The Fibonacci Numbers Algorithm with Tabulation

In this lesson, we will revisit the Fibonacci numbers algorithm and implement it in a bottom-up manner.

We'll cover the following

- Simple recursion
- Bottom-up dynamic programming
- Explanation
- Time and space complexity

Simple recursion

Following is the simple implementation of the Fibonacci numbers algorithm. We covered this in one of the lessons from the first chapter. Please feel free to look at it before we continue to the bottom-up solution.

Bottom-up dynamic programming

Notice how we started from `Fib(6)` and went on to break it into `Fib(5)` and `Fib(4)`. We saw that these had not been evaluated so we broke them further until we hit the base case. Now let's look at an alternate approach, where we start from the base case, i.e., `Fib(0)` and `Fib(1)`, and then build-up to the solution. We will keep storing all the subproblems along the way.

So, the plan is to start from `Fib(0)` and evaluate every Fibonacci number between `0` and `n` giving us the value of `Fib(n)`.

Look at the coding playground below.

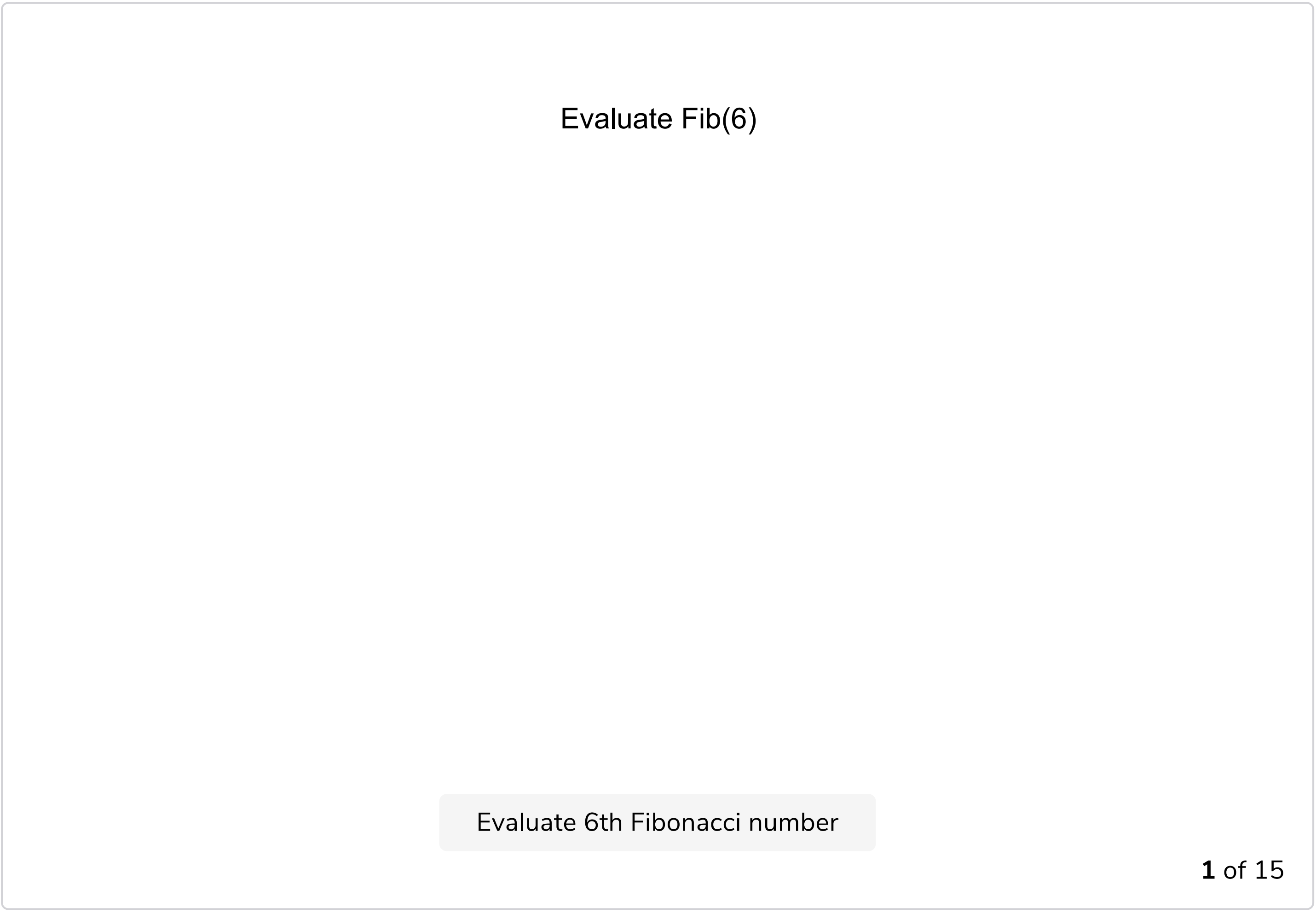
```
def fib(n):
  if n == 0:
    return 0
  if n == 1:
    return 1
  # table for tabulation
  table = [None] * (n+1)
  table[0] = 0         # base case 1, fib(0) = 0
  table[1] = 1        # base case 2, fib(1) = 1
  # filling up tabulation table starting from 2 and going upto n
  for i in range(2,n+1):
    # we have result of i-1 and i-2 available because these had been evaluated already
    table[i] = table[i-1] + table[i-2]
  # return the value of n in tabulation table
  return table[n]

print(fib(10000))
```

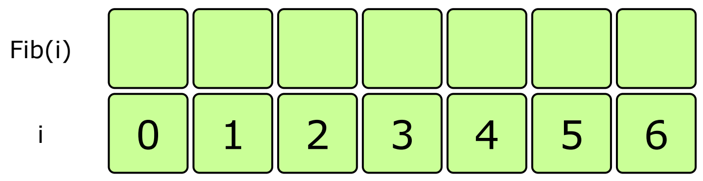
Explanation

Let’s see what’s happening here. First, we created a list for the purpose of tabulation called `table`. We set its size equal to `n+1` since we are going to evaluate `n+1` total Fibonacci numbers (`0,1,2...n`). Next, we updated the value for the base cases in the table by setting 0^{th} and 1^{st} index to `0` and `1` respectively. The only thing left to do is iterate over the table to fill it up. At the end of the iteration, we will have n^{th} Fibonacci number at the last index of `table`.

Look at the following visualization for a better understanding of the algorithm.



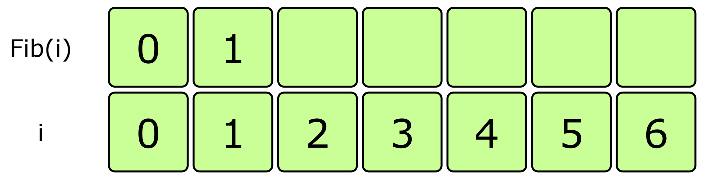
Evaluate Fib(6)



Construct an empty list of size 7 for tabulation

2 of 15

Evaluate Fib(6)



Update value of 0 and 1, since these are base cases

3 of 15

Evaluate Fib(6)

Fib(i)	0	1					
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1					
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1	1				
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

6 of 15

Evaluate Fib(6)

Fib(i)	0	1	1				
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

7 of 15

Evaluate Fib(6)

Fib(i)	0	1	1	2			
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

8 of 15

Evaluate Fib(6)

Fib(i)	0	1	1	2			
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

9 of 15

Evaluate Fib(6)

Fib(i)	0	1	1	2	3		
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1	1	2	3		
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1	1	2	3	5	
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1	1	2	3	5	
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1	1	2	3	5	8
i	0	1	2	3	4	5	6

Now iterate over the table and fill each value using last two

Evaluate Fib(6)

Fib(i)	0	1	1	2	3	5	8
i	0	1	2	3	4	5	6

Table filled, Fib(6) is at 6th place in table

Time and space complexity

We are reusing values from the table and avoiding recomputation. Thus, we will only evaluate every Fibonacci number once making the total run time complexity **O(n)**. To avoid recomputation, we need to store all the Fibonacci numbers until `n`, making the space complexity of this algorithm **O(n)**. But think about it; do we really need to store all the entries in the table or can we do better?

In the next lesson, we will optimize this algorithm to reduce the space complexity.