## **Dynamic Programming**

Bottom Up



## **©Lecture Flow**

- Prerequisites
- What it is
- How to convert top down to bottom up?
- Common Variants
- Common Pitfalls
- Practice Questions
- Quote of the day



## Prerequisites

DP - Top Down Approach



## Recap Questions

- What is the Time Complexity of DP in terms of States
  - o Imagine you have a DP solution with three states
    - i:0 < i < 1,000
    - i: 0 < j < 10
    - k: 0 < k < 10
- Is DP brute force? Explain?
- What are problems that have Optimal Substructure, Overlapping Subproblems and both together?
- What are the three main things we need to figure out with a top-down approach?





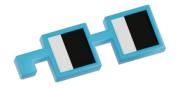
A method for solving complex problems by **breaking them down** into simpler subproblems and **reusing solutions** to these subproblems to construct the solution to the original problem.



## Revision: Use cases

When the problem exhibits the properties of **overlapping subproblems** and **optimal substructure**.





# What is bottom-up (tabulation) dp?



## What is bottom-up dp?

- Bottom-up DP: Builds from smaller to larger problems.
- Top-down DP: Breaks a larger problem into smaller sub-problems.

#### Fibonacci

- Bottom-up: dp[i-2] + dp[i-1] known.
- Top-down: dp[i-2] + dp[i-1] initially unknown.



## Goals

- Has better time-efficiency, we don't have to pay for recursion.
- Opens doors to memory optimization.



## Non-goals

Better asymptotic time complexity.





# How do we go from top-down to bottom-up dp?



## Recipe: Top-down to bottom-up DP

- 1. Represent your state-space with some kind of data structure
- 2. Initialize the base cases in the data structure
- 3. Figure-out the **state transitions.**
- 4. **Fill the answers** in our data-structure (while respecting the dependency order).
- 5. **Return the answer** for the initial state.



### 509. Fibonacci Number

The **Fibonacci numbers**, commonly denoted F(n) form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F(0) = 0$$
,  $F(1) = 1$   
 $F(n) = F(n - 1) + F(n - 2)$ , for  $n > 1$ .

Given n, calculate F(n).





## Step 1: Represent your state-space.

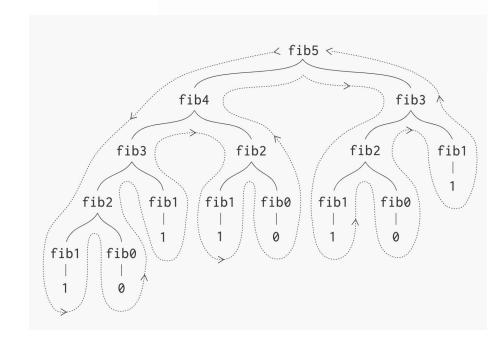
What would be a good fit? Array? Matrix?



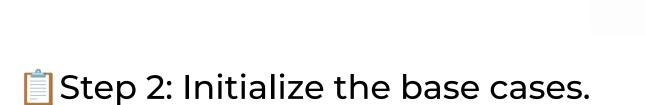
## Step 1: Represent your state-space.

#### An array, right?

$$dp = [0, 1, 1, 2, ...$$
fib0, fib1, fib2, fib3, ...







What are the base cases?



## Step 2: Initialize the base cases.

$$dp[0] = 0$$
 $dp[1] = 1$ 
 $dp = [0, 1, _, _, _, _, _, ...]$ 





What do you refer when calculating for fib(x)?



## Step 3: Figure-out the state transitions.

Focus on the dependencies.

$$dp = [0, 1, 1, 2, 3, 5, 8, ...]$$

$$0, 1, 2, 3, 4, 5, 6, ...$$





But, in what order?



## Dependency order:

$$[0, 1] => 2 => 3 => 4 => 5$$





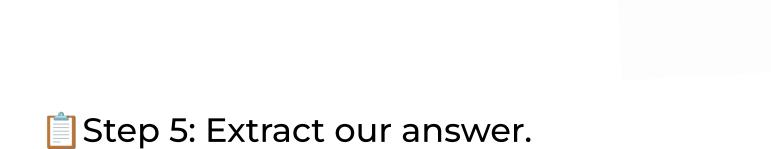


answers = 
$$\begin{bmatrix} 0, & 1, & 1 & , & 2 & , & _ & , & _ & ] \\ 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

answers = 
$$\begin{bmatrix} 0, & 1, & 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$



answers = 
$$\begin{bmatrix} 0, & 1, & 1, & 2, & 3, & 5 \end{bmatrix}$$
  
0 1 2 3 4 5



Which cell contains our answer?



## Step 5: Extract our answer.

answers = 
$$[0, 1, 1, 2, 3, 5]$$
  
0 1 2 3 4 5



## Questions?





## Let's do another one.



#### 62. Unique Paths

There is a robot on an  $m \times n$  grid. The robot is initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m-1][n-1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to  $2 * 10^9$ .



## Top-down approach

```
class Solution:
  def uniquePaths(self, m: int, n: int) -> in
       @cache
       def uniquePathsRec(coord):
           row , col = coord
           if coord == (m - 1, n - 1):
               return 1
           ways = 0
           if row + 1 < m:
               ways += uniquePathsRec((row + 1, col))
           if col + 1 < n:
               ways += uniquePathsRec((row, col + 1))
           return ways
       return uniquePathsRec((0,0))
```



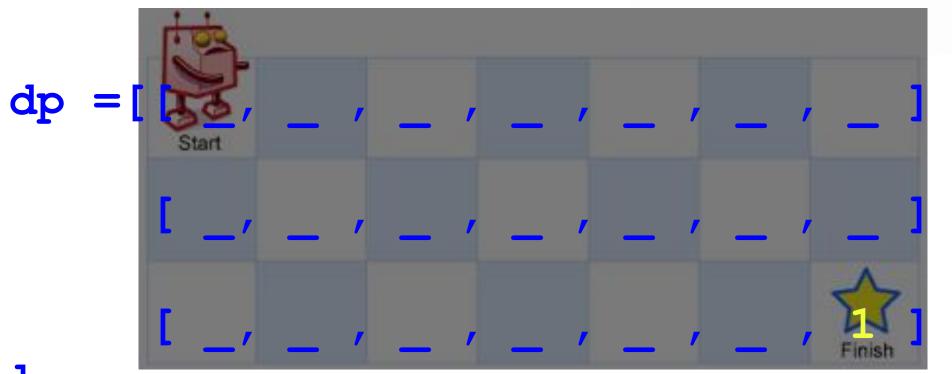
## Step 1: What does the state space look like?

Total Park						
(0, 0) Start	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6) Finish

## Step 1: What does the state space look like?

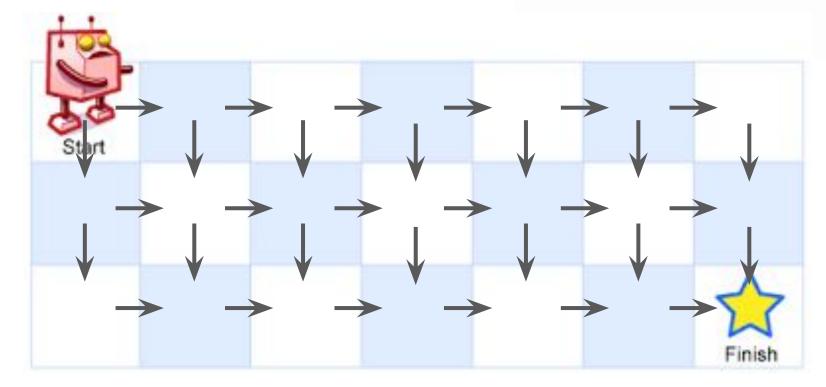


## Step 2: What is the base case?





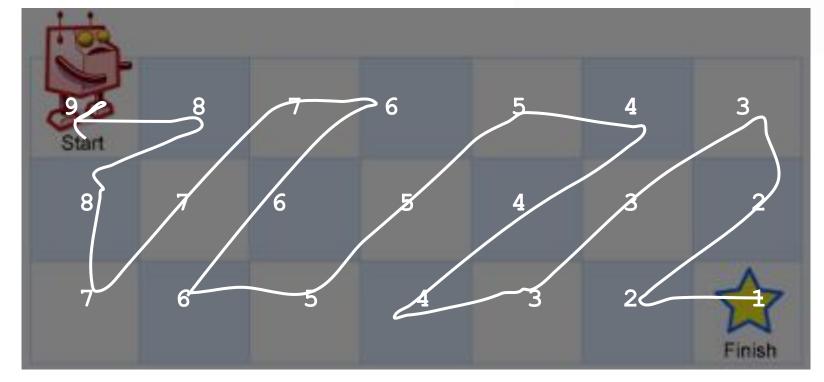
## Step 3: What do the moves look like?



# Step 4: Fill the answers in our array.

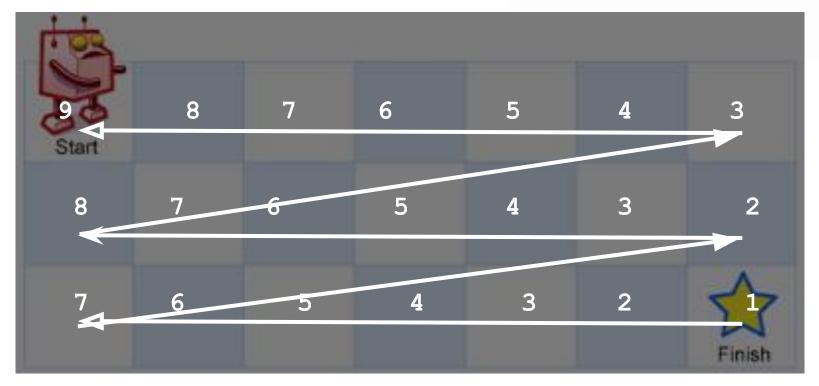


## Order 1: Default



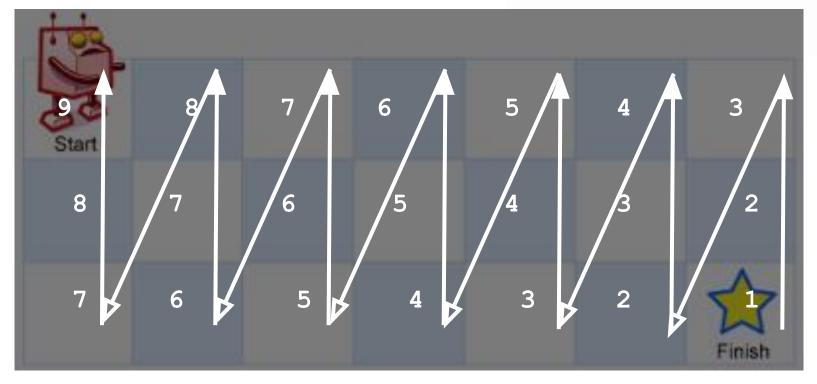


## Order 2: Row-wise



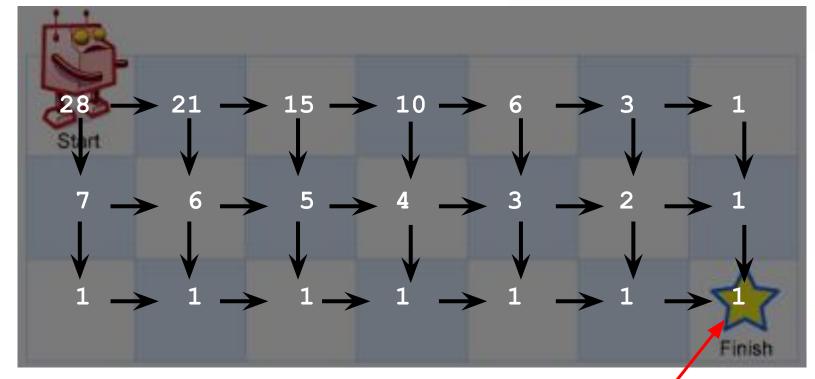


## Order 3: Column-wise

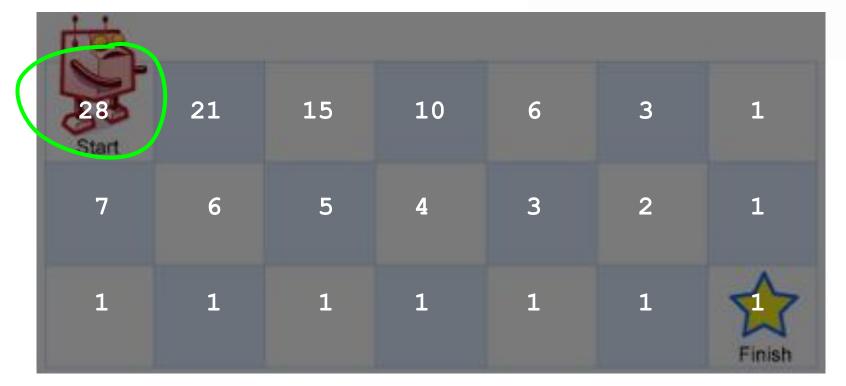




## Step 4: Fill the answers in our array.



## Step 5: Extract the answer.





- N-th Tribonacci Number
- Minimum Falling Path Sum



### ∠Common Variants

- Counting (distinct ways)
  - Usually looks like this: dp[state] = sum(dp[state\_1], dp[state\_2], dp[..]..)
- Optimization (maximum, minimum, and longest/shortest)
  - Usually looks like this: dp[state] = min/max(dp[state\_1], dp[state\_2], dp[..]..)
- **Decision** (yes or no)
  - Usually looks like this: dp[state] = all/any(dp[state\_1], dp[state\_2], dp[..]..)





- Coin Change II
- Solving Questions With Brainpower





- Is Subsequence
- Distinct Subsequences





- Unique Paths II
- Dungeon Game





- Partition to K Equal Sum Subsets
- Minimum XOR Sum of Two Arrays









## Assuming bottom-up dp has better time complexity

- Both bottom-up and top-down DP share the same asymptotic time complexity.
- Despite this, bottom-up DP is typically more efficient than top-down DP.





### **Incorrect Initialization:**

```
def dp(states):
    # if we are minimizing, we should use float('inf')
    dp = [[0 for _ in range(len)] for i in range(n+1)]
```





## **Handling Edge Cases in Bottom-Up DP:**

- Edge cases might be overlooked in standard bottom-up DP, resulting in errors.
- Ensure consideration of smallest, largest, and special input cases for accurate solutions.



### Variant to think about:

#### States

- One Index referring to an element in an array
- Two indices referring to a range in an array
- Numerical Constraints like do something in k steps
- Status on whether some action has taken place or not
- Bitmask/tuple to indicate something is visited or not visited

### Recurrence Relation

- Static recurrence relation function: fib(n) = fib(n-1) + fib(n-2)
- Iteration in your recurrence relation

#### State Reduction

Trying to represent one state in terms of the others

#### Context

- Array
- Matrix
- Graph
- Simulation



# Resources

- Leetcode General Discussion
- <u>Leetcode Explore Card</u>
- <u>Dynamic Programming lecture #1 Fibonacci, iteration vs recursion</u>
- <u>Competitive Programmer's Handbook</u>
- <u>Dynamic programming for Coding Interviews: A bottom-up approach to problem solving</u>
- Traveling Salesperson Problem DP with Bitmask





Longest Arithmetic Subsequence of Given Difference

Combination Sum IV

Perfect Squares

**Triangle** 

Solving Questions With Brainpower

Minimum Falling Path Sum

Minimum Path Cost in a Grid

Champagne Tower



"We worry top-down, but we invest bottom-up."

- Seth Klarman

