dp[sum] = count of subsequences with this sum
    dp[0] = 1  # Empty subsequence

    for num in nums:
        new_dp = dp.copy()
        for s in dp:
            new_sum = s + num
            new_dp[new_sum] = new_dp.get(new_sum, 0) + dp[s]
        dp = new_dp

    return dp.get(k, 0)
```

Benefit:

- No need to allocate large array
- Only store reachable states

DP + Monotonic Deque

Use case: Sliding window optimization in DP

```
from collections import deque

# Problem: Max subarray sum with length constraint (<= k)
def max_subarray_sum_with_constraint(nums, k):
    n = len(nums)
    dp = [0] * n
    dp[0] = nums[0]

    # Monotonic deque maintains decreasing order of dp values
    dq = deque([0])
    result = dp[0]

    for i in range(1, n):
        # Remove elements outside window
        while dq and dq[0] < i - k:
            dq.popleft()

        # dp[i] = max(nums[i], nums[i] + max(dp[j]) for j in [i-k, i-1])
        dp[i] = nums[i]
        if dq:
            dp[i] = max(dp[i], nums[i] + dp[dq[0]])

        # Maintain monotonic property
        while dq and dp[dq[-1]] <= dp[i]:
            dq.pop()
        dq.append(i)
        result = max(result, dp[i])

    return result
```

DP + Trie

Use case: String matching problems

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False

def word_break(s, word_dict):
    # Build Trie
    root = TrieNode()
    for word in word_dict:
        node = root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_word = True

    # DP with Trie
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True

    for i in range(1, n + 1):
        node = root
        for j in range(i - 1, -1, -1):
            if s[j] not in node.children:
                break
            node = node.children[s[j]]
            if node.is_word and dp[j]:
                dp[i] = True
```

Common Data Structure Combinations

Data Structure	Use Case	Example
Hash Map	Fast state lookup	Two Sum, Subarray Sum
Segment Tree	Range queries	LIS with range max
Priority Queue	Track k best/worst	K-th largest
Monotonic Stack	Maintain order	Next greater element
Monotonic Deque	Sliding window	Window maximum
Trie	String prefixes	Word Break
Union-Find	Components	Islands with DP

Key Insight:

- DP handles optimal substructure
- Data structure optimizes state transitions
- Often reduces time complexity by a factor

Common Pitfalls and Patterns

Pitfall 1: Wrong Base Case

Wrong:

```python
def climb_stairs(n):
    dp = [0] * (n + 1)
    dp[1] = 1  # Missing dp[2]
    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]  # Fails for n=2
```

Correct:

```python
def climb_stairs(n):
    if n <= 2:
        return n
    dp = [0] * (n + 1)
    dp[1], dp[2] = 1, 2
    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Lesson: Always verify base cases with hand calculation

Pitfall 2: Index Out of Bounds

Wrong:

```
for i in range(n):
    # Error when i=0
    dp[i] = dp[i-1] + nums[i]
```

Correct:

```
dp[0] = nums[0]
for i in range(1, n):
    dp[i] = dp[i-1] + nums[i]
```

Lesson: Handle first/last elements separately if needed

Pitfall 3: Incorrect Iteration Order

Wrong (0/1 Knapsack):

```
# Forward iteration
for i in range(n):
    for w in range(weights[i], W+1):
        dp[w] = max(dp[w],
            dp[w-weights[i]] + values[i])
# Allows using same item multiple times!
```

Correct:

```
# Backward iteration
for i in range(n):
    for w in range(W, weights[i]-1, -1):
        dp[w] = max(dp[w],
            dp[w-weights[i]] + values[i])
# Each item used at most once
```

Lesson: Iteration order matters for in-place updates

Pitfall 4: Wrong Initialization

Wrong:

```python
# Using 0 for max problem
dp = [0] * n
for i in range(n):
    dp[i] = max(dp[i-1] + nums[i],
                nums[i])
# Wrong if all nums negative
```

Correct:

```python
# Use -inf for max problems
dp = [float('-inf')] * n
dp[0] = nums[0]
for i in range(1, n):
    dp[i] = max(dp[i-1] + nums[i],
                nums[i])
```

Lesson: Initialize based on problem (min: $+\infty$, max: $-\infty$)

Common DP Patterns Summary

1. **Linear DP**: `dp[i]` depends on `dp[i-1]`, `dp[i-2]`
   - Fibonacci, climbing stairs, house robber
2. **Grid DP**: `dp[i][j]` depends on `dp[i-1][j]`, `dp[i][j-1]`
   - Unique paths, minimum path sum
3. **Knapsack**: `dp[i][capacity]` or `dp[capacity]`
   - 0/1 knapsack, coin change, partition
4. **Interval DP**: `dp[i][j]` for range [i, j]
   - Matrix chain multiplication, burst balloons
5. **State Machine**: `dp[i][state]` for different modes
   - Stock trading with cooldown
6. **Bitmask DP**: `dp[mask]` for subsets
   - TSP, assignment problem

Problem-Solving Checklist

When approaching a DP problem:

1. ✓ Identify if DP is applicable
   - Overlapping subproblems?
   - Optimal substructure?
2. ✓ Define state clearly
   - What does `dp[i]` mean?
3. ✓ Write recurrence relation
4. ✓ Identify base cases
5. ✓ Determine iteration order
   - Which states depend on which?
6. ✓ Consider space optimization
7. ✓ Test with small examples
8. ✓ Handle edge cases
   - Empty input, single element, etc.

Debugging DP Solutions

Techniques:

- **Print DP table**
  - Visualize how values are computed
  - Spot incorrect transitions
- **Verify base cases**
  - Hand-calculate smallest instances
- **Check recurrence**
  - Does it match problem statement?
- **Test simple inputs first**
  - Edge cases: $n = 0, 1, 2$
- **Compare approaches**
  - Memoization vs tabulation should give same result

Summary

Dynamic Programming: Key Takeaways

Core Concepts:

- DP optimizes recursive solutions by caching subproblem results
- Requires overlapping subproblems + optimal substructure
- Two approaches: top-down (memoization) vs bottom-up (tabulation)

Design Process:

1. Define state (what varies?)
2. Find recurrence relation
3. Set base cases
4. Determine iteration order
5. Optimize space if needed

Advanced Techniques:

- Space optimization (rolling array, state compression)
- Combining with data structures (hash map, deque, trie)
- Recognizing common patterns (linear, grid, knapsack, interval, etc.)

Complexity Analysis

Time Complexity:

- Number of states $\times$ time per state
- Linear DP: $O(n)$
- 2D DP: $O(n^2)$ or $O(n \times m)$
- Knapsack: $O(n \times W)$
- Interval DP: $O(n^3)$
- Bitmask DP: $O(2^n \times n)$

Space Complexity:

- Without optimization: same as time
- With optimization: often $O(n)$ or $O(1)$
- Top-down: add $O(n)$ for recursion stack

Practice Problems

Beginner:

- Fibonacci, Climbing Stairs
- Min Cost Climbing Stairs
- House Robber

Intermediate:

- Longest Increasing Subsequence
- Coin Change
- Edit Distance
- Unique Paths

Advanced:

- 0/1 Knapsack
- Longest Common Subsequence
- Matrix Chain Multiplication
- Stock Trading with Cooldown
- Traveling Salesman Problem

Resources

Online Judges:

- LeetCode DP tag problems
- Codeforces DP problems
- AtCoder Educational DP Contest

Books:

- *Introduction to Algorithms* (CLRS) - Chapter 15
- *Algorithm Design* by Kleinberg & Tardos
- *Competitive Programming 3* by Halim

Tips:

- Practice regularly - DP requires pattern recognition
- Start with simple problems and build up
- Understand the recurrence, not just memorize solutions