Dynamic Programming
Fibonacci numbers are defined recursively as

\[
\begin{align*}
F(0) &= F(1) = 1 \\
F(n) &= F(n-1) + F(n-2)
\end{align*}
\]

You can turn this into a recursive implementation which runs in time \( T(n) = T(n-1) + T(n-2) = F(n) \) (which is exponential). Notice that this algorithm recomputes a lot of intermediate results (e.g. \( F(n) = F(n-1) + F(n-2) = F(n-2) + F(n-3) + F(n-2) \)). This suggests that we store the intermediate values \( F(i) \) and re-use them. Following this intuition, we can define an efficient recursive (“memoized”) implementation:

```python
1     Fib <- { array 1..n of results }
2     fib(n)
3         if n < 2
4             return 1
5         else
6             fib(n - 1)
7             Fib[n] <- Fib[n-1] + Fib[n-2]
8             return Fib[n]
```

Dynamic programming combines storing intermediate results with a second idea: instead of computing the subproblems top-down, we compute them bottom-up:

```python
1     fib(n)
2         Fib <- { array 1..n of results }
3         for i = 2 .. n
4             Fib[i] <- Fib[i-1] + Fib[i-2]
5         return Fib[n]
```