CHAPTER 8

Loops

THE OBJECTIVES OF THIS CHAPTER ARE TO ENABLE YOU TO LEARN TO:

- Program (or code) determinate loops with `for`.
- Program (or code) indeterminate loops with `while`.

CONTENTS

In Chapter 2 we introduced the powerful `for` statement, which is used to repeat a block of statements a fixed number of times. This type of structure, where the number of repetitions must be determined in advance, is sometimes called *determinate repetition*. However, it often happens that the condition to end a loop is only satisfied *during the execution of the loop itself*. Such a structure is called *indeterminate*. We begin with a discussion of determinate repetition.

8.1 DETERMINATE REPETITION WITH `for`

8.1.1 Binomial coefficient

The *binomial coefficient* is widely used in mathematics and statistics. It is defined as the number of ways of choosing r objects out of n without regard to order, and is given by

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}. \tag{8.1}$$

If this form is used, the factorials can get very big, causing an overflow. But a little thought reveals we can simplify Equation (8.1) as follows:

$$\binom{n}{r} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{r!}, \tag{8.2}$$

$$\text{e.g., } \binom{10}{3} = \frac{10!}{3!\times 7!} = \frac{10\times 9\times 8}{1\times 2\times 3}.$$

Using Equation (8.2) is computationally much more efficient:

Essential MATLAB for Engineers and Scientists. DOI:10.1016/B978-0-08-100877-5.00009-8

```
ncr = 1;
n = ...
r = ...

for k = 1:r
  ncr = ncr * (n - k + 1) / k;
end

disp( ncr )
```

The binomial coefficient is sometimes pronounced 'n-see-r'. Work through the program by hand with some sample values.

8.1.2 Update processes

Many problems in science and engineering involve modeling a process where the main variable is repeatedly updated over a period of time. Here is an example of such an *update process*.

A can of orange juice at temperature 25 °C is placed in a fridge, where the ambient temperature F is 10 °C. We want to find out how the temperature of the orange juice changes over a period of time. A standard way of approaching this type of problem is to break the time period up into a number of small steps, each of length dt. If T_i is the temperature at the *beginning* of step i, we can use the following model to get T_{i+1} from T_i:

$$T_{i+1} = T_i - K\,dt\,(T_i - F), \tag{8.3}$$

where K is a constant parameter depending on the insulating properties of the can, and the thermal properties of orange juice. Assume that units are chosen so that time is in minutes.

The following script implements this scheme. To make the solution more general, take the starting time as a, and the end time as b. If dt is very small, it will be inconvenient to have output displayed after every step, so the script also asks you for the *output interval* opint. This is the time (in minutes) between successive rows of output. It checks that this interval is an integer multiple of dt. Try the script with some sample values, e.g., $dt = 0.2$ minutes and opint $= 5$ minutes (these are the values used for the output below).

```
K = 0.05;
F = 10;
a = 0;                      % start time
b = 100;                    % end time
time = a;                   % initialize time
T = 25;                     % initialize temperature
load train                  % prepare to blow the whistle
```

```
dt = input( 'dt: ' );
opint = input( 'output interval (minutes): ' );
if opint/dt ~= fix(opint/dt)
  sound(y, Fs)          % blow the whistle!
  disp( 'output interval is not a multiple of dt!' );
  break
end

clc
format bank
disp( '         Time          Temperature' );
disp( [time T] )      % display initial values

for time = a+dt : dt : b
  T = T - K * dt * (T - F);
  if abs(rem(time, opint)) < 1e-6    % practically zero!
    disp( [time T] )
  end
end
```

Output:

```
Time          Temperature
    0         25.00
 5.00         21.67
  ...
95.00         10.13
100.00        10.10
```

Note:

1. The function `rem` is used to display the results every `opint` minutes: when `time` is an integer multiple of `opint` its remainder when divided by `opint` should be zero. However, due to rounding error, the remainder is not always *exactly* zero. It is therefore better to test whether its absolute value is less than some very small value. (Rounding error is discussed in Chapter 11.)
2. While this is probably the most obvious way of writing the script, we cannot easily plot the graph of temperature against time this way, since `time` and `T` are scalars which are repeatedly updated. To plot the graph they both need to be vectors (see Chapter 9).
3. Note how sound is implemented. See `help audio` for other interesting sounds supplied by MATLAB.
4. In case you are wondering how I got the headings in the right place, I'll let you into the secret. Run the script without a heading but with the numerical output as you want it. Then simply paste the `disp` statement with the

headings into the command window and edit it until the headings fall in the right place. Paste the final version of the `disp` statement into the script.

5. Note the use of `break` to stop the script prematurely if the user gives bad input. See below for more general use of `break`.

8.1.3 Nested `for`s

As we saw in Chapter 2 (**Vectorizing nested `for`s**), `for` loops can be nested inside each other. The main point to note is that the index of the *inner* `for` moves faster.

8.2 INDETERMINATE REPETITION WITH `while`

Determinate loops all have in common the fact that you can work out in principle exactly how many repetitions are required *before the loop starts*. But in the next example, there is no way *in principle* of working out the number of repeats, so a different structure is needed.

8.2.1 A guessing game

The problem is easy to state. MATLAB 'thinks' of an integer between 1 and 10 (i.e., generates one at random). You have to guess it. If your guess is too high or too low, the script must say so. If your guess is correct, a message of congratulations must be displayed.

A little more thought is required here, so a structure plan might be helpful:

```
1. Generate random integer
2. Ask user for guess
3. While guess is wrong:
       If guess is too low
           Tell her it is too low
       Otherwise
           Tell her it is too high
       Ask user for new guess
4. Polite congratulations
5. Stop.
```

Here is the script:

```
matnum = floor(10 * rand + 1);
guess = input( 'Your guess please: ' );
load splat

while guess ~= matnum
sound(y, Fs)

  if guess > matnum
```

```
      disp( 'Too high' )
    else
      disp( 'Too low' )
    end;

    guess = input( 'Your next guess please: ' );
  end

  disp( 'At last!' )
  load handel
  sound(y, Fs)                  % hallelujah!
```

Try it out a few times. Note that the `while` loop repeats as long as `matnum` is not equal to `guess`. There is no way in principle of knowing how many loops will be needed before the user guesses correctly. The problem is truly indeterminate.

Note that `guess` has to be input in two places: firstly to get the `while` loop going, and secondly during the execution of the `while`.

8.2.2 The `while` statement

In general the `while` statement looks like this:

```
while condition
  statements
end
```

The `while` construct repeats *statements* WHILE its *condition* remains true. The condition therefore is the condition to repeat once again. The condition is tested each time BEFORE *statements* are repeated. Since the condition is evaluated before *statements* are executed, it is possible to arrange for *statements* not to be executed at all under certain circumstances. Clearly, *condition* must depend on *statements* in some way, otherwise the loop will never end.

Recall that a *vector* condition is considered true only if *all* its elements are non-zero.

The command-line form of `while` is:

```
while condition statements, end
```

8.2.3 Doubling time of an investment

Suppose we have invested some money which draws 10% interest per year, compounded. We would like to know how long it takes for the investment to double. More specifically, we want a statement of the account each year, *until* the balance has doubled. The English statement of the problem hints heavily that we should use an indeterminate loop with the following structure plan:

```
1. Initialize balance, year, interest rate
2. Display headings
3. Repeat
       Update balance according to interest rate
       Display year, balance
   until balance exceeds twice original balance
4. Stop.
```

A program to implement this plan would be

```
a = 1000;
r = 0.1;
bal = a;
year = 0;
disp( 'Year       Balance' )

while bal < 2 * a
  bal = bal + r * bal;
  year = year + 1;
  disp( [year bal] )
end
```

Note that the more natural phrase in the structure plan 'until balance exceeds twice original balance' must be coded as

```
while bal < 2 * a ...
```

This condition is checked each time before another loop is made. Repetition occurs only if the condition is true. Here's some output (for an opening balance of $1000):

```
Year       Balance

   1       1100.00
   2       1210.00
   3       1331.00
   4       1464.10
   5       1610.51
   6       1771.56
   7       1948.72
   8       2143.59
```

Note that when the last loop has been completed, the condition to repeat is false for the first time, since the new balance ($2143.59) is more than $2000.

Note also that a determinate for loop *cannot* be used here because we don't know how many loops are going to be needed until after the script has run (although in this particular example perhaps you *could* work out in advance how many repeats are needed?).

If you want to write the new balance only while it is *less* than $2000, all that has to be done is to move the statement

```
    disp( [year bal] )
```

until it is the first statement in the while loop. Note that the initial balance of $1000 is displayed now.

8.2.4 Prime numbers

Many people are obsessed with prime numbers, and most books on programming have to include an algorithm to test if a given number is prime. So here's mine.

A number is prime if it is not an exact multiple of any other number except itself and 1, i.e., if it has no factors except itself and 1. The easiest plan of attack then is as follows. Suppose P is the number to be tested. See if any numbers N can be found that divide into P without remainder. If there are none, P is prime. Which numbers N should we try? Well, we can speed things up by restricting P to odd numbers, so we only have to try odd divisors N. When do we stop testing? When $N = P$? No, we can stop a lot sooner. In fact, we can stop once N reaches $\sqrt{P}$, since if there is a factor greater than $\sqrt{P}$ there must be a corresponding one less than $\sqrt{P}$, which we would have found. And where do we start? Well, since $N = 1$ will be a factor of any P, we should start at $N = 3$. The structure plan is as follows:

```
1. Input $P$
2. Initialize $N$ to 3
3. Find remainder $R$ when $P$ is divided by $N$
4. While $R \neq 0$ and $N < \sqrt{P}$ repeat:
     Increase $N$ by 2
     Find $R$ when $P$ is divided by $N$
5. If $R \neq 0$ then
     $P$ is prime
   Else
     $P$ is not prime
6. Stop.
```

Note that there may be no repeats—R might be zero the first time. Note also that there are *two* conditions under which the loop may stop. Consequently, an if is required after completion of the loop to determine which condition stopped it.

See if you can write the script. Then try it out on the following numbers: 4 058 879 (not prime), 193 707 721 (prime) and 2 147 483 647 (prime). If such things interest you, the largest *known* prime number at the time of writing was $2^{6\,972\,593} - 1$ (discovered in June 1999). It has 2 098 960 digits and would occupy about 70 pages if it was printed in a newspaper. Obviously our algorithm cannot test such a large number, since it's unimaginably greater than the largest number which can be represented by MATLAB. Ways of testing such huge numbers for primality are described in D.E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms* (Addison-Wesley, 1981). This particular whopper was found by the GIMPS (Great Internet Mersenne Prime Search). See http://www.utm.edu/research/primes/largest.html for more information on the largest known primes.

8.2.5 Projectile trajectory

In Chapter 3 we considered the flight of a projectile, given the usual equations of motion (assuming no air resistance). We would like to know now when and where it will hit the ground. Although this problem can be solved with a determinate loop (if you know enough applied mathematics), it is of interest also to see how to solve it with an indeterminate `while` loop. The idea is to calculate the trajectory repeatedly with increasing time, *while* the vertical displacement (y) remains positive. Here's the script:

```
dt = 0.1;
g = 9.8;
u = 60;
ang = input( 'Launch angle in degrees: ' );
ang = ang * pi / 180;              % convert to radians
x = 0; y = 0; t = 0;               % for starters
more(15)

while y >= 0
  disp( [t x y] );
  t = t + dt;
  y = u * sin(ang) * t - g * t^2 / 2;
  x = u * cos(ang) * t;
end
```

The command `more(n)` gives you `n` lines of output before pausing. This is called *paging*. To get another single line of output press **Enter**. To get the next page of `n` lines press the spacebar. To quit the script press **q**.

Try the script for some different launch angles. Can you find the launch angle which gives the maximum horizontal range (x)? What launch angle keeps it in the air for the longest time?

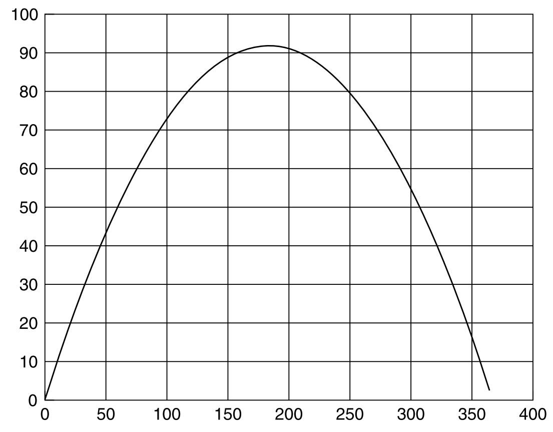

FIGURE 8.1 Projectile trajectory.

Note that when the loop finally ends, the value of y will be negative (check this by displaying y). However, the position of the disp statement ensures that only positive values of y are displayed. If for some reason you need to record the last value of t, say, before y becomes negative, you will need an if statement inside the while, e.g.,

```
if y >= 0
  tmax = t;
end
```

Change the script so that it displays the last time for which y was positive (tmax), *after* the while loop has ended.

Now suppose we want to plot the trajectory, as shown in Figure 8.1. Note in particular how the trajectory stops above the *x*-axis. We need to use vectors now. Here is the script:

```
dt = 0.1;
g = 9.8;
u = 60;
ang = input( 'Launch angle in degrees: ' );
ang = ang * pi / 180;              % convert to radians
xp = zeros(1); yp = zeros(1);      % initialize
y = 0; t = 0;
i = 1;                             % initial vector subscript

while y >= 0
  t = t + dt;
  i = i + 1;
  y = u * sin(ang) * t - g * t^2 / 2;
  if y >= 0
    xp(i) = u * cos(ang) * t;
    yp(i) = y;
  end
end

plot(xp, yp),grid
```

Note that the function `zeros` is used to initialize the vectors. This also clears any vector of the same name hanging around in the workspace from previous runs.

Note also the use of an `if` inside the `while` loop to ensure that only co-ordinates of points above the ground are added to the vectors `xp` and `yp`.

If you want the last point above the ground to be closer to the ground, try a smaller value of `dt`, e.g., 0.01.

8.2.6 `break` and `continue`

Any loop structure you are likely to encounter in scientific programming can be coded with either 'pure' `for` or `while` loops, as illustrated by the examples in this chapter. However, as a concession to intellectual laziness I feel obliged to mention in passing the `break` and `continue` statements.

If there are a number of different conditions to stop a `while` loop you may be tempted to use a `for` with the number of repetitions set to some accepted cut-off value (or even `Inf`) but enclosing `if` statements which `break` out of the `for` when the various conditions are met. Why is this not regarded as the best programming style? The reason is simply that when you read the code months later you will have to wade through the whole loop to find all the conditions to end it, rather than see them all paraded at the start of the loop in the `while` clause.

If you are going to insist on using `break` you will have to look it up in `help` for yourself!

The `continue` statement is somewhat less virulent than `break` ...

8.2.7 Menus

Try the following program, which sets up a menu window, as in Figure 8.2:

```
k = 0;

while k ~= 3
```

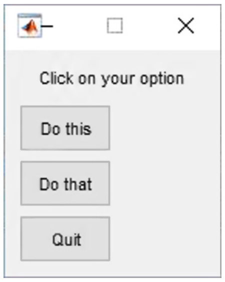

FIGURE 8.2 Menu window.

```
       k = menu( 'Click on your option', 'Do this', ...
                 'Do that', 'Quit' );
       if k == 1
         disp( 'Do this ... press any key to continue ...' )
         pause
       elseif k == 2
         disp( 'Do that ... press any key to continue ...' )
         pause
       end
     end
```

Note:

1. The `menu` function enables you to set up a menu of choices for a user.
2. `menu` takes only string arguments. The first one is the title of the menu. The second and subsequent strings are the choices available to the user.
3. The value returned by `menu` (`k` here) numbers the user's choices.
4. Since one has no idea how many choices the user will make, `menu` is properly enclosed in an indeterminate `while` loop. The loop continues to present the menu until the last option (in this example) is selected.
5. You can design much more sophisticated menu-driven applications with the MATLAB GUIDE (Graphical User Interface Development Environment).

SUMMARY

- A `for` statement should be used to program a determinate loop, where the number of repeats is known (in principle) *before* the loop is encountered. This situation is characterized by the general structure plan:

```
    Repeat N times:
        Block of statements to be repeated
```

  where N is known or computed *before* the loop is encountered for the first time, and is not changed by the block.

- A `while` statement should be used to program an indeterminate repeat structure, where the exact number of repeats is *not* known in advance. Another way of saying this is that these statements should be used whenever the truth value of the condition for repeating is changed in the body of the loop. This situation is characterized by the following structure plan:

```
While condition is true repeat:
    statements to be repeated (reset truth value of condition).
```

  Note that *condition* is the condition to repeat.
- The statements in a `while` construct may sometimes never be executed.
- Loops may be nested to any depth.
- The `menu` statement inside a `while` loop may be used to present a user with a menu of choices.

EXERCISES

8.1 A person deposits \$1000 in a bank. Interest is compounded monthly at the rate of 1% per month. Write a program which will compute the monthly balance, but write it only *annually* for 10 years (use nested `for` loops, with the outer loop for 10 years, and the inner loop for 12 months). Note that after 10 years, the balance is \$3300.39, whereas if interest had been compounded annually at the rate of 12% per year the balance would only have been \$3105.85.
See if you can vectorize your solution.

8.2 There are many formulae for computing π (the ratio of a circle's circumference to its diameter). The simplest is

$$\frac{\pi}{4} = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots \tag{8.4}$$

which comes from putting $x = 1$ in the series

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \dots \tag{8.5}$$

(a) Write a program to compute π using Equation (8.4). Use as many terms in the series as your computer will reasonably allow (start modestly, with 100 terms, say, and re-run your program with more and more each time). You should find that the series converges very slowly, i.e. it takes a lot of terms to get fairly close to π.

(b) Rearranging the series speeds up the convergence:

$$\frac{\pi}{8} = \frac{1}{1\times 3} + \frac{1}{5\times 7} + \frac{1}{9\times 11} \dots$$

Write a program to compute π using this series instead. You should find that you need fewer terms to reach the same level of accuracy that you got in (a).

(c) One of the fastest series for π is

$$\frac{\pi}{4} = 6\arctan\frac{1}{8} + 2\arctan\frac{1}{57} + \arctan\frac{1}{239}.$$

Use this formula to compute π. Don't use the MATLAB function `atan` to compute the arctangents, since that would be cheating. Rather use Equation (8.5).

(d) Can you vectorize any of your solutions (if you haven't already)?

8.3 The following method of computing π is due to Archimedes:

```
1. Let $A = 1$ and $N = 6$
2. Repeat 10 times, say:
       Replace $N$ by $2N$
       Replace $A$ by $[2 - \sqrt{(4 - A^2)}]^{1/2}$
       Let $L = NA/2$
       Let $U = L/\sqrt{1 - A^2/2}$
       Let $P = (U + L)/2$ (estimate of $\pi$)
       Let $E = (U - L)/2$ (estimate of error)
       Print $N$, $P$, $E$
3. Stop.
```

Write a program to implement the algorithm.

8.4 Write a program to compute a table of the function

$$f(x) = x\sin\left[\frac{\pi(1 + 20x)}{2}\right]$$

over the (closed) interval $[-1, 1]$ using increments in x of (a) 0.2, (b) 0.1 and (c) 0.01.

Use your tables to sketch graphs of $f(x)$ for the three cases (by hand), and observe that the tables for (a) and (b) give totally the wrong picture of $f(x)$.

Get your program to draw the graph of $f(x)$ for the three cases, superimposed.

8.5 The transcendental number e (2.71828182845904 ...) can be shown to be the limit of

$$(1 + x)^{1/x}$$

as x tends to zero (from above). Write a program which shows how this expression converges to e as x gets closer and closer to zero.

8.6 A square wave of period T may be defined by the function

$$f(t) = \begin{cases} 1 & (0 < t < T) \\ -1 & (-T < t < 0). \end{cases}$$

The Fourier series for $f(t)$ is given by

$$F(t) = \frac{4}{\pi}\sum_{k=0}^{\infty}\frac{1}{2k+1}\sin\left[\frac{(2k+1)\pi t}{T}\right].$$

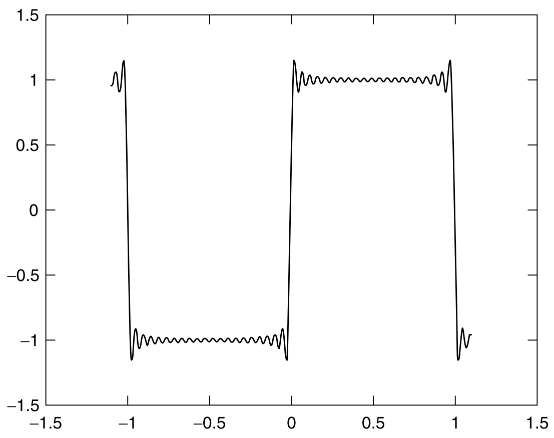

FIGURE 8.3 Fourier series: The Gibbs phenomenon.

It is of interest to know how many terms are needed for a good approximation to this infinite sum. Taking $T = 1$, write a program to compute and plot the sum to n terms of the series for t from -1.1 to 1.1 in steps of 0.01, say. Run the program for different values of n, e.g. 1, 3, 6, etc. Superimpose plots of $F(t)$ against t for a few values of n.
On each side of a discontinuity a Fourier series exhibits peculiar oscillatory behavior known as the Gibbs phenomenon. Figure 8.3 shows this clearly for the above series with $n = 20$ (and increments in t of 0.01). The phenomenon is much sharper for $n = 200$ and t increments of 0.001.

8.7 If an amount of money A is invested for k years at a nominal annual interest rate r (expressed as a decimal fraction), the value V of the investment after k years is given by

$$V = A(1 + r/n)^{nk}$$

where n is the number of compounding periods per year. Write a program to compute V as n gets larger and larger, i.e. as the compounding periods become more and more frequent, like monthly, daily, hourly, etc. Take $A = 1000$, $r = 4\%$ and $k = 10$ years. You should observe that your output gradually approaches a limit. **Hint**: use a `for` loop which doubles n each time, starting with $n = 1$.
Also compute the value of the formula Ae^{rk} for the same values of A, r and k (use the MATLAB function `exp`), and compare this value with the values of V computed above. What do you conclude?

8.8 Write a program to compute the sum of the series $1^2 + 2^2 + 3^2 \ldots$ such that the sum is as large as possible without exceeding 1000. The program should display how many terms are used in the sum.

8.9 One of the programs in Section 8.2 shows that an amount of \$1000 will double in eight years with an interest rate of 10%. Using the same interest rate, run the program with initial balances of \$500, \$2000 and \$10000 (say) to see how long they all take to double. The results may surprise you.

8.10 Write a program to implement the structure plan of Exercise 3.2.

8.11 Use the Taylor series

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

to write a program to compute $\cos x$ correct to four decimal places (x is in radians). See how many terms are needed to get 4-figure agreement with the MATLAB function `cos`. Don't make x too large; that could cause rounding error.

8.12 A student borrows \$10 000 to buy a used car. Interest on her loan is compounded at the rate of 2% per month while the outstanding balance of the loan is more than \$5000, and at 1% per month otherwise. She pays back \$300 every month, except for the last month, when the repayment must be less than \$300. She pays at the end of the month, *after* the interest on the balance has been compounded. The first repayment is made one month after the loan is paid out. Write a program which displays a monthly statement of the balance (after the monthly payment has been made), the final payment, and the month of the final payment.

8.13 A projectile, the equations of motion of which are given in Chapter 3, is launched from the point O with an initial velocity of 60 m/s at an angle of 50° to the horizontal. Write a program which computes and displays the time in the air, and horizontal and vertical displacement from the point O every 0.5 s, as long as the projectile remains above a horizontal plane through O.

8.14 When a resistor (R), capacitor (C) and battery (V) are connected in series, a charge Q builds up on the capacitor according to the formula

$$Q(t) = CV(1 - e^{-t/RC})$$

if there is no charge on the capacitor at time $t = 0$. The problem is to monitor the charge on the capacitor every 0.1 s in order to detect when it reaches a level of 8 units of charge, given that $V = 9$, $R = 4$ and $C = 1$. Write a program which displays the time and charge every 0.1 seconds until the charge first exceeds 8 units (i.e. the last charge displayed must exceed 8). Once you have done this, rewrite the program to display the charge only while it is strictly less than 8 units.

8.15 Adapt your program for the prime number algorithm in Section 8.2 to find all the prime factors of a given number (even or odd).