AD Midterm

1. The Master Logic Table

Scan this first to identify which problem you are facing.

Problem	Goal	The Recurrence / Core Logic	Time Complexity
Climbing Stairs	Count ways to reach top.	`dp[i] = dp[i-1] + dp[i-2]`	$O(N)$
Min Cost Stairs	Min energy to reach top.	`dp[i] = cost[i] + min(dp[i-1], dp[i-2])`	$O(N)$
Cut Rod	Max profit cutting a rod.	`val[i] = max(price[k] + val[i-k])` for all cuts `k`	$O(N^2)$
Min Coin	Min coins for amount `V`.	`dp[v] = 1 + min(dp[v - c])` for all coins `c`	$O(V \cdot N)$
0/1 Knapsack	Max value with weight limit.	`max( val[i] + dp[w-wt[i]], dp[w] )` (Include vs Exclude)	$O(N \cdot W)$
LCS	Longest Common Subsequence.	Match? `1 + dp[i-1][j-1]`. No Match? `max(dp[i-1][j], dp[i][j-1])`	$O(N \cdot M)$
Edit Distance	Min ops to convert A to B.	Match? `dp[i-1][j-1]`. No? `1 + min(Insert, Delete, Replace)`	$O(N \cdot M)$
MCS	Max contiguous subarray sum.	`current = max(num, current + num)` (Kadane's)	$O(N)$
Balance Split	Split set into two equal sums.	`subset_sum(i-1, current_sum - nums[i])` (Target = Total/2)	$O(N \cdot Sum)$
Combination	Generate all subsets.	Select `x[i]=1`, recurse; Unselect `x[i]=0`, recurse.	$O(2^N)$

---

2. Universal Code Templates

If the question asks for code, copy one of these and rename variables.

A. The "Memoization" Template (Top-Down)

Best for: "Min Coin", "Knapsack", "Cut Rod", "Edit Distance". Safest to write.

```
import sys
sys.setrecursionlimit(20000) # ALWAYS WRITE THIS

memo = {}

def solve(i, capacity): # i = index/item, capacity = weight/money/length
    state = (i, capacity)

    # 1. CHECK MEMO
    if state in memo: return memo[state]

    # 2. BASE CASES
    if capacity == 0: return 0            # Found solution / Filled bag
    if capacity < 0: return float('-inf') # Invalid (capacity exceeded)
    if i < 0: return 0                    # No items left

    # 3. RECURSIVE CHOICES (Knapsack Example)
    # Option A: Skip item 'i'
    res_skip = solve(i - 1, capacity)

    # Option B: Take item 'i'
    res_take = val[i] + solve(i - 1, capacity - wt[i])
```

```
    # 4. STORE & RETURN
    memo[state] = max(res_skip, res_take) # Use min() for cost problems
    return memo[state]
```

B. The "Tabulation" Template (Bottom-Up)

Best for: "Climbing Stairs", "LCS", or if the question demands a "Table".

```
# Setup table. Rows = Items (0 to n), Cols = Capacity (0 to W)
# Initialize with 0 (for maximization) or infinity (for minimization)
dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

for i in range(1, n + 1):
    for w in range(1, W + 1):
        if weight[i-1] <= w:
            # Logic: max( exclude, include )
            dp[i][w] = max(dp[i-1][w], val[i-1] + dp[i-1][w - weight[i-1]])
        else:
            dp[i][w] = dp[i-1][w]

print(dp[n][W]) # Answer is usually in the bottom-right
```

---

3. Problem-Specific Cheat Codes

1. Minimum Coin Change

- **Logic:** You can use the same coin **unlimited** times.
- **Code Tweak:** In the recursive step, call `solve(i, ...)` (stay at same coin) instead of `solve(i-1, ...)` (move to next).
- **Base Case:** If `v == 0` return 0. If `v < 0` return `infinity`.

2. Longest Common Subsequence (LCS)

- **Visual:** Comparing String A (rows) and B (cols).
- **Logic:**
  - If `A[i] == B[j]`: They match! Cost is `1 + solve(i-1, j-1)`.
  - If `A[i] != B[j]`: No match. Try skipping A's char `solve(i-1, j)` vs skipping B's char `solve(i, j-1)`. Take `max`.

3. Edit Distance (Levenshtein)

- **Logic:** Same grid as LCS, but 3 options for mismatch.
- **Code Tweak:**
  - `Insert`: `solve(i, j-1)`
  - `Delete`: `solve(i-1, j)`
  - `Replace`: `solve(i-1, j-1)`
  - Take `1 + min(Insert, Delete, Replace)`.

4. Climbing Stairs

- **Trick:** `dp` array size is `N+1`.
- **Init:** `dp[0]=cost[0]`, `dp[1]=cost[1]`.
- **Loop:** `range(2, n)`.

5. Balance Split

- **Logic:** It is a hidden Knapsack problem.
- **Check:** First, calculate `Total_Sum`. If it's **odd**, return `False` (impossible to split equally).
- **Goal:** Find a subset that sums to `Total_Sum / 2`.

6. Ultimate Greyness

- **Likely Context:** This sounds like "Optimal Binary Search Tree" or "Matrix Chain Multiplication" (minimizing cost of combining things).
- **Logic:** It's an **Interval DP** problem.
- **Loop:** You iterate over `length` (from 2 to n), then `i` (start), then `k` (split point).
- **Recurrence:** `dp[i][j] = min(dp[i][k] + dp[k+1][j]) + Cost(i, j)`

---

4. Essay Modules (For the text questions)

Q1: "Why did you use Dynamic Programming?"

> *"I chose Dynamic Programming because this problem exhibits two properties:*
>
> 1. ***Overlapping Subproblems:*** *A recursive solution would repeatedly solve the same small sub-problems (like calculating the cost for step i multiple times), leading to exponential time complexity.*
> 2. *Optimal Substructure: The optimal solution can be constructed from the optimal solutions of its sub-problems (e.g., the min cost to reach step i relies on the min cost of steps $i-1$ and $i-2$).*
>
> *Using a Memoization table allows us to store these results, reducing complexity to Polynomial time."*

Q2: "Analyze the Complexity."

> - ***Time:*** *"Proportional to the number of states. Here, we have N items and capacity W, so there are $N \times W$ unique states. Each state takes $O(1)$ to compute. Total: $O(N \cdot W)$."*
> - ***Space:*** *"Required for the recursion stack ($O(N)$) and the memoization table ($O(N \cdot W)$). Total: $O(N \cdot W)$."*

How to use this:

1. **Don't Panic.** You have the knowledge; this is just an index for your brain.
2. **Identify the Problem:** Use the "Mega-Map" (Section 1) to figure out what kind of problem it is.
3. **Choose Your Weapon:** For code questions, grab the appropriate **Master Template** (Section 2) and plug in the problem-specific logic from Section 3.
4. **Answer Essays:** Use the pre-written **Essay Modules** (Section 5) to structure your theoretical answers quickly.

---

1. The "Mega-Map": Logic & Complexity At-A-Glance

Scan this first to match the exam question to a known problem type.

Problem Topic	Core Goal	Key Logic / Recurrence	Time Complexity (DP/Greedy)
Combination	Generate all binary strings of length N.	`solve(i+1)` after setting `x[i]=0`, then `x[i]=1`	$O(2^N)$

Problem Topic	Core Goal	Key Logic / Recurrence	Time Complexity (DP/Greedy)
Balance Split	Can set be partitioned into two equal sums?	Subset Sum: `isSubsetSum(i-1, sum) or isSubsetSum(i-1, sum - set[i-1])`	$O(N \cdot Sum)$
MCS (Kadane's)	Find max sum subarray.	`curr_max = max(nums[i], curr_max + nums[i])`	$O(N)$
Climbing Stairs	Count ways to reach the Nth step.	`dp[i] = dp[i-1] + dp[i-2]`	$O(N)$
Min Cost Stairs	Min cost to reach the top.	`dp[i] = cost[i] + min(dp[i-1], dp[i-2])`	$O(N)$
Min Coin Change	Min coins to make amount V.	`dp[v] = 1 + min(dp[v - coin])` for all coins	$O(V \cdot N_{coins})$
Cut Rod	Max profit from cutting a rod of length N.	`dp[i] = max(price[j] + dp[i-j-1])` for all cuts `j`	$O(N^2)$
0/1 Knapsack	Max value within capacity W.	`max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])`	$O(N \cdot W)$
LCS	Longest Common Subsequence length.	Match: `1 + dp[i-1][j-1]`. No Match: `max(dp[i-1][j], dp[i][j-1])`	$O(N \cdot M)$
Edit Distance	Min ops to convert str1 to str2.	Match: `dp[i-1][j-1]`. Mismatch: `1 + min(Insert, Delete, Replace)`	$O(N \cdot M)$
Greedy Coins	Min coins (specific systems only).	Always pick the largest denomination $\leq$ remaining amount.	$O(N_{coins})$ or $O(V)$
Activity Selection	Max non-overlapping activities.	Sort by finish time. Pick next if it starts after last finished.	$O(N \log N)$
Fractional Knapsack	Max value (can take fractions).	Sort by value/weight ratio. Take whole items, then fraction of last.	$O(N \log N)$
"Ultimate Greyness"	(Assumed Interval DP like Matrix Chain)	`dp[i][j] = min(dp[i][k] + dp[k+1][j] + cost(i,k,j))` for $i \leq k < j$	$O(N^3)$

---

2. Master Code Templates (Python)

Copy-paste these and fill in the blanks based on the problem type.

Template A: Memoization (Top-Down DP)

Best for: Knapsack, LCS, Edit Distance, Min Coin, Cut Rod.

```
import sys
sys.setrecursionlimit(10000) # CRITICAL for deep recursion

memo = {}

def solve(i, j): # State variables change based on problem (e.g., index, capacity)
    state = (i, j)
    if state in memo: return memo[state]

    # --- BASE CASES ---
    # e.g., if i == 0 or j == 0: return 0
    if i < 0 or j < 0: return float('inf') if minimizing else float('-inf')

    # --- RECURSIVE STEP (The "Hook") ---
    # Plug in problem-specific logic here. Example for Knapsack:
    # res = max(solve(i-1, j), val[i-1] + solve(i-1, j-wt[i-1]))

    memo[state] = res # Store result
    return res
```

Template B: Tabulation (Bottom-Up DP)

Best for: Climbing Stairs, LCS, Edit Distance, or when iterative is required.

```
def solve_dp(n, m): # n, m are dimensions (e.g., string lengths, capacity)
    # Initialize table with base values (0 for max, inf for min)
    # Create (n+1) x (m+1) table. Rows=i, Cols=j
    dp = [[0 for _ in range(m + 1)] for _ in range(n + 1)]

    # --- FILL TABLE ---
    # Outer loops iterate through state space
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            # --- DP LOGIC (The "Hook") ---
            # Plug in the recurrence relation here. Example for LCS:
            # if S1[i-1] == S2[j-1]:
            #     dp[i][j] = 1 + dp[i-1][j-1]
            # else:
            #     dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[n][m] # Answer is usually in the bottom-right corner
```

---

3. Problem-Specific "Hooks" & Details

Insert these into the templates above.

DP Problems (The "Big 8")

1. **Climbing Stairs & Min Cost Stairs**
   - **State:** `i` (current step).
   - **Tabulation Hook:** `dp[i] = dp[i-1] + dp[i-2]` (or with costs).
   - **Base Cases:** `dp[0]`, `dp[1]` are fixed. Loop starts from 2.
2. **Min Coin Change (Unbounded)**
   - **State:** `v` (current amount).
   - **Memo Hook:** `res = 1 + min(solve(v - coin) for coin in coins)`
   - **Important:** Can reuse coins, so recursion stays at same coin index (or iterates all coins for a 1D DP state).
3. **Cut Rod (Unbounded)**
   - **State:** `n` (remaining length).
   - **Memo Hook:** `res = max(prices[i] + solve(n - (i+1)) for i in range(n))`
   - **Similar to Min Coin:** Unbounded choices at each step.
4. **0/1 Knapsack**
   - **State:** `i` (item index), `w` (remaining capacity).
   - **Memo Hook:** `res = max(solve(i-1, w), values[i-1] + solve(i-1, w - weights[i-1]))`
   - **Important:** `i-1` because you can't reuse items.
5. **LCS (Longest Common Subsequence)**
   - **State:** `i`, `j` (indices in strings S1, S2).
   - **Memo Hook:**

     ```
     if S1[i-1] == S2[j-1]:
         res = 1 + solve(i-1, j-1)
     else:
         res = max(solve(i-1, j), solve(i, j-1))
     ```
6. **Edit Distance**

- **State:** `i`, `j` (indices in strings S1, S2).
- **Memo Hook:**

```
if S1[i-1] == S2[j-1]:
    res = solve(i-1, j-1) # Match, cost 0
else:
    res = 1 + min(solve(i, j-1),   # Insert
                  solve(i-1, j),   # Delete
                  solve(i-1, j-1)) # Replace
```

7. **Balance Split (Partition Problem)**
   - **Logic:** It's a 0/1 Subset Sum problem. Target sum is `TotalSum / 2`.
   - **Quick Check:** If `TotalSum % 2 != 0`, return `False` immediately.
   - **DP State:** `dp[i][s]` = is it possible to get sum `s` using first `i` items?
8. **MCS (Maximum Contiguous Subarray Sum)**
   - **Algorithm:** Kadane's Algorithm (Iterative, not typical DP table).
   - **Code:**

```
max_so_far = nums[0]
current_max = nums[0]
for i in range(1, len(nums)):
    current_max = max(nums[i], current_max + nums[i])
    max_so_far = max(max_so_far, current_max)
return max_so_far
```

Greedy Algorithms

- **Core Idea:** Make the locally optimal choice at each step and hope it leads to the global optimum. No backtracking, no memoization table.

1. **Activity Selection**
   - **Logic:** Pick the next activity that finishes earliest and doesn't overlap.
   - **Code Snippet:**

```
activities.sort(key=lambda x: x[1]) # Sort by finish time
last_finish_time = 0
count = 0
for start, finish in activities:
    if start >= last_finish_time:
        count += 1
        last_finish_time = finish
return count
```

2. **Fractional Knapsack**
   - **Logic:** Pick items with the highest value-to-weight ratio first. Take fractions if needed.
   - **Code Snippet:**

```
# items is list of (value, weight)
items.sort(key=lambda x: x[0]/x[1], reverse=True) # Sort by V/W ratio
total_value = 0
for val, wt in items:
    if capacity >= wt: # Take whole item
        capacity -= wt
        total_value += val
    else: # Take fraction
        total_value += val * (capacity / wt)
```

```
        break # Knapsack full
return total_value
```

---

4. The "Ultimate Greyness" (Interval DP)

Assuming this refers to Matrix Chain Multiplication or Optimal Binary Search Tree style problems.

- **State:** `dp[i][j]` represents the optimal cost for the interval from index `i` to `j`.
- **Base Case:** `dp[i][i] = 0` (Cost of a single item is 0).
- **Tabulation Template:**

```
n = len(dims) - 1
dp = [[0]*n for _ in range(n)]
for length in range(2, n + 1): # Chain length from 2 to n
    for i in range(n - length + 1):
        j = i + length - 1
        dp[i][j] = float('inf')
        for k in range(i, j): # Try all split points k
            # cost() is problem-specific (e.g., dims[i]*dims[k+1]*dims[j+1])
            q = dp[i][k] + dp[k+1][j] + cost(i, k, j)
            if q < dp[i][j]:
                dp[i][j] = q
return dp[0][n-1]
```

---

5. Essay Modules (Cut & Paste Theory)

Module A: Why use Dynamic Programming?

> ***Question:*** *Explain the intuition behind using DP for this problem.*
>
> ***Answer:*** *"I chose a Dynamic Programming approach because this problem exhibits two essential properties that make DP suitable:*
>
> 1. ***Overlapping Subproblems:*** *A naive recursive solution would repeatedly solve the same small sub-problems multiple times, leading to exponential time complexity. For example, calculating* `solve(i, j)` *would happen many times across different branches of the recursion tree.*
> 2. *Optimal Substructure: The optimal solution to the larger problem can be constructed efficiently from the optimal solutions of its smaller sub-problems.*
>
>   *By using [Memoization/Tabulation], we store the results of these sub-problems, ensuring each is solved only once. This drastically reduces the time complexity from exponential to polynomial, specifically* $O(\text{number of states})$*."*

Module B: Greedy vs. Dynamic Programming

> ***Question:*** *Why is a Greedy approach not suitable here? / Compare Greedy and DP.*
>
> ***Answer:*** *"A Greedy algorithm makes the locally optimal choice at each step with the hope of finding the global optimum. It does not reconsider previous choices.*
>
> - ***Greedy works*** *only when the problem has the 'Greedy Choice Property', meaning a local optimum always leads to a global optimum (e.g., Fractional Knapsack).*
> - ***DP is required*** *when a locally optimal choice might lead to a suboptimal global solution. DP explores all possible combinations of choices (implicitly or explicitly) but does so efficiently by storing results of*

> *subproblems. For problems like 0/1 Knapsack or Shortest Path in a graph with negative edges, a greedy choice can lead to a dead end or a non-optimal result, making DP necessary."*

Module C: Complexity Analysis

> ***Question:*** *Analyze the time and space complexity of your DP solution.*
>
> ***Answer:***
>
> - ***Time Complexity:*** *The time complexity of a DP solution is generally $O(\text{number of states} \times \text{time per state})$. In this problem, the state is defined by [dimensions, e.g., index `i` and capacity `w`]. The number of states is $O(N \cdot W)$. Calculating each state takes $O(1)$ time [or $O(N)$ if there's an inner loop like in Cut Rod]. Therefore, the total time complexity is $O(N \cdot W)$.*
> - ***Space Complexity:*** *The space complexity is determined by the size of the DP table (or memoization dictionary) used to store subproblem results, which is $O(\text{number of states})$. Thus, the space complexity is $O(N \cdot W)$. [Mention if space optimization to $O(W)$ is possible for 2D DP problems relying only on the previous row].*

Final Pre-Exam Checklist:

1. **Set Recursion Limit:** `import sys; sys.setrecursionlimit(20000)` at the top of Python recursive code.
2. **Base Cases First:** Always write the base cases of your recursion before the recursive step.
3. **Check Indices:** Be careful with 0-based vs 1-based indexing, especially when mapping problem inputs to DP table rows/columns.
4. **Min vs. Max:** Are you minimizing cost or maximizing value? Initialize with `float('inf')` or `float('-inf')`/`0` respectively.