Optimizing the Fibonacci Numbers Algorithm

In this lesson, we will learn a better way to tabulate the Fibonacci numbers algorithm for better space complexity.

We'll cover the following

Fibonacci numbers algorithm with tabulation

In the last lesson, we saw a bottom-up implementation of the Fibonacci numbers algorithm. We discussed how the time complexity of that algorithm was *O(n)*, thanks to tabulation. However, due to tabulation, our space complexity also rose to **O(n)**.

Do we really need this much space for this algorithm? Or can we do better by using our space smartly?

Space complexity from O(n) to O(1)

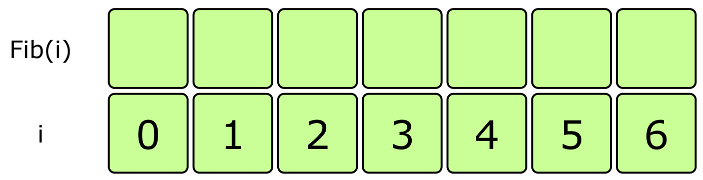
To evaluate any Fibonacci number, we need the preceding two Fibonacci numbers. Nothing else. Now that we are evaluating our Fibonacci numbers from the start, we may not need to store all preceding terms for higher Fibonacci numbers. This means that we only need to store the answer to `Fib(4)` to compute `Fib(5)` and `Fib(6)`. `Fib(7)` or any other later numbers do not require the answer to `Fib(4)`. Thus, instead of maintaining the state of all the `n-1` previous Fibonacci numbers, if we maintain the values of only the last two preceding numbers, our algorithm will run fine. Look at the following visualization.

Evaluate Fib(6)

Evaluate 6th Fibonacci number

1 of 9

Evaluate Fib(6)



Construct an empty list of size 7 for tabulation

2 of 9

Evaluate Fib(6)

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

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

3 of 9

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

4 of 9

Evaluate Fib(6)

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

We need last two values to fill this one

5 of 9

Evaluate Fib(6)

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

using 0 and 1, fill 2

6 of 9

Evaluate Fib(6)

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

For filling up the rest of the table, we are never going to need 0

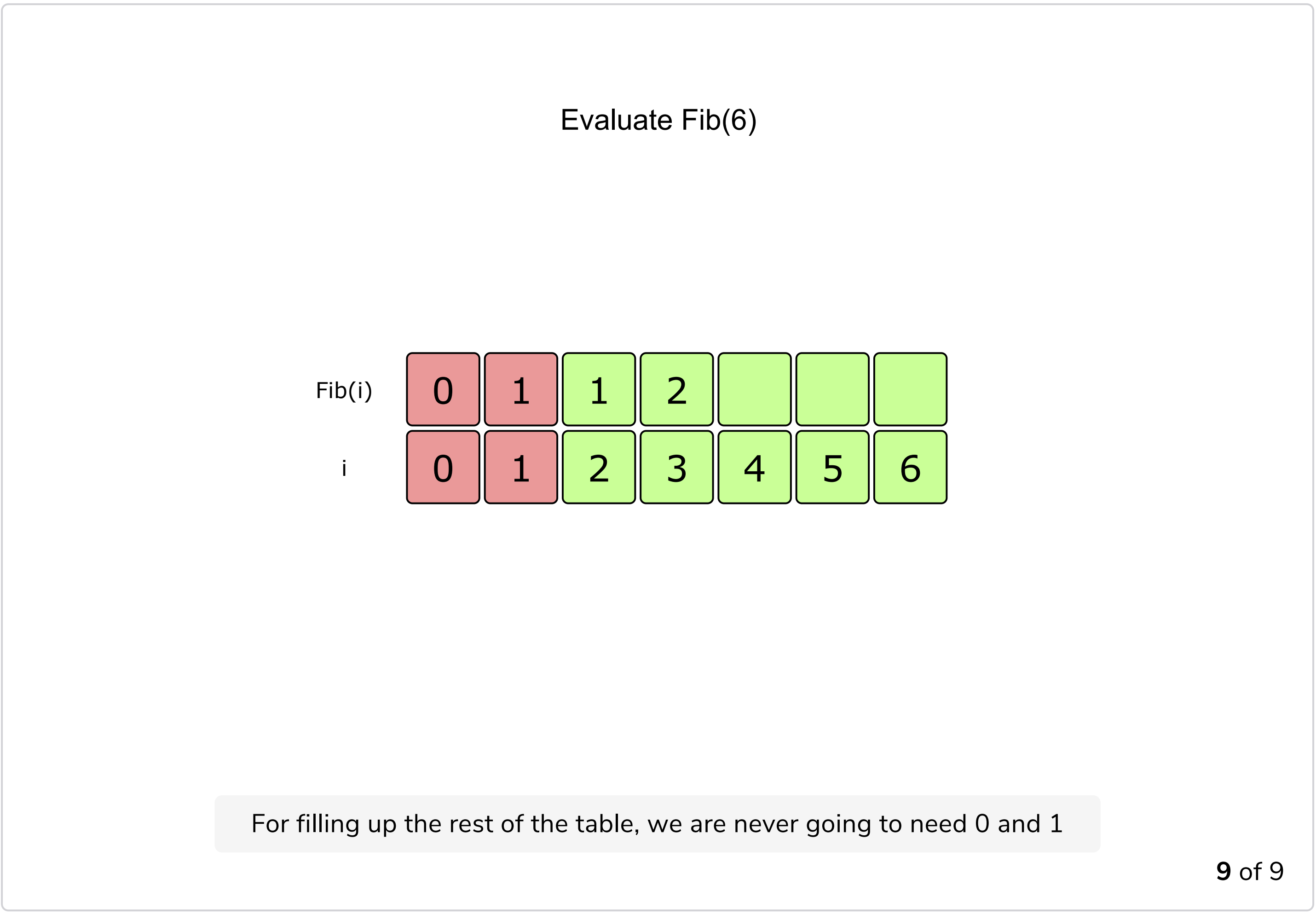
7 of 9

Evaluate Fib(6)

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

using 1 and 2, fill 3

8 of 9



Evaluate Fib(6)
Fib(i)
0
1
1
2
i
0
1
2
3
4
5
6
For filling up the rest of the table, we are never going to need 0 and 1
9 of 9

So, if we maintain only two previous numbers and keep updating them as we go along, we will be able to get nth Fibonacci number without keeping a table of size n.

Let's see the implementation of this algorithm below.

```python
def fib(n):
  if n == 0:            # base cases
    return 0
  if n == 1:            # base cases
    return 1
  secondLast = 0        # base case 1, fib(0) = 0
  last = 1              # base case 2, fib(1) = 1
  current = None        # initially set to None
  for i in range(1,n):    # iterate n times to evaluate n-th fibonacci
    # storing ith fibonacci in current by summing up i-1th and i-2th fibonacci
    current = secondLast + last
    secondLast = last   # updating for next iteration
    last = current
  return current      # return the value of n in tabulation table

print(fib(6))
```

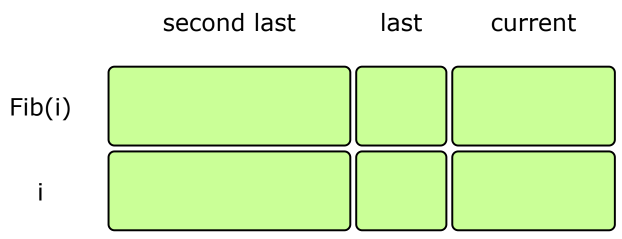
We are simply updating the variables to store the last and second last Fibonacci numbers by leveraging the fact that no larger Fibonacci number is going to need answers to numbers other than its preceding two numbers.

Evaluate Fib(6)

Evaluate 6th Fibonacci number

1 of 13

Evaluate Fib(6)



Have two variables, one for second last Fibonacci number, other for last one

Evaluate Fib(6)

	second last	last	current
Fib(i)	0	1	
i	0	1	

For base cases have 0th number in second last and 1st in last

Evaluate Fib(6)

	second last	last	current
Fib(i)	0	1	1
i	0	1	2

To evaluate next Fibonacci number, just add second last and last one

Evaluate Fib(6)

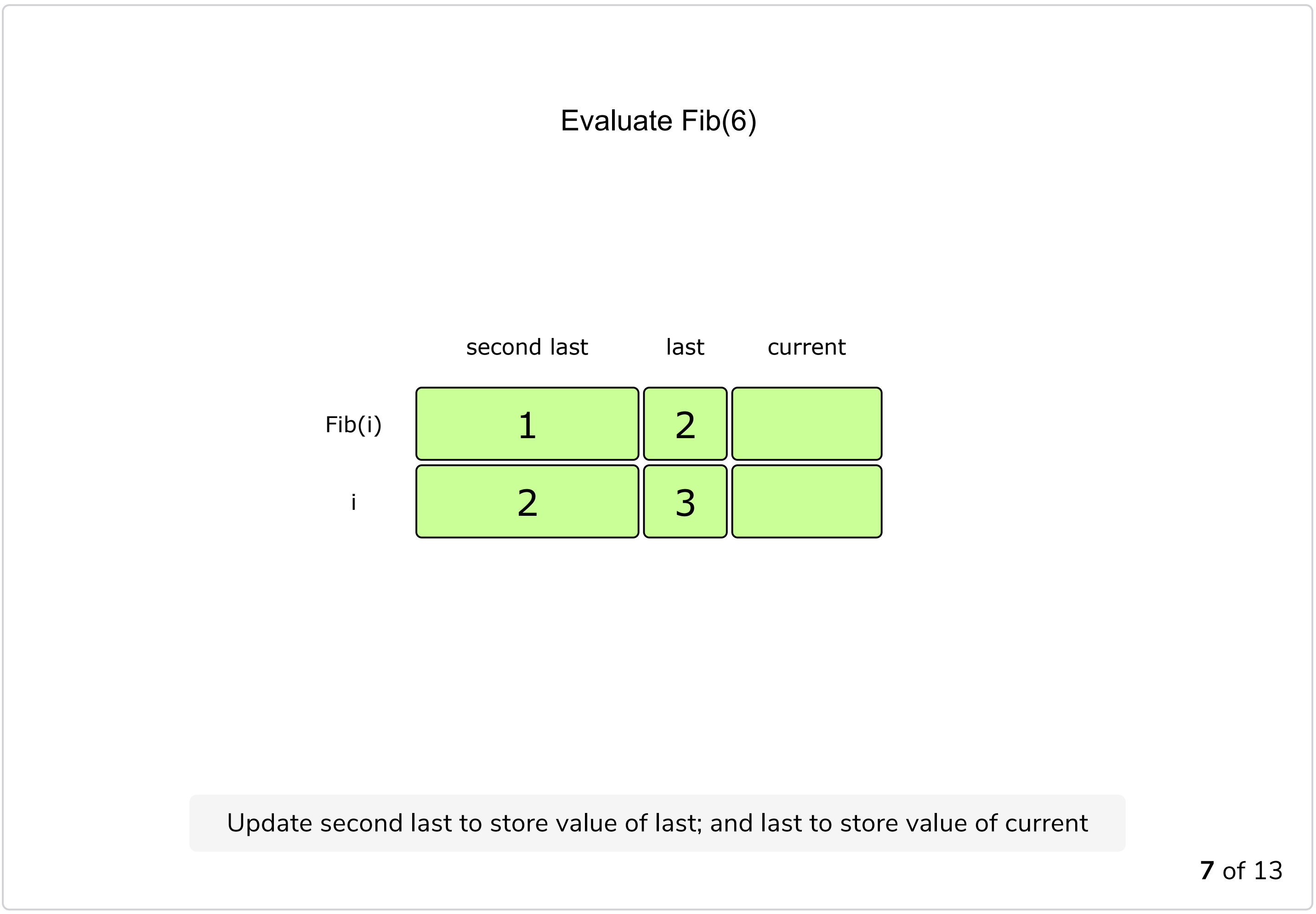
	second last	last	current
Fib(i)	1	1	
i	1	2	

Update second last to store value of last; and last to store value of current

Evaluate Fib(6)

	second last	last	current
Fib(i)	1	1	2
i	1	2	3

To evaluate next Fibonacci number, just add second last and last one

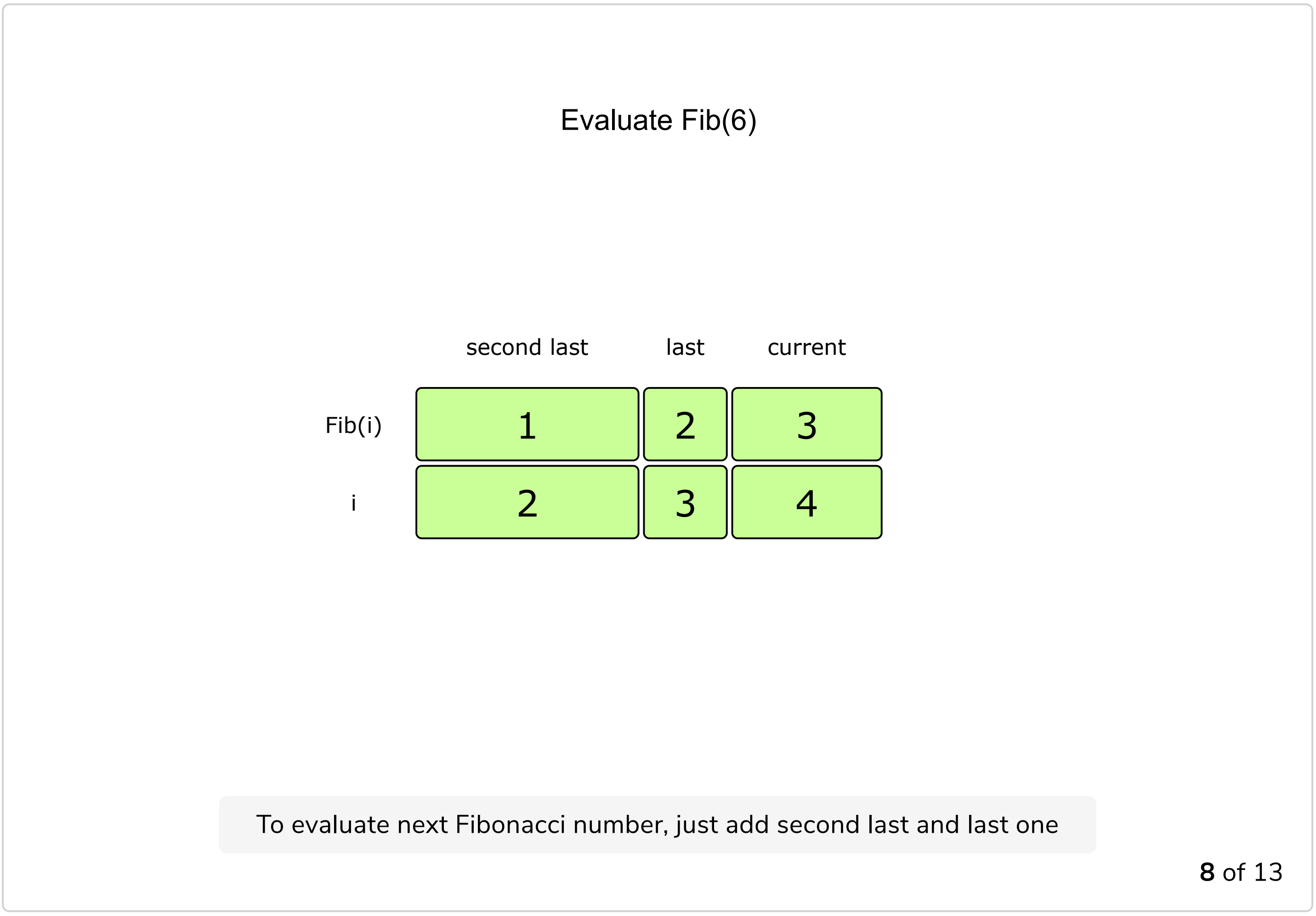


Evaluate Fib(6)

	second last	last	current
Fib(i)	1	2	
i	2	3	

Update second last to store value of last; and last to store value of current

7 of 13



Evaluate Fib(6)

	second last	last	current
Fib(i)	1	2	3
i	2	3	4

To evaluate next Fibonacci number, just add second last and last one

8 of 13

Evaluate Fib(6)

	second last	last	current
Fib(i)	2	3	
i	3	4	

Update second last to store value of last; and last to store value of current

Evaluate Fib(6)

	second last	last	current
Fib(i)	2	3	5
i	3	4	5

To evaluate next Fibonacci number, just add second last and last one

Evaluate Fib(6)

	second last	last	current
Fib(i)	3	5	
i	4	5	

Update second last to store value of last; and last to store value of current

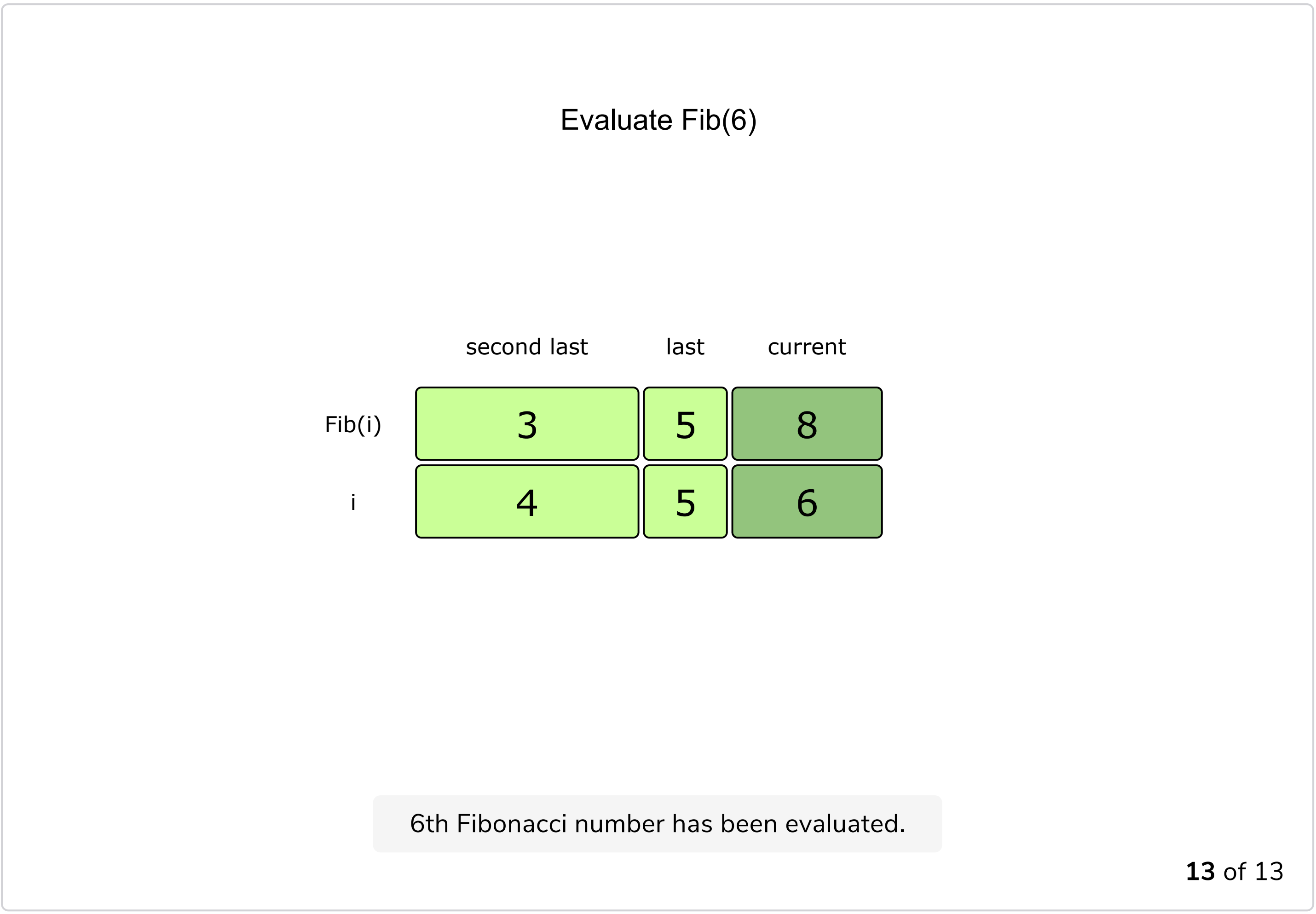
11 of 13

Evaluate Fib(6)

	second last	last	current
Fib(i)	3	5	8
i	4	5	6

To evaluate next Fibonacci number, just add second last and last one

12 of 13



Now that we have eliminated the list of `n` numbers from our algorithm and modified it to store only the last two, our space complexity has reduced from *O(n)* to **O(1)**.

Implication

We were able to do this by exploiting the fact that we are building our solution from the smallest possible subproblem and know exactly what subproblems bigger problems require. Thus, we can use this information to store only the required state.

We cannot do this in the top-down approach because subproblems are accessed in random order there.

Therefore, when solving a problem with bottom-up dynamic programming, look for the minimum state you need. Look at what subproblems are required by every bigger problem and only maintain that much state.

In the next lesson, you will solve your first coding challenge using bottom-up dynamic programming.