Caching

May 20, 2020

0.0.1 What is Caching?

Caching can be defined as the process of storing data into a temporary data storage to avoid recomputation or to avoid reading the data from a relatively slower part of memory again and again. Thus cachig serves as a fast "look-up" storage allowing programs to execute faster.

Let's use caching to chalk out an efficient solution for ***a problem from the Recursion lesson****.*

0.0.2 Problem Statement - (Recursion) - Repeat Exercise

A child is running up a staircase and can hop either 1 step, 2 steps or 3 steps at a time. Given that the staircase has a total `n` steps, write a function to count the number of possible ways in which child can run up the stairs.

For e.g.

- `n == 1` then `answer = 1`
- `n == 3` then `answer = 4` The output is `4` because there are four ways we can climb the staircase:
  - 1 step + 1 step + 1 step
  - 1 step + 2 steps
  - 2 steps + 1 step
  - 3 steps
- `n == 5` then `answer = 13`

Hint You would need to make use of the Inductive Hypothesis, which comprises of the following two steps: 1. **The Inductive Hypothesis**: Calculate/assume the results for base case. In our problem scenario, the base cases would be when n = 1, 2, and 3.

2. **The Inductive Step**: Prove that for every $n >= 3$, if the results are true for n , then it holds for $(n + 1)$ as well. In other words, assume that the statement holds for some arbitrary natural number n , and prove that the statement holds for $(n + 1)$.

```
In [1]: def staircase(n):
            # Base Case - What holds true for minimum steps possible i.e., n == 1? Return the nu

            # Inductive Hypothesis - What holds true for n == 2 or n == 3? Return the number of
```

```
            # Inductive Step (n > 3) - use Inductive Hypothesis to formulate a solution

            pass
```

In [2]:
```
def test_function(test_case):
    answer = staircase(test_case[0])
    if answer == test_case[1]:
        print("Pass")
    else:
        print("Fail")
```

In [3]:
```
test_case = [4, 7]
test_function(test_case)
```

```
Fail
```

In [4]:
```
test_case = [5, 13]
test_function(test_case)
```

```
Fail
```

In [5]:
```
test_case = [3, 4]
test_function(test_case)
```

```
Fail
```

In [6]:
```
test_case = [20, 121415]
test_function(test_case)
```

```
Fail
```

Hide Solution

In [ ]:
```
def staircase(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    elif n == 3:
        return 4
    return staircase(n - 1) + staircase(n - 2) + staircase(n - 3)
```

0.0.3 Problem Statement - (Caching)

While using recursion for the above problem, you might have noticed a small problem with efficiency.

Let's take a look at an example.

- Say the total number of steps are `5`. This means that we will have to call at `(n=4)`, `(n=3)`, `and (n=2)`
- To calculate the answer for `n=4`, we would have to call `(n=3)`, `(n=2) and (n=1)`

You can notice that even for a small number of staircases (here 5), we are calling `n=3` and `n=2` multiple times. Each time we call a method, additional time is required to calculate the solution. In contrast, instead of calling on a particular value of `n` again and again, we can **calculate it once and store the result** to speed up our program.

Which data structure are you thinking to store the results?

Your job is to use any data-structure that you have used until now to write a faster implementation of the function you wrote earlier while using recursion.

```
In [ ]: def staircase(n):
            pass

In [ ]: test_case = [4, 7]
        test_function(test_case)

In [ ]: test_case = [5, 13]
        test_function(test_case)

In [ ]: test_case = [3, 4]
        test_function(test_case)

In [ ]: test_case = [20, 121415]
        test_function(test_case)
```

Hide Solution

```
In [ ]: def staircase(n):
            num_dict = dict({})
            return staircase_faster(n, num_dict)

        def staircase_faster(n, num_dict):
            if n == 1:
                output = 1
            elif n == 2:
                output = 2
            elif n == 3:
                output = 4
            else:
```

```
    if (n - 1) in num_dict:
        first_output = num_dict[n - 1]
    else:
        first_output =  staircase_faster(n - 1, num_dict)

    if (n - 2) in num_dict:
        second_output = num_dict[n - 2]
    else:
        second_output = staircase_faster(n - 2, num_dict)

    if (n - 3) in num_dict:
        third_output = num_dict[n - 3]
    else:
        third_output = staircase_faster(n - 3, num_dict)

    output = first_output + second_output + third_output

num_dict[n] = output;
return output
```