Chapter 7

Recurrences and Memoization: The Fibonacci Sequence

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

The Fibonacci sequence occurs frequently in nature and has closed form \( f(n) = f(n - 1) + f(n - 2) \), where \( f(1) = 1 \), \( f(0) = 1 \). This can be computed via a simple recursive implementation (Listing 7.1). On a large \( n \), the recursive Fibonacci can be quite slow or can either run out of RAM or surpass Python’s allowed recursion limit\(^1\). But if we wanted to know the precise runtime of our recursive Fibonacci, this is difficult to say. To the untrained eye, it may look like a \( 2^n \) runtime, because it bifurcates at every non-leaf in the recursion; however, this is not correct because the call tree of a \( 2^n \) algorithm corresponds to a perfect binary tree, while the call trees from our recursive Fibonacci will be significantly deeper in some areas.

An iterative approach (Listing 7.2) can re-use previous computations and improve efficiency, computing the \( n \)th Fibonacci number in \( O(n) \). This iterative strategy is a type of “dynamic programming”, a technique used to solve problems in a bottom-up fashion that reuses computations. Note that in calling \texttt{fib(100)} with our iterative method, it would only compute \texttt{fib(7)} a single time. In contrast, the naive recursive approach would compute \texttt{fib(7)} several times.

---

\(^1\)The Python equivalent of a “stack overflow” error.
Listing 7.1: Recursive Fibonacci.

def fib(n):
    if n==0 or n==1:
        return 1
    return fib(n-1) + fib(n-2)

Listing 7.2: Iterative Fibonacci.

def fib(n):
    last_result = 1
    result = 1
    for i in xrange(n-1):
        next_result = last_result + result
        last_result = result
        result = next_result
    return result

N=100
for i in xrange(N):
    print fib(i)

7.1 Memoization

“Memoization” is the top-down counterpart to dynamic programming: rather than a programmer deliberately designing the algorithm to reuse computations in a bottom-up manner, memoization performs recursive calls, but caches previously computed answers. Like dynamic programming, memoization prevents redundant computations. Listing 7.3 shows a memoized implementation of the Fibonacci sequence.

7.1.1 Graphical proof of runtime

To compute the runtime of the memoized variant, consider the call tree: As with the recursive version, fib(n) calls fib(n-1) (left subtree) and will

\footnote{For simplicity, assume an $O(1)$ dictionary lookup for our cache. In the case of Fibonacci, we could always use an array of length $n$ instead of a dictionary, should we need to.}
7.1. MEMOIZATION

Later call \( \text{fib}(n-2) \) (right subtree) \(^3\) \( \text{fib}(n-1) \) is called first, and that will call \( \text{fib}(n-2) \) (left subtree) and will later call \( \text{fib}(n-3) \) (right subtree). Proceeding in this manner, we can see that the base cases will be reached and then \( \text{fib}(2) \) will be computed and cached. That \( \text{fib}(2) \) was called by \( \text{fib}(3) \) (that \( \text{fib}(2) \) was the left subtree of its parent, and thus it was the \( \text{fib}(n-1) \) call, not the \( \text{fib}(n-2) \) call). \( \text{fib}(3) \) then calls \( \text{fib}(1) \) (which is the base case), to compute \( \text{fib}(3) \) and add it to the cache. That \( \text{fib}(3) \) was called by \( \text{fib}(4) \), which will also call \( \text{fib}(2) \); \( \text{fib}(2) \) is already in the cache. Note that every non-base case right subtree call will already be in the cache. Thus, when we draw the call tree, the right subtrees will all be leaves. The depth of the tree will be \( n - 1 \) because that is the distance traveled before \( n \) decreases to either base cases \((n = 1 \text{ or } n = 0)\) through calls that decrease \( n \) by 1 in each recursion. Thus the total number of nodes in the call tree will be \( \in \Theta(n) \) and the runtime of the memoized Fibonacci function will be \( \in \Theta(n) \).

7.1.2 Algebraic proof of runtime

Consider that each value can be added to the cache at most once, and since the work done in each of these recursive calls (an addition) costs \( O(1) \), then the runtime is \( \in O(n) \). Furthermore, \( \text{fib}(i) \) must be computed for \( i \in \{0, 1, 2, \ldots, n\} \), because we need \( \text{fib}(n-1) \) and \( \text{fib}(n-2) \) to compute \( \text{fib}(n) \). Hence, the runtime is \( \in \Omega(n) \). Thus we verify what we saw above: the runtime of the memoized Fibonacci function will be \( \in \Theta(n) \).

Listing 7.3: Memoized Fibonacci.

```python
# if no cache argument is given, start with an empty cache:
def fib(n, cache={}):
    if n==0 or n==1:
        return 1

    if n not in cache:
        cache[n] = fib(n-1,cache) + fib(n-2,cache)

    # i must be in cache now:
    return cache[n]
```

\(^3\)With the caveat that the memoized version also passes the cache as an additional parameter
CHAPTER 7. FIBONACCI

N=100
print 'with empty cache every time (slower):'
for i in xrange(N):
    print fib(i)
print

print 'keeping previous work (faster):'
cache={}
for i in xrange(N):
    print fib(i, cache)

Furthermore, we can reuse the cache in subsequent recursive calls. If we do this, it will ensure computing every Fibonacci number from 0 to n will cost $O(n)$ in total (regardless of the order in which we compute them). Equivalently, it means that if we compute every Fibonacci number from 0 to n (regardless of the order in which we compute them), the amortized runtime per call will be $\tilde{O}(1)$.

7.2 Recurrence closed forms and “eigendecomposition”

The question then arises: can we do better than our memoized version? Perhaps, but we need to step back and consider this problem from a mathematical perspective. First, let’s write the Fibonacci computation using linear algebra:

$$f(n) = f(n-1) + f(n-2)$$

$$= \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}.$$

This does nothing useful yet; it simply restates this in a standard mathematical form, revealing that multiplication with the vector $[1 \ 1]$ advances two neighboring Fibonacci numbers to compute the following Fibonacci number. But if we want to chain this rule together and use it iteratively, we have a problem: our function takes

$$\begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$
7.2. **RECURRENCE CLOSED FORMS**

a length-2 vector, as an input, but it produces a single numeric result. For this reason, as stated, the above mathematical formalism cannot be easily applied multiple times.

We can easily adapt this linear algebra formulation to produce a length-2 output: feeding in the previous two Fibonacci numbers $f(n-1)$ and $f(n-2)$ should yield the next pair $f(n)$ and $f(n-1)$:

$$
\begin{bmatrix}
f(n) \\
f(n-1)
\end{bmatrix} =
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
f(n-1) \\
f(n-2)
\end{bmatrix}.
$$

The “characteristic matrix” of the Fibonacci recurrence is

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$  

We can see that we start with base case values

$$\begin{bmatrix} f(1) \\
f(0) \end{bmatrix} = \begin{bmatrix} 1 \\
1 \end{bmatrix};$$

therefore, we can compute the $n^{th}$ via

$$A \cdot A \cdots A \cdot \begin{bmatrix} 1 \\
1 \end{bmatrix}$$

\[ \underbrace{A \cdot A \cdots A}_{n-1 \text{ terms}} \cdot \begin{bmatrix} 1 \\
1 \end{bmatrix} = A^{n-1} \cdot \begin{bmatrix} 1 \\
1 \end{bmatrix}. \]

This does not yet help us compute our Fibonacci values faster than $O(n)$, but we have no moved into a more theoretical domain where useful answers may exist.

Passing a vector through a square matrix is a bit like shooting an arrow into a storm: it may accelerate, decelerate, turn, or reverse (or some combination) as it goes through the storm, and a new arrow will be shot out. There are special directions where shooting the arrow in will only accelerate or decelerate it, but not turn it. These are called the “eigenvectors”\[4\]. Each eigenvector has a corresponding eigenvalue, the amount by which the vector is stretched after being shot through the storm. For example, an eigenvector

\[\text{From the German “eigen” meaning “self”}\]
v_1 with paired eigenvalue \( \lambda_1 \) will mean that \( A v_1 = \lambda_1 v_1 \), i.e., that shooting through our little storm stretched the vector by constant \( \lambda_1 \).

Since we have a two-dimensional problem, we need at most two eigenvectors to fully describe the space (they are like axes). For this reason, we can think of our initial

\[
\begin{bmatrix}
1 \\
1
\end{bmatrix}
\]

vector as being composed of some amounts from each of those two eigenvector ingredients. Thus, we have

\[
\begin{bmatrix}
\ f(n) \\
\ f(n-1)
\end{bmatrix} = A^{n-1} \cdot \begin{bmatrix}
1 \\
1
\end{bmatrix}
= A^{n-1} \cdot (c_1 v_1 + c_2 v_2)
= c_1 \lambda_1^{n-1} v_1 + c_2 \lambda_2^{n-1} v_2
f(n) = c_1 \lambda_1^{n-1} v_1[0] + c_2 \lambda_2^{n-1} v_2[0].
\]

From this we see that this would result in a two-term exponential sequence with four free parameters, \( d_1, \lambda_1, d_2, \) and \( \lambda_2 \) (where \( d_1 = c_1 \cdot v_1[0] \) and \( d_2 = c_2 \cdot v_2[0] \)). If we fit the free parameters that would produce \( f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 3 \), we find

\[
f(n) = \frac{1}{\sqrt{5}} \left( \left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1} \right)
= \frac{1}{\sqrt{5}} \left(1.618\ldots^{n+1} - (-0.618\ldots)^{n+1} \right)
\]

Interestingly, the larger of the eigenvalues, \( \frac{1+\sqrt{5}}{2} \) equals the golden ratio \( \phi \approx 1.618 \) (we can also compute these numerically via numpy, as shown in Listing 7.4). This is our Fibonacci closed form, which runs in \( \Theta(1) \) if we have access to an \( O(1) \) function for computing \( a^b \) for arbitrary \( a \) and \( b \). An implementation of this Fibonacci closed form can be seen in Listing 7.5.

Listing 7.4: Numerically computing the closed form from the characteristic matrix via numpy.

```python
import numpy

# Fibonacci characteristic matrix:
```
7.2. **RECURRENCE CLOSED FORMS**

\[ A = \text{numpy.matrix([[1,1],[1,0]])} \]

# Eigenvalues:
\[ \lambda_1, \lambda_2 = \text{numpy.linalg.eigvals}(A) \]

# Solve for coefficients d1,d2:
# \( z(t) = d_1 \lambda_1^{(t-1)} + d_2 \lambda_2^{(t-1)} \):
# \( \leftrightarrow M \cdot [d_1, d_2] = [1,1] \)
\[ M = \text{numpy.matrix([[\lambda_1^{-1}, \lambda_2^{-1}], [\lambda_1^0, \lambda_2^0]])} \]
\[ d = \text{numpy.linalg.solve}(M, [[1],[1]]) \]
\[ d_1, d_2 = d.T[0] \]

```python
def z(t):
    return d1*lambda1**(t-1) + d2*lambda2**(t-1)
```

for t in xrange(10):
    print z(t)

When we do not have access to an \( O(1) \) function for computing \( a^b \), we can compute our Fibonacci function in \( \Theta(\log(n)) \) using the recurrence

\[
a^b = \begin{cases} 
(\frac{a^{\frac{b}{2}}}{2})^2 & b \text{ is even} \\
\frac{a \cdot (a^{\frac{b-1}{2}})}{2} & \text{else}.
\end{cases}
\]

The recurrence for \( a^b \) is easily found by memoizing while computing \( a^b \).

Listing 7.5: Closed form of Fibonacci sequence in Python.

```python
import numpy as np

def fib(n):
    root_of_5 = np.sqrt(5)
    lambda_1=(1+root_of_5)/2.0
    lambda_2=(1-root_of_5)/2.0
    return 1/root_of_5 * ( lambda_1**(n+1) - lambda_2**(n+1) )

N=100
for i in xrange(N):
    print i, fib(i)
```
7.3 The runtime of the naive recursive Fibonacci

Each 1 added to the result can come only from a leaf (i.e., base cases) in the call tree. If we only include these leaves in the runtime (excluding any of the other nodes in the call tree or additions performed), we get

\[
\text{runtime} \geq \text{runtime from leaves} \\
\geq \text{number of leaves} \\
\in \Omega(\phi^n).
\]

For that reason we can see that the runtime of Fibonacci is bounded below: 
\[f(n) \in \Omega(\phi^n) \subset \Omega(1.618^n)\] (because \(\phi > 1.618\)).

7.4 Generic memoization

Connoisseurs of computer science will observe the above trajectory from recurrence to closed form and will ask, “Why don’t we just use the recurrence to identify which famous sequence this is, and then look up its closed form?” The answer to this is simple: what would we do for a sequence that isn’t famous?

Likewise, we should resist the urge to always insist on a closed form: we cannot always construct a closed form using known techniques. Indeed, there are recurrent functions that seem to always terminate, but where this has not yet been proven, in spite of real effort. For this reason, memoization can still be a surprisingly vital tool in its own right.

Listing 7.6 shows how we can make a generic “decorator” in Python, which will can be used to adapt new functions to memoized forms. For example, consider

\[
g(a, b) = \begin{cases} 
1 & \text{if } a \leq 0 \text{ or } b \leq 0 \\
g(a - 2, b + 1) + g(a + 1, b - 2) + g(a - 1, b - 1) & \text{else.}
\end{cases}
\]

Without computing a closed form, we can still compute this function without fear of recomputing the same values again and again (as we did with our initial naive Fibonacci implementation).

\[^5\text{See the “hailstone sequence”.}\]
Listing 7.6: Generic memoization using a “decorator” in Python.

```python
# class used as function "decorator":
class Memoized:
    def __init__(self, function):
        self._function = function
        self._cache = {}
    def __call__(self, *args):
        if args not in self._cache:
            # not in the cache: call the function and store the result in
            # the cache
            self._cache[args] = self._function(*args)

        # the result must now be in the cache:
        return self._cache[args]

@Memoized
def fib(n):
    if n==0 or n==1:
        return 1
    return fib(n-1) + fib(n-2)

print fib(10)

@Memoized
def g(a,b):
    if a<=0 or b<=0:
        return 1
    return g(a-2,b+1) + g(a+1,b-2) + g(a-1,b-1)

print g(10,3)
```