Python Programming

Jian Zhang

Nov. 30, 2023@PHBS

https://jianzhang.tech/

Dynamic Programming

- **Dynamic programming (DP)** is usually based on a recursive formula and one or more initial states.
- The solution of the current subproblem **will be derived** from the solutions of the previous subproblems.
- Using dynamic programming to solve problems only requires **polynomial time complexity**, so it is much faster than a recursive method and a violent method.

Dynamic Programming

- **State:** used to describe the solution of each subproblem.
- **State Transition Equation (STE):** a relational expression describing how states transition.
- **Optimal Substructure (OS):** the optimal solution of the problem contains the optimal solution of the subproblem.

The basic principle of DP: find the optimal solution of a certain state, and then find the optimal solution of the next state with its help.

Dynamic Programming

- **Four elements:**

① **Recursion + Memorization**

② **Definition of State:** f(n), f(i, j), f(i, j, k), …

③ **State Transition Equation :** f(n) = **Best_of**(f(n-1), f(n-2), …)

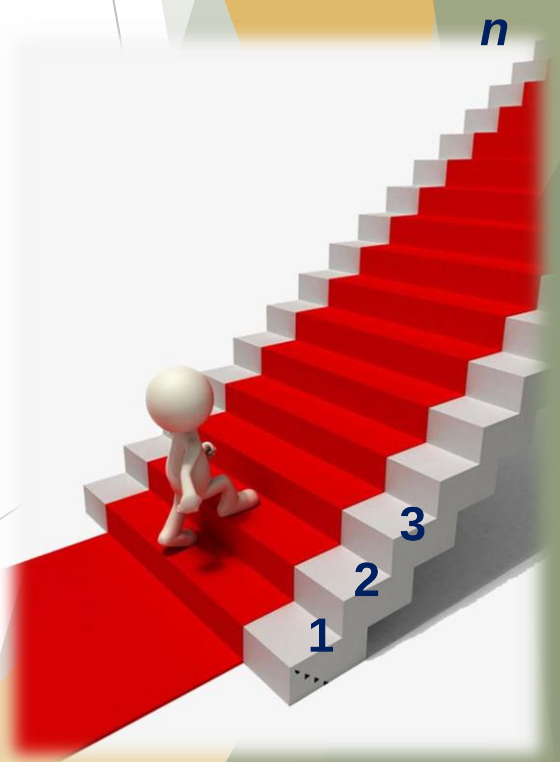
④ **Optimal Substructure**

Climbing Stairs Problem

You are climbing a staircase. It takes ***n*** steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many **distinct ways** can you climb to the top?

Note: Give ***n*** will be a positive integer.



Example 1:

Input: ***n*** = 2
Output: 2

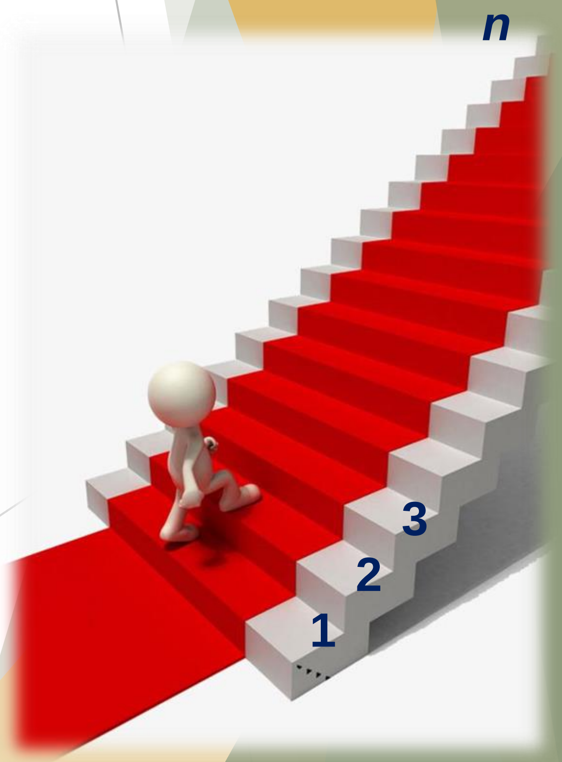
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps

Climbing Stairs Problem

You are climbing a staircase. It takes ***n*** steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many **distinct ways** can you climb to the top?

Note: Give ***n*** will be a positive integer.



Example 2:

Input: ***n*** = 3
Output: 3

Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 2 steps + 1 step
3. 1 step + 2 steps

Climbing Stairs Problem

Example 1:

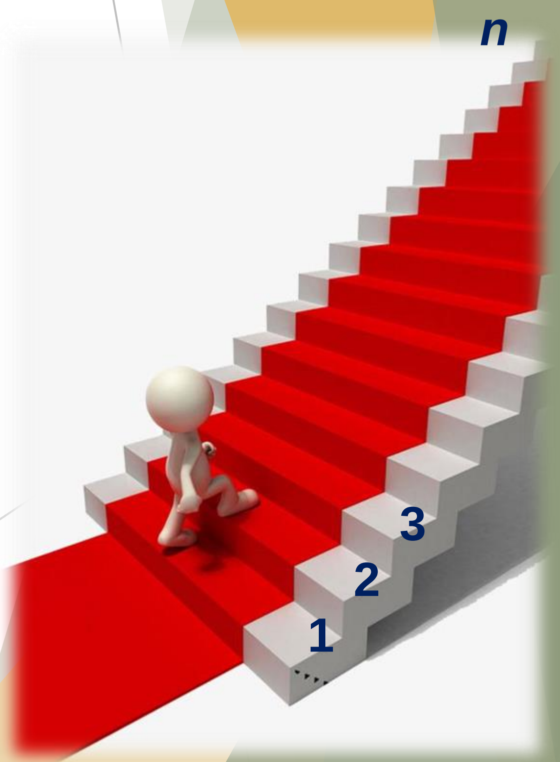
Input: ***n*** = 2
Output: 2

Explanation: two ways.
1. 1 step + 1 step
2. 2 steps

Example 2:

Input: ***n*** = 3
Output: 3

Explanation: three ways.
1. 1 step + 1 step + 1 step
2. 2 steps + 1 step
3. 1 step + 2 steps



Climbing Stairs Problem

Example 1:

Input: n = 2
Output: 2

Explanation: two ways.
1. 1 step + 1 step
2. 2 steps

Example 2:

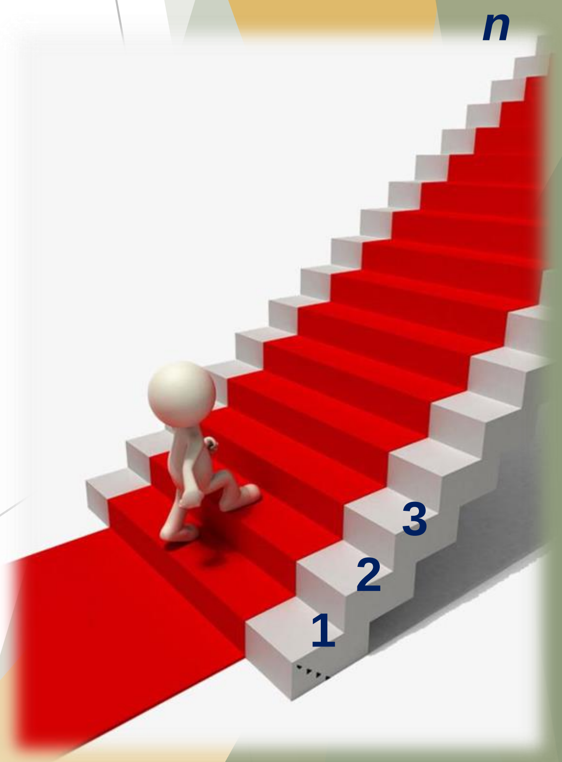
Input: n = 3
Output: 3

Explanation: three ways.
1. 1 step + 1 step + 1 step
2. 2 steps + 1 step
3. 1 step + 2 steps

Example 3:

Input: n = 4
Output: 5

Explanation: five ways to climb to the top.
1. 1 step + 1 step + 1 step + 1 step
2. 2 steps + 1 step + 1 step
3. 1 step + 2 steps + 1 step
4. 1 step + 1 step + 2 steps
5. 2 steps + 2 steps



Climbing Stairs Problem

Example 1:

Input: $n = 2$
Output: 2

Explanation: two ways.
1. 1 step + 1 step
2. 2 steps

Example 2:

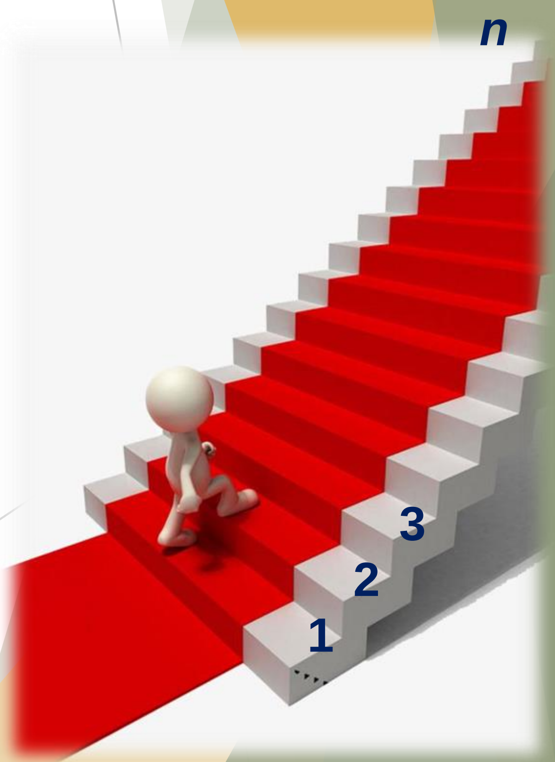
Input: $n = 3$
Output: 3

Explanation: three ways.
1. 1 step + 1 step + 1 step
2. 2 steps + 1 step
3. 1 step + 2 steps

Example 3:

Input: $n = 4$
Output: 5

Explanation: five ways to climb to the top.
1. 1 step + 1 step + 1 step + ~~1 step~~
2. 2 steps + 1 step + ~~1 step~~
3. 1 step + 2 steps + ~~1 step~~
4. 1 step + 1 step + ~~2 steps~~
5. 2 steps + ~~2 steps~~



Climbing Stairs Problem

Example 1:

Input: ***n*** = 2
Output: 2

Explanation: two ways.
1. 1 step + 1 step
2. 2 steps

Example 2:

Input: ***n*** = 3
Output: 3

Explanation: three ways.
1. 1 step + 1 step + 1 step
2. 2 steps + 1 step
3. 1 step + 2 steps

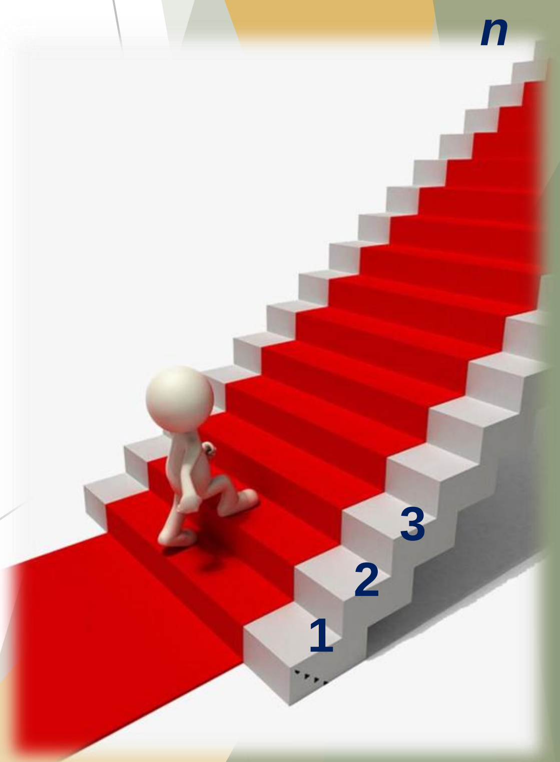
Example 3:

Input: ***n*** = 4
Output: 5

Explanation: five ways to climb to the top.
1. 1 step + 1 step + 1 step + ~~1 step~~
2. 2 steps + 1 step + ~~1 step~~
3. 1 step + 2 steps + ~~1 step~~
4. 1 step + 1 step + ~~2 steps~~
5. 2 steps + ~~2 steps~~

n = 3 (Example 2)

n = 2 (Example 1)



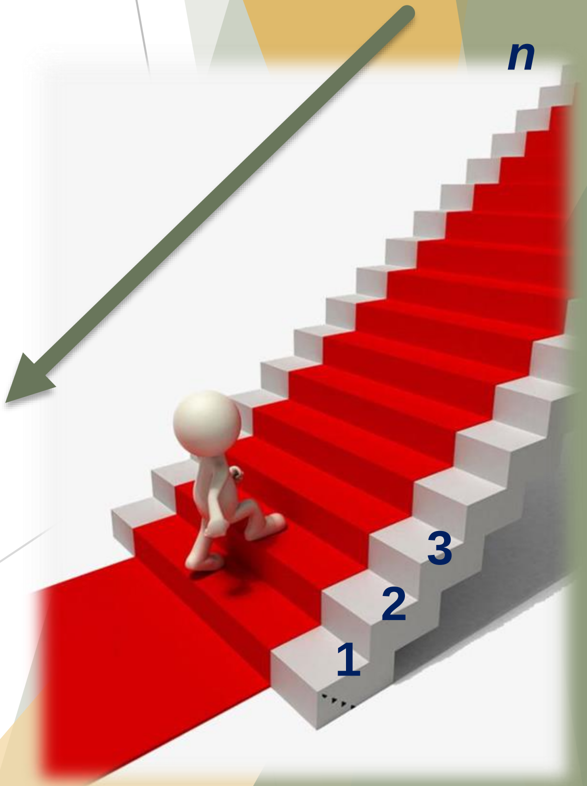
Climbing Stairs Problem

State: Let f(n) denote the number of distinct ways to climb n steps.

State Transition Equation: $f(n) = f(n-1) + f(n-2)$

Top-Down Thinking

Boundary: $f(1) = 1, f(2) = 2$



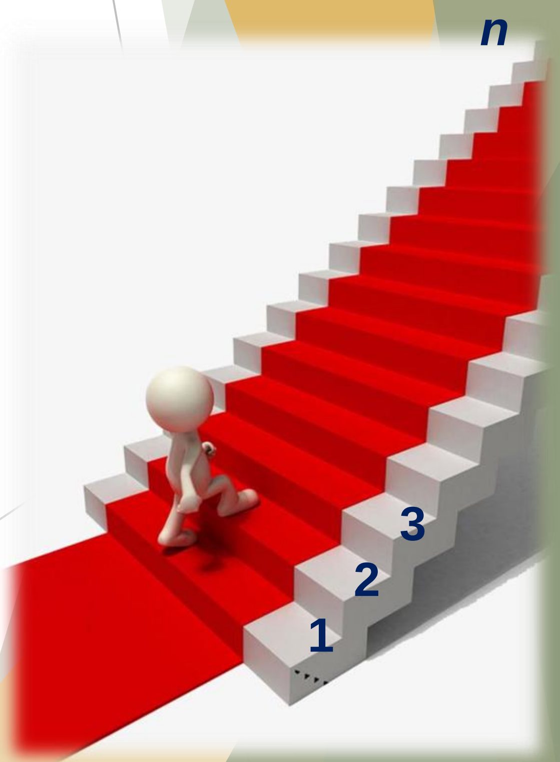
Solution 1: Recursion

```
def climbStairs(n):
    if n == 1:
        return 1
    if n == 2:
        return 2
    else:
        return climbStairs(n-1) + climbStairs(n-2)
```

Climbing Stairs Problem

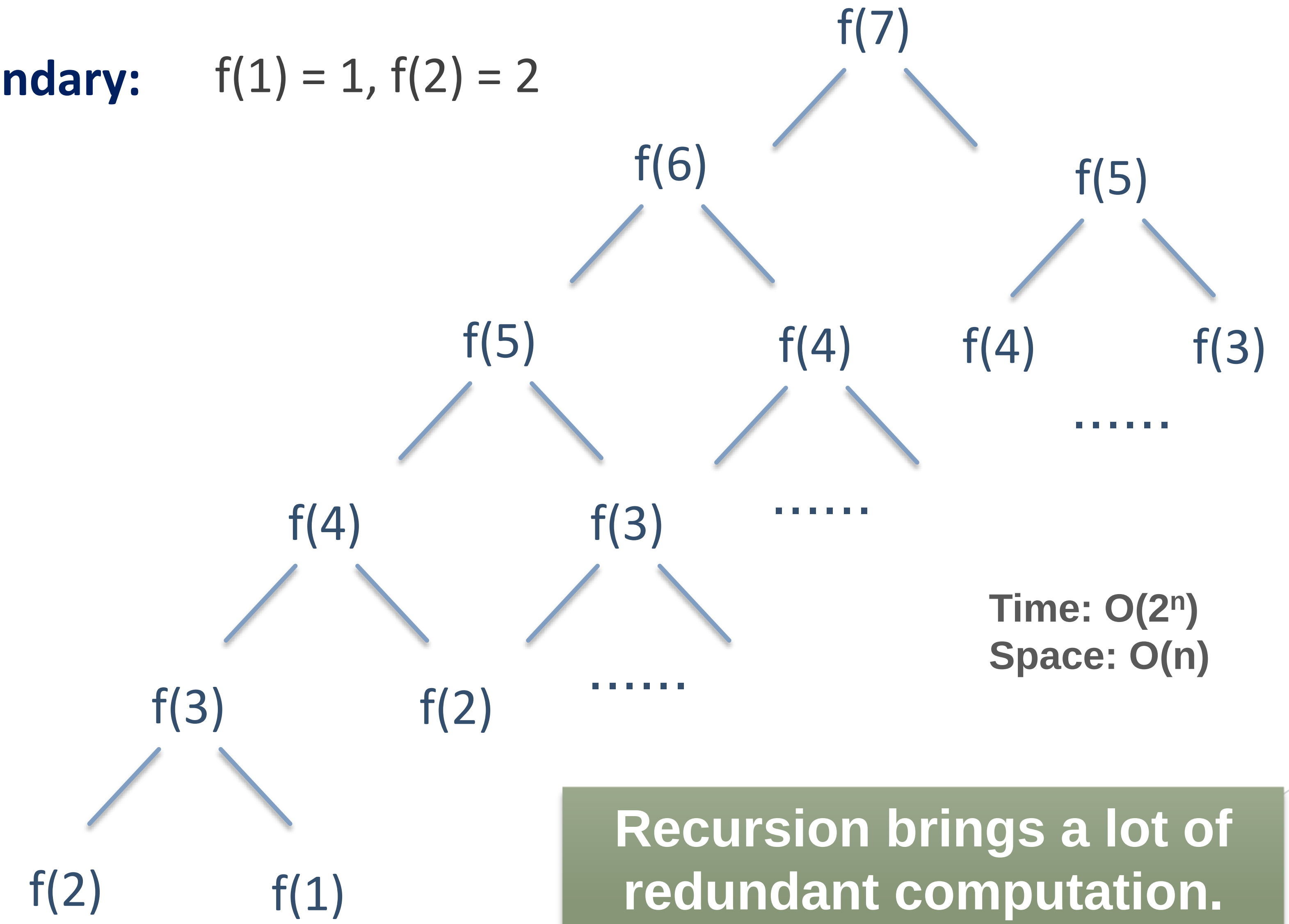
f(7)

Boundary: f(1) = 1, f(2) = 2



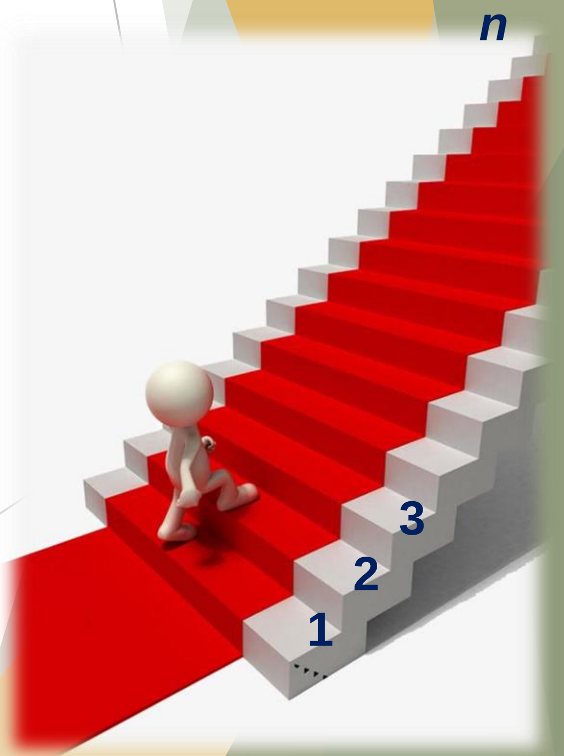
Climbing Stairs Problem

Boundary: f(1) = 1, f(2) = 2



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

Recursion brings a lot of redundant computation.



Climbing Stairs Problem

State: Let f(n) denote the number of distinct ways to climb n steps.

State Transition Equation: $f(n) = f(n-1) + f(n-2)$

Bottom-Up Thinking

Boundary: $f(1) = 1, f(2) = 2$

Solution 2: Recursion + Memorization (DP)

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

Climbing Stairs Problem

State: Let f(n) denote the number of distinct ways to climb n steps.

State Transition Equation: $f(n) = f(n-1) + f(n-2)$

Bottom-Up Thinking

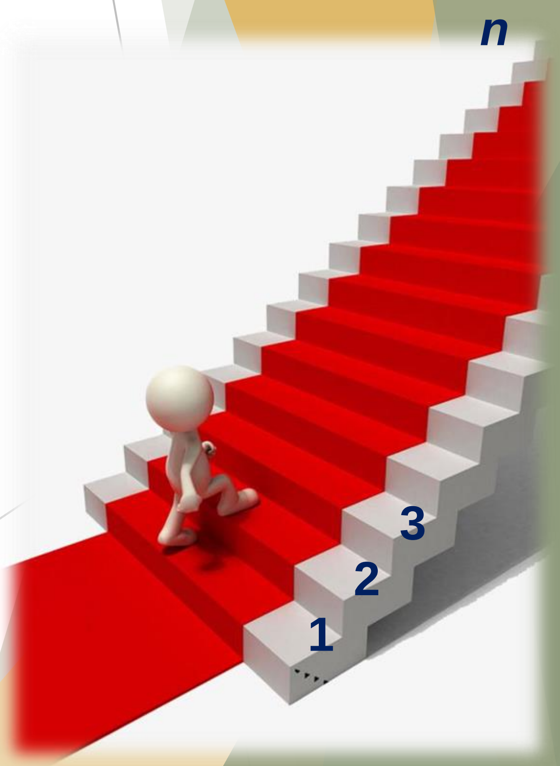
Boundary: $f(1) = 1, f(2) = 2$

Solution 2: Recursion + Memorization (DP)

$f(3) = f(1) + f(2) = 3$, $f(4) = f(2) + f(3) = 5$, ...

Time: O(n)
Space: O(n)

State: [f(1), f(2), f(3), f(4), ..., f(n)]



Climbing Stairs Problem

State: Let f(n) denote the number of distinct ways to climb n steps.

State Transition Equation: $f(n) = f(n-1) + f(n-2)$

Bottom-Up Thinking

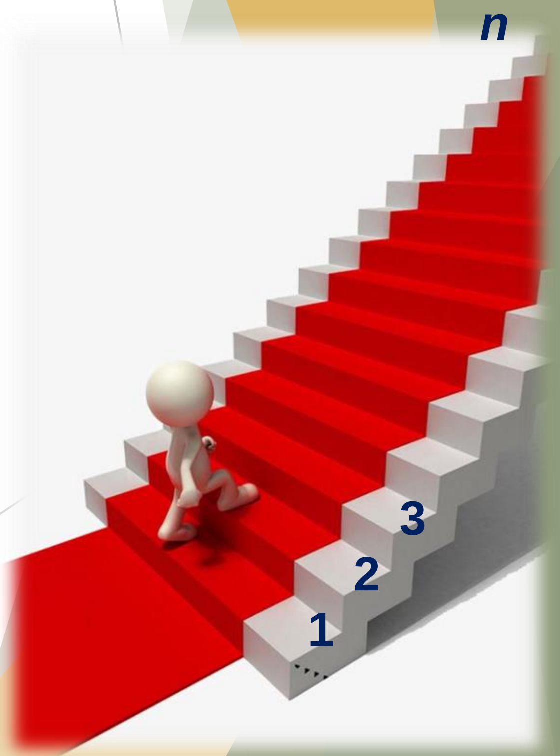
Boundary: $f(1) = 1, f(2) = 2$

Solution 3: Recursion + Saving Memory

```
def climbStairs(n):
    prev, current = 0, 1
    for i in range(n):
        prev, current = current, prev + current
    return current
```

Time: O(n)
Space: O(1)

Fibonacci sequence



Make Change Problem

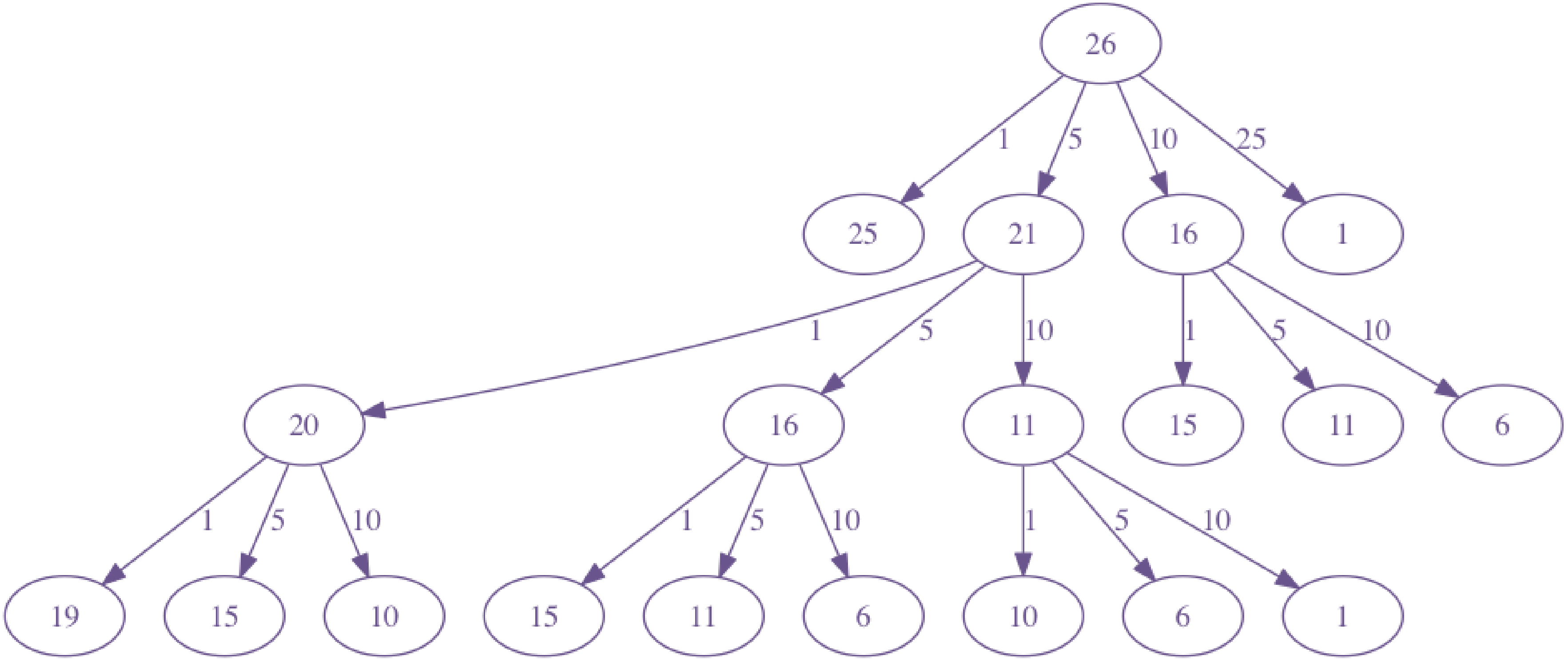
coinValueList = [1,5,10,25]
change = 26

$$numCoins = min \begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

Make Change Problem

coinValueList = [1,5,10,25]
change = 26

$$numCoins = min \begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$



recursion induces redundancy

Make Change Problem

```
def recMC(coinValueList, change):

    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList, change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins
```

Make Change Problem

```
def recDC(coinValueList, change, knownResults):

    minCoins = change
    if change in coinValueList:
        knownResults[change] = 1
        return 1
    elif knownResults[change] > 0:
        return knownResults[change]
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recDC(coinValueList, change-i, knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins
```

Make Change Problem

```
def dpMakeChange(coinValueList,change,minCoins):
    for cents in range(change+1):
        coinCount = cents
        for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
                coinCount = minCoins[cents-j]+1
        minCoins[cents] = coinCount
    return minCoins[change]
```

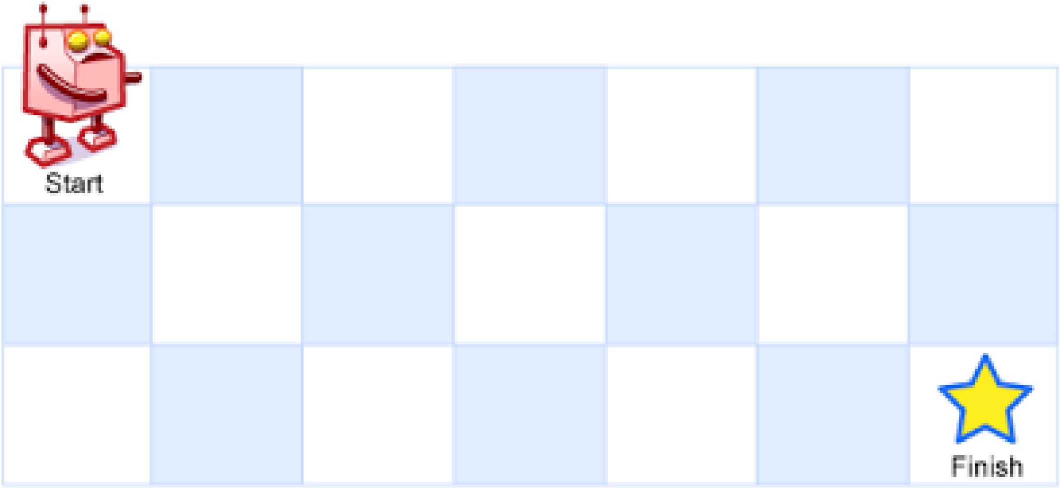
$$numCoins = min \begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

Homework1

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



Input: m = 3, n = 2
Output: 3

Explanation:
From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down

Homework2

Given an integer array nums, return the length of the longest strictly increasing subsequence.

A subsequence is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].

Example 1:
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

Example 2:
Input: nums = [0,1,0,3,2,3]
Output: 4

Questions?