Problem	Recurrence Formula
Fibonacci Numbers	dp[i] = dp[i-1] + dp[i-2]
Climbing Stairs	dp[i] = dp[i-1] + dp[i-2]
Min Cost Climbing Stairs	dp[i] = cost[i] + min(dp[i-1], dp[i-2])
House Robber	dp[i] = max(dp[i-1], dp[i-2] + nums[i])
Decode Ways	dp[i] = dp[i-1] + dp[i-2] (if valid)
Tiling Problem	dp[i] = dp[i-1] + dp[i-2]
Generalized k-step	dp[i] = dp[i-1]+dp[i-2]+...+dp[i-k]

Cheat Sheet Table : Fibonacci Family Tree (1D DP Problems)

That "1D DP" Means

- The DP state has **one dimension** (like dp[i]).
- Each state depends only on previous states in that same dimension.

```
dp[0] = 0
dp[1] = 1
for i in range(2, n+1):
    dp[i] = dp[i-1] + dp[i-2]
```

Cheat Sheet Table : Knapsack Variant

- **Base Problem: 0/1 Knapsack**
  - **Given:** n items with value[i], weight[i] and capacity W.
  - **Goal:** Maximize total value without exceeding W.
  - **Choice:** For each item i, take it (if weight ≤ capacity) or skip it.

```
dp[i][w] = max(
  dp[i-1][w],                          # skip item i
  value[i-1] + dp[i-1][w - weight[i-1]]    # take item i
)
```

Variant	State	Recurrence
0/1 Knapsack	dp[i][w]	max(dp[i-1][w], val+dp[i-1][w-wt])
Subset Sum	dp[i][s] (True/False)	dp[i-1][s] or dp[i-1][s-arr[i]]

Partition Equal Subset Sum	Same as Subset Sum	Check sum/2
Count of Subsets with Given Sum	Count ways	dp[i-1][s]+dp[i-1][s-arr[i]]
Min Subset Sum Diff	Based on achievable sums	Pick closest to sum/2
Unbounded Knapsack	Unlimited copies	max(dp[i-1][w], val+dp[i][w-wt])
Rod Cutting	Like Unbounded Knapsack	Max profit by lengths
Coin Change	Ways/Min coins (Unbounded)	Ways: dp[i][amt]=dp[i-1][amt]+dp[i][amt-coin[i]]
Bounded Knapsack	Limited copies	Convert to multiple 0/1
Multi-Dimensional Knapsack	Multi-constraints	dp[i][w1][w2] etc.
Fractional Knapsack	Greedy	Sort by value/weight

🌳 Grid / Matrix DP Problem Family

The **base structure**:
You have a **2D grid/matrix**.
Your state = dp[i][j] (usually row, col).
You fill the grid based on **neighbors** or **previous rows**.

Problem	State dp	Recurrence
Unique Paths	Ways to reach (i,j)	dp[i][j]=dp[i-1][j]+dp[i][j-1]
Min Path Sum	Min cost to (i,j)	grid[i][j]+min(dp[i-1][j],dp[i][j-1])
Falling Path Sum	Min cost from top to bottom	grid[i][j]+min(dp[i-1][j-1],dp[i-1][j],dp[i-1][j+1])
Triangle Min Path Sum	Min cost in triangle	triangle[i][j]+min(dp[i+1][j],dp[i+1][j+1])
Maximal Square	Largest square of 1's	1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
LCS	LCS length up to i,j	match: 1+dp[i-1][j-1] else max(dp[i-1][j],dp[i][j-1])
Edit Distance	Min edits up to i,j	if match: dp[i-1][j-1] else 1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
Cherry Pickup	Two travelers DP	dp[x1][y1][x2]

Problem	State dp	Recurrence
Unique Paths	Ways to reach (i,j)	dp[i][j]=dp[i-1][j]+dp[i][j-1]
Min Path Sum	Min cost to (i,j)	grid[i][j]+min(dp[i-1][j],dp[i][j-1])

Falling Path Sum	Min cost from top to bottom	grid[i][j]+min(dp[i-1][j-1],dp[i-1][j],dp[i-1][j+1])
Triangle Min Path Sum	Min cost in triangle	triangle[i][j]+min(dp[i+1][j],dp[i+1][j+1])
Maximal Square	Largest square of 1’s	1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
LCS	LCS length up to i,j	match: 1+dp[i-1][j-1] else max(dp[i-1][j],dp[i][j-1])
Edit Distance	Min edits up to i,j	if match: dp[i-1][j-1] else 1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
Cherry Pickup	Two travelers DP	dp[x1][y1][x2]

Strings DP

Problem	**State & Recurrence**
LCS	if match:1+dp[i-1][j-1] else max(dp[i-1][j],dp[i][j-1])
Longest Common Substring	if match:1+dp[i-1][j-1] else 0
Edit Distance	if match:dp[i-1][j-1] else 1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
Longest Palindromic Subsequence	if s[i]==s[j]:2+dp[i+1][j-1] else max(dp[i+1][j],dp[i][j-1])
Palindrome Partitioning	dp[i]=min(dp[j-1]+1) if s[j..i] palindrome
Decode Ways	dp[i]=dp[i-1]+dp[i-2] if valid
Distinct Subsequences	if match:dp[i-1][j-1]+dp[i-1][j] else dp[i-1][j]
Interleaving String	dp[i][j]= (dp[i-1][j] or dp[i][j-1]) if chars match
Regex Matching	Handles . and *
Wildcard Matching	Handles ? and *

Pattern	**Typical DP Table Shape**
Two strings (compare)	`dp[i][j]` from diagonal/top/left
One string (palindrome)	`dp[i][j]` on substrings (i..j)
One string (count ways)	`dp[i]` for prefix length
Pattern matching	`dp[i][j]` for string vs pattern