The Coding Test - Unlocked Work

SOLUTION

You just heard from one of our GWC Alum, here are some additional ways you can think through this coding test.

Let's think about this test with the following question: What is the very last step that is done?

The very last hop the child makes, the one that lands her on the nth step, was either a
- 3-step hop,
- 2-step hop, or
- 1-step hop

How many ways then are there to get up to the nth step? We don't know yet, but we can relate it to some subproblems.

If we thought about all of the paths to the nth step, we could just build them off of the paths to the three previous steps. We can get up to the nth step by any of the following:

- Going to the (n - l)st step and hopping 1 step.
- Going to the (n -2)nd step and hopping 2 steps.
- Going to the (n - 3)rd step and hopping 3 steps.

Therefore, we just need to add the number of these paths together.

NOTE: Be very careful here. A lot of people want to multiply them. Multiplying one path with another would signify taking one path and then taking the other. That's not what's happening here.

Alternative Solution 1: The Brute Force Solution

This is a fairly straightforward algorithm to implement recursively. We just need to follow logic like this:

countWays(n-1) + countWays(n-2) + countWays(n-3)

The one tricky bit is defining the base case. If we have 0 steps to go (we're currently standing on the step), are there zero paths to that step or one path? That is, what is countWays(0)? Is it 1 or 0? You could define it either way. There is no "right" answer here. However, it's a lot easier to define it as 1. If you defined it as 0, then you would need some additional base cases (or else you'd just wind up with a series of 0s getting added).

A simple implementation of this code is as follows.

```
int countWays(int n) {
  if (n < 0) {
    return 0;
  } else if (n == 0) {
    return 1;
  } else {
    return countWays(n-1) + countWays(n-2) + countWays(n-3);
  }
}
```

Like the Fibonacci problem, the runtime of this algorithm is exponential (roughly O (n3)), since each call branches out to three more calls.

Alternative Solution 2: The Memoization Solution

The previous solution for countWays is called on many times for the same values, which is unnecessary. We can fix this through memoization. Essentially, if we've seen this value of n before, return the cached value. Each time we compute a fresh value, add it to the cache. Typically, we use a HashMap<Integer, Integer> for a cache. In this case, the keys will be exactly 1 through n. It's more compact to use an integer array.

```
int countWays(int n) {
  int[] memo = new int[n + 1];
  Arrays.fill(memo, -1);
  return countWays(n, memo);
}

int countWays(int n, int[] memo) {
  if (n < 0) {
    return 0;
  } else if (n == 0) {
    return 1;
  } else if (memo[n] > -1) {
    return memo[n];
  } else {
    memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
              countWays(n - 3, memo);
    return memo[n];
  }
}
```

Regardless of whether or not you use memoization, note that the number of ways will quickly overflow the bounds of an integer. By the time you get to just n 37, the result has already overflowed. Using a long will delay, but not completely solve, this issue.

It is great to communicate this issue to your interviewer. They probably won't ask you to work around it. (Although you could, with a Big integer class). But it's important to demonstrate that you think about these issues.