Staircase

We'll cover the following

Problem Statement

Given a stair with ‘n’ steps, implement a method to count how many possible ways are there to reach the top of the staircase, given that, at every step you can either take 1 step, 2 steps, or 3 steps.

Example 1:

```
Number of stairs (n) : 3
Number of ways = 4
Explanation: Following are the four ways we can climb : {1,1,1}, {1,2}, {2,1}, {3}
```

Example 2:

```
Number of stairs (n) : 4
Number of ways = 7
Explanation: Following are the seven ways we can climb : {1,1,1,1}, {1,1,2}, {1,2,
1}, {2,1,1},
{2,2}, {1,3}, {3,1}
```

Let’s first start with a recursive brute-force solution.

Basic Solution

At every step, we have three options: either jump 1 step, 2 steps, or 3 steps. So our algorithm will look like this:

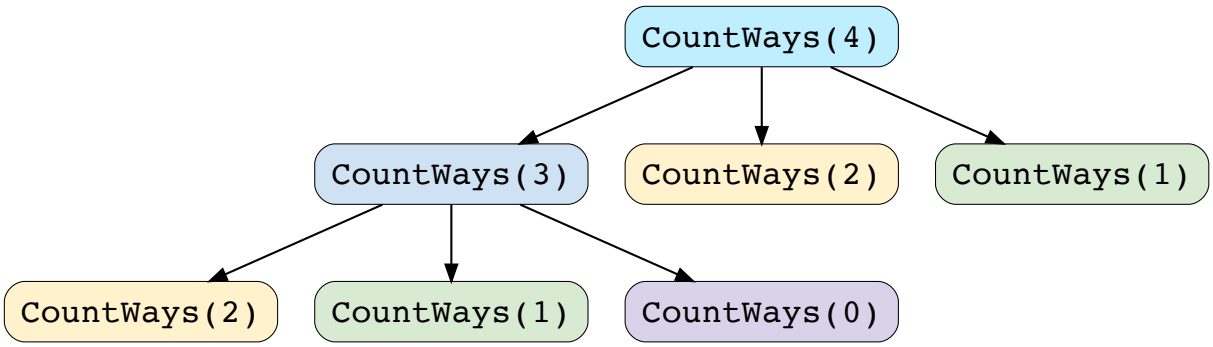
Java | JS | Python3 | C++

```
def count_ways(n):
  if n == 0:
    return 1  # base case, we don't need to take any step, so there is only one way
```

```
4
5    if n == 1:
6      return 1  # we can take one step to reach the end, and that is the only way
7
8    if n == 2:
9      return 2  # we can take one step twice or jump two steps to reach at the top
10
11   # if we take 1 step, we are left with 'n-1' steps;
12   take1Step = count_ways(n - 1)
13   # similarly, if we took 2 steps, we are left with 'n-2' steps;
14   take2Step = count_ways(n - 2)
15   # if we took 3 steps, we are left with 'n-3' steps;
16   take3Step = count_ways(n - 3)
17
18   return take1Step + take2Step + take3Step
19
20
21 def main():
22
23   print(count_ways(3))
24   print(count_ways(4))
25   print(count_ways(5))
26
27
28 main()
29
```

The time complexity of the above algorithm is exponential $O(3^n)$ as we are making three recursive call in the same function. The space complexity is $O(n)$ which is used to store the recursion stack.

Let's visually draw the recursion for `CountWays(4)` to see the overlapping subproblems:



Recursion tree for calculating Fibonacci numbers

We can clearly see the overlapping subproblem pattern: `CountWays(2)` and `CountWays(1)` have been called twice. We can optimize this using memoization.

Top-down Dynamic Programming with Memoization

We can use an array to store the already solved subproblems. Here is the code:

Java | JS | Python3 | C++

```
1  def count_ways(n):
```

```
  dp = [0 for x in range(n+1)]
  return count_ways_recursive(dp, n)


def count_ways_recursive(dp, n):
  if n == 0:
    return 1  # base case, we don't need to take any step, so there is only one way

  if n == 1:
    return 1  # we can take one step to reach the end, and that is the only way

  if n == 2:
    return 2  # we can take one step twice or jump two steps to reach at the top

  if dp[n] == 0:
    # if we take 1 step, we are left with 'n-1' steps;
    take1Step = count_ways_recursive(dp, n - 1)
    # similarly, if we took 2 steps, we are left with 'n-2' steps;
    take2Step = count_ways_recursive(dp, n - 2)
    # if we took 3 steps, we are left with 'n-3' steps;
    take3Step = count_ways_recursive(dp, n - 3)

    dp[n] = take1Step + take2Step + take3Step

  return dp[n]


def main():

  print(count_ways(3))
  print(count_ways(4))
  print(count_ways(5))


main()

```

What is the time and space complexity of the above solution? Since our memoization array `dp[n+1]` stores the results for all the subproblems, we can conclude that we will not have more than $n + 1$ subproblems (where 'n' represents the total number of steps). This means that our time complexity will be $O(N)$. The space complexity will also be $O(n)$; this space will be used to store the recursion-stack.

Bottom-up Dynamic Programming

Let's try to populate our `dp[]` array from the above solution, working in a bottom-up fashion. As we saw in the above code, every `CountWaysRecursive(n)` is the sum of the previous three counts. We can use this fact to populate our array.

Code

Here is the code for our bottom-up dynamic programming approach:

Java | JS | Python3 | C++

```
def count_ways(n):
  dp = [0 for x in range(n+1)]
```

```
  dp = [0 for x in range(n+1)]
  dp[0] = 1
  dp[1] = 1
  dp[2] = 2

  for i in range(3, n+1):
    dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]

  return dp[n]


def main():

  print(count_ways(3))
  print(count_ways(4))
  print(count_ways(5))


main()
```

The above solution has time and space complexity of $O(n)$.

Memory optimization

We can optimize the space used in our previous solution. We don't need to store all the counts up to 'n', as we only need three previous numbers to calculate the next count. We can use this fact to further improve our solution:

Java | JS | Python3 | C++

```
def count_ways(n):
  if n < 2:
    return 1
  if n == 2:
    return 2
  n1, n2, n3 = 1, 1, 2
  for i in range(3, n+1):
    n1, n2, n3 = n2, n3, n1+n2+n3
  return n3


def main():

  print(count_ways(3))
  print(count_ways(4))
  print(count_ways(5))


main()
```

The above solution has a time complexity of $O(n)$ and a constant space complexity $O(1)$.

Fibonacci number pattern

We can clearly see that this problem follows the Fibonacci number pattern. The only difference is that in Fibonacci numbers every number is a sum of the two preceding numbers, whereas in this problem every count is a sum of three preceding counts. Here is the recursive formula for this problem:

```
    CountWays(n) = CountWays(n-1) + CountWays(n-2) + CountWays(n-3), for n >=3
```

This problem can be extended further. Instead of taking 1, 2, or 3 steps at any time, what if we can take up to 'k' steps at any time? In that case, our recursive formula will look like:

```
    CountWays(n) = CountWays(n-1) + CountWays(n-2) + CountWays(n-3) + ... + CountW
ays(n-k), for n >= k
```

Back
Fibonacci numbers

Next
Number factors

Mark as Completed

Report an Issue
Ask a Question (https://discuss.educative.io/tag/staircase__pattern-3-fibonacci-numbers__grokking-dynamic-programming-patterns-for-coding-interviews)