70. Climbing Stairs

Easy Memoization Math Dynamic Programming Leetcode Link

Problem Description

The problem presents us with a scenario where we are required to climb a staircase that consists of `n` individual steps. The objective is to determine the total number of distinct ways to reach the top of the staircase.

The constraints that define our approach to climbing are:

- We can take one step at a time (climb `1` step).
- We can take two steps at a time (climb `2` steps).

This means that for each step we take, we have two choices - to take a single step or a double step - unless we are at a point where only one step is needed to reach the top.

To solve this problem, we need to think about it in terms of finding the total combinations of 1-step and 2-step moves that would lead us to the top.

Intuition

To understand the solution's intuition, we can notice that the way to reach a particular step is a combination of the ways to reach the previous two steps (since we can take a step from either of them). This is a classic example of a dynamic programming problem that exhibits the property of overlapping subproblems.

If we look at the smaller subproblems, we can see that:

- To reach the first step, there is exactly 1 way (climb `1` step).
- To reach the second step, there are 2 ways (climb `1` step + `1` step or climb `2` steps directly).

For any higher step `i`, the ways to reach it is the sum of the ways to reach step `(i-1)` and step `(i-2)`. This is because we can arrive at step `i` by taking a single step from `(i-1)` or a double step from `(i-2)`.

We translate this recursive relationship into an iterative solution by using two variables, `a` and `b`, to store the number of ways to reach the current and the next step. We then use a simple loop to iterate through the steps, updating `a` and `b` at each iteration. At each step in our loop, `a` takes the value of `b` (the number of ways to get to the previous step), and `b` becomes `a+b` (the number of ways to get to the current step from the two steps before it). By the time we reach the top of the staircase, `b` will hold the total number of distinct ways to reach the top.

Solution Approach

The implementation of the solution uses a simple iterative algorithm that employs the dynamic programming approach to solve the problem efficiently by avoiding recomputation of the same subproblems.

Here is a step-by-step breakdown of the algorithm, reflecting on the provided Python code:

1. First, we initialize two variables, `a` and `b`, to 0 and 1, respectively. These variables will help us keep track of the number of ways to reach the current step and the next step. In dynamic programming terms, `a` represents the solution to the subproblem at `i-1` and `b` represents the solution to the subproblem at `i`.
2. We then use a `for` loop to iterate exactly `n` times, where `n` is the total number of steps in the staircase. In each iteration, we perform the following actions:
   - We update the variable `a` to hold the value of `b`, which represents the number of ways to reach the current step.
   - We update the variable `b` to hold the sum of the old values of `a` and `b`. The value of `b` now represents the total number of ways to reach the next step as it sums the ways of getting to the two preceding steps (exploiting the fact that steps `(i-1)` and `(i-2)` contribute to the number of ways to reach step `i`).
3. This process is repeated until we have iterated through all the steps, effectively building up the number of ways to reach higher steps based on the number of ways to reach the previous two steps.
4. At the end of the loop, the variable `b` will contain the total number of distinct ways we can climb to the top of the staircase.

No additional data structures are needed since we're only keeping track of the solution for the two most recent subproblems at any time, making the space complexity `O(1)` - constant space. The time complexity of this algorithm is `O(n)` since we are only looping once through the steps from `1` to `n`.

The elegance of this solution lies in its simple yet powerful usage of the Fibonacci sequence pattern, where the number of ways to reach a given step is the sum of the number of ways to reach the two preceding steps, just like the Fibonacci numbers are the sum of the two preceding numbers in the sequence.

Example Walkthrough

Let's use an example where `n = 4`, which represents a staircase with 4 steps to illustrate the solution approach.

For the first step (`n = 1`), there is 1 distinct way to reach the top: by taking 1 step.

For the second step (`n = 2`), there are 2 distinct ways:

- Take 1 step twice (1 + 1).
- Take 2 steps once.

For the third step (`n = 3`), we need to sum the ways to reach `n = 1` and `n = 2` because we can get to the third step by taking:

- 1 step from the second step (which has 2 ways as calculated before) or
- 2 steps from the first step (which has 1 way as calculated before). So the total number of ways to reach `n = 3` is `1 (from n=1) + 2 (from n=2) = 3` ways.

Now, let's calculate the distinct ways to reach the top of the staircase with 4 steps (`n = 4`):

1. Initialize two variables: `a = 0` (representing `n = 0`), `b = 1` (representing `n = 1`).
2. We iterate starting with `n = 2` to `n = 4`.
   - For `n = 2`: `a` becomes `b` (1), and `b` becomes `a + b` (0 + 1 = 1). Now `a = 1`, `b = 1`.
   - For `n = 3`: `a` becomes `b` (1), and `b` becomes `a + b` (1 + 1 = 2). Now `a = 1`, `b = 2`.
   - For `n = 4`: `a` becomes `b` (2), and `b` becomes `a + b` (1 + 2 = 3). Now `a = 2`, `b = 3`.
3. After completing the loop, the value of `b` is 3, which represents the total number of distinct ways to reach the top of a staircase with 4 steps.

Thus, there are 3 distinct ways to climb to the top of a 4-step staircase:

- 1 + 1 + 1 + 1 (taking one step at a time)
- 1 + 2 + 1 (taking a double step after the first step)
- 2 + 1 + 1 (taking a double step from the bottom)

By following the solution approach given in the content, we can deduce that for any given number of steps `n`, we can use the dynamic programming iterative algorithm to find the total number of distinct ways to climb to the top. This approach efficiently solves the problem by building upon the number of ways to reach the smaller steps and is optimized to use only constant space.

Python Solution

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        # Initialize two variables that represent the base cases:
        # prev_step represents the number of ways to reach the step before the current one
        # curr_step represents the number of ways to reach the current step. Initially set to 1 because there's 1 way to be at the fi
        prev_step, curr_step = 0, 1

        # Loop for each step up to the nth step
        for _ in range(n):
            # At each step, the number of ways to reach the current step is the sum of the ways to reach the previous two steps.
            prev_step, curr_step = curr_step, prev_step + curr_step

        # After finishing the loop, curr_step contains the total number of ways to reach the nth step.
        return curr_step

```

Java Solution

```java
class Solution {

    // This method calculates the number of distinct ways to climb to the top.
    public int climbStairs(int n) {
        int first = 0, second = 1;

        // Loop through number of steps n
        for (int i = 0; i < n; i++) {
            // Calculate next number in the series
            int next = first + second;

            // Update the previous two numbers for next iteration
            first = second;
            second = next;
        }

        // The 'second' variable holds the total ways to reach the top
        return second;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of ways to climb n stairs,
    // where each time you can climb 1 or 2 steps.
    int climbStairs(int n) {
        // Initialize variables to store previous two results
        int prevStep = 0, // Corresponds to the number of ways to climb to the previous step
        int currentStep = 1; // Corresponds to the number of ways to climb to the current step

        // Loop through the number of stairs to calculate the number of ways
        for (int i = 0; i < n; ++i) {
            // Calculate the number of ways to climb to the next step
            int nextStep = prevStep + currentStep; // Sum of the previous two results

            // Update variables for the next iteration
            prevStep = currentStep; // The current step becomes the previous step for the next iteration
            currentStep = nextStep; // The calculated next step becomes the current step for the next iteration
        }

        // Return the number of ways to climb to the top step
        return currentStep;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the number of distinct ways to climb to the top of a staircase
// with n steps, where you can either climb 1 or 2 steps at a time.
function climbStairs(n: number): number {
    // Initialize two variables to store the number of ways to climb to the
    // current step (current) and the previous step (previous).
    let previous = 1;
    let current = 1;

    // Loop through the steps from 1 to n - 1.
    // No loop iteration for the first step as there is only one way to climb one step.
    for (let i = 1; i < n; i++) {
        // Update the number of ways for the current step by adding the number
        // of ways to climb to the previous step to the number of ways for
        // the step before it (previous + current will give us the new current).
        [previous, current] = [current, previous + current];
    }

    // Return the number of ways to climb to the top of the staircase, which will
    // be stored in 'current' after the loop finishes.
    return current;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the input number of steps. This is because there is a single loop which iterates `n` times, and within each iteration, a constant-time operation is performed.

Space Complexity

The space complexity of the code is `O(1)` as only two variables `a` and `b` are used irrespective of the input size, which means the amount of space used does not grow with the input size.