Limitations of Top-Down Dynamic Programming

In this lesson, we will see some limitations of top-down dynamic programming approach.

We'll cover the following

- Massive amount of recursion
- Expensive recursion
- How to avoid recursion?

Massive amount of recursion

Top-down dynamic programming consists of recursive calls to the subproblems. The answer is not evaluated until all the subproblems have been evaluated; which in turn depends on their own subproblems. We quickly have a huge number of incomplete recursive calls on our program stack. If we keep making these recursive calls and no previous calls have evaluated; we will run out of space on the stack. This will end up in erroneous termination of the program. Run the following code of the Fibonacci numbers algorithm we discussed in the last chapter.

```
memo = {} #dictionary for Memoization

def fib(n):
  if n == 0: # base case 1
    return 0
  if n == 1: # base case 2
    return 1
  elif n in memo: # Check if result for n has already been evaluated
    return memo[n] # return the result if it is available
  else: # otherwise recursive step
    memo[n] = fib(n-1) + fib(n-2) # store the result of n in memoization dictionary
    return memo[n] # return the value

print (fib(1000))
```

The program will terminate and if you scroll down to the error log you will see this line:

```
RecursionError: maximum recursion depth exceeded in comparison
```

This error means that we ran out of the stack memory allocated to our program.

> 💡 **Stack** is the memory provided to the program to store temporary variables and return addresses. When a program makes function calls, variables and return addresses are pushed onto the stack. When a function call has been evaluated, the return address is popped from the stack and the program continues execution from the statement at the return address.

Expensive recursion

In addition to being costly in terms of the stack memory, recursion is also incredibly expensive in terms of program execution. Every function call means pushing that function's variables and return address onto the stack and moving the instruction pointer to the address of that function. When a function is returned it pops the address from the stack and moves the instruction pointer to the return address. While this may look trivial at first, this cost adds on with every recursive call. Compare this to a simple loop. A loop involves just a conditional jump to the address, which is not an expensive operation at all. There is no creation of new variables, storing of return addresses, or manipulation of the stack. This makes loops faster and more scalable than recursion. This is why most of the programs are written iteratively (using loops) instead of recursively, even though recursion is more intuitive.

How to avoid recursion?

This is why we need a new approach to dynamic programming that does not include recursion. Remember how *recursion* was nothing but the breaking down of a problem into subproblems until those subproblems were evaluated; and when subproblems were evaluated, the main problem also evaluated. Now if we want to avoid recursion, we will go the opposite way. We will solve subproblems first and then whenever the main problem requires answers to the subproblems, we will return the answers we had already evaluated. This approach of starting with the evaluation of subproblems and then building up to the solution of the main

evaluation of subproblems and then building up to the solution of the main problem is called **bottom-up dynamic programming**.

In the next lesson, we will learn about *tabulation*, a technique to store the results of evaluated subproblems.