Dynamic Programming

1. Climbing Stairs

Pattern: Dynamic Programming (1D — Bottom-Up)

Problem Statement

> 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?

Sample Input & Output

```
Input: n = 2
Output: 2
Explanation: There are two ways: (1+1) or (2).
```

```
Input: n = 3
Output: 3
Explanation: (1+1+1), (1+2), (2+1).
```

```
Input: n = 1
Output: 1
Explanation: Only one way: (1).
```

LeetCode Editorial Solution + Inline Tests

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        # STEP 1: Initialize structures
        #   - Use two variables to track ways to reach
        #     previous two steps (space-optimized DP).
        if n == 1:
            return 1

        prev2 = 1  # ways to reach step 1
        prev1 = 2  # ways to reach step 2

        # STEP 2: Main loop / recursion
        #   - For each step i from 3 to n, total ways =
        #     ways(i-1) + ways(i-2)
        #   - This maintains the Fibonacci recurrence.
        for i in range(3, n + 1):
            curr = prev1 + prev2  # new total ways
            # STEP 3: Update state / bookkeeping
            #   - Shift window: prev2 becomes prev1,
            #     prev1 becomes curr for next iteration.
            prev2 = prev1
            prev1 = curr

        # STEP 4: Return result
        #   - prev1 holds ways to reach step n
        return prev1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.climbStairs(3) == 3, "Test 1 failed"

    #  Test 2: Edge case
    assert sol.climbStairs(1) == 1, "Test 2 failed"

    #  Test 3: Tricky/negative
    assert sol.climbStairs(5) == 8, "Test 3 failed"
    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `climbStairs(5)` step by step:

1. **Initial check**: `n = 5` 1 → skip return.
2. Set `prev2 = 1` (ways to reach step 1).
   Set `prev1 = 2` (ways to reach step 2).
3. Enter loop for `i = 3` to `5`:

   - **i = 3**:
     `curr = 2 + 1 = 3`
     Update: `prev2 = 2`, `prev1 = 3`
   - **i = 4**:
     `curr = 3 + 2 = 5`
     Update: `prev2 = 3`, `prev1 = 5`
   - **i = 5**:
     `curr = 5 + 3 = 8`
     Update: `prev2 = 5`, `prev1 = 8`

4. Return `prev1 = 8`.

Final state:
- `prev2 = 5` (ways for step 4)
- `prev1 = 8` (ways for step 5) → **output = 8**

This matches the 8 distinct sequences:
(1,1,1,1,1), (1,1,1,2), (1,1,2,1), (1,2,1,1), (2,1,1,1),
(1,2,2), (2,1,2), (2,2,1).

Key insight: The problem follows the **Fibonacci sequence**, where each step's count depends only on the two before it — perfect for **space-optimized DP**.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Single loop from 3 to `n` → runs `n - 2` times → linear.

- **Space Complexity**: `O(1)`

  Only three integer variables used (`prev2`, `prev1`, `curr`) — constant extra space.

2. House Robber

Pattern: Dynamic Programming (1D)

Problem Statement

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Sample Input & Output

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total = 1 + 3 = 4.
```

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (2), house 3 (9), and house 5 (1).
Total = 2 + 9 + 1 = 12.
```

```
Input: nums = [2]
Output: 2
Explanation: Only one house - rob it.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def rob(self, nums: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - Use two variables to track max profit up to previous
        #     two houses (dp[i-1] and dp[i-2]).
        #   - Avoid O(n) space by rolling variables.
        if not nums:
            return 0
        n = len(nums)
        if n == 1:
            return nums[0]

        # prev2 = max profit up to house i-2
        # prev1 = max profit up to house i-1
        prev2 = nums[0]
        prev1 = max(nums[0], nums[1])

        # STEP 2: Main loop / recursion
        #   - For each house i (starting at index 2), decide:
        #       rob i + prev2   OR   skip i → keep prev1
        #   - Maintain invariant: prev1 always holds best up to i-1
        for i in range(2, n):
            current = max(prev1, prev2 + nums[i])

            # STEP 3: Update state / bookkeeping
            #   - Shift window: prev2 ← prev1, prev1 ← current
            #   - Critical: update in correct order to avoid overwrite
            prev2 = prev1
            prev1 = current
        # STEP 4: Return result
        #   - After loop, prev1 holds max for all houses
        return prev1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.rob([1, 2, 3, 1]) == 4

    #  Test 2: Edge case
    assert sol.rob([2]) == 2

    #  Test 3: Tricky/negative
    assert sol.rob([2, 1, 1, 2]) == 4  # Rob first and last
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `rob([2, 1, 1, 2])` step by step.

Initial state:
- `nums = [2, 1, 1, 2]`
- `n = 4` → skip base cases (`n > 1`)
- `prev2 = nums[0] = 2`
- `prev1 = max(2, 1) = 2`

Loop starts at i = 2 (third house, value = 1):
- `current = max(prev1=2, prev2=2 + nums[2]=1) = max(2, 3) = 3`
- Update: `prev2 = prev1 = 2` → `prev1 = current = 3`
- **State**: `prev2=2`, `prev1=3`

i = 3 (fourth house, value = 2):
- `current = max(prev1=3, prev2=2 + nums[3]=2) = max(3, 4) = 4`
- Update: `prev2 = 3`, `prev1 = 4`
- **State**: `prev2=3`, `prev1=4`

Loop ends → return `prev1 = 4`.

Final output: `4`
Key insight: At each house, we only need the best totals from **two steps back** (so we can safely rob current) and **one step back** (so we skip current). This avoids storing full DP array.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Single pass through the array (`for` loop runs `n - 2` times). Each step does O(1) work.

- **Space Complexity**: `O(1)`

  Only two extra variables (`prev1`, `prev2`) used — constant space regardless of input size.

3. Decode Ways

Pattern: Dynamic Programming (1D)

Problem Statement

> A message containing letters from `A-Z` can be encoded into numbers using the following mapping:
> `'A' -> "1"`, `'B' -> "2"`, …, `'Z' -> "26"`.
> To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways).
> Given a string `s` containing only digits, return the **number of ways** to decode it.
> The test cases are generated so that the answer fits in a **32-bit integer**.

Sample Input & Output

```
Input: "12"
Output: 2
Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).
```

```
Input: "226"
Output: 3
Explanation: "226" could be decoded as "BZ" (2 26), "VF" (22 6),
or "BBF" (2 2 6).
```

```
Input: "06"
Output: 0
Explanation: "06" cannot be mapped to 'F' because leading zero
is invalid ("06"   "6").
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def numDecodings(self, s: str) -> int:
        # STEP 1: Initialize structures
        # - dp[i] = number of ways to decode s[:i]
        # - Use two variables to save space: prev2 = dp[i-2], prev1 = dp[i-1]
        if not s or s[0] == '0':
            return 0

        n = len(s)
        # dp[-1] = 1 (empty string has 1 way), dp[0] = 1
        prev2 = 1  # dp[i-2]
        prev1 = 1  # dp[i-1]

        # STEP 2: Main loop / recursion
        #   - Iterate from index 1 to n-1
        #   - At each step, consider single-digit and two-digit decoding
        for i in range(1, n):
            current = 0
            # Check single-digit decode (s[i])
            if s[i] != '0':
                current += prev1

            # Check two-digit decode (s[i-1:i+1])
            two_digit = int(s[i-1:i+1])
            if 10 <= two_digit <= 26:
                current += prev2

            # STEP 3: Update state / bookkeeping
            #   - If current is 0, no valid decoding → early break
            if current == 0:
                return 0

            # Shift window: prev2 <- prev1, prev1 <- current
            prev2, prev1 = prev1, current

        # STEP 4: Return result
        #   - prev1 holds dp[n], the total ways
        return prev1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert (sol.numDecodings("12") == 2,
            f"Expected 2, got {sol.numDecodings('12')}")

    #   Test 2: Edge case - leading zero
    assert (sol.numDecodings("06") == 0,
            f"Expected 0, got {sol.numDecodings('06')}")

    #   Test 3: Tricky/negative - valid two-digit but zero in middle
    assert (sol.numDecodings("226") == 3,
            f"Expected 3, got {sol.numDecodings('226')}")

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `s = "226"` step by step.

Initial state:
- `s = "226"`, length = 3
- `s[0] = '2'` '0' → valid start
- `prev2 = 1` (ways to decode empty prefix)
- `prev1 = 1` (ways to decode "2" → just "B")

Iteration i = 1 (processing second char `'2'`):
- `current = 0`
- Single-digit: `s[1] = '2'` '0' → add `prev1 = 1` → `current = 1`
- Two-digit: `s[0:2] = "22"` → 22 [10,26] → add `prev2 = 1` → `current = 2`
- Update: `prev2 = 1`, `prev1 = 2`

State: `prev2=1`, `prev1=2` → 2 ways to decode "22": ("B","B") or ("V")

Iteration i = 2 (processing third char `'6'`):
- `current = 0`
- Single-digit: `s[2] = '6'` '0' → add `prev1 = 2` → `current = 2`
- Two-digit: `s[1:3] = "26"` → 26 [10,26] → add `prev2 = 1` → `current = 3`
- Update: `prev2 = 2`, `prev1 = 3`

Final return: `prev1 = 3`

Output: `3`

Key insight: At each step, we only need the last two results — classic **space-optimized DP**.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the string once. Each step does constant work (digit checks, integer conversion of 2-char substring).

- **Space Complexity**: `O(1)`

  Only two integer variables (`prev2`, `prev1`) are used, regardless of input size. No recursion stack or DP array.

4. Combination Sum IV

Pattern: Dynamic Programming (1D / Unbounded Knapsack with Permutations)

Problem Statement

Given an array of **distinct** integers `nums` and a target integer `target`, return the number of possible **combinations** that add up to `target`.

The test cases are generated so that the answer can fit in a 32-bit integer.

Note: Different sequences are counted as different combinations. For example, `[1,2]` and `[2,1]` are two distinct combinations.

Sample Input & Output

```
Input: nums = [1,2,3], target = 4
Output: 7
Explanation: The possible combination ways are:
(1,1,1,1), (1,1,2), (1,2,1), (2,1,1), (1,3), (3,1), (2,2)
```

```
Input: nums = [9], target = 3
Output: 0
Explanation: No combination possible since 9 > 3.
```

```
Input: nums = [1], target = 1
Output: 1
Explanation: Only one way: [1].
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize DP array
        #   - dp[i] = number of ways to form sum i
        #   - dp[0] = 1 (one way to make sum 0: choose nothing)
        dp = [0] * (target + 1)
        dp[0] = 1

        # STEP 2: Main loop over all sums from 1 to target
        #   - For each sum i, try every num in nums
        #   - If num <= i, we can extend combinations that sum to (i - num)
        for i in range(1, target + 1):
            for num in nums:
                if num <= i:
                    dp[i] += dp[i - num]

        # STEP 3: Return result
        #   - dp[target] holds total permutations that sum to target
        return dp[target]

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.combinationSum4([1, 2, 3], 4) == 7

    #  Test 2: Edge case - no solution
    assert sol.combinationSum4([9], 3) == 0
```

```
#   Test 3: Tricky/negative - single element match
assert sol.combinationSum4([1], 1) == 1

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell $\rightarrow$ run directly $\rightarrow$ instant feedback.

Example Walkthrough

We'll trace `nums = [1, 2, 3]`, `target = 4`.

Initial state:
- `dp = [1, 0, 0, 0, 0]` (length 5, index 0 to 4)

Step-by-step:

1. **i = 1**
   - Try `num = 1`: 1 1 $\rightarrow$ `dp[1] += dp[0]` $\rightarrow$ `dp[1] = 1`
   - `num = 2`: $2 > 1 \rightarrow$ skip
   - `num = 3`: $3 > 1 \rightarrow$ skip
     $\rightarrow$ `dp = [1, 1, 0, 0, 0]`
2. **i = 2**
   - `num = 1`: 1 2 $\rightarrow$ `dp[2] += dp[1]` $\rightarrow$ `dp[2] = 1`
   - `num = 2`: 2 2 $\rightarrow$ `dp[2] += dp[0]` $\rightarrow$ `dp[2] = 1 + 1 = 2`
   - `num = 3`: skip
     $\rightarrow$ `dp = [1, 1, 2, 0, 0]`
3. **i = 3**
   - `num = 1`: `dp[3] += dp[2]` $\rightarrow 0 + 2 = 2$
   - `num = 2`: `dp[3] += dp[1]` $\rightarrow 2 + 1 = 3$
   - `num = 3`: `dp[3] += dp[0]` $\rightarrow 3 + 1 = 4$
     $\rightarrow$ `dp = [1, 1, 2, 4, 0]`

4. **i = 4**
   - `num = 1`: `dp[4] += dp[3]` $\rightarrow 0 + 4 = 4$
   - `num = 2`: `dp[4] += dp[2]` $\rightarrow 4 + 2 = 6$
   - `num = 3`: `dp[4] += dp[1]` $\rightarrow 6 + 1 = 7$
     $\rightarrow$ `dp = [1, 1, 2, 4, 7]`

Final output: `dp[4] = 7`

> **Key insight**: Unlike classic "combination sum" (which counts unique sets), this counts **permutations** — so order matters. That's why we iterate **sum first**, then **nums**: this allows different orders to be counted separately.

Complexity Analysis

- **Time Complexity**: `O(target * len(nums))`

  We iterate over each sum from 1 to `target`, and for each, loop through all `nums`. Each operation inside is O(1).

- **Space Complexity**: `O(target)`

  The `dp` array has size `target + 1`. No recursion stack (iterative DP), so space scales linearly with target.

5. Coin Change

Pattern: Dynamic Programming (DP) — Unbounded Knapsack / Minimum Coin Change

---

Problem Statement

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

Sample Input & Output

```
Input: coins = [1, 3, 4], amount = 6
Output: 2
Explanation: 3 + 3 = 6 → uses 2 coins (optimal)
```

```
Input: coins = [2], amount = 3
Output: -1
Explanation: Cannot form 3 using only coin 2.
```

```
Input: coins = [1], amount = 0
Output: 0
Explanation: Zero amount requires zero coins.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # STEP 1: Initialize DP array
        #   - dp[i] = min coins to make amount i
        #   - Use amount + 1 as INF (larger than any possible answer)
        INF = amount + 1
        dp = [INF] * (amount + 1)
        dp[0] = 0  # Base: 0 coins needed for amount 0

        # STEP 2: Main loop over all amounts from 1 to amount
        #   - For each amount, try every coin
        #   - Invariant: dp[i] holds optimal coins for amount i
        for i in range(1, amount + 1):
            for coin in coins:
                # STEP 3: Update state only if coin fits
                #   - Skip if coin > current amount
                #   - Otherwise, update dp[i] using dp[i - coin]
                if coin <= i:
                    dp[i] = min(dp[i], dp[i - coin] + 1)

        # STEP 4: Return result
        #   - If dp[amount] still INF → impossible → return -1
        return dp[amount] if dp[amount] != INF else -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.coinChange([1, 3, 4], 6) == 2

    #  Test 2: Edge case - amount = 0
    assert sol.coinChange([1], 0) == 0

    #  Test 3: Tricky/negative - impossible amount
    assert sol.coinChange([2], 3) == -1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `coinChange([1, 3, 4], amount=6)` step by step.

Initial setup: - `amount = 6` - `INF = 7` - `dp = [0, 7, 7, 7, 7, 7, 7]` (indices 0 to 6)

Now iterate `i` from 1 to 6:

---

i = 1
Try coins: 1, 3, 4
- coin=1: 1 1 → `dp[1] = min(7, dp[0]+1) = min(7, 1) = 1`
- coin=3: skip (3 > 1)
- coin=4: skip
→ `dp = [0, 1, 7, 7, 7, 7, 7]`

---

i = 2
- coin=1: `dp[2] = min(7, dp[1]+1) = 2`
- others too big
→ `dp = [0, 1, 2, 7, 7, 7, 7]`

---

i = 3
- coin=1: `dp[3] = min(7, dp[2]+1) = 3`
- coin=3: `dp[3] = min(3, dp[0]+1) = 1` ← better!
- coin=4: skip
→ `dp = [0, 1, 2, 1, 7, 7, 7]`

---

i = 4
- coin=1: `dp[4] = dp[3]+1 = 2`
- coin=3: `dp[4] = min(2, dp[1]+1 = 2)` → still 2
- coin=4: `dp[4] = min(2, dp[0]+1 = 1)` → now 1!
→ `dp = [0, 1, 2, 1, 1, 7, 7]`

---

i = 5
- coin=1: `dp[5] = dp[4]+1 = 2`
- coin=3: `dp[5] = min(2, dp[2]+1 = 3)` → keep 2
- coin=4: `dp[5] = min(2, dp[1]+1 = 2)` → still 2

→ `dp = [0, 1, 2, 1, 1, 2, 7]`

---

i = 6
- coin=1: `dp[6] = dp[5]+1 = 3`
- coin=3: `dp[6] = min(3, dp[3]+1 = 2)` → better!
- coin=4: `dp[6] = min(2, dp[2]+1 = 3)` → keep 2

→ `dp = [0, 1, 2, 1, 1, 2, 2]`

Final result: `dp[6] = 2` → return `2`.

Matches expected output!

Complexity Analysis

- **Time Complexity**: `O(amount × len(coins))`

  We iterate through all amounts from `1` to `amount`, and for each, we try every coin. So total operations = `amount × number of coins`.

- **Space Complexity**: `O(amount)`

  We store a 1D DP array of size `amount + 1`. No recursion stack or extra structures.

6. Partition Equal Subset Sum

Pattern: Dynamic Programming (0/1 Knapsack)

Problem Statement

Given an integer array `nums`, return `true` if you can partition the array into two subsets such that the sum of the elements in both subsets is equal, or `false` otherwise.

Constraints:
- `1 <= nums.length <= 200`
- `1 <= nums[i] <= 100`

Clarification: This is only possible if the total sum is even. If the total sum is odd, equal partition is impossible.

Sample Input & Output

```
Input: [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1,5,5] and [11],
both summing to 11.
```

```
Input: [1,2,3,5]
Output: false
Explanation: Total sum = 11 (odd), so equal partition impossible.
```

```
Input: [100]
Output: false
Explanation: Only one element → cannot split into two non-empty subsets.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        # STEP 1: Initialize structures
        #   - Compute total sum; if odd → impossible.
        total = sum(nums)
        if total % 2 != 0:
            return False
        target = total // 2

        #   - dp[j] = True if subset sum j is achievable
        #   - Size = target + 1 to include sum 0 through target
        dp = [False] * (target + 1)
        dp[0] = True  # Base: sum 0 always possible (empty subset)

        # STEP 2: Main loop / recursion
        #   - For each number, update dp backwards to avoid reuse
        for num in nums:
            # Traverse from target down to num
            for j in range(target, num - 1, -1):
                # STEP 3: Update state / bookkeeping
                #   - If we could make (j - num), then we can make j
                if dp[j - num]:
                    dp[j] = True
                # Early exit if target becomes reachable
                if dp[target]:
                    return True

        # STEP 4: Return result
        #   - dp[target] indicates if half-sum is achievable
        return dp[target]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.canPartition([1, 5, 11, 5]) == True

    #  Test 2: Edge case - odd total sum
    assert sol.canPartition([1, 2, 3, 5]) == False

    #  Test 3: Tricky/negative - single element
    assert sol.canPartition([100]) == False
    print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `canPartition([1, 5, 11, 5])`.

Step 0: Initialization
- `nums = [1,5,11,5]`
- `total = 1+5+11+5 = 22` → even → `target = 11`
- `dp = [True, False, False, ..., False]` (length 12)

Step 1: Process `num = 1`
- Loop `j` from 11 down to 1:
- `j=1`: `dp[1 - 1] = dp[0] = True` → set `dp[1] = True`
- Now `dp[1] = True`
- `dp = [T, T, F, F, ..., F]`

Step 2: Process `num = 5`
- Loop `j` from 11 down to 5:
- `j=6`: `dp[6-5]=dp[1]=True` → `dp[6]=True`
- `j=5`: `dp[0]=True` → `dp[5]=True`
- Now `dp[5] = True`, `dp[6] = True`
- `dp = [T, T, F, F, F, T, T, F, ..., F]`

Step 3: Process `num = 11`
- Loop `j` from 11 down to 11:
- `j=11`: `dp[11-11]=dp[0]=True` → `dp[11]=True`
- Since `dp[11]` is now `True`, **return `True` immediately**

 Final output: `True`
We found a subset (`[11]`) that sums to 11 → the rest (`[1,5,5]`) also sums to 11.

Complexity Analysis

- **Time Complexity**: `O(n * target)`

  We iterate over `n` numbers. For each, we scan up to `target` values.
  Worst-case `target`   `sum/2`, and `sum`   `200 * 100 = 20,000`, so feasible.

- **Space Complexity**: `O(target)`

  We use a 1D DP array of size `target + 1`. No recursion stack.
  This is optimized from the 2D knapsack version by iterating backwards.

7. Unique Paths

Pattern: Dynamic Programming (2D Grid)

Problem Statement

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** `(0, 0)` and tries to reach the **bottom-right corner** `(m - 1, n - 1)`.
The robot can only move either **down** or **right** at any point in time.
Given the integers `m` and `n`, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

Sample Input & Output

```
Input: m = 3, n = 7
Output: 28
Explanation: There are 28 distinct ways to reach (2,6) from (0,0)
using only right/down moves.
```

```
Input: m = 3, n = 2
Output: 3
Explanation: Paths: (R→R→D), (R→D→R), (D→R→R)
```

```
Input: m = 1, n = 1
Output: 1
Explanation: Already at destination - one trivial path.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        # STEP 1: Initialize DP table
        #   - dp[i][j] = number of ways to reach cell (i, j)
        #   - All cells in first row/col have only 1 path (straight line)
        dp = [[1] * n for _ in range(m)]

        # STEP 2: Main loop - fill grid from (1,1) onward
        #   - Each cell = paths from top + paths from left
        #   - This maintains the DP recurrence relation
        for i in range(1, m):
            for j in range(1, n):
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

        # STEP 3: Return result
        #   - Bottom-right cell holds total unique paths
        return dp[m - 1][n - 1]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.uniquePaths(3, 7) == 28, "Normal case failed"

    #  Test 2: Edge case - single cell
    assert sol.uniquePaths(1, 1) == 1, "Edge case (1x1) failed"

    #  Test 3: Tricky/negative - narrow grid
    assert sol.uniquePaths(3, 2) == 3, "Tricky case (3x2) failed"
    print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `uniquePaths(3, 2)` step by step:

1. **Initialize `dp`** as a 3x2 grid filled with 1s:

   ```
   dp = [
     [1, 1],
     [1, 1],
     [1, 1]
   ]
   ```

2. **Start nested loops**:

   - `i = 1`, `j = 1`:
     - `dp[1][1] = dp[0][1] + dp[1][0] = 1 + 1 = 2`

     - Now `dp[1][1] = 2`

3. **Next iteration**:

   - `i = 2`, `j = 1`:
     - `dp[2][1] = dp[1][1] + dp[2][0] = 2 + 1 = 3`

4. **Final `dp` state**:

   ```
   dp = [
     [1, 1],
     [1, 2],
     [1, 3]
   ]
   ```

5. **Return `dp[2][1]` → `3`**, which matches expected output.

Key insight: Every cell accumulates paths from the only two possible directions (top and left), building up the solution bottom-up.

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  We visit each cell exactly once in the nested loops.

- **Space Complexity**: `O(m * n)`

  We store a 2D DP table of size `m x n`.
  (Note: Can be optimized to O(n) using 1D array, but this version prioritizes clarity for pattern mastery.)

8. Maximal Square

Pattern: Dynamic Programming (2D)

Problem Statement

Given an `m x n` binary matrix filled with `'0'`s and `'1'`s, find the **largest square** containing only `'1'`s and return its **area**.

Sample Input & Output

```
Input: matrix = [["1","0","1","0","0"],
                 ["1","0","1","1","1"],
                 ["1","1","1","1","1"],
                 ["1","0","0","1","0"]]
Output: 4
Explanation: The largest square of '1's has side length 2 → area = 4.
```

```
Input: matrix = [["0","1"],["1","0"]]
Output: 1
Explanation: Only isolated '1's exist → max square area = 1.
```

```
Input: matrix = [["0"]]
Output: 0
Explanation: No '1' present → area = 0.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        # STEP 1: Initialize structures
        #   - dp[i][j] = side length of largest square ending at (i-1, j-1)
        #   - Use extra row/col to avoid boundary checks
        if not matrix or not matrix[0]:
            return 0

        rows, cols = len(matrix), len(matrix[0])
        dp = [[0] * (cols + 1) for _ in range(rows + 1)]
        max_side = 0

        # STEP 2: Main loop / recursion
        #   - Traverse each cell; if '1', extend square from top-left
        for i in range(1, rows + 1):
            for j in range(1, cols + 1):
                if matrix[i - 1][j - 1] == "1":
                    # STEP 3: Update state / bookkeeping
                    #- Square side = 1 + min(neighbors) → ensures full square
                    dp[i][j] = 1 + min(
                        dp[i - 1][j],      # top
                        dp[i][j - 1],      # left
                        dp[i - 1][j - 1]   # top-left
                    )
                    max_side = max(max_side, dp[i][j])
                # else: dp[i][j] remains 0

        # STEP 4: Return result
        #   - Area = side^2; handles all-zero matrix naturally
        return max_side * max_side
```

```python
# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    matrix1 = [
        ["1","0","1","0","0"],
        ["1","0","1","1","1"],
        ["1","1","1","1","1"],
        ["1","0","0","1","0"]
    ]
    print(sol.maximalSquare(matrix1))  # Expected: 4

    #  Test 2: Edge case - isolated 1s
    matrix2 = [["0","1"],["1","0"]]
    print(sol.maximalSquare(matrix2))  # Expected: 1

    #  Test 3: Tricky/negative - all zeros
    matrix3 = [["0"]]
    print(sol.maximalSquare(matrix3))  # Expected: 0
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** step-by-step:

Initial state:
- `matrix` as given (4×5)
- `dp` is a 5×6 grid of zeros
- `max_side = 0`

Now iterate `i = 1 to 4`, `j = 1 to 5` (1-indexed for dp):

- **(i=1, j=1)** → `matrix[0][0] = "1"`
  → `dp[1][1] = 1 + min(dp[0][1]=0, dp[1][0]=0, dp[0][0]=0) = 1`
  → `max_side = 1`
- **(i=1, j=2)** → `matrix[0][1] = "0"` → skip → `dp[1][2] = 0`

- **(i=2, j=1)** → `matrix[1][0] = "1"` → `dp[2][1] = 1` → `max_side` still 1
- **(i=2, j=3)** → `matrix[1][2] = "1"`
  → neighbors: top=dp[1][3]=1, left=dp[2][2]=0, diag=dp[1][2]=0
  → `dp[2][3] = 1 + min(1,0,0) = 1`
- **(i=3, j=3)** → `matrix[2][2] = "1"`
  → neighbors: top=dp[2][3]=1, left=dp[3][2]=1, diag=dp[2][2]=1
  → `dp[3][3] = 1 + 1 = 2` → `max_side = 2`
- **(i=3, j=4)** → `matrix[2][3] = "1"`
  → neighbors: top=dp[2][4]=1, left=dp[3][3]=2, diag=dp[2][3]=1
  → `dp[3][4] = 1 + min(1,2,1) = 2` → `max_side` remains 2
- **(i=3, j=5)** → `matrix[2][4] = "1"`
  → neighbors: top=dp[2][5]=1, left=dp[3][4]=2, diag=dp[2][4]=1
  → `dp[3][5] = 2` → still max_side=2

No larger square found. Final `max_side = 2` → area = **4**.

Key insight: `dp[i][j]` only grows if **all three neighbors** support a larger square — ensuring the region is fully filled with `'1'`s.

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  We visit each cell exactly once. Each update is O(1) (min of three values).

- **Space Complexity**: `O(m * n)`

  The `dp` table has `(m+1) * (n+1)` entries. This can be optimized to O(n) with rolling rows, but the standard solution uses full 2D for clarity and pattern consistency.

9. Longest Increasing Subsequence

Pattern: Dynamic Programming (DP) — Classic LIS

Problem Statement

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.

Sample Input & Output

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

```
Input: nums = [0,1,0,3,2,3]
Output: 4
Explanation: LIS = [0,1,2,3] or [0,1,3,3] is invalid
(not strictly increasing), so [0,1,2,3] → length 4.
```

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
Explanation: All elements equal → no strictly increasing pair → LIS length = 1.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        # STEP 1: Initialize DP array
        #   - dp[i] = length of LIS ending at index i
        #   - Every element is at least a subsequence of length 1
        n = len(nums)
        dp = [1] * n
        # STEP 2: Main loop - for each position i, check all j < i
        #   - If nums[j] < nums[i], we can extend the subsequence ending at j
        #   - Maintain invariant: dp[i] stores best LIS ending exactly at i
        for i in range(1, n):
            for j in range(i):
                if nums[j] < nums[i]:
                    dp[i] = max(dp[i], dp[j] + 1)

        # STEP 3: Update state - already done in inner loop

        # STEP 4: Return result
        #   - Answer is max over all dp[i], since LIS can end anywhere
        return max(dp)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.lengthOfLIS([10,9,2,5,3,7,101,18]) == 4

    #  Test 2: Edge case - all equal
    assert sol.lengthOfLIS([7,7,7,7,7,7,7]) == 1

    #  Test 3: Tricky/negative - decreasing then increasing
    assert sol.lengthOfLIS([5,4,3,2,1,2,3,4,5]) == 5
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `nums = [10, 9, 2, 5]` step by step.

Initial state:
- `nums = [10, 9, 2, 5]`
- `dp = [1, 1, 1, 1]` (each element starts with LIS length = 1)

Step 1: `i = 1` (nums[1] = 9)
- Loop `j = 0`: nums[0] = 10 → 10 < 9? No → skip
- `dp` remains `[1, 1, 1, 1]`

Step 2: `i = 2` (nums[2] = 2)
- `j = 0`: 10 < 2?
- `j = 1`: 9 < 2?
- No updates → `dp = [1, 1, 1, 1]`

Step 3: `i = 3` (nums[3] = 5)
- `j = 0`: 10 < 5?
- `j = 1`: 9 < 5?
- `j = 2`: 2 < 5? → `dp[3] = max(1, dp[2] + 1) = max(1, 1+1) = 2`
- Now `dp = [1, 1, 1, 2]`

Final step: `max(dp) = max([1,1,1,2]) = 2` → correct (LIS = [2,5])

This matches expected behavior. The algorithm checks every earlier element to see if it can extend a valid increasing subsequence.

Complexity Analysis

- **Time Complexity**: `O(n²)`

  Two nested loops: outer runs `n-1` times, inner runs up to `i` times → total n(n-1)/2 → O(n^2).

- **Space Complexity**: `O(n)`

  The `dp` array stores one integer per input element → scales linearly with input size.

10. Jump Game

Pattern: Greedy

Problem Statement

You are given an integer array `nums`. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Return `true` if you can reach the last index, or `false` otherwise.

Sample Input & Output

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 where the maximum jump
length is 0, so you cannot reach the last index.
```

```
Input: nums = [0]
Output: true
Explanation: Already at the last (and only) index.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def canJump(self, nums: List[int]) -> bool:
        # STEP 1: Initialize structures
        #   - `max_reach` tracks the farthest index we can reach so far.
        #   - Start at index 0, so initial reach is 0.
        max_reach = 0
        # STEP 2: Main loop / recursion
        #   - Iterate through each index up to the last reachable position.
        #   - If current index exceeds `max_reach`, we're stuck → return False.
        for i in range(len(nums)):
            if i > max_reach:
                return False

            # STEP 3: Update state / bookkeeping
            #   - From index `i`, we can reach up to `i + nums[i]`.
            #   - Update `max_reach` to the furthest possible.
            max_reach = max(max_reach, i + nums[i])

            # Early exit: if we can already reach the end, return True.
            if max_reach >= len(nums) - 1:
                return True

        # STEP 4: Return result
        #- Should not reach here due to early return, but included for safety.
        return max_reach >= len(nums) - 1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.canJump([2,3,1,1,4]) == True, "Test 1 failed"

    #  Test 2: Edge case - single element
    assert sol.canJump([0]) == True, "Test 2 failed"

    #  Test 3: Tricky/negative - zero trap before end
    assert sol.canJump([3,2,1,0,4]) == False, "Test 3 failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `nums = [2,3,1,1,4]` step by step:

1. **Initialize**:
   - `max_reach = 0`
   - `len(nums) = 5`, so last index is `4`.
2. **i = 0**:
   - Check: `0 > max_reach (0)`? → No.
   - Update `max_reach = max(0, 0 + 2) = 2`.
   - Is `2 >= 4`? → No. Continue.
3. **i = 1**:
   - Check: `1 > 2`? → No.
   - Update `max_reach = max(2, 1 + 3) = 4`.
   - Is `4 >= 4`? → **Yes!** → Return `True`.

Final Output: `True`

Key Insight: We never simulate every jump. Instead, we greedily track the *farthest* we could possibly reach at any point. If we can reach or surpass the last index, success!

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the array at most once. Each step does O(1) work.
- **Space Complexity**: `O(1)`

  Only a single variable (`max_reach`) is used, independent of input size.

11. Maximum Subarray

Pattern: Kadane's Algorithm (Dynamic Programming)

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Sample Input & Output

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

```
Input: nums = [1]
Output: 1
Explanation: Only one element → subarray is [1].
```

```
Input: nums = [-5,-2,-8,-1]
Output: -1
Explanation: All negatives → pick least negative (largest single element).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - current_sum tracks max sum ending at current index
        #   - best_sum tracks global maximum seen so far
        current_sum = nums[0]
        best_sum = nums[0]

        # STEP 2: Main loop / recursion
        #   - For each num (starting from index 1), decide:
        #     → extend existing subarray (current_sum + num)
        #     → OR start fresh from current num
        for i in range(1, len(nums)):
            num = nums[i]
            current_sum = max(num, current_sum + num)

            # STEP 3: Update state / bookkeeping
            #   - Always update best_sum if current_sum is better
            best_sum = max(best_sum, current_sum)

        # STEP 4: Return result
        #   - best_sum holds max subarray sum by end of loop
        return best_sum

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.maxSubArray([-2,1,-3,4,-1,2,1,-5,4]) == 6

    #  Test 2: Edge case (single element)
    assert sol.maxSubArray([1]) == 1

    #  Test 3: Tricky/negative (all negatives)
    assert sol.maxSubArray([-5,-2,-8,-1]) == -1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]` step by step.

Initial state:
- `current_sum = -2` (max subarray ending at index 0)
- `best_sum = -2` (global best so far)

Step 1: `i = 1`, `num = 1`
- Compute: `max(1, -2 + 1) = max(1, -1) = 1`
- Update: `current_sum = 1`
- Update `best_sum = max(-2, 1) = 1`

→ Now: `current_sum=1`, `best_sum=1`

Step 2: `i = 2`, `num = -3`
- Compute: `max(-3, 1 + (-3)) = max(-3, -2) = -2`
- Update: `current_sum = -2`
- `best_sum = max(1, -2) = 1` (unchanged)

→ Now: `current_sum=-2`, `best_sum=1`

Step 3: `i = 3`, `num = 4`
- Compute: `max(4, -2 + 4) = max(4, 2) = 4`
- Update: `current_sum = 4`
- `best_sum = max(1, 4) = 4`

→ Now: `current_sum=4`, `best_sum=4`

Step 4: `i = 4`, `num = -1`
- Compute: `max(-1, 4 + (-1)) = max(-1, 3) = 3`
- Update: `current_sum = 3`
- `best_sum = max(4, 3) = 4`

→ Now: `current_sum=3`, `best_sum=4`

Step 5: `i = 5`, `num = 2`
- Compute: `max(2, 3 + 2) = max(2, 5) = 5`
- Update: `current_sum = 5`
- `best_sum = max(4, 5) = 5`

→ Now: `current_sum=5`, `best_sum=5`

Step 6: `i = 6`, `num = 1`
- Compute: `max(1, 5 + 1) = 6`
- Update: `current_sum = 6`
- `best_sum = max(5, 6) = 6`

→ Now: `current_sum=6`, `best_sum=6`

Step 7: `i = 7`, `num = -5`
- Compute: `max(-5, 6 + (-5)) = max(-5, 1) = 1`
- Update: `current_sum = 1`
- `best_sum = 6` (unchanged)

Step 8: `i = 8`, `num = 4`
- Compute: `max(4, 1 + 4) = max(4, 5) = 5`
- Update: `current_sum = 5`
- `best_sum = 6` (still best)

Final return: `6`

Key insight: At every step, we **either extend the best subarray ending at the previous index or start fresh**—never carry forward a negative-sum prefix.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Single pass through the array (`n-1` iterations after initialization). Each step does O(1) work.

- **Space Complexity**: `O(1)`

  Only two extra variables (`current_sum`, `best_sum`) used—no scaling with input size.

12. Maximum Product Subarray

Pattern: Dynamic Programming (Track Min & Max)

Problem Statement

Given an integer array `nums`, find a contiguous subarray (containing at least one number) which has the largest product, and return its product.

The test cases are generated so that the answer will fit in a 32-bit integer.

Sample Input & Output

```
Input: [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product = 6.
```

```
Input: [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.
```

```
Input: [-2,3,-4]
Output: 24
Explanation: Entire array [-2,3,-4] → (-2)×3×(-4) = 24.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - max_prod tracks the max product ending at current index
        #   - min_prod tracks the min product (for negative flips)
        #   - result stores global maximum seen so far
        if not nums:
            return 0

        max_prod = min_prod = result = nums[0]

        # STEP 2: Main loop / recursion
        #   - For each number, compute new max/min products
        #   - Why? A negative number can flip min to max
        for i in range(1, len(nums)):
            num = nums[i]

            # STEP 3: Update state / bookkeeping
            #   - Store old max before overwriting
            temp_max = max_prod
            max_prod = max(num, num * max_prod, num * min_prod)
            min_prod = min(num, num * temp_max, num * min_prod)

            # Update global result
            result = max(result, max_prod)

        # STEP 4: Return result
        #   - Handles all edge cases (single element, zeros, negatives)
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.maxProduct([2, 3, -2, 4]) == 6

    #   Test 2: Edge case (zero resets product)
    assert sol.maxProduct([-2, 0, -1]) == 0

    #   Test 3: Tricky/negative (even negatives → positive)
    assert sol.maxProduct([-2, 3, -4]) == 24

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `maxProduct([-2, 3, -4])` step by step.

Initial state:
- `nums = [-2, 3, -4]`
- `max_prod = min_prod = result = -2`

Step 1: `i = 1`, `num = 3`
- `temp_max = max_prod = -2`
- Compute new `max_prod`:
`max(3, 3 * (-2), 3 * (-2)) = max(3, -6, -6) = 3`
- Compute new `min_prod`:
`min(3, 3 * (-2), 3 * (-2)) = min(3, -6, -6) = -6`
- Update `result = max(-2, 3) = 3`

State after step 1:
`max_prod = 3`, `min_prod = -6`, `result = 3`

Step 2: `i = 2`, `num = -4`
- `temp_max = max_prod = 3`
- Compute new `max_prod`:
`max(-4, -4 * 3, -4 * (-6)) = max(-4, -12, 24) = 24`
- Compute new `min_prod`:
`min(-4, -4 * 3, -4 * (-6)) = min(-4, -12, 24) = -12`
- Update `result = max(3, 24) = 24`

State after step 2:
`max_prod = 24`, `min_prod = -12`, `result = 24`

Final return: `24`

> **Key insight**: Because multiplying two negatives gives a positive, we must track **both** the maximum and minimum products at each step. The minimum (most negative) can become the maximum if the next number is negative.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Single pass through the array (`n = len(nums)`). Each step does constant-time comparisons and multiplications.

- **Space Complexity**: `O(1)`

Only a few scalar variables (`max_prod`, `min_prod`, `result`, `temp_max`) are used — no extra arrays or recursion stack.