Recursion and Memoization

Memoization

Memoization is the idea of saving and reusing previously computed values of a function rather than recomputing them. To illustrate the idea, we consider the example of computing the Fibonacci numbers using a simple recursive program. The Fibonacci numbers are defined by specifying that the first two are equal to 1, and every successive one is the sum of the preceding two. Letting F_n denote the nth Fibonacci number, we have

$$F_0 = F_1 = 1$$

and for $n \geq 2$,

$$F_n = F_{n-1} + F_{n-2}.$$

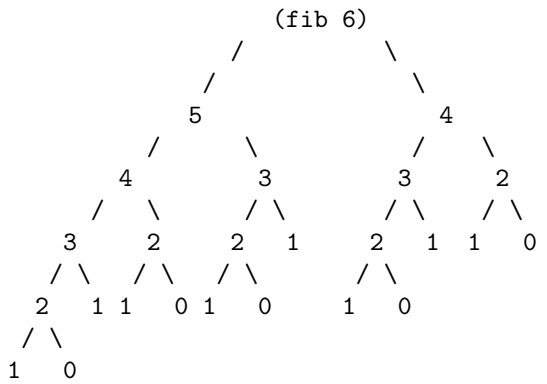
The sequence begins

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

We define a function `(fib n)` to return the value of F_n.

```
(define fib
  (lambda (n)
    (if (<= n 1)
        1
        (+ (fib (- n 1)) (fib (- n 2))))))
```

We diagram the tree of calls for `(fib 6)`, showing only the arguments for the calls.



The value of `(fib 6)` is 13, and there are 13 calls that return 1 from the base cases of argument 0 or 1. Because this is a binary tree, there is one fewer non-base case calls, for a total of 25 calls. In fact, to compute `(fib n)` there will be $2F_n - 1$ calls to the `fib` procedure.

How fast does F_n grow as a function of n? The answer is: exponentially. It is not doubling at every step, but it is more than doubling at every other step, because

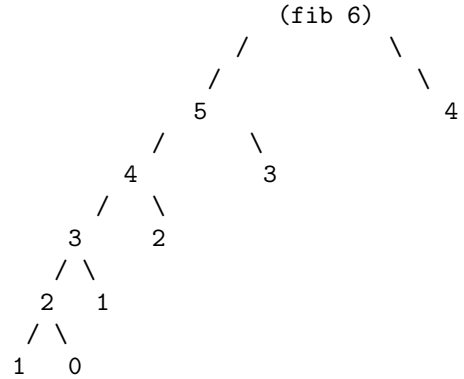
$$F_n = F_{n-1} + F_{n-2} > 2F_{n-2}.$$

It is not difficult to show that for all $n \geq 2$,

$$F_n \geq 2^{n/2} = (\sqrt{2})^n.$$

In fact, the growth rate of F_n is proportional to $((\sqrt{5}+1)/2)^n$, that is, the golden ratio (1.618..) raised to the power n.

There are a very large number, $2F_n - 1$, of calls to the `fib` function on a relatively small number of different arguments, namely the integers between 0 and n. This is a symptom of a situation in which memoization can help produce a more efficient solution. We keep track of previously computed values of the procedure, and use the stored values instead of recomputing them. If we do this for the above call tree, we get the following structure.



The second time we compute `(fib 2)`, we look it up instead of recomputing it, and similarly for the second time we need `(fib 3)` and `(fib 4)`. Thus, we have a total of 11 calls, or in general, $2n-1$ calls to compute `(fib n)`, an exponential improvement.

We implement this idea in Scheme by using a vector to store a table of previously computed values, indexed by the argument to the procedure. Note that this depends on the behavior of the `fib` procedure being *functional*, that is, purely determined by the value of its argument, without any internal state or randomization.

```
(define memo-fib
  (lambda (n) (let ((table (make-vector (+ n 1) #f))) (memo-fib-help n table))))

(define memo-fib-help
   (lambda (n table)
     (cond
       ((vector-ref table n) (vector-ref table n))
       ((<= n 1) 1)
       (else (begin (vector-set! table n
                                 (+ (memo-fib-help (- n 1) table)
                                    (memo-fib-help (- n 2) table)))
                    (vector-ref table n))))))
```

The top-level procedure creates a vector with indices 0 through n, initialized to `#f`, and then calls an auxiliary procedure. The auxiliary procedure checks to see if the function value for n is already in the table (any non `#f` value will test as true) and returns it if so. The second clause of the `cond` returns the correct value for the base cases of 0 and 1. Otherwise, the auxiliary procedure calls itself recursively on $n-1$ and $n-2$, places the result in the table, and returns it. Here is an example with the auxiliary procedure traced to see the structure of the calls.

```
> (memo-fib 5)
```

```
CALL memo-fib-help 5 #(#f #f #f #f #f #f)
 CALL memo-fib-help 4 #(#f #f #f #f #f #f)
  CALL memo-fib-help 3 #(#f #f #f #f #f #f)
   CALL memo-fib-help 2 #(#f #f #f #f #f #f)
    CALL memo-fib-help 1 #(#f #f #f #f #f #f)
    RETN memo-fib-help 1
    CALL memo-fib-help 0 #(#f #f #f #f #f #f)
    RETN memo-fib-help 1
   RETN memo-fib-help 2
   CALL memo-fib-help 1 #(#f #f 2 #f #f #f)
   RETN memo-fib-help 1
  RETN memo-fib-help 3
  CALL memo-fib-help 2 #(#f #f 2 3 #f #f)
  RETN memo-fib-help 2
 RETN memo-fib-help 5
 CALL memo-fib-help 3 #(#f #f 2 3 5 #f)
 RETN memo-fib-help 3
RETN memo-fib-help 8
8
```

Note that this implementation returns values for arguments 0 and 1 without placing them in the table.

This is not a great way to compute the nth Fibonacci number. A simple constant-space iterative method works as follows: set variables X and Y to 1 and 1 and then repeatedly set Z to $X + Y$, Y to Z and X to Y. A straightforward Scheme implementation of this idea might be the following.

```
(define fib-better
  (lambda (n)
    (fib-better-help n 1 1)))

(define fib-better-help
  (lambda (n x y)
     (if (<= n 0) x (fib-better-help (- n 1) y (+ x y)))))
```

The following shows a call with the auxiliary procedure traced.

```
> (fib-better 4)
CALL fib-better-help 4 1 1
 CALL fib-better-help 3 1 2
  CALL fib-better-help 2 2 3
   CALL fib-better-help 1 3 5
    CALL fib-better-help 0 5 8
    RETN fib-better-help 5
   RETN fib-better-help 5
  RETN fib-better-help 5
 RETN fib-better-help 5
RETN fib-better-help 5
5
```

There is an even more efficient method of computing the nth Fibonacci number, which we do not cover here. However, the point of the memoized version of the Fibonacci procedure is to illustrate the memoization of

a simple function, not to produce a good implementation of Fibonacci! We shall consider the problem of parsing a context free grammar as a nontrivial example of a recursive algorithm that can be memoized, leading to a dynamic programming algorithm for parsing.