**Dynamic Programming**

Dynamic Programming (DP) is a powerful technique used in algorithm design to solve problems by breaking them down into smaller overlapping subproblems and **solving each subproblem only once**. It is particularly useful for optimization problems where the solution can be **constructed efficiently using previously computed results.**

**Key Concepts of Dynamic Programming**

1. **Overlapping Subproblems**
   * The problem can be divided into smaller subproblems that are reused multiple times.
   * Example: Fibonacci series computation, where fib(n) = fib(n-1) + fib(n-2).
2. **Optimal Substructure**
   * The optimal solution to a problem can be formed using the optimal solutions of its subproblems.
   * Example: The shortest path in a graph, where the shortest path from A → C is the shortest path from A → B + shortest path from B → C.

**Approaches to Dynamic Programming**

There are two main ways to implement DP:

1. **Top-Down Approach (Memoization)**
   * Solve the problem recursively and store the results of subproblems to avoid recomputation.
   * Useful when only some subproblems need to be solved.
   * Example: Fibonacci with Memoization

def fib\_memo(n, memo={}):  
 if n in memo: *# Check if result is already computed* return memo[n]  
 if n <= 1:  
 return n  
 memo[n] = fib\_memo(n - 1, memo) + fib\_memo(n - 2, memo) *# Store result in memo* return memo[n]  
  
print(fib\_memo(10)) *# Output: 55*print(fib\_memo(100)) *# Output: 354224848179261915075 (Efficiently computed)*

1. **Bottom-Up Approach (Tabulation)**
   * Solve all subproblems first and use them to build the final solution iteratively.
   * More efficient in terms of space since it avoids recursion.
   * Example: Fibonacci with Tabulation

def fib\_tabulation(n):  
 if n <= 1:  
 return n  
 dp = [0] \* (n + 1) *# Create DP table* dp[1] = 1 *# Base case* for i in range(2, n + 1):  
 dp[i] = dp[i - 1] + dp[i - 2] *# Compute Fibonacci iteratively* return dp[n]  
  
print(fib\_tabulation(10)) *# Output: 55*print(fib\_tabulation(100)) *# Output: 354224848179261915075*

### **1️⃣ Understanding Top-Down (Memoization) vs. Bottom-Up (Tabulation)**

|  |  |  |
| --- | --- | --- |
| Feature | **Top-Down (Memoization)** | **Bottom-Up (Tabulation)** |
| **Approach** | Recursive (solves subproblems on demand) | Iterative (solves subproblems in a predefined order) |
| **Computation Order** | Solves only necessary subproblems first | Solves all subproblems first before reaching the final answer |
| **Storage** | Uses a hashmap or array to store results of already computed subproblems | Uses a table (array) to store results iteratively |
| **Performance** | Can lead to deep recursion (stack overflow for large inputs) | Avoids recursion, so no stack overhead |
| **Ease of Implementation** | More intuitive (direct translation of recursion) | Sometimes harder to figure out the iterative approach |
| **Use Case** | Useful when only a few subproblems are needed | Preferred when all subproblems are required for the solution |
| **Example Execution** (for fib(5)) | Solves fib(5), calls fib(4), fib(3), etc., storing results as needed | Computes fib(0), fib(1), fib(2), ..., up to fib(5) in order |

### **2️⃣ When to Prefer Memoization vs. Tabulation?**

✅ **Use Memoization (Top-Down) when:**

✔ The problem has a **natural recursive structure**.  
✔ Not all subproblems need to be computed (e.g., solving a specific case instead of filling a table).  
✔ You want an **easier implementation** by just modifying a recursive approach.

❌ **Avoid Memoization if:**

✘ Recursion depth is too large, which may cause **stack overflow**.  
✘ You need all subproblems anyway, making recursion unnecessary.

✅ **Use Tabulation (Bottom-Up) when:**

✔ The problem has **clear ordering of subproblems** (e.g., DP[i] depends on DP[i-1], DP[i-2], etc.).  
✔ You want to **optimize space complexity** by only storing the necessary results.  
✔ You need better **performance** since it avoids recursion overhead.

❌ **Avoid Tabulation if:**

✘ Finding the **iteration order is difficult**.  
✘ The problem is **more naturally solved with recursion**.

### **4️⃣ Key Takeaways**

🚀 **If recursion is natural for the problem, use Memoization (Top-Down).**  
⚡ **If iteration is straightforward, use Tabulation (Bottom-Up).**  
🔄 **For space efficiency, modify Tabulation to use only a few variables instead of an entire table (e.g., optimized Fibonacci uses just two variables).**

**Common DP Problems**

1. **Fibonacci Numbers** (Simple DP)
2. **Knapsack Problem** (0/1 and Unbounded)
3. **Longest Common Subsequence (LCS)**
4. **Longest Increasing Subsequence (LIS)**
5. **Coin Change Problem** (Minimum number of coins)
6. **Edit Distance** (Convert one string to another)
7. **Matrix Chain Multiplication**
8. **Subset Sum Problem**
9. **Rod Cutting Problem**
10. **Minimum Path Sum in a Grid**