Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution 1

The problem seems to be a *dynamic programming* one. **Hint**: the tag also suggests that! Here are the steps to get the solution incrementally.

- Base cases:
  if n <= 0, then the number of ways should be zero. if n == 1, then there is only way to climb the stair. if n == 2, then there are two ways to climb the stairs. One solution is one step by another; the other one is two steps at one time.
- The key intuition to solve the problem is that given a number of stairs n, if we know the number ways to get to the points `[n-1]` and `[n-2]` respectively, denoted as `n1` and `n2` , then the total ways to get to the point `[n]` is `n1 + n2`. Because from the `[n-1]` point, we can take one single step to reach `[n]`. And from the `[n-2]` point, we could take two steps to get there. There is NO overlapping between these two solution sets, because we differ in the final step.

Now given the above intuition, one can construct an array where each node stores the solution for each number n. Or if we look at it closer, it is clear that this is basically a fibonacci number, with the starting numbers as 1 and 2, instead of 1 and 1.

The implementation in Java as follows:

```
public int climbStairs(int n) {
    // base cases
    if(n <= 0) return 0;
    if(n == 1) return 1;
    if(n == 2) return 2;

    int one_step_before = 2;
    int two_steps_before = 1;
    int all_ways = 0;

    for(int i=2; i<n; i++){
        all_ways = one_step_before + two_steps_before;
        one_step_before = two_steps_before;
        two_steps_before = all_ways;
    }
    return all_ways;
}
```

written by liaison original link here

Solution 2

Hi guys, I come up with this arithmetic way. Find the inner logic relations and get the answer.

```
public class Solution {

public int climbStairs(int n) {
    if(n == 0 || n == 1 || n == 2){return n;}
    int[] mem = new int[n];
    mem[0] = 1;
    mem[1] = 2;
    for(int i = 2; i < n; i++){
        mem[i] = mem[i-1] + mem[i-2];
    }
    return mem[n-1];
}
```

}

written by tangxukai original link here

Solution 3

Same simple algorithm written in every offered language. Variable `a` tells you the number of ways to reach the current step, and `b` tells you the number of ways to reach the next step. So for the situation one step further up, the old `b` becomes the new `a`, and the new `b` is the old `a+b`, since that new step can be reached by climbing 1 step from what `b` represented or 2 steps from what `a` represented.

Ruby wins, and *"the C languages"* all look the same.

Ruby (60 ms)

```ruby
def climb_stairs(n)
    a = b = 1
    n.times { a, b = b, a+b }
    a
end
```

C++ (0 ms)

```cpp
int climbStairs(int n) {
    int a = 1, b = 1;
    while (n--)
        a = (b += a) - a;
    return a;
}
```

Java (208 ms)

```java
public int climbStairs(int n) {
    int a = 1, b = 1;
    while (n-- > 0)
        a = (b += a) - a;
    return a;
}
```

Python (52 ms)

```python
def climbStairs(self, n):
    a = b = 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

C (0 ms)

```
int climbStairs(int n) {
    int a = 1, b = 1;
    while (n--)
        a = (b += a) - a;
    return a;
}
```

C# (48 ms)

```
public int ClimbStairs(int n) {
    int a = 1, b = 1;
    while (n-- > 0)
        a = (b += a) - a;
    return a;
}
```

Javascript (116 ms)

```
var climbStairs = function(n) {
    a = b = 1
    while (n--)
        a = (b += a) - a
    return a
};
```

written by StefanPochmann original link here