Chapter 3

Sequences

In Chapter 2 we played with the sequence of regular n-gon areas $\{a_n\}$ where

$$A_n = \frac{n}{2}\sin\left(\frac{2\pi}{n}\right).$$

We numerically "discovered" that

$$\lim_{n\to\infty} A_n = \pi\ ,$$

a fact that is consistent with our geometric intuition.

In this chapter we build our "n-th term expertise" by exploring sequences that are specified in various ways. At the top of our agenda are sequences of sums like

$$S_n = 1 + \frac{1}{4} + \frac{1}{9} + \cdots + \frac{1}{n^2}.$$

Many important functions can be approximated by very simple summations, e.g.,

$$\exp(x) \approx 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}.$$

The quality of the approximation depends upon the value of x and the integer n.

Sometimes a sequence is defined *recursively*. The n-term may be specified as a function of previous terms, e.g.,

$$f_n = \begin{cases} 1 & \text{if } n = 1 \text{ or } 2 \\ f_{n-1} + f_{n-2} & \text{if } n \geq 3 \end{cases}$$

Sequence problems of this variety give us the opportunity to practice some difficult formula-to-program transitions.

3.1 Summation

The summation of a sequence of "regularly scheduled" numbers is such a common enterprise that a special notation is used. This is the "sigma" notation and here is an example:

$$\sum_{k=0}^{n} \frac{x^k}{k!} \;=\; 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}.$$

The value of many regular summations is known. For example, the sum of the first n positive integers is given by

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}. \tag{3.1.1}$$

It is interesting to write programs that check rules like this. The program `Example4_1` confirms that the sum of the first n integers is given by the above rule for $n = 1$ to 20.

```
% Example3_1:  Check the rule for the summation of the first n integers

nmax= 20;     % The no.  of n-values used in the rule checking
s= 0;         % The sum of the sequence so far

% Print the column headings
fprintf('\n');
fprintf('\n n \t Sum \t n(n+1)/2 \n');
fprintf('---------------------------\n');

% Compute summation of first n integers
for n= 1:nmax
   s= s+n;
   rhs= n*(n+1)/2;
   fprintf('%3d \t %3d \t %8d \n', n, s, rhs);
end
```

Output:

```
 n   Sum   n(n+1)/2
----------------
  1    1          1
  2    3          3
  3    6          6
           ⋮
 19  190        190
 20  210        210
```

The program uses a `for`-loop to actually compute the summation. The result is then printed side-by-side with the value of the summation formula. The program is *not* a proof of Equation

(3.1.1); it merely checks its correctness for a small set of possible n.

Problem 3.1. Modify `Example3_1` so that it confirms the following for $n = 1..20$:

$$\sum_{k=1}^{n} k^2 \;=\; \frac{n(n+1)(2n+1)}{6}\,.$$

Problem 3.2. It is possible to find real numbers $a, b, c, d,$ and e so that

$$\sum_{k=1}^{n} k^3 \;=\; an^4 + bn^3 + cn^2 + dn + e$$

for all n. Note that if n is large enough, then

$$\sum_{k=1}^{n} k^3 \;\approx\; an^4.$$

By dividing both sides by n^4 and assuming that n is large, we see that

$$a \;\approx\; \left(\sum_{k=1}^{n} k^3\right)/n^4$$

Write a program that estimates a using this approximation for $n = 1, \ldots, 50$.

Problem 3.3. If $r \neq 1$, then it can be shown that

$$\sum_{k=0}^{n} r^k \;=\; \frac{1 - r^{n+1}}{1 - r}$$

Modify `Example3_1` so that it confirms this for $n = 1, \ldots, 20$. Design the modification so that the value of r is obtained as input.

Problem 3.4. Modify `Example3_1` so that it confirms the following summation for $n = 1..12$:

$$(1 + 2 + \cdots + n)^2 = 1^3 + 2^3 + \cdots + n^3$$

for $n = 1, \ldots, 20$.

Given a real number x, define S_n to be the summation

$$S_n \;=\; \sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$$

i.e.,

$$S_n = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

The general term

$$a_k = (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

for this summation looks formidable. It appears that each term requires (a) an exponentiation of -1, (b) an exponentiation of x, and (c) a factorial. But with a little thought, it is clear that we do not have to start these computations "from scratch" with the generation of each new term. For example, if

$$a_6 = \frac{x^{13}}{13!}$$

is available, then we can compute a_7 from the formula

$$a_7 = -\frac{x^{15}}{15!} = -\frac{x^2 x^{13}}{15 \cdot 14 \cdot 13!} = -a_6 \frac{x^2}{15 \cdot 14}$$

In general, since $x^{2k+1} = x^2 \cdot x^{2k-1}$ and

$$(2k+1)! = (2k+1)(2k)(2k-1)!,$$

we have

$$\frac{a_k}{a_{k-1}} = (-1)^k \frac{x^{2k+1}}{(2k+1)!} \Big/ (-1)^{k-1} \frac{x^{2k-1}}{(2k-1)!} = \frac{-x^2}{(2k+1)(2k)}$$

and so

$$a_k = \frac{-x^2}{(2k+1)(2k)} a_{k-1}.$$

Given this recursion, here is a fragment that assigns to `s` the value of S_n assuming that `n` and `x` are initialized:

```
s= x;
numerator= -x*x;
a= x;   % {a(0)}
for k= 1:n
   a= a * (numerator / (2*k*(2*k+1)));   % {a(k)}
   s= s + a;
end
```

The key idea is to develop a *recursion* that relates the current term a_k to its predecessor a_{k-1}. One can write a mathematically equivalent fragment that computes the term $a_k = (-1)^k \frac{x^{2k+1}}{(2k+1)!}$ independently for each k. In that case, we calculate the components $(-1)^k$, x^{2k+1}, and $(2k+1)!$ from scratch for each k, so the program has the following structure:

```
s= 0;
for k= 0:n
   ⟨Task 1: calculate $(-1)^k$, store in variable minusOnePower⟩
   ⟨Task 2: calculate $x^{2k+1}$, store in variable xPower⟩
   ⟨Task 3: calculate $(2k+1)$, store in variable factorialTerm⟩
   s= s + minusOnePower*xPower/factorialTerm;
end
```

Task 1, evaluating $(-1)^k$, can be accomplished simply by using a conditional statement:

```
% Task 1:
if  (mod(k,2)==1)
   minusOnePower= -1;
else
   minusOnePower= 1;
end
```

To calculate the second component x^{2k+1}, one can use Matlab's power operator ^ or multiply x by itself a number of times. We'll use a `for`-loop:

```
% Task 2:
xPower= 1;
for j= 1:2*k+1
   xPower= x*xPower;
end
```

This loop deals with the evaluation of a power of x only, using a new index variable j. Looking back at the outline for calculating S_n, we see that this fragment for Task 2, which involves a loop, will be put "inside" another loop. We are not concerned by this since we have "itemized" the individual tasks in the outline above and we are now simply "filling in the detail" for one item. We will look at how these loops execute after we write the code for Task 3 for computing a factorial:

```
% Task 3:
factorialTerm= 1;
for i= 1:2*k+1
   factorialTerm= i*factorialTerm;
end
```

Again, we need a loop for calculating the factorial. Notice, however, that the loop index variables `j` and `i` for Tasks 2 and 3, respectively, take on the *same* values as the loop bodies are executed. This tells us that we can use one loop for Tasks 2 and 3 together. Suppose we choose to use the index variable `j`, then we have

```
% Tasks 2 and 3:
xPower= 1;
factorialTerm= 1;
for j= 1:2*k+1
   xPower= x*xPower;
   factorialTerm= j*factorialTerm;
end
```

Putting all the fragments for Tasks 1, 2, and 3 into the "outline" for calculating S_n, we have

```
s= 0;
for k= 0:n
   % Task 1:
      if  (mod(k,2)==1)
         minusOnePower= -1;
      else
         minusOnePower= 1;
      end
   % Tasks 2 and 3:
      xPower= 1;
      factorialTerm= 1;
```

```
        for j= 1:2*k+1
            xPower= x*xPower;
            factorialTerm= j*factorialTerm;
        end
    s= s + minusOnePower*xPower/factorialTerm;
end
```

This fragment contains *nested loops*, i.e., one loop is inside another. We will refer to the loop on the outside, with index `k`, as the "outer loop" and the loop with index `j` as the "inner loop." How do the loops execute? In exactly the same way as we have discussed in Chapter 2! Looking at the "big picture" first, we see that index `k` takes on its first value, 0, then the outer loop's body is executed. Then `k` takes on its next value, 1, and execute the outer loop's body. This goes on until `k` takes on its final value, `n`, and execute the outer loop's body a final time. Whenever we say "execute the outer loop's body," we can replace that phrase with the detail below, keeping in mind that `k` holds a single value at the moment:

- Task 1 is executed.
- Tasks 2 and 3 begins execution.
- Index j takes on its first value, 1.
- The inner loop's body is executed
- Index j takes on its next value, 2.
- The inner loop's body is executed

  ⋮
- Index j takes on its last value, $2k + 1$
- The inner loop's body is executed
- Tasks 2 and 3 have been completed. Update the value of `s` (last statement inside the outer loop).

If you analyze the values of the the two index variables `k` and `j` throughout the execution of the fragment, you will see the following "time line":

Time	$\longrightarrow$				
k	0	1 1 1	2 2 2 2 2	...	n $\cdots$ n
j	1	1 2 3	1 2 3 4 5		1 2 $\cdots$ $2n$ $2n+1$

The nested loops may appear complex when you look at the entire program, but in fact we never had to worry about the nesting when we *developed* the program. We first *itemized* the tasks within the loop with index `k`. At that time, there was only one loop, the `k`-loop. When we filled in the detail for items 2 and 3, we created a second loop, the `j`-loop. Since items 2 and 3 were itemized inside the `k`-loop, the entire `j`-loop went inside the `k`-loop. Therefore, the nested structure is a simple *consequence* of our systematic, "top-down" approach to developing the program rather than a program feature that you have to worry about at the outset. Of course, as you develop more programs you will begin to "foresee" nested structures when they are needed. For right now, however, concentrate on *decomposing a problem* into an itemized list of individual tasks to be accomplished. Then the nested structures will simply “reveal” themselves as you “fill in the detail” for the individual tasks.

We have shown now two approaches to calculating the summation S_n: (1) determine (by hand) the recursion that relates the term a_k to a_{k-1} and then writing a simple, single-loop program, and (2) write a program that calculates the terms independently “from scratch” for each k. Although we shall study the issue of efficiency more formally later on, it is obvious that the “from scratch” approach[1] is inferior in terms of the amount of required computation.

Now let us consider n for the summation S_n. It is known that for large n, $\sin(x) \approx S_n$. Thus, instead of building the summation for a fixed n, we may wish to continue the process of adding in terms until $|S_n - \sin(x)|$ is small. e.g.,

```
s= x;
numerator= -x*x;
a= x;   % {a(0)}
sinx= sin(x);
k= 0;
while (  abs(s-sinx) > 0.0001 )
   k= k+1;  a= a * (numerator / (2*k*(2*k+1)));   % {a(k)}
   s= s + a;
end
```

A danger with this kind of exploration is that it may take an inordinate number of iterations before the termination criterion is satisfied. To guard against this possibility it is advisable to “put a lid” on the maximum number of steps. All we need to do is change the `while` statement to include a second criterion:

```
while (  abs(s-sinx) > 0.0001 && (k < 20) )
```

Unlike the above example, it is usually the case that the limiting value of a summation is unknown and the termination criteria cannot be based upon the proximity to the limit. In this case, a reasonable course of action is to terminate when the value of the term about to be added is small, e.g.,

```
while (  abs(a) > 0.0001 && (k < 20) )
```

It is sometimes preferable to terminate when the current term is small *relative* to the current sum, e.g.,

```
while (  abs(a) > 0.0001*abs(s) && (k < 20) )
```

Taking this last approach to termination, program `Example3_2` provides an interactive framework that enables us to examine S_n as an approximation to $\sin(x)$. Notice the nested `while`-loops. The outer loop guard is part of the “interactive framework” that we developed in Chapter 2.

[1]The “from scratch” approach can take a very compact form by making use of MATLAB’s power operator `^` and built-in function `factorial`. In that case there will be one loop only and the loop body consists of the single statement `s= s + (-1)^k * x^(2*k+1) / factorial(2*k+1)`. Although this *appears* compact, it involves

```
% Example3_2:  Interactive exploration of the series for sin(x)

anotherEg= 'y';
% Loop until the user quits the program
while ( anotherEg ~= 'n')

   x= input('Enter x:  ');
   fprintf(' n \t Approximation \t Error\n');
   fprintf('--------------------------------\n');
   % Compute the sine series
   s= x;   % Current sum
   a= x;   % ratio between a_k+1, a_k
   k= 0;
   while (  abs(a) > .0001*abs(s) && k<20 )
      k= k + 1;
      a= a * ((-x*x)/(2*k*(2*k+1)));
      s= s + a;
      fprintf('%3d \t %8.5f \t %8.5f \n', k, s, abs(s-sin(x)));
   end

   anotherEg = input('\nAnother example (y/n)?  ','s');
end.
```

Sample output:

```
Enter x:  1

   n    Approximation   Error
 ---------------------------------
   1    0.83333         0.00814
   2    0.84167         0.00020
   3    0.84147         0.00000
   4    0.84147         0.00000

Another example (y/n)?  n
```

Within this interactive framework, we put the code for the sine series, which consists of its own loop. In developing the code, you wouldn't worry about both loops at the same time. Instead, first put down the "big picture":

```
anotherEg= 'y';
% Loop until the user quits the program
while ( anotherEg ~= 'n')

   % Compute the sine series

   anotherEg = input('\nAnother example (y/n)?  ','s');
end
```

The `while`-loop of the interactive framework is simple, so we put that down first and then just *name* the task yet to be accomplished using a comment—do not worry about all the details of the problem all at once. At this stage, you can test your program-in-progress by running it. After you get a working interactive framework, *then* you start writing the code for the task "compute the sine series." This code involves a loop, but that is not a problem at all, as you will deal with just one single loop for the series calculation and you can ignore the already completed outer loop. *Decomposing* the problem into these two main tasks simplifies your work since each task is smaller than the original problem.

PROBLEM 3.5. For large n,

$$R_n = 1 - \frac{1}{3} + \cdots - \frac{(-1)^{n+1}}{2n-1} = \sum_{k=1}^{n} \frac{(-1)^{k+1}}{2k-1} \approx \frac{\pi}{4}$$

$$T_n = 1 + \frac{1}{2^2} + \cdots + \frac{1}{n^2} = \sum_{k=1}^{n} \frac{1}{k^2} \approx \frac{\pi^2}{6}$$

$$U_n = 1 + \frac{1}{2^4} + \cdots + \frac{1}{n^4} = \sum_{k=1}^{n} \frac{1}{n^4} \approx \frac{\pi^4}{90}$$

Write a single program (with three loops) that computes and prints the smallest n so that

$$\left| R_n - \frac{\pi}{4} \right| \le 0.001,$$

the smallest n so that

$$\left| T_n - \frac{\pi^2}{6} \right| \le 0.001,$$

and the smallest n so that

$$\left| U_n - \frac{\pi^4}{90} \right| \le 0.001.$$

just about the same amount of computation as the "from scratch" approach given in the main text because each component of each term is still computed independently without taking advantage of the computation done for the previous term. *If computational efficiency is not a concern*, then this compact form does have one advantage: the general formula is stated explicitly in the program.

PROBLEM 3.6. Each of the following sequences converge to π:

$$\begin{aligned} a_n &= \frac{6}{\sqrt{3}}\sum_{k=0}^{n}\frac{(-1)^k}{3^k(2k+1)} \\ b_n &= 16\sum_{k=0}^{n}\frac{(-1)^k}{5^{2k+1}(2k+1)} - 4\sum_{k=0}^{n}\frac{(-1)^k}{239^{2k+1}(2k+1)} \end{aligned}$$

Write a single program that prints $a_0, \ldots, a_n$ where n is the smallest integer so $|a_n - \pi| \le .000001$ and prints $b_0, \ldots, b_n$ where n is the smallest integer so $|b_n - \pi| \le .000001$.

PROBLEM 3.7. For all positive n, define

$$a_n = \sum_{j=1}^{n^2}\frac{n}{n^2+j^2}\ .$$

Write a program that prints $a_2, \ldots, a_n$ where n is the smallest integer such that $|a_{n-1} - a_n| \le .01$. Hint. Structure your solution as follows:

```
⟨Compute $a_1$ and $a_2$.⟩
n= 2;
while ( $|a_{n-1} - a_n| > 0.01$)
    n= n+1;
    ⟨Compute $a_n$⟩
end
```

The computation of a_n requires a loop itself and so this is a nested-loop problem.

PROBLEM 3.8. Explore the following approximations by modifying `Example3_2`. Warning: some of the approximations deteriorate very rapidly as $|x|$ gets large.

$$\begin{aligned} \textbf{(a)}\ \cos(x) &\approx \sum_{j=0}^{n}\frac{(-x^2)^j}{(2j)!} \\ \textbf{(b)}\ \csc(x) = 1/\sin(x) &\approx \frac{1}{x} + 2x\sum_{j=1}^{n}\frac{(-1)^j}{x^2 - k^2\pi^2} \\ \textbf{(c)}\ \sinh(x) = \frac{e^x - e^{-x}}{2} &\approx \sum_{j=0}^{n}\frac{x^{2j+1}}{(2j+1)!} \\ \textbf{(d)}\ \cosh(x) = \frac{e^x + e^{-x}}{2} &\approx \sum_{j=0}^{n}\frac{x^{2j}}{(2j)!} \\ \textbf{(e)}\ \exp(x) &\approx \sum_{j=0}^{n}\frac{x^k}{k!} \end{aligned}$$

PROBLEM 3.9. Write a program that verifies the inequalities

$$\frac{2}{3}n\sqrt{n} \le \sum_{k=1}^{n}\sqrt{k} \le \frac{4n+3}{6}\sqrt{n}$$

for $n = 1, \ldots, 100$.

Problem 3.10. Define

$$E_n = \left(\sum_{k=1}^{n} \frac{1}{k} \right) - \ln(n)$$

It is known that E_n converges to the *Euler constant* for large n. Write a program that prints E_{100k} for $k = 1, \ldots, 100$.

Analogous to the summation problem is the "product problem." Consider the following sequence:

$$P_0 = 2, \quad P_1 = 2\left(\frac{2}{1}\frac{2}{3}\right), \quad P_2 = 2\left(\frac{2}{1}\frac{2}{3}\right)\left(\frac{4}{3}\frac{4}{5}\right), \quad P_3 = 2\left(\frac{2}{1}\frac{2}{3}\right)\left(\frac{4}{3}\frac{4}{5}\right)\left(\frac{6}{5}\frac{6}{7}\right), \text{etc.}$$

In general,

$$P_k = P_{k-1}\frac{2k}{2k-1}\frac{2k}{2k+1} = P_{k-1}\frac{4k^2}{4k^2-1}$$

and we have

```
product= 2;
for k= 1:n
   factor= 4*k*k/(4*k*k -1);
   product= product*factor;
end
```

In order to specify products succinctly, there is a notation analogous to the Σ-notation. If $a_0, a_1, \ldots$ then

$$P_k = \prod_{j=0}^{k} a_j \; = \; a_0 a_1 a_2 \cdots a_k.$$

Thus, in the above example, $a_0 = 2$ and $a_k = 4k^2/(4k^2 - 1)$ for $k \geq 1$.

Problem 3.11. Using the interactive framework, explore the quality of the approximation

$$\sin(x) \;\approx\; x \prod_{j=1}^{n} \left(1 - \frac{x^2}{j^2\pi^2}\right)$$

Use $n = 50$ and print all the partial products and their errors.

Problem 3.12. Numerically determine the value of

$$P_n = \prod_{k=2}^{n} \frac{k^3 - 1}{k^3 + 1}$$

as n gets large.

3.2 Recursions

The simplest way that a sequence $\{a_n\}$ can be specified is with an explicit recipe for each term, e.g., $a_n = 2^{-n}$. Sometimes a sequence is defined by giving the first term and then a rule for all the successors:

$$a_n = \begin{cases} 1 & \text{if } n = 0 \\ \\ n \cdot a_{n-1} & \text{if } n \geq 1 \end{cases} .$$

This is an example of a *one-term* recurrence and we see that

$$\begin{array}{lll} a_1 & = 1 \cdot a_0 & = 1 \\ a_2 & = 2 \cdot a_1 & = 2 \\ a_3 & = 3 \cdot a_2 & = 6 \\ a_4 & = 4 \cdot a_3 & = 24 \end{array}$$

The fragment

```
a= 1;
for n= 1:4
   a= a*n;
   fprintf('%d \t %d \n', n, a)
end
```

produces a short table with the same values. It is not hard to see that values of the factorial function are being reported:

$$a_n = n! = 1 \cdot 2 \cdot 3 \cdots n$$

The n-th term for this particular one-term recursion can be specified explicitly, but this typically is not the case. An interesting example that does not permit the "closed formula" expression for the general term is the "up and down" sequence:

$$a_n \;=\; \begin{cases} \text{any positive integer} & \text{if } n = 1 \\ \\ a_{n-1}/2 & \text{if } n > 1 \text{ and } a_{n-1} \text{ is even} \\ \\ 3a_{n-1} + 1 & \text{if } n > 1 \text{ and } a_{n-1} \text{ is odd} \end{cases}$$

Thus, if $a_1 = 17$, then the sequence

$$17,\ 52,\ 26,\ 13,\ 40,\ 20,\ 10,\ 5\ 16,\ 8,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1, \ldots$$

is produced. Notice that once the number one "is reached", the cycle 1,4,2,1,4,2,.. begins. It is known that the up and down sequence always reaches one no matter what the choice of a_1 is. There is no simple, explicit recipe for a_n as in the case for the factorial sequence.

Let $f(m)$ designate the smallest integer so $a_{f(m)} = 1$ given that $a_1 = m$. From the above example we see that $f(17) = 13$. We may also conclude from that same example that $f(1) = 1$, $f(4) = 3$, and $f(52) = 12$. If `m` houses the starting integer m, then the fragment

```
a= m;
k= 1;
while (  a ~= 1 )
   % {a = a(k)}
   if  ( mod(a,2) == 0 )
      a= a/2;
   else
      a= 3*a + 1;
   end
   k= k + 1;
end
fprintf('f(m) = %d \n', k)
```

prints the value of $f(m)$.

Now let's augment this fragment so that it also prints the largest value encountered along the up-and-down route from m to 1. This is the first of many look-for-the-max problems that we shall encounter. It requires the maintenance of a variable whose mission is to keep track of the largest integer encountered "so far." The program `Example3_3` presents the details. Notice that `aMax` is initially set to m, the starting value. Each time a new a-value is generated, it is compared to the value of `aMax`. The variable `aMax` always houses the largest a-value that has arisen since the loop started. More precisely,

$$\texttt{aMax} = \max\{a_1, \ldots, a_k\}$$

If `a > aMax` is true, then a new largest value has been encountered and `aMax` is revised accordingly.

PROBLEM 3.13. Modify `Example3_3` so that for starting values $m = 1, 2, \ldots, 100$, it prints m, $f(m)$, and the associated max value.

The *Fibonacci sequence* gives us an opportunity to see what an iteration looks like that is based upon a *two-term recurrence.* Here is a table of the first 8 Fibonacci numbers $f_1, \ldots, f_8$:

k	1	2	3	4	5	6	7	8
f_k	1	1	2	3	5	8	13	21

The pattern should be clear. Once we "get going," each Fibonacci number is the sum of its two predecessors:

$$f_k = \begin{cases} 1 & \text{if } k = 1 \\ 1 & \text{if } k = 2 \\ f_{k-1} + f_{k-2} & \text{if } k > 2 \end{cases}$$

This is an example of a *two-term recurrence* and a loop that generates such a sequence requires the maintenance of two variables. One variable (call it `current`) is needed for the current Fibonacci number and another (call it `old`) is needed for its predecessor. The triplet

```
% Example3_3:  The Up-and-Down Sequence

fprintf('\n m \t f(m) \t max \n')
fprintf('---------------------\n')

for m= 50:60
   a= m;     % a = a(k)
   k= 1;
   aMax= m;  % Max value of a so far
   while (  a ~= 1 )
      if  ( mod(a,2) == 0 )
         a= a/2;
      else
         a= 3*a + 1;
      end
      if  (a > aMax)
         aMax= a;
      end
      k= k + 1;
   end
fprintf('%3d \t %4d \t %4d\n', m, k, aMax);
end
```

Output:

```
 m   f(m)       max
-------------------
50     25        88
51     25       232
52     12        52
53     12       160
54    113      9232
55    113      9232
56     20        56
57     33       196
58     20        88
59     33       304
60     20       160
```

```
new=current+old; old=current; current=new;
```

updates these two variables so that they respectively house the next Fibonacci number and *its* predecessor. FIGURE 3.1 depicts the changes these variables undergo assuming that initially `current = 5`, `old = 3`, and `new = 5`.

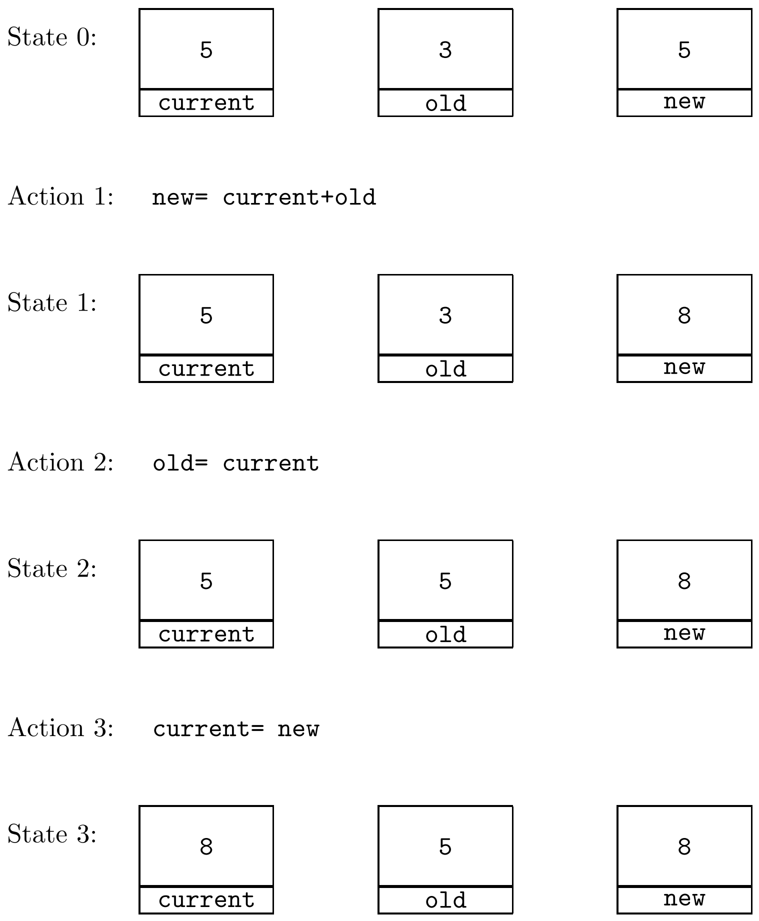

FIGURE 3.1 *The Fibonacci Update*

`Example3_4` puts the Fibonacci updates under the control of a `while`-loop and prints a list of all the Fibonacci numbers that are less than one million. Notice that the first two Fibonacci numbers are set up before the loop begins.

```
% Example3_4:  Print all Fibonacci numbers less than a given upper bound
current= 1; % f(k)
old= 1;     % f(k-1)
k= 2;
bound= 1000000;

fprintf('\n k \t f(k) \n');
fprintf('-----------------\n');
fprintf('%3d \t %8d \n', 1, old);
fprintf('%3d \t %8d \n', 2, current)

while ( current+old < bound)
   new= current+old; % f(k+1)
   old= current;
   current= new;
   k= k+1;
   fprintf('%3d \t %8d \n', k, current)
end
```

Output:

```
 k        f(k)
--------------
 1           1
 2           1
 3           2
 4           3
 5           5
 6           8
       ⋮
28      317811
29      514229
30      832040
```

PROBLEM 3.14. Modify `Example3_4` so that it reads in an integer $x > 1$ and prints Fibonacci numbers f_k and f_{k+1} where $f_k \le x < f_{k+1}$.

PROBLEM 3.15. Modify `Example3_4` so that it reads in a positive integer and prints a message that indicates whether or not it is a Fibonacci number.

PROBLEM 3.16. Define $r_k = f_k/f_{k-1}$, the ratio of f_k to its predecessor. Modify `Example3_4` so that it prints $r_3, \ldots, r_n$ where n is the smallest integer with the property that $|r_n - r_{n-1}| \le 0.000001$.

PROBLEM 3.17. Define

$$\begin{aligned}
t_0 &= \sqrt{1+0}\\
t_1 &= \sqrt{1+1}\\
t_2 &= \sqrt{1+2}\\
t_3 &= \sqrt{1+2\sqrt{1+3}}\\
t_4 &= \sqrt{1+2\sqrt{1+3\sqrt{1+4}}}\\
t_5 &= \sqrt{1+2\sqrt{1+3\sqrt{1+4\sqrt{1+5}}}}
\end{aligned}$$

Pick up the pattern and develop a program that prints $t_1, \ldots, t_{26}$. A loop is required for each t_k.

PROBLEM 3.18. Let m be a positive integer and consider the sequence

$$\begin{aligned}
t_1 &= \sqrt{m}\\
t_2 &= \sqrt{m-\sqrt{m}}\\
t_3 &= \sqrt{m-\sqrt{m+\sqrt{m}}}\\
t_4 &= \sqrt{m-\sqrt{m+\sqrt{m-\sqrt{m}}}}\\
t_5 &= \sqrt{m-\sqrt{m+\sqrt{m-\sqrt{m+\sqrt{m}}}}}
\end{aligned}$$

Pick up the pattern and write a program that helps you determine the limit of t_n as n gets large. Use the interactive framework, soliciting m and iterating until $|t_n - t_{n-1}| \le .0001$. For your information, the limit is an integer if m = 7, 13, 21, 31, or 43. A loop is required for each t_k.