# Solution Review: The Staircase Problem

In this lesson, we will review the staircase problem from the coding challenge in last lesson.

**We'll cover the following**



## Solution 1: Simple recursion #

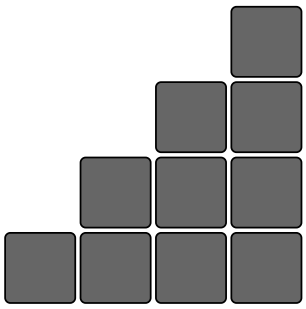
```
def staircase(n, m):
  # base case of when there is no stair
  if n == 0:
    return 1
  ways = 0
  # iterate over number of steps, we can take
  for i in range(1,m+1):
    # if steps remaining is smaller than the jump step, skip
    if i <= n:
      # recursive call with n i units lesser where i is the number of steps taken here
      ways += staircase(n-i, m)
  return ways

print(staircase(4,2))
```

### Explanation #

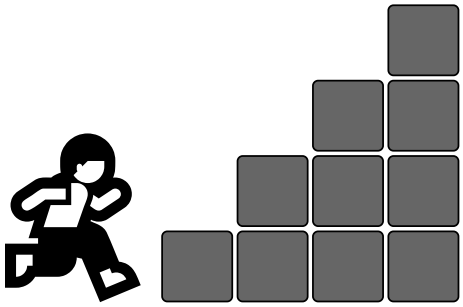
The problem is similar to the Fibonacci numbers algorithm. Instead of binary recursion, we have an m-ary recursion here. Which means every recursive makes,

at most, m recursive calls.

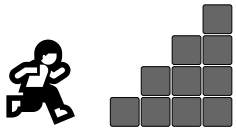
The algorithm is pretty intuitive as well. At any step, we have the option to take steps of `m` different lengths, i.e., we can either take `1` step, `2` steps, or so on up to `m` steps. Each different step will result in a different path. Therefore, we make recursive calls with each number of steps (from `1` to `m`). We subtract `i` from n for each call since we have taken `i` number of steps at this instance (*lines 7-11*). Our algorithm will conclude when `n` becomes equal to `0`. This would mean we have reached the top of the staircase, and the path we took to get there constitutes one of the many paths, therefore we will return 1 here (*lines 3-4*). This is our base case. Look at the following visualization for a dry run of this algorithm.



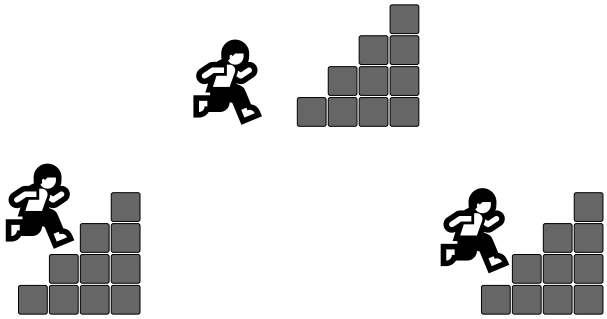
Climb staircase with n=4, and m=2



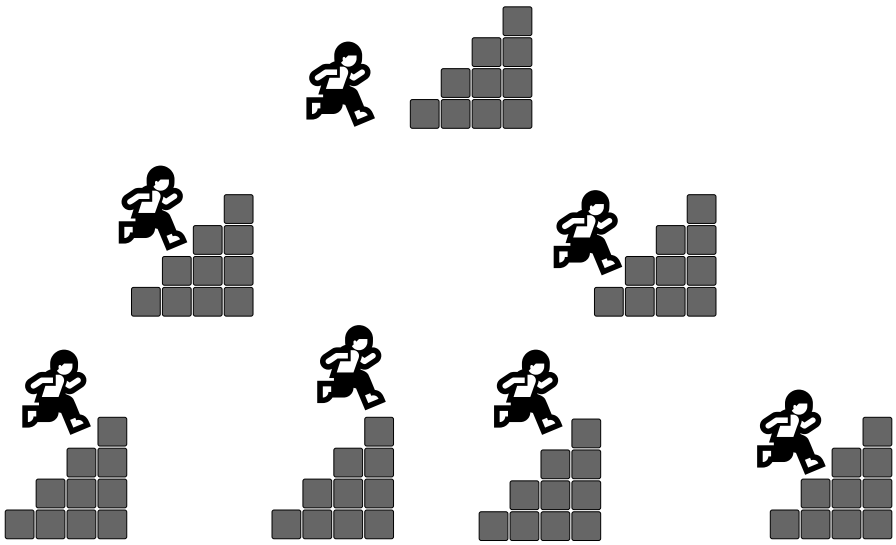
Nick can go up either one or two steps



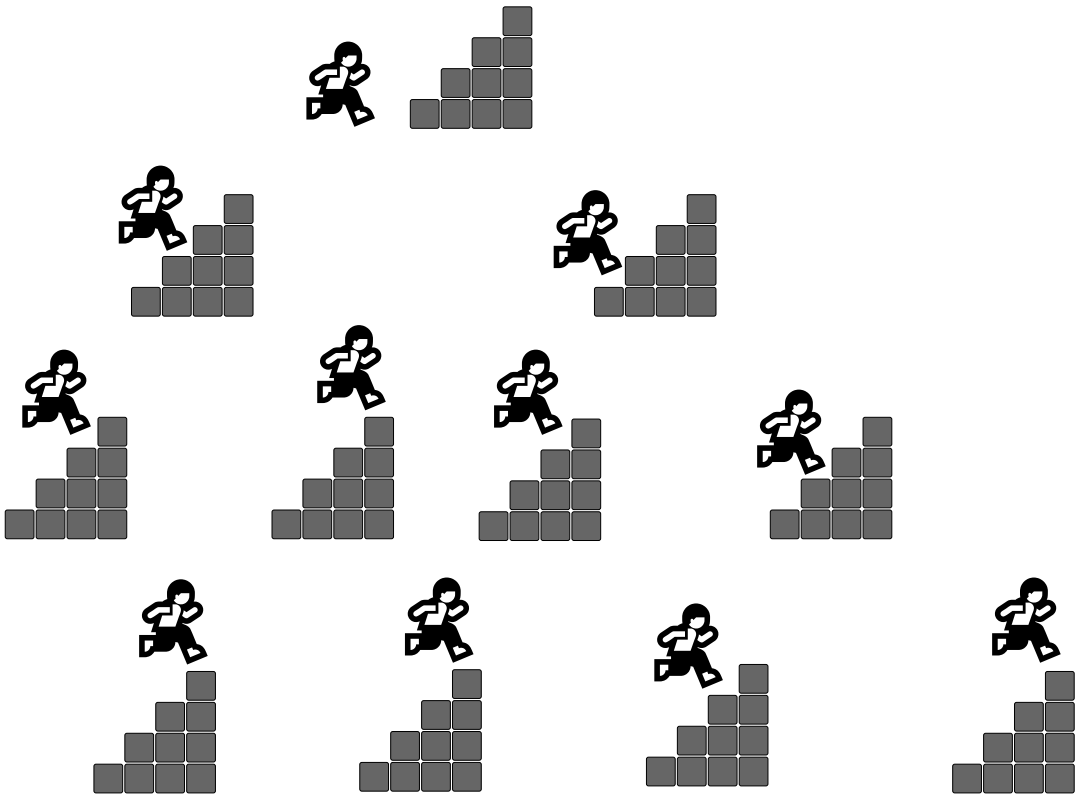
Nick can go up either one or two steps



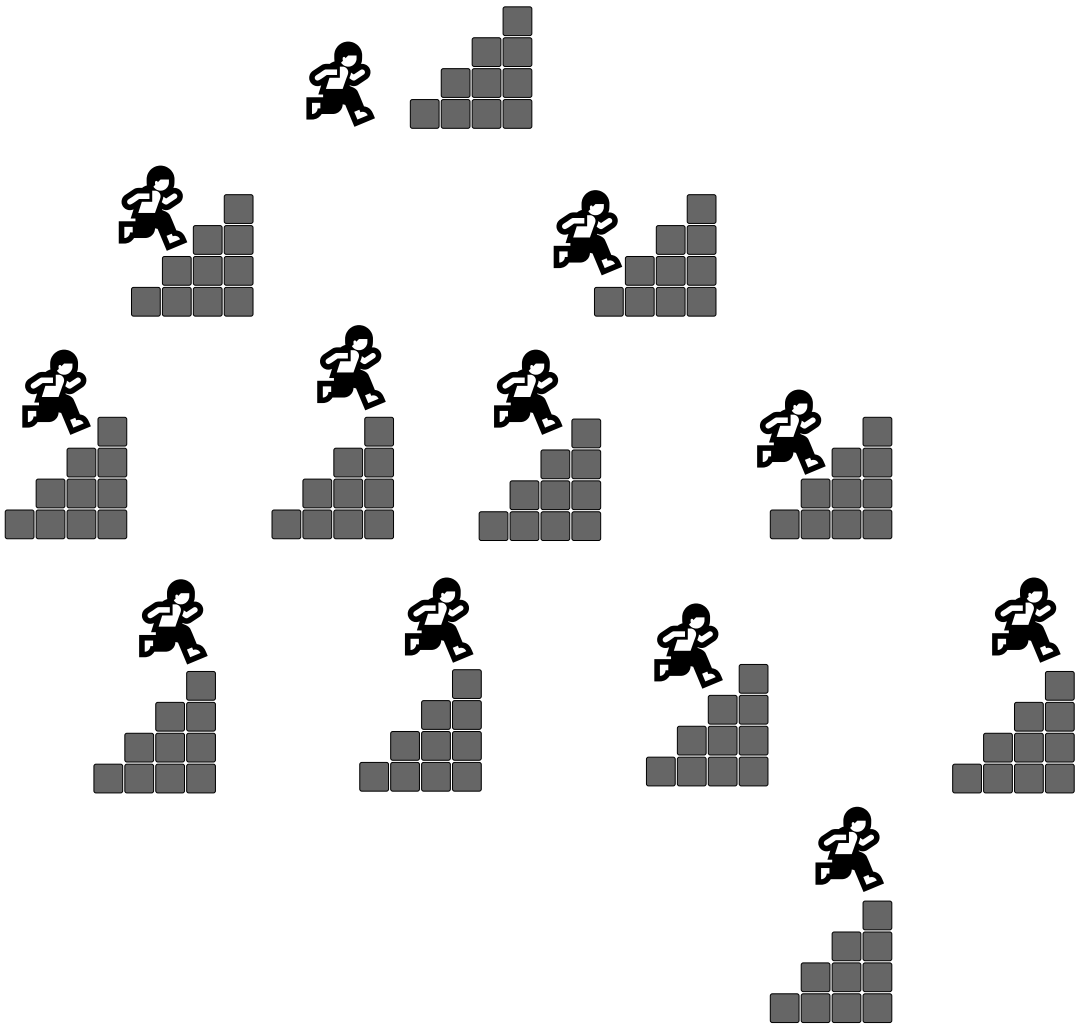
Nick can go up either one or two steps



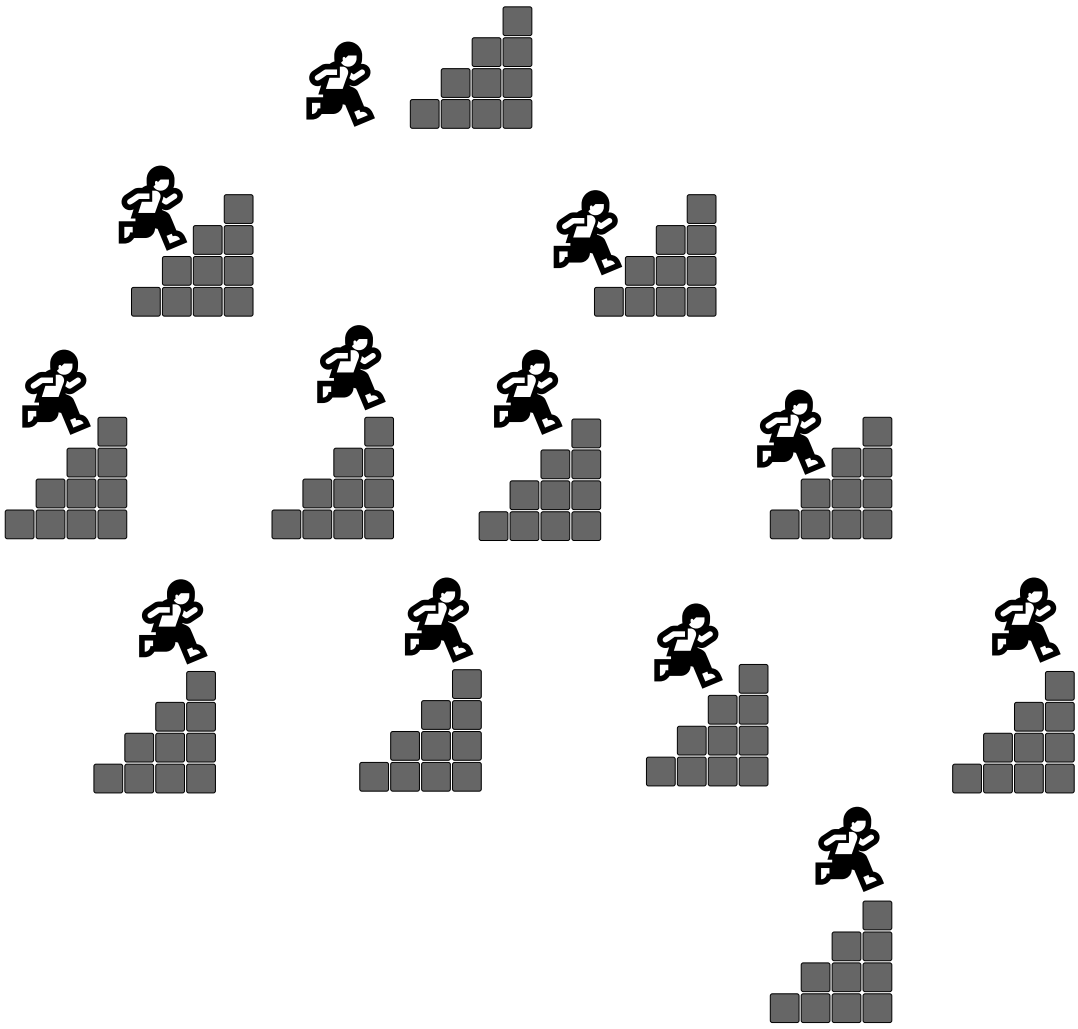
One of the subproblem has already concluded while others have not



Most of the calls have concluded, only one remaining



All calls concluded, count leaf nodes



All calls concluded, count leaf nodes

**8** of 8

## Time complexity #

Every time we can make up to `m` recursive calls. `m` recursive calls from `m` recursive calls and so on results in exponential time complexity. That means $m \times m \times m ... m = m^n$, this time complexity is $\mathbf{O(m^n)}$.

# Dynamic programming properties #

Notice how this problem satisfies both conditions of dynamic programming.

## Optimal substructure #

## Optimal substructure #

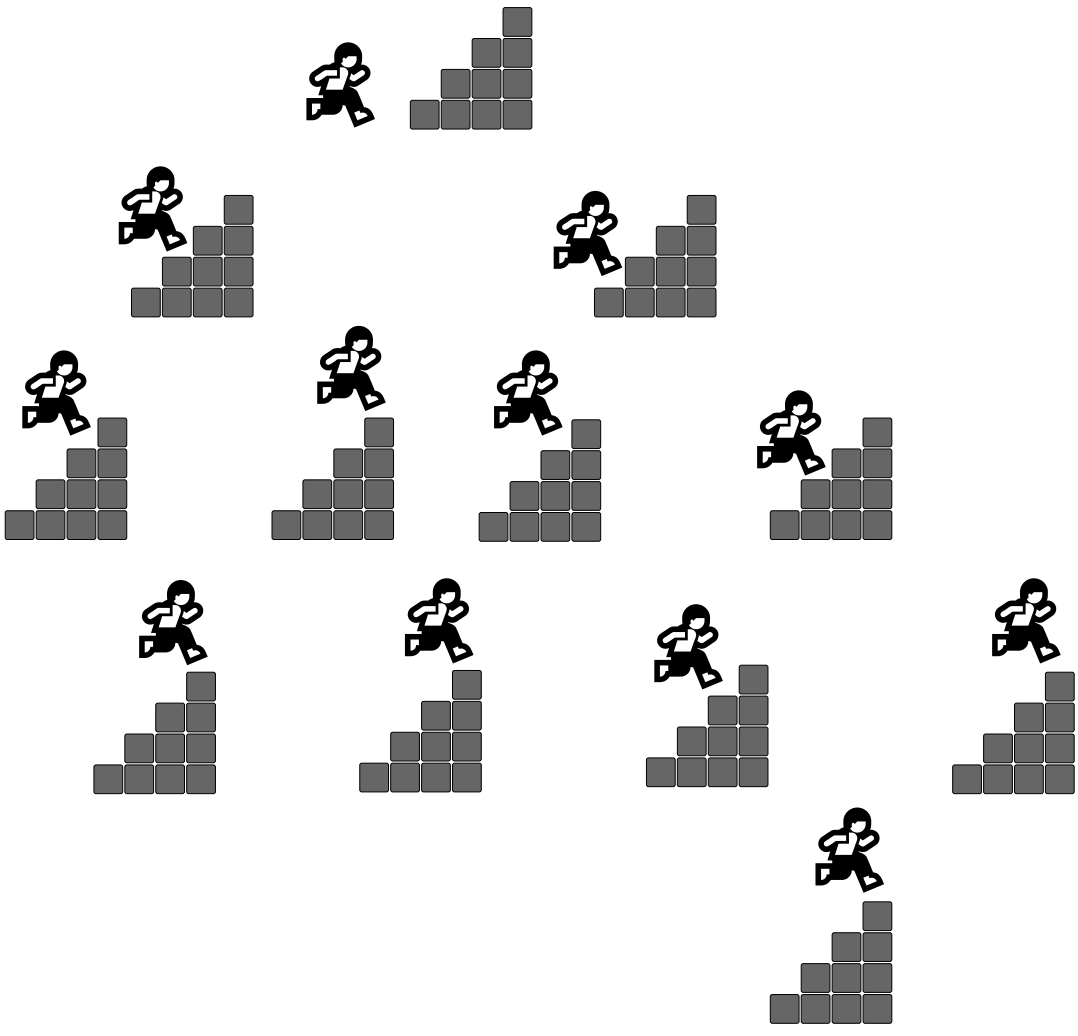
If we want to evaluate the answer for `staircase(n, m)`, and we have answers for `staircase(n-1, m)`, `staircase(n-2, m)`, ... `staircase(n-m, m)`, we can simply sum up all these answers to get the answer for the `staircase(n, m)`.

## Overlapping subproblems #

Notice how `staircase(n-2, m)` will appear as a subproblem of `staircase(n, m)` and `staircase(n-1, m)`.

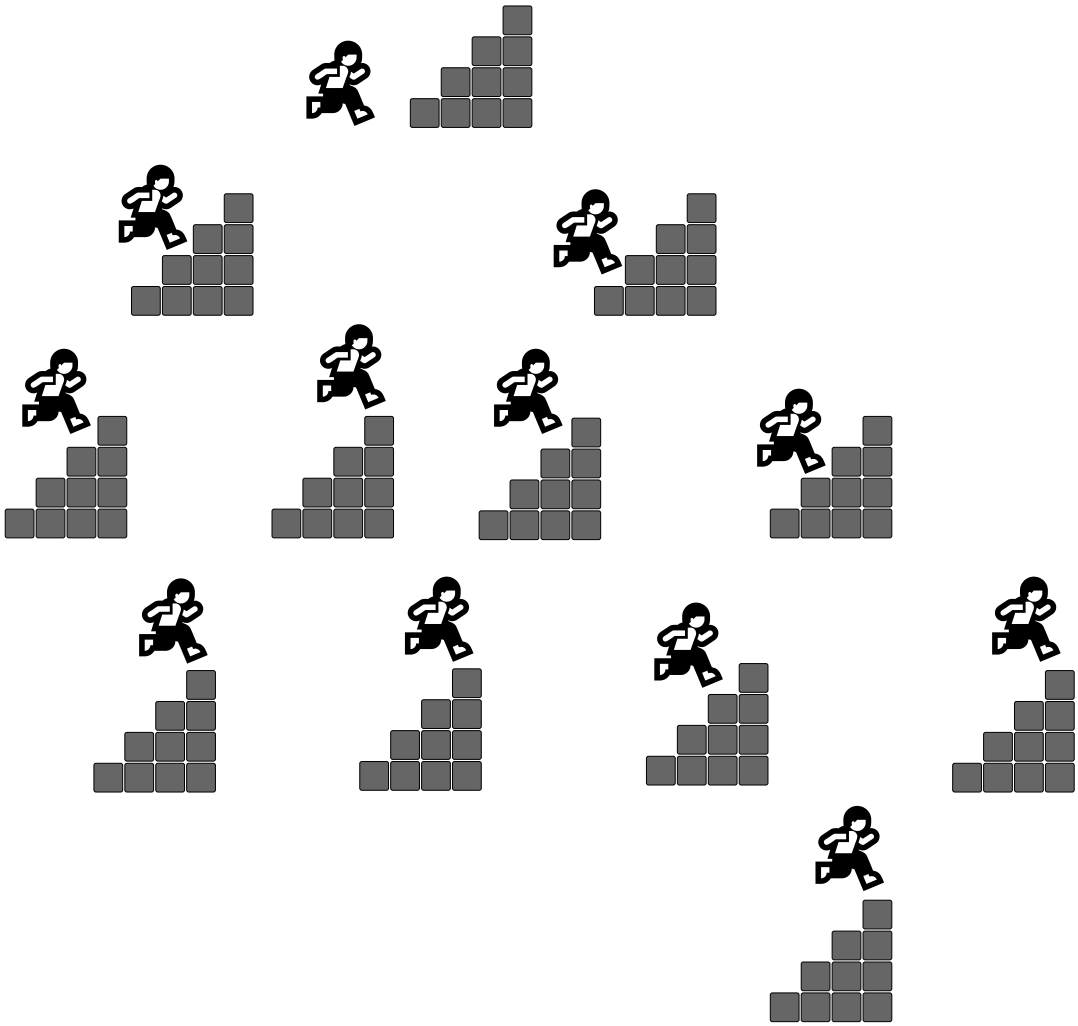
If you look at the final slide, there are some repeating sub-problems. Can you identify them?

If we have come two steps up the staircase, we know there are only two ways to take the next two steps. Thus the call `staircase(2,2)` will always evaluate to `2`. We are reevaluating it two times. The same is the case for the call `staircase(3,2)`, we are re-evaluating it three times.

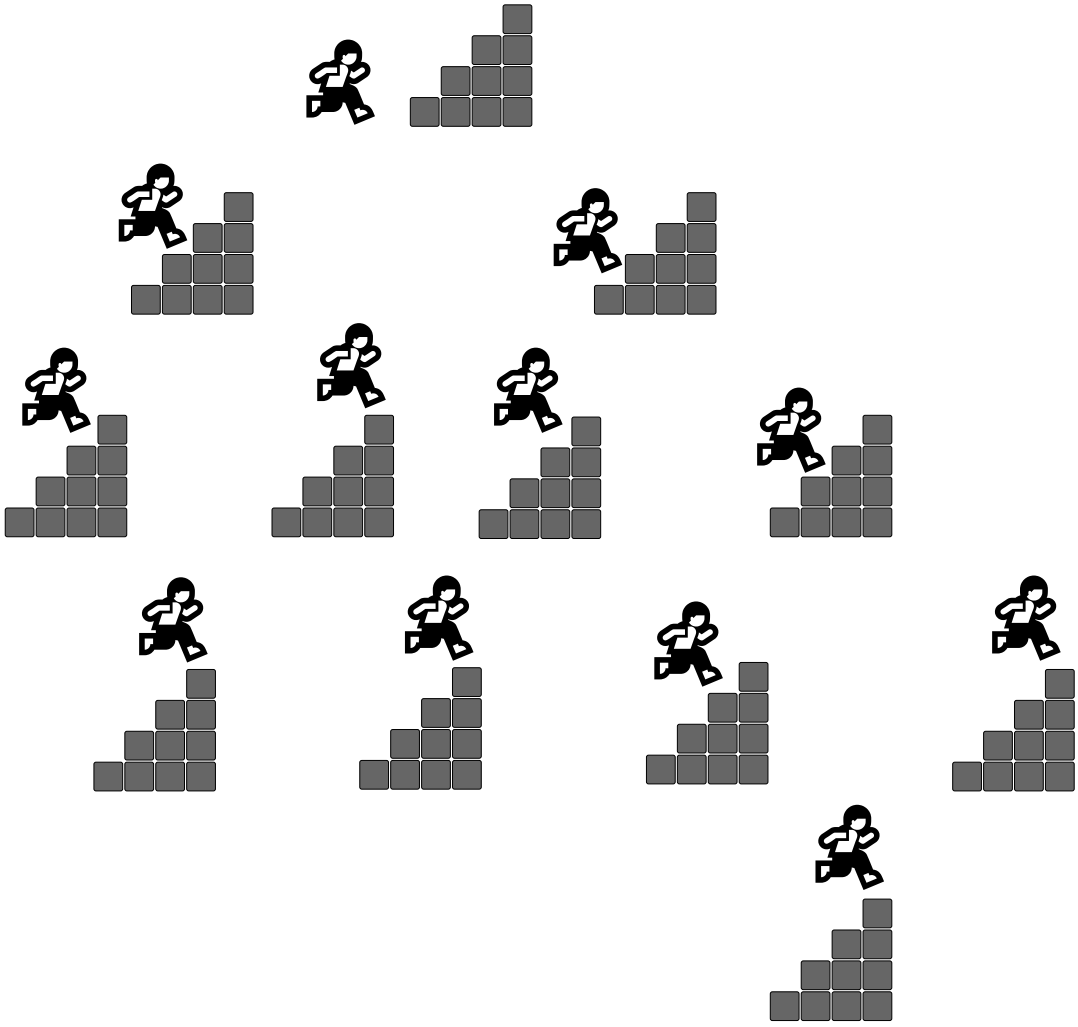


Identify the repeating sub-problems

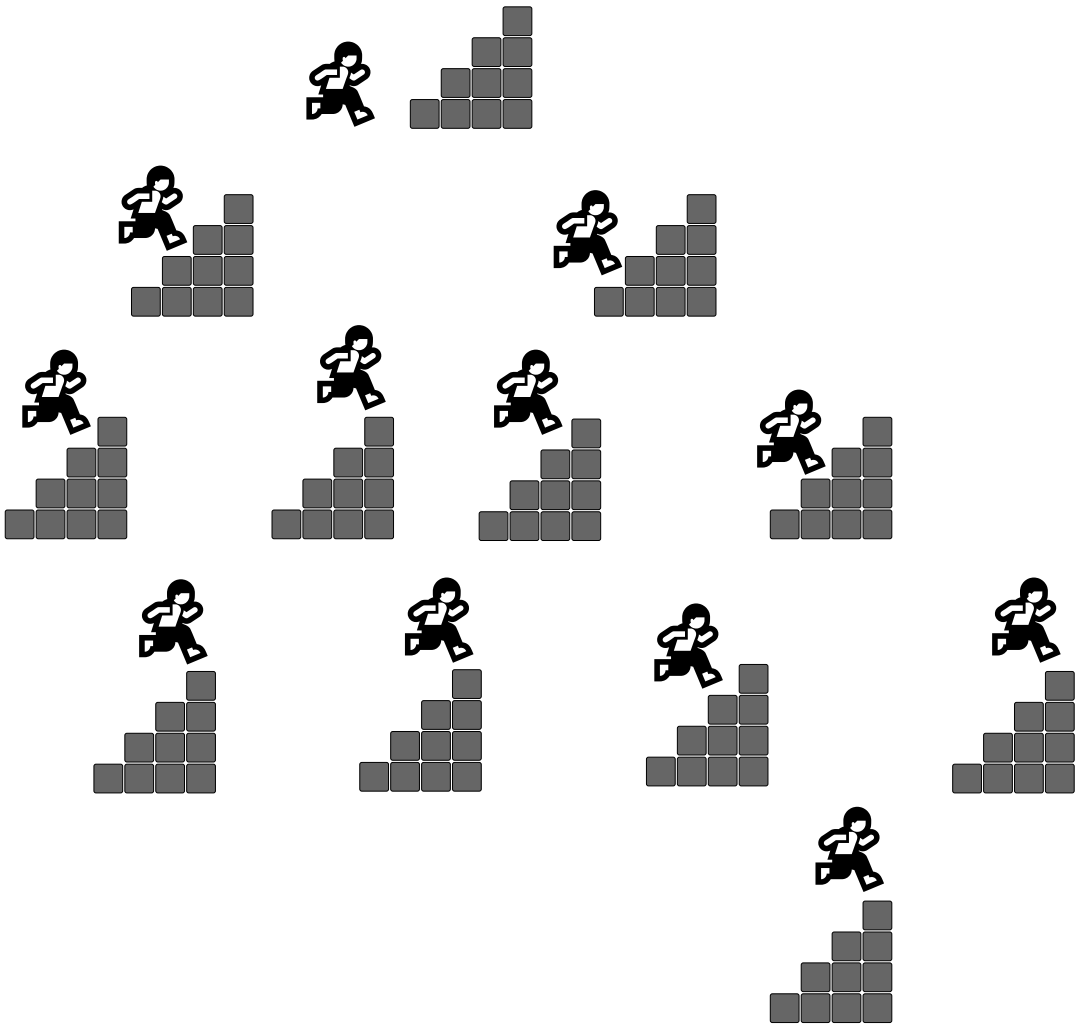




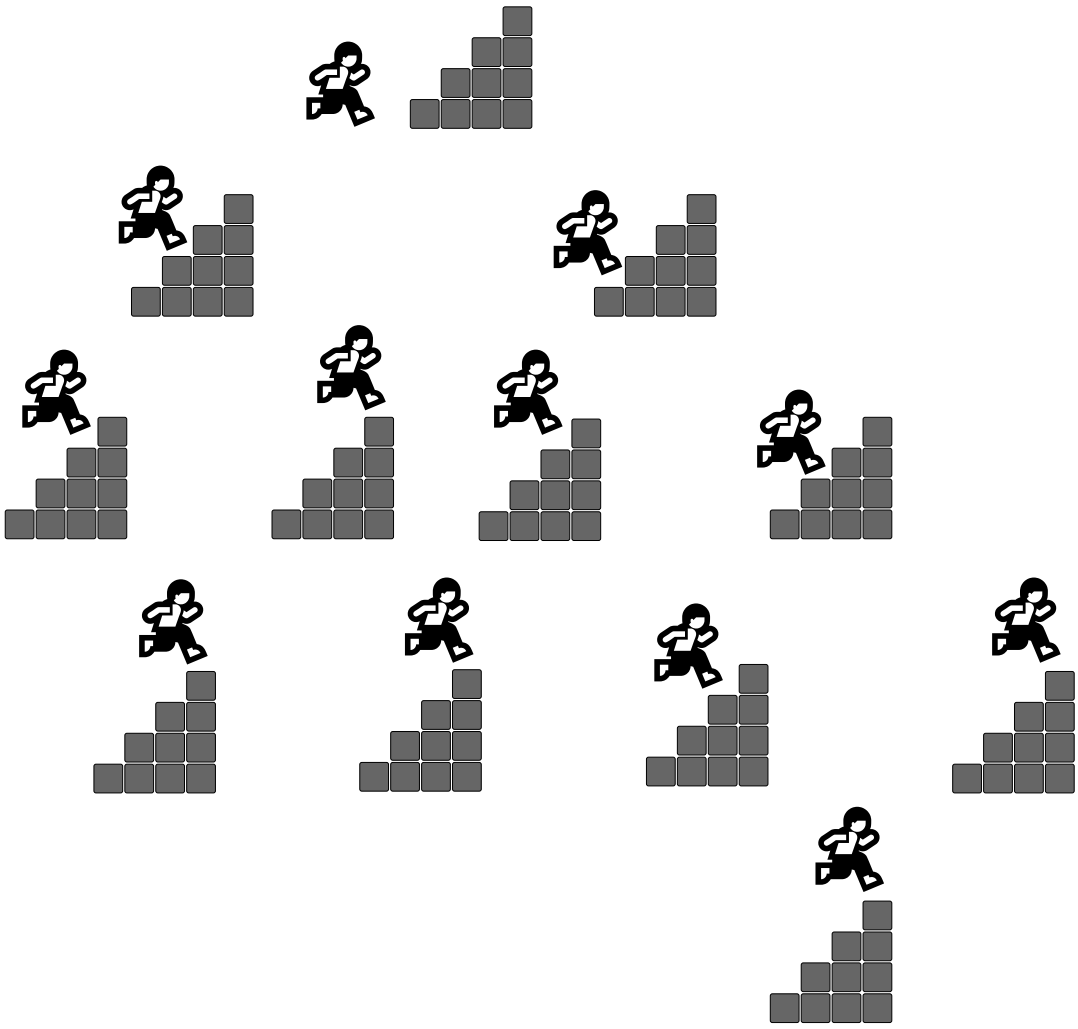
Identify the repeating sub-problems



Identify the repeating sub-problems



Identify more repeating sub-problems



Identify more repeating sub-problems

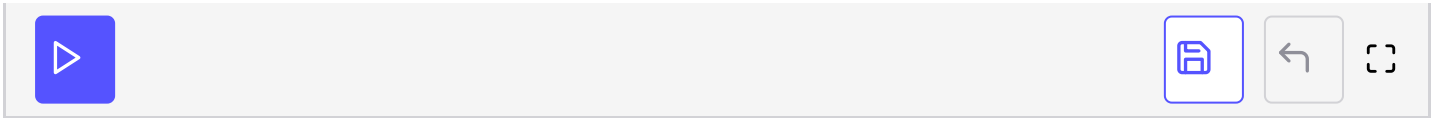
**5** of 5

Now that you can identify some repeating subproblems. Let's see a solution that uses memoization.

## Solution 2: Recursion with memoization #

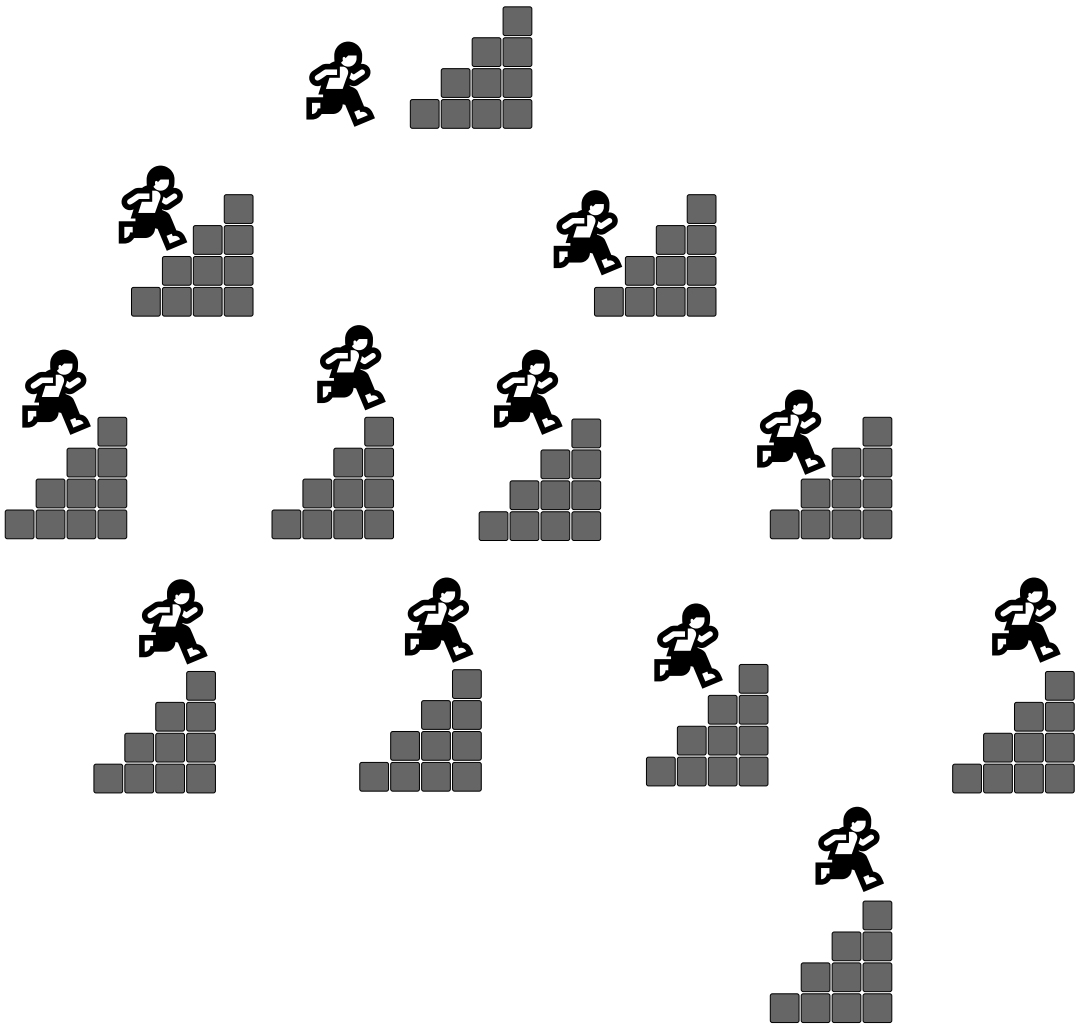
```
def nthStair(n, m, memo):
  # base case of when there is no stair
  if n == 0:
    return 1
  # before recursive step check if result is memoized
  if n in memo:
    return memo[n]
```

```
  ways = 0
  # iterate over number of steps, we can take
  for i in range(1,m+1):
    # if steps remaining is smaller than the jump step, skip
    if i <= n:
      #recursive call with n i units lesser where i is the number of steps taken here
      ways += nthStair(n-i, m, memo)
  # memoize result before returning
  memo[n] = ways
  return ways

def staircase(n, m):
  memo = {}
  # helper function to add memo dictionary to function
  return nthStair(n, m, memo)

print(staircase(100, 6))
```

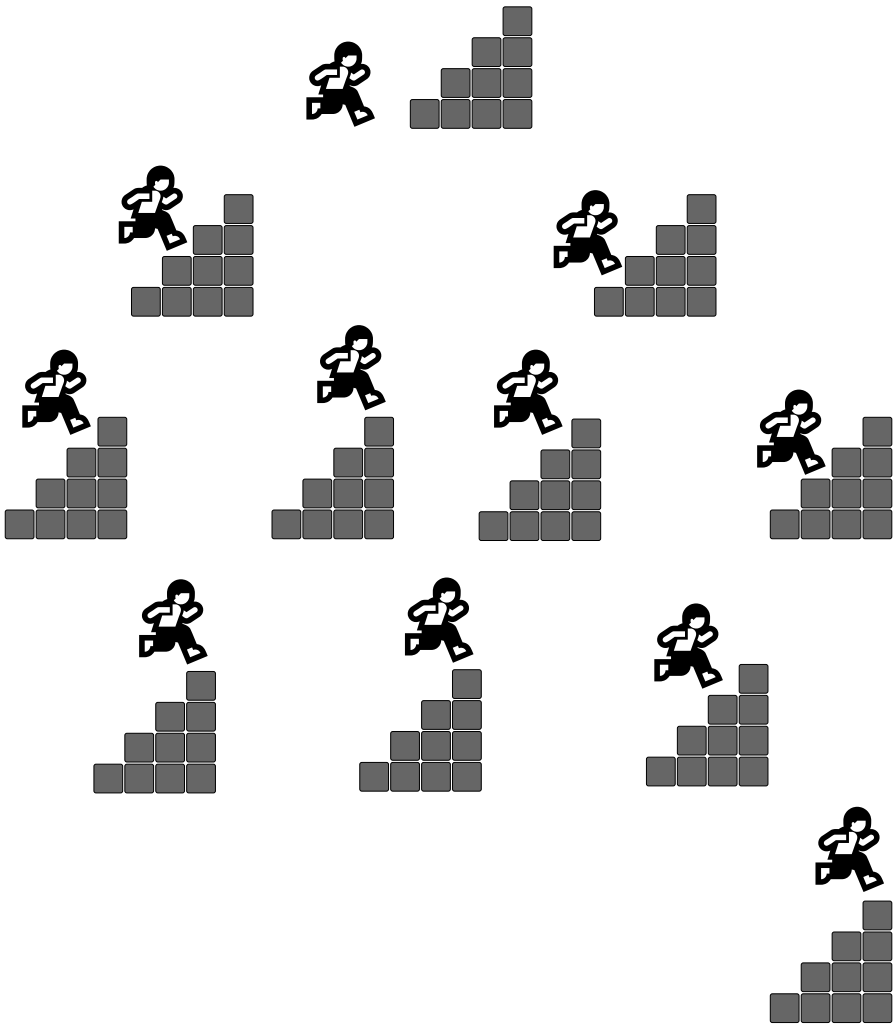


## Explanation

The only thing different from the first solution here is the addition of memoization. We have created a new function, `nthStair`, just to pass a memoization dictionary `memo` in the arguments of the function (*line 22*). Now, before we make the recursive call, we check if we have the result in the `memo` dictionary, if we do we can use it. Otherwise, we make the recursive call and then store the result in the dictionary. Look at the following visualization to see how memoization helps us in our solution.



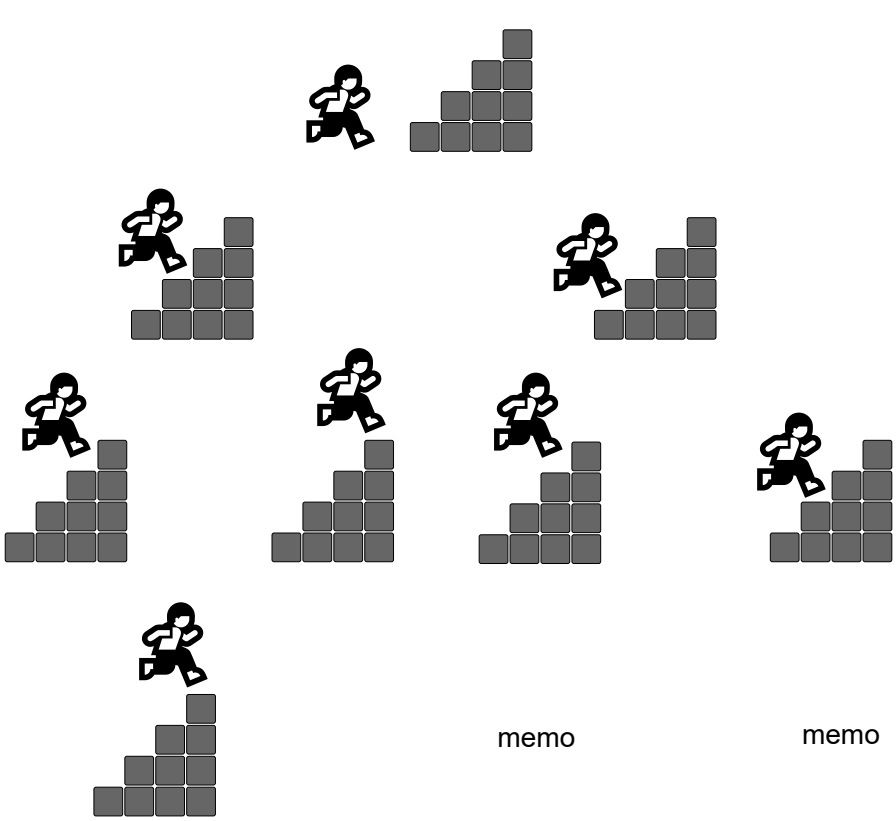
Build a memo table



| n | value |
|---|---|
| 2 | 2 |
| 1 | 1 |

memo table

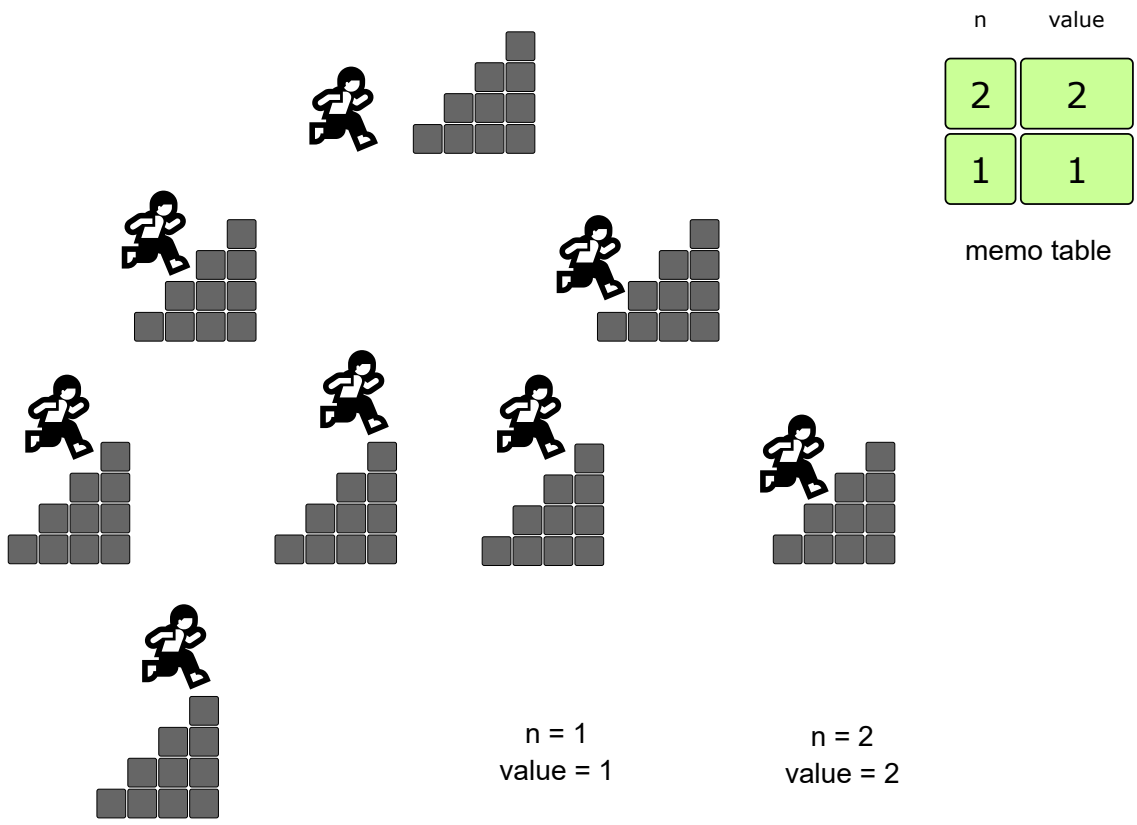
Identify more repeating sub-problems




| n | value |
|---|---|
| 2 | 2 |
| 1 | 1 |

memo table

Check memo for repeating sub-problems




## Time complexity #

Now `staircase(i, m)` will be evaluated only once for each value of `i` from `1` to `n`. And for each such call, we will just have to add value for each value of stairs i.e. `1` to `m`. Thus, just like Fibonacci numbers, we will not have a tree-like structure. This way, the time complexity will be **O(nm)**.

---

In the next lesson, you will solve another coding challenge.