<https://leetcode.com/problems/climbing-stairs/description/>

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?

**Example 1:**

**Input:** n = 2

**Output:** 2

**Explanation:** There are two ways to climb to the top.1. 1 step + 1 step2. 2 steps

**Example 2:**

**Input:** n = 3

**Output:** 3

**Explanation:** There are three ways to climb to the top.1. 1 step + 1 step + 1 step2. 1 step + 2 steps3. 2 steps + 1 step

**Constraints:**

1 <= n <= 45

**Attempt 1: 2025-05-26**

**Solution 1: Native DFS (10 min, 31/45)**

class Solution {

public int climbStairs(int n) {

return helper(n);

}

private int helper(int n) {

if(n <= 2) {

return n;

}

return helper(n - 1) + helper(n - 2);

}

}

Time Complexity: O(2^n)

Space Complexity: O(n)

For the native DFS (recursive) solution you provided, let's analyze its time and space complexity:

**Time Complexity:**

The time complexity is O(2^n). Here's why:

Each call to helper(n) branches into two recursive calls: helper(n-1) and helper(n-2).

This creates a binary tree of recursive calls with a height of n.

The number of nodes in this tree grows exponentially, roughly following the Fibonacci sequence, which has an exponential time complexity of O(2^n).

**Space Complexity:**

The space complexity is O(n) due to the recursion stack:

The maximum depth of the recursion stack is n (when you take all steps as 1-step moves).

Each recursive call adds a new frame to the call stack until it reaches the base case (n <= 2).

**Why is this inefficient?**

The recursive solution recalculates the same subproblems multiple times (e.g., helper(3) is computed many times for larger n).

This leads to redundant computations, making it much slower than the dynamic programming approach (which is O(n) time and O(1) space).

**Example:**

For n = 5, the recursion tree looks like:

helper(5)

/ \

helper(4) helper(3)

/ \ / \

helper(3) helper(2) helper(2) helper(1)

/ \

helper(2) helper(1)

**Notice how helper(3) is computed twice, helper(2) three times, etc.**

**Better Approach:**

Use memoization (top-down DP) or bottom-up DP to store computed results and avoid redundant work:

**This reduces time complexity to O(n) (each subproblem is computed once) and space complexity remains O(n) (for memoization and recursion stack). Further optimization to O(1) space is possible using the iterative DP approach.**

**Solution 2: Memoziation (10 min)**

class Solution {

    public int climbStairs(int n) {

        Integer[] memo = new Integer[n + 1];

        return helper(n, memo);

    }

    private int helper(int n, Integer[] memo) {

        if(memo[n] != null) {

            return memo[n];

        }

        if(n <= 2) {

            return n;

        }

        return memo[n] = helper(n - 1, memo) + helper(n - 2, memo);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 3: DP (10 min)**

class Solution {

    public int climbStairs(int n) {

        if(n <= 2) {

            return n;

        }

        int[] dp = new int[n + 1];

        dp[0] = 0;

        dp[1] = 1;

        dp[2] = 2;

        for(int i = 3; i <= n; i++) {

            dp[i] = dp[i - 1] + dp[i - 2];

        }

        return dp[n];

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 4: Space Optimization DP (10 min)**

class Solution {

public int climbStairs(int n) {

if(n <= 2) {

return n;

}

int a = 1;

int b = 2;

int c = 0;

for(int i = 3; i <= n; i++) {

c = a + b;

a = b;

b = c;

}

return c;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

**How the native DFS solution convert into below DP solution**

Yes! The conversion from the native DFS (recursive) solution to the DP (iterative) solution follows a standard "recursion-to-DP" transformation process. Here’s how it works step-by-step:

**Step 1: Identify the Recursive Formula**

The recursive solution is based on the recurrence relation:

helper(n) = helper(n - 1) + helper(n - 2)

with base cases:

helper(1) = 1

helper(2) = 2

This is the same as the Fibonacci sequence, where each step depends on the previous two.

**Step 2: Recognize Overlapping Subproblems**

The recursive solution recalculates the same subproblems repeatedly (e.g., helper(3) is computed multiple times).

→ This inefficiency leads to O(2ⁿ) time and O(n) space (due to recursion stack).

To optimize, we can store computed results to avoid redundant calculations.

**Step 3: Convert Recursion to DP (Memoization → Tabulation)**

**Option 1: Top-Down DP (Memoization)**

Modify the recursive solution to store results in a memo array.

Before computing helper(n), check if it’s already computed.

class Solution {

public int climbStairs(int n) {

int[] memo = new int[n + 1];

return helper(n, memo);

}

private int helper(int n, int[] memo) {

if (n <= 2) return n;

if (memo[n] != 0) return memo[n]; // Return cached result

memo[n] = helper(n - 1, memo) + helper(n - 2, memo); // Store result

return memo[n];

}

}

Time: O(n) (each subproblem computed once)

Space: O(n) (for memoization + recursion stack)

**Option 2: Bottom-Up DP (Tabulation)**

Replace recursion with iteration and compute results from the smallest subproblems upwards.

Use an array dp[] where dp[i] stores the number of ways to reach step i.

class Solution {

public int climbStairs(int n) {

if (n <= 2) return n;

int[] dp = new int[n + 1];

dp[1] = 1; // Base case

dp[2] = 2; // Base case

for (int i = 3; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2]; // Reuse stored results

}

return dp[n];

}

}

Time: O(n)

Space: O(n) (but can be optimized further)

**Step 4: Optimize Space to O(1)**

Since dp[i] only depends on dp[i-1] and dp[i-2], we don’t need the entire dp array. Instead, use two variables to store the last two values.

class Solution {

public int climbStairs(int n) {

if (n <= 2) return n;

int prev1 = 1, prev2 = 2;

for (int i = 3; i <= n; i++) {

int curr = prev1 + prev2;

prev1 = prev2;

prev2 = curr;

}

return prev2;

}

}

Time: O(n)

Space: O(1) (only 3 variables used)

**Key Takeaways**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Time | Space | Notes |
| Native DFS (Recursive) | O(2ⁿ) | O(n) | Inefficient due to repeated work |
| Top-Down DP (Memoization) | O(n) | O(n) | Caches results to avoid recomputation |
| Bottom-Up DP (Tabulation) | O(n) | O(n) | Iterative, computes from base cases |
| Optimized DP (Variables) | O(n) | O(1) | Best space efficiency |

The DP solution eliminates recursion and stores intermediate results, making it much faster than the naive approach. The iterative version is preferred for its O(1) space optimization. 🚀

**Refer to**

<https://leetcode.com/problems/climbing-stairs/solutions/3708750/4-method-s-beat-s-100-c-java-python-beginner-friendly/>

**Intuition:**

To calculate the number of ways to climb the stairs, we can observe that when we are on the nth stair,

we have two options:

either we climbed one stair from the (n-1)th stair or

we climbed two stairs from the (n-2)th stair.

By leveraging this observation, we can break down the problem into smaller subproblems and apply the concept of the Fibonacci series.

The base cases are when we are on the 1st stair (only one way to reach it) and the 2nd stair (two ways to reach it).

By summing up the number of ways to reach the (n-1)th and (n-2)th stairs, we can compute the total number of ways to climb the stairs. This allows us to solve the problem efficiently using various dynamic programming techniques such as recursion, memoization, tabulation, or space optimization.

**Approach 1: Recursion ❌ TLE ❌**

**Explanation**: The recursive solution uses the concept of Fibonacci numbers to solve the problem. It calculates the number of ways to climb the stairs by recursively calling the climbStairs function for (n-1) and (n-2) steps. However, this solution has exponential time complexity (O(2^n)) due to redundant calculations.

class Solution {

public int climbStairs(int n) {

if (n == 0 || n == 1) {

return 1;

}

return climbStairs(n-1) + climbStairs(n-2);

}

}

**Approach 2: Memoization**

**Explanation**: The memoization solution improves the recursive solution by introducing memoization, which avoids redundant calculations. We use an unordered map (memo) to store the already computed results for each step n. Before making a recursive call, we check if the result for the given n exists in the memo. If it does, we return the stored value; otherwise, we compute the result recursively and store it in the memo for future reference.

class Solution {

public int climbStairs(int n) {

Map<Integer, Integer> memo = new HashMap<>();

return climbStairs(n, memo);

}

private int climbStairs(int n, Map<Integer, Integer> memo) {

if (n == 0 || n == 1) {

return 1;

}

if (!memo.containsKey(n)) {

memo.put(n, climbStairs(n-1, memo) + climbStairs(n-2, memo));

}

return memo.get(n);

}

}

**Approach 3: Tabulation**

**Explanation**: The tabulation solution eliminates recursion and uses a bottom-up approach to solve the problem iteratively. It creates a DP table (dp) of size n+1 to store the number of ways to reach each step. The base cases (0 and 1 steps) are initialized to 1 since there is only one way to reach them. Then, it iterates from 2 to n, filling in the DP table by summing up the values for the previous two steps. Finally, it returns the value in the last cell of the DP table, which represents the total number of ways to reach the top.

class Solution {

public int climbStairs(int n) {

if (n == 0 || n == 1) {

return 1;

}

int[] dp = new int[n+1];

dp[0] = dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i-1] + dp[i-2];

}

return dp[n];

}

}

**Approach 4: Space Optimization**

**Explanation**: The space-optimized solution further reduces the space complexity by using only two variables (prev and curr) instead of an entire DP table. It initializes prev and curr to 1 since there is only one way to reach the base cases (0 and 1 steps). Then, in each iteration, it updates prev and curr by shifting their values. curr becomes the sum of the previous two values, and prev stores the previous value of curr.

class Solution {

public int climbStairs(int n) {

if (n == 0 || n == 1) {

return 1;

}

int prev = 1, curr = 1;

for (int i = 2; i <= n; i++) {

int temp = curr;

curr = prev + curr;

prev = temp;

}

return curr;

}

}

**Refer to**

[L746.Min Cost Climbing Stairs (Ref.L70)](note://WEB8aa33f758e5532cf447424bcc598b222)

[L509.Fibonacci Number (Ref.L70,L746,L842,L873)](note://WEB5e91be389fb43d760647ae414e6240e8)

[L1137.N-th Tribonacci Number (Ref.L70,L509)](note://WEB58ee17f8bf9cf9e5980ec12989780df7)

[L2244.Minimum Rounds to Complete All Tasks (Ref.L2451)](note://WEB836f01dee30b51b6bf5f6e87afbd8795)

[L2320.Count Number of Ways to Place Houses (Ref.L70,L198)](note://WEBe7b6043ebce73029e2e8890edb8e11af)

[L2400.Number of Ways to Reach a Position After Exactly k Steps (Ref.L62,L70)](note://WEB0d086d1f5849b8970f5b9498bd68e08d)

[L2466.Count Ways To Build Good Strings (Ref.L70)](note://WEB80a2405e09daf617142ef6cfe28dee0a)