Quant Puzzle #006: Stair Climbing.

Problem Statement

You need to ascend n stairs. You may take 1 step or 2 steps with each climb. How many distinct ways can you reach the top?

Follow-Up Question

Can you code up your solutions and talk about their time complexities?

Context & Key Insights

This puzzle tests familiarity with combinatorics and dynamic programming. The central approach is to decompose the problem into sub-problems and derive a recurrence relation that captures the structure. While conceptually straightforward, it highlights the ability to formalize recursive reasoning, derive closed-form solutions, and analyse algorithmic complexity.

This problem is classified as **Easy** because it involves a simple one-dimensional recurrence relation with two base cases, whose structure is akin to the well-known Fibonacci sequence, and it can be solved efficiently using elementary techniques like dynamic programming or solving a characteristic equation, with minimal risk of subtle errors.

Core Ideas

Key Tools:

- By considering trivial cases, establish the base cases of our recurrence relation.
- Form a recurrence relation. Solve it to find the general solution.

Solution

Let us consider some trivial cases. For $n = 0$ steps there is only one possible way to climb the stairs; they are already climbed. For $n = 1$ steps there is again only one possible way to climb the stairs; take a single step.

For $n = 2$ steps we now have a choice. We can climb a single step, at which point the problem reduces to the $n = 1$ case, or we can climb both steps in one go. The result is two distinct ways of climbing $n = 2$ steps.

The form of the recurrence should be apparent. Let's define f_n to be the number of distinct ways of climbing n steps. From the $n = 2$ case we constructed: $f_2 = f_1 + f_0$. It should be apparent that for larger n we can either take a single step (reducing the problem to the $n - 1$ sub-problem) or we can take two steps (reducing the problem to the $n - 2$ sub-problem). Thus we arrive at the recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

We will solve this recurrence algebraically to find a general closed-form solution. To solve this for a general formula we seek solutions of the form Ar^n, where A is an arbitrary constant. The recurrence relation is hence

$$Ar^n - Ar^{n-1} - Ar^{n-2} = 0$$

which reduces to

$$r^2 - r - 1 = 0$$

Solving this quadratic we find that the values of r can be $r = \frac{1\pm\sqrt{5}}{2}$. Any linear combination of the two solutions is itself a solution to the recurrence relation and so we have that

$$f_n = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$$

where A and B are arbitrary constants. Applying the base cases:

$$\begin{aligned} f_0 &= A + B = 1 \\ f_1 &= \left(\frac{1+\sqrt{5}}{2}\right)A + \left(\frac{1-\sqrt{5}}{2}\right)B = 1 \end{aligned}$$

Solving this set of simultaneous equations yields

$$\begin{aligned} A &= \frac{1+\sqrt{5}}{2\sqrt{5}} \\ B &= \frac{\sqrt{5}-1}{2\sqrt{5}} \end{aligned}$$

Incorporating these into the general solution we arrive at

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$$

which is the final result.

An Alternative Programming Solution

Rather than solve the recurrence relation for an explicit closed-form formula, we can instead leverage the power of a computer to recursively compute the solution using the recurrence relation and its base cases. We can construct a size $n + 1$ array to hold the answer to each sub-problem and, starting from the base cases, compute each n-th solution sequentially.

```cpp
int fn(int n) {
    std::vector<int> cache(n+1);
    cache[0] = 1; cache[1] = 1;
    for(int i = 2; i <= n; i++)
    {
        cache[i] = cache[i-1] + cache[i-2];
    }
    return cache[n];
}
```

This has an $\mathcal{O}(n)$ time and space complexity. The space complexity can be reduced to $\mathcal{O}(1)$ with the following alternative that realises that sub-problems more than two steps away contribute nothing to the solution since only the last two values are required to compute the next.

```
int fn(int n) {
    int f0 = 1; int f1 = 1;
        for(int i = 2; i <= n; i++)
        {
            int tmp = f1;
            f1 = f0+f1;
            f0 = tmp;
        }
        return f1;
}
```

Of course to improve the time complexity further to $\mathcal{O}(1)$ we simply code up the closed-form formula.

```
int fn(int n) {
    double root5 = sqrt(5);
    return (1/root5)*(pow((1+root5)/2, n+1) - pow((1-root5)/2, n+1));
}
```

It is worth noting that the use of the `sqrt` function is subject to precision errors and the use of integers as a return type will only suffice for small n as the answer grows exponentially with n.

Takeaways

The main difficulty of this problem lies in arriving at the correct recurrence relation, including base cases. A subtle mistake such as taking $f_0 = 0$ can throw the solution off. Solving this puzzle proves that you can:

- You can solve problems by breaking them into smaller, simpler problems.
- Identify recurrences from recursive logic.
- Correctly formulate the recurrence problem and its base cases and subsequently solve it.
- Show familiarity with advanced concepts such as dynamic programming and can discuss the time and space complexities of different programmatical approaches.

More Editorials: https://github.com/UIronsMaths/Quant-Puzzle-Editorials