Potes enim videre in hac margine, qualiter hoc operati fuimus, scilicet quod iunximus primum numerum cum secundo, videlicet 1 cum 2; et secundum cum tercio; et tercium cum quarto; et quartum cum quinto, et sic deinceps. . . .

[You can see in the margin here how we have worked this; clearly, we combined the first number with the second, namely 1 with 2, and the second with the third, and the third with the fourth, and the fourth with the fifth, and so forth. . . .]

— Leonardo Pisano, *Liber Abaci* (1202)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás,
The Life of Reason, Book I: Introduction and Reason in Common Sense (1905)

You know what a learning experience is?
A learning experience is one of those things that says,
"You know that thing you just did? Don't do that."

— Douglas Adams, *The Salmon of Doubt* (2002)

3

Dynamic Programming

3.1 Mātrāvṛtta

One of the earliest examples of recursion arose in India more than 2000 years ago, in the study of poetic meter, or prosody. Classical Sanskrit poetry distinguishes between two types of syllables (*akṣara*): *light* (*laghu*) and *heavy* (*guru*). In one class of meters, variously called *mātrāvṛtta* or *mātrāchandas*, each line of poetry consists of a fixed number of "beats" (*mātrā*), where each light syllable lasts one beat and each heavy syllable lasts two beats. The formal study of *mātrā-vṛtta* dates back to the *Chandaḥśāstra*, written by the scholar Piṅgala between 600BCE and 200BCE. Piṅgala observed that there are exactly five 4-beat meters: ——, —••, •—•, ••—, and ••••. (Here each "—" represents a long syllable and each "•" represents a short syllable.)[1]

[1]In Morse code, a "dah" lasts three times as long as a "dit", but each "dit" or "dah" is followed by a pause with the same duration as a "dit". Thus, each "dit-pause" is a *laghu akṣara*, each

Although Piṅgala's text *hints* at a systematic rule for counting meters with a given number of beats,[2] it took about a millennium for that rule to be stated explicitly. In the 7th century CE, another Indian scholar named Virahāṇka wrote a commentary on Piṅgala's work, in which he observed that the number of meters with n beats is the sum of the number of meters with $(n-2)$ beats and the number of meters with $(n-1)$ beats. In more modern notation, Virahāṇka's observation implies a recurrence for the total number $M(n)$ of n-beat meters:

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

It is not hard to see that $M(0) = 1$ (there is only one empty meter) and $M(1) = 1$ (the only one-beat meter consists of a single short syllable).

The same recurrence reappeared in Europe about 500 years after Virahāṇka, in Leonardo of Pisa's 1202 treatise *Liber Abaci*, one of the most influential early European works on "algorism". In full compliance with Stigler's Law of Eponymy,[3] the modern *Fibonacci numbers* are defined using Virahāṇka's recurrence, but with different base cases:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

In particular, we have $M(n) = F_{n+1}$ for all n.

Backtracking Can Be Slow

The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them. Here is the same algorithm written in pseudocode:

"dah-pause" is a *guru akṣara*, and there are exactly five letters (M, D, R, U, and H) whose codes last four *mātrā*.

[2]The *Chandaḥśāstra* contains *two* systematic rules for listing all meters with a given number of *syllables*, which correspond roughly to writing numbers in binary from left to right (like Greeks) or from right to left (like Egyptians). The same text includes a recursive algorithm to compute 2^n (the number of meters with n syllables) by repeated squaring, and (arguably) a recursive algorithm to compute binomial coefficients (the number of meters with k short syllables and n syllables overall).

[3]*"No scientific discovery is named after its original discoverer."* In his 1980 paper that gives the law its name, the statistician Stephen Stigler jokingly claimed that this law was first proposed by sociologist Robert K. Merton. However, similar statements were previously made by Vladimir Arnol'd in the 1970's ("Discoveries are rarely attributed to the correct person."), Carl Boyer in 1968 ("Clio, the muse of history, often is fickle in attaching names to theorems!"), Alfred North Whitehead in 1917 ("Everything of importance has been said before by someone who did not discover it."), and even Stephen's father George Stigler in 1966 ("If we should ever encounter a case where a theory is named for the correct man, it will be noted."). We will see *many* other examples of Stigler's law in this book.

```
RecFibo($n$):
  if $n = 0$
      return 0
  else if $n = 1$
      return 1
  else
      return RecFibo($n-1$) + RecFibo($n-2$)
```

Unfortunately, this naive recursive algorithm is horribly slow. Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. Let $T(n)$ denote the number of recursive calls to RecFibo; this function satisfies the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1,$$

which looks an awful lot like the recurrence for Fibonacci numbers themselves! Writing out the first several values of $T(n)$ suggests the closed-form solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$, which we can verify by induction (hint, hint). So computing F_n using this algorithm takes about twice as long as just counting to F_n. Methods beyond the scope of this book[4] imply that $F_n = \Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803$ is the so-called *golden ratio*. In short, the running time of this recursive algorithm is exponential in n.

We can actually see this exponential growth directly as follows. Think of the recursion tree for RecFibo as a binary tree of additions, with only 0s and 1s at the leaves. Since the eventual output is F_n, exactly F_n of the leaves must have value 1; these leaves represent the calls to RecRibo(1). An easy inductive argument (hint, hint) implies that RecFibo(0) is called exactly F_{n-1} times. (If we just want an asymptotic bound, it's enough to observe that the number of calls to RecFibo(0) is at most the number of calls to RecFibo(1).) Thus, the recursion tree has exactly $F_n + F_{n-1} = F_{n+1} = O(F_n)$ leaves, and therefore, because it's a full binary tree, $2F_{n+1} - 1 = O(F_n)$ nodes altogether.

Memo(r)ization: Remember Everything

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecFibo(n) results in one recursive call to RecFibo($n-1$), two recursive calls to RecFibo($n-2$), three recursive calls to RecFibo($n-3$), five recursive calls to RecFibo($n-4$), and in general F_{k-1} recursive calls to RecFibo($n-k$) for any integer $0 \le k < n$. Each call is recomputing some Fibonacci number from scratch.

We can speed up our recursive algorithm considerably by writing down the results of our recursive calls and looking them up again if we need them later.

[4]See http://algorithms.wtf for notes on solving backtracking recurrences.

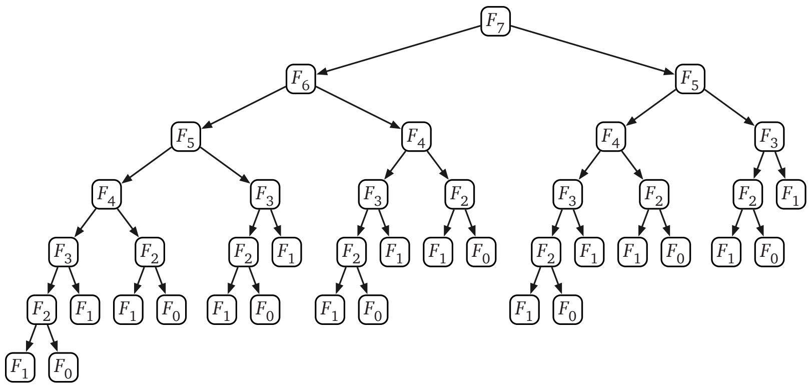

Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

This optimization technique, now known as *memoization* (yes, without an R), is usually credited to Donald Michie in 1967, but essentially the same technique was proposed in 1959 by Arthur Samuel.[5]

```
MemFibo(n):
  if $n = 0$
      return 0
  else if $n = 1$
      return 1
  else
      if $F[n]$ is undefined
          $F[n] \leftarrow$ MemFibo($n-1$) + MemFibo($n-2$)
      return $F[n]$
```

Memoization clearly decreases the running time of the algorithm, but by how much? If we actually trace through the recursive calls made by MemFibo, we find that the array $F[\,]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. This pattern can be verified by induction: Each entry $F[i]$ is filled only after its predecessor $F[i-1]$. If we ignore the time spent in recursive calls, it requires only constant time to evaluate the recurrence for each Fibonacci number F_i. But by design, the recurrence for F_i is evaluated only once for each index i. We conclude that MemFibo performs only $O(n)$ additions, an *exponential* improvement over the naïve recursive algorithm!

[5]Michie proposed that programming languages should support an abstraction he called a "memo function", consisting of both a standard function ("rule") and a dictionary ("rote"), instead of separately supporting arrays and functions. Whenever a memo function computes a function value for the first time, it "memorises" (yes, with an R) that value into its dictionary. Michie was inspired by Samuel's use of "rote learning" to speed up the recursive evaluation of checkers game trees; Michie describes his more general proposal as "enabling the programmer to 'Samuelize' any functions he pleases." (As far as I can tell, Michie never used the term "memoisation" himself.) Memoization was used even earlier by Claude Shannon's maze-solving robot "Theseus", which he designed and constructed in 1950.

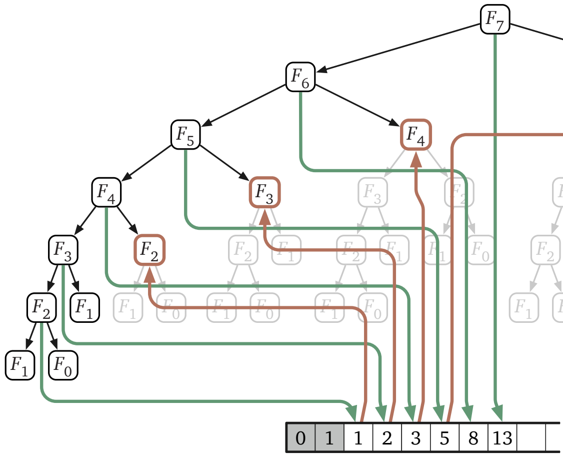

Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Dynamic Programming: Fill Deliberately

Once we see how the array $F[\,]$ is filled, we can replace the memoized recurrence with a simple for-loop that *intentionally* fills the array in that order, instead of relying on a more complicated recursive algorithm to do it for us accidentally.

```
ITERFIBO($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

Now the time analysis is immediate: IterFibo clearly uses ***O*(*n*) *additions*** and stores $O(n)$ *integers*.

This is our first explicit ***dynamic programming*** algorithm. The dynamic programming paradigm was formalized and popularized by Richard Bellman in the mid-1950s, while working at the RAND Corporation, although he was far from the first to use the technique. In particular, this iterative algorithm for Fibonacci numbers was already proposed by Virahāṇka and later Sanskrit prosodists in the 12th century, and again by Fibonacci at the turn of the 13th century![6]

[6]More general dynamic programming techniques were independently deployed several times in the late 1930s and early 1940s. For example, Pierre Massé used dynamic programming algorithms to optimize the operation of hydroelectric dams in France during the Vichy regime. John von Neumann and Oskar Morgenstern developed dynamic programming algorithms to determine the winner of any two-player game with perfect information (for example, checkers). Alan Turing and his cohorts used similar methods as part of their code-breaking efforts at