What is Tabulation?

In this lesson, we will learn about a technique to store results of evaluated subproblems in bottom-up dynamic programming.

Tabulation

As we discussed in the **bottom-up** approach, we start from the bottom, the smallest subproblem, or the base case. After evaluating the base case, we evaluate a slightly bigger problem and store its result. We continue doing this, i.e., build on the solution of smaller subproblems to evaluate bigger and bigger subproblems until we find the solution to our main problem.

In bottom-up dynamic programming, the process of storing results of evaluated subproblems in an array is called **tabulation**.

How is it different from memoization?

In memoization and top-down dynamic programming, we evaluated and stored results on an ad-hoc basis. We only evaluated something when it was needed. This means if a problem depends on only two subproblems, only those subproblems would have been evaluated. This comes naturally from the design of recursion. Moreover, the order in which we evaluated subproblems did not matter either. Evaluating *Fibonacci(n)* before *Fibonacci(n-1)* or vice versa did not have any effect on the result or running time of the algorithm.

This is not the case with tabulation. Here, we will have to evaluate almost every subproblem that lies between the base case and the main problem and we will have to do so in order. Thus, in tabulation, we typically use lists and arrays instead of hashtables. Lists force us to start from scratch and solve everything sequentially, which is the crux of bottom-up dynamic programming as well. For example, if we were writing an algorithm for the factorial of a number `n` using a bottom-up approach, we first establish a factorial of `0`, then we will use it for factorial of `1`, `2` and so on until we reach `n`. We can tabulate the results by keeping a list of size `n`, where each index will store the factorial against that index. For example at index `4`, we will store `4!`. Look at the code of the factorial algorithm below.

We have also given the code for a recursive algorithm to evaluate the factorial for comparison.

BottomUp Recursion

```
def factorial(n):
  #tabulation table of size n+1
  table = [0]*(n+1)
  # base case of 0! = 1
  table[0] = 1
  # iterate until n
  for i in range(1,n+1):
    # using answer to i-1th problem from tabulation to build answer for ith problem
    table[i] = table[i-1]*i
  # return answer; the nth factorial
  return table[n]

print(factorial(30))
```

There are a couple of key takeaways from this coding example. Notice how we are building from the smallest subproblem up to the bigger subproblems. We started with 0^{th} factorial (*line 5*) and then we used it to build bigger factorials through for loop (*lines 7-9*). The second important takeaway is that we could not have had `factorial(n)` evaluated before `factorial(n-1)` because the former requires the latter to evaluate. Therefore, the order in which you build up your tabulation table is extremely crucial and varies with problem to problem. In some problems, you might need multi-dimensional lists for tabulation.

In the next lesson, we will see the bottom-up implementation of the Fibonacci numbers algorithm.