

Chapter 3

Loops. Part II

3.1 The “while-do” loop

The “for-do” loop discussed in Chapter 2 is suitable for repeated computation when the number (or the upper bound) of the repetition is known. There are many cases where computation needs to be repeated for an unknown number of times until a certain condition eventually becomes false. The “while-do” loop is a programming feature for those situations, as shown in the following example.

Example 3.1 A fixed point iteration. It is known that the iteration

$$x_{k+1} = \cos x_k, \quad \text{for } k = 0, 1, \dots$$

converges to the unique “fixed point” x_* of $\cos x$, namely $x_* = \cos x_*$, from any starting point x_0 . However, it is not known how many iteration steps is needed to achieve a desired accuracy. It may be desirable to keep the iteration running until the error estimate $|x_k - x_{k-1}| < \delta$ where δ is a prescribed error tolerance. In other words, the iteration continues *while* $|x_k - x_{k-1}| > \delta$. The iteration can be carried out accordingly using a “while-do” loop directly in a Maple worksheet:

```
> x[0], x[1] := 1.0, evalf(cos(1.0)):   delta := 10.0^(-8):  
  
k := 1:  
while abs(x[k]-x[k-1]) >= delta do  
    k := k + 1:  
    x[k] := evalf( cos(x[k-1]) ):      # fixed point iteration  
end do:  
  
seq(x[j],j=1..k);                    # show all the iterates
```

```

1., 0.5403023059, 0.8575532158, 0.6542897905, 0.7934803587, 0.7013687737, 0.7639596829, 0.7221024250,
0.7504177618, 0.7314040424, 0.7442373549, 0.7356047404, 0.7414250866, 0.7375068905, 0.7401473356,
0.7383692041, 0.7395672022, 0.7387603199, 0.7393038924, 0.7389377567, 0.7391843998, 0.7390182624,
0.7391301765, 0.7390547908, 0.7391055719, 0.7390713653, 0.7390944074, 0.7390788860, 0.7390893414,
0.7390822985, 0.7390870427, 0.7390838470, 0.7390859996, 0.7390845496, 0.7390855263, 0.7390848684,
0.7390853116, 0.7390850131, 0.7390852141, 0.7390850787, 0.7390851699, 0.7390851085, 0.7390851499,
0.7390851220, 0.7390851408, 0.7390851281, 0.7390851367

```

It takes 46 iteration steps from $x_0 = 1$ to reach the approximate fixed point of prescribed accuracy. □

The syntax of the while-do loop is

```

while loop_condition do
    [
    [ block of statements to be repeated ]
    [
end do;

```

The block of statements inside the while-do loop will be repeated as long as the *loop_condition* remain true. Before writing a while-do loop, it is important to be sure that the *loop_condition* will become false and the loop will not run forever. If an upper bound is known on the number of repetitions, or if the the programmer wants to impose such an upper bound (say 100), the loop in Example 3.1 can be replaced by a “for-while-do” loop

```

...
for k from 1 to 100 while abs(x[k]-x[k-1]) >= delta do
    x[k+1] := evalf( cos(x[k]) );
end do;
...

```

or a for-do loop combined with a conditional “break” statement

```

...
for k from 1 to 100 do
    if abs(x[k]-x[k-1]) < delta then
        break;
    else
        x[k+1] := evalf( cos(x[k]) );
    end if
end do;
...

```

The **break** statement is quite useful to terminate a loop early when certain condition is met.

Chapter 3

Loops. Part II

3.1 The “while-do” loop

The “for-do” loop discussed in Chapter 2 is suitable for repeated computation when the number (or the upper bound) of the repetition is known. There are many cases where computation needs to be repeated for an unknown number of times until a certain condition eventually becomes false. The “while-do” loop is a programming feature for those situations, as shown in the following example.

Example 3.1 A fixed point iteration. It is known that the iteration

$$x_{k+1} = \cos x_k, \quad \text{for } k = 0, 1, \dots$$

converges to the unique “fixed point” x_* of $\cos x$, namely $x_* = \cos x_*$, from any starting point x_0 . However, it is not known how many iteration steps is needed to achieve a desired accuracy. It may be desirable to keep the iteration running until the error estimate $|x_k - x_{k-1}| < \delta$ where δ is a prescribed error tolerance. In other words, the iteration continues *while* $|x_k - x_{k-1}| > \delta$. The iteration can be carried out accordingly using a “while-do” loop directly in a Maple worksheet:

```
> x[0], x[1] := 1.0, evalf(cos(1.0)):   delta := 10.0^(-8):  
  
k := 1:  
while abs(x[k]-x[k-1]) >= delta do  
  k := k + 1:  
  x[k] := evalf( cos(x[k-1]) ):      # fixed point iteration  
end do:  
  
seq(x[j],j=1..k);                   # show all the iterates
```

```

1., 0.5403023059, 0.8575532158, 0.6542897905, 0.7934803587, 0.7013687737, 0.7639596829, 0.7221024250,
0.7504177618, 0.7314040424, 0.7442373549, 0.7356047404, 0.7414250866, 0.7375068905, 0.7401473356,
0.7383692041, 0.7395672022, 0.7387603199, 0.7393038924, 0.7389377567, 0.7391843998, 0.7390182624,
0.7391301765, 0.7390547908, 0.7391055719, 0.7390713653, 0.7390944074, 0.7390788860, 0.7390893414,
0.7390822985, 0.7390870427, 0.7390838470, 0.7390859996, 0.7390845496, 0.7390855263, 0.7390848684,
0.7390853116, 0.7390850131, 0.7390852141, 0.7390850787, 0.7390851699, 0.7390851085, 0.7390851499,
0.7390851220, 0.7390851408, 0.7390851281, 0.7390851367

```

It takes 46 iteration steps from $x_0 = 1$ to reach the approximate fixed point of prescribed accuracy. □

The syntax of the while-do loop is

```

while loop_condition do
    [
    [ block of statements to be repeated ]
    ]
end do;

```

The block of statements inside the while-do loop will be repeated as long as the *loop_condition* remain true. Before writing a while-do loop, it is important to be sure that the *loop_condition* will become false and the loop will not run forever. If an upper bound is known on the number of repetitions, or if the the programmer wants to impose such an upper bound (say 100), the loop in Example 3.1 can be replaced by a “for-while-do” loop

```

...
for k from 1 to 100 while abs(x[k]-x[k-1]) >= delta do
    x[k+1] := evalf( cos(x[k]) );
end do;
...

```

or a for-do loop combined with a conditional “break” statement

```

...
for k from 1 to 100 do
    if abs(x[k]-x[k-1]) < delta then
        break;
    else
        x[k+1] := evalf( cos(x[k]) );
    end if
end do;
...

```

The `break` statement is quite useful to terminate a loop early when certain condition is met.

3.2 The golden section method *(Optimization)*

3.2.1 The unimodal function

The method of the golden section can be applied to find the maximum value of an unimodal function. A function is called *unimodal* if it is strictly increasing, reaches a maximum, and then strictly decreases. For example:

```
> f:=x->-x^2+3*x-2;
```

$$f := x \rightarrow -x^2 + 3x - 2$$

```
> plot(f,0..2);
```

The graph is shown in Fig. 3.1. The maximum value of this unimodal function occurs at $x = 1.5$.

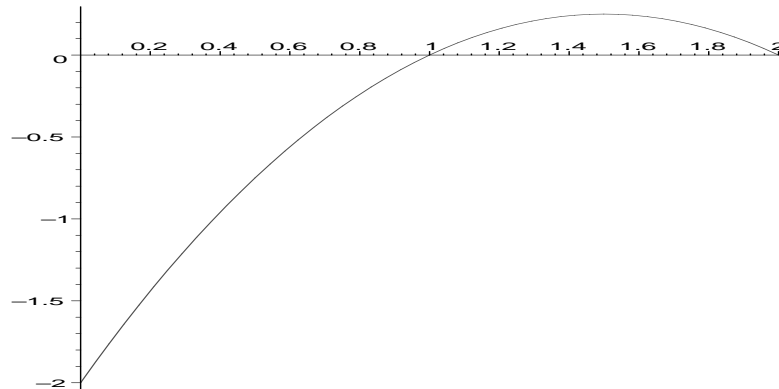


Figure 3.1: A unimodal function

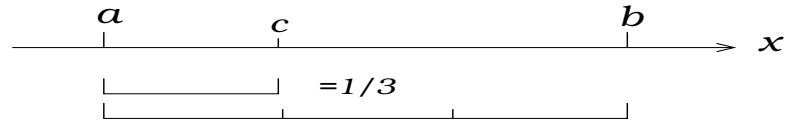
The problem of maximizing a unimodal function arise in many practical applications with or without using computers. A scenario can easily be constructed involving such an application:

A chicken roast problem: A barbecue chain store plans to develop a new brand of roast chicken using their standardized oven. Those 4-pound chicken are slightly undercooked if roasted for 30 minutes but overcooked at 1 hour mark. To maximize the potential profit, the store wants to establish the optimal oven time that maximize the taste appeal, which can be quantified by assembling a focus group of typical customers to score the taste of the roast chicken from 0 to 10. It is obviously reasonable to expect this “taste function” to be a unimodal function of oven time with interval domain $[30, 60]$ in minutes.

The method of golden section is perhaps the best method to solve such problems.

3.2.2 The golden ratio

Let $[a, b]$ be an interval with a point c inside. The location of c in $[a, b]$ can be described in terms of a fraction of the length of the interval $[a, b]$. For example, one may say c is one third to the right of a , meaning the length of $[a, c]$ is $\frac{1}{3}$ of that of $[a, b]$. This “one third” is the *section ratio* of c in $[a, b]$.



In this case, the value of c equals that of a plus the length of $[a, c]$, which is $\frac{1}{3}$ of the length of $[a, b]$. That is,

$$c = a + (c - a) = a + \frac{1}{3}(b - a) = \left(1 - \frac{1}{3}\right)a + \frac{1}{3}b \quad (3.1)$$

In general, a point $c \in [a, b]$ with a section ratio of t (with $0 \leq t \leq 1$) is

$$c = (1 - t)a + tb$$

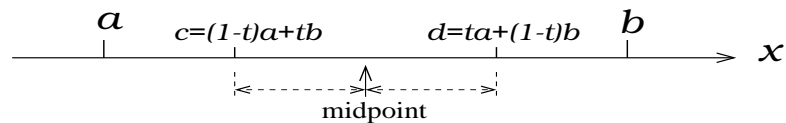
so the section ratio

$$t = \frac{c - a}{b - a} = \frac{\text{length of } [a, c]}{\text{length of } [a, b]}.$$

The point symmetric to c about the midpoint of $[a, b]$ is

$$d = ta + (1 - t)b$$

For any section ratio t , point c and d above are said to be *conjugate* points associated with the section ratio.

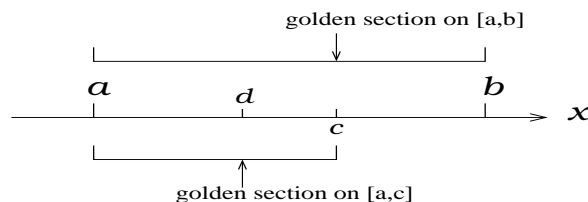


There is a special section ratio τ , called the *golden section ratio*, for which c in the interval $[a, b]$ is

$$c = (1 - \tau)a + \tau b$$

and its conjugate point d cuts the interval $[a, c]$ with the same ratio

$$d = \tau a + (1 - \tau)b = (1 - \tau)a + \tau c$$



It can be verified that the golden section ratio is

$$\tau = \frac{-1 + \sqrt{5}}{2} \approx 0.618$$

Rich and varied topics can be found in the literature and internet related to the golden section or the “golden mean”. with fascinating mathematics and applications.

3.2.3 Method of the golden section

Let $f(x)$ be a unimodal function on $[a, b]$, . A maximum point x_* of $f(x)$ exists in (a, b) . The objective is to shrink the interval and zero in on x_* .

As shown in Figure 3.2, let m_l and m_r be a pair of conjugate section points (mid-left and mid-right points respectively) corresponding to the golden section ratio. Either the left subinterval $[a, m_r]$ or the right subinterval $[m_l, b]$ contains the unique maximum point x_* , depending on which function value, $f(m_l)$ or $f(m_r)$, is larger. If $f(m_r) > f(m_l)$ as illustrated in Figure 3.2, then the interval $[m_l, b]$ containing m_r is the interval of choice. Similarly, if $f(m_l) > f(m_r)$, one would shrink $[a, b]$ to $[a, m_r]$.

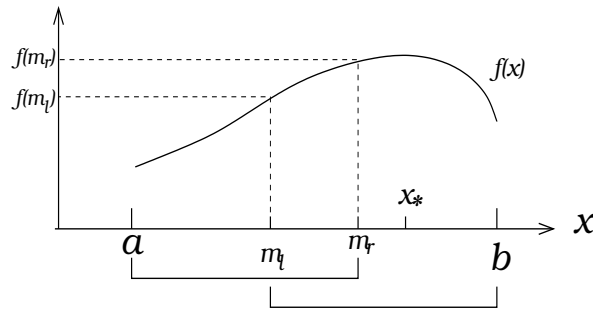


Figure 3.2: Method of the golden section: interval $[m_l, b]$ is chosen over $[a, m_r]$ because $f(m_r) > f(m_l)$

A simple rule can be summarized for choosing subintervals:

- If the ‘left value’ $f(m_l)$ is greater, choose the ‘left subinterval’ $[a, m_r]$.
- If the ‘right value’ $f(m_r)$ is greater, choose the ‘right subinterval’ $[m_l, b]$.

The beauty of the method is that m_l is one of the conjugate points of golden section in $[a, m_r]$, so is m_r in $[m_l, b]$. There is only one function evaluation in each subinterval at the other conjugate point.

Using the chicken roast problem as an example, the store R&D team would initially roast two chicken at

$$\begin{aligned} m_r &= (1 - \tau) \cdot 30 + \tau \cdot 60 \approx 48.5 \quad \text{and} \\ m_l &= \tau \cdot 30 + (1 - \tau) \cdot 60 \approx 41.5 \end{aligned}$$

minutes. The focus group scores two chicken with, say, 7.8 and 6.2 points in taste. So the new interval containing the optimal over time is $[m_l, b] = [41.5, 60]$. To shrink this new interval, the R&D team only needs to roast one chicken at 52.9 minutes since its conjugate is 48.5 and the score 7.8 is already known! Let the taste score at 52.9 minutes be, say, 7.2. Then the optimal time is in the interval $[41.5, 52.9]$ and the next chicken is to be roasted for 45.8 minutes, scored by the focus group, and compared with the score 7.8 at the conjugate time 48.5. An approximate optimal oven time will be obtained by continuing this process.

The method of the golden section can be described in the following pseudo-code:

```

Input:  $a, b, f$  and error tolerance  $\delta$ 
start with a working interval  $[\alpha, \beta] = [a, b]$  (more on this later)
calculate golden section points  $m_l$  and  $m_r$ , along with  $v_l = f(m_l)$  and  $v_r = f(m_r)$ 
While  $|\beta - \alpha| > \delta$ , repeat the following as a loop
  If  $v_l > v_r$  then
    Replace  $[\alpha, \beta]$  with  $[\alpha, m_r]$  (that is, set  $\beta = m_r$ )
    Set  $m_r$  and  $v_r$  to the current  $m_l$  and  $v_l$  respectively
    Replace  $m_l$  with  $\tau\alpha + (1 - \tau)\beta$ , calculate  $v_l = f(m_l)$ 
  else
    Replace  $[\alpha, \beta]$  with  $[m_l, \beta]$ 
    Set  $m_l$  and  $v_l$  to the current  $m_r$  and  $v_r$  respectively
    Replace  $m_r$  with  $(1 - \tau)\alpha + \tau\beta$ , calculate  $v_r = f(m_r)$ 
end if

```

A program based on the pseudo-code:

```

# A program to calculate the maximum point of an unimodal function
#
goldsec := proc( f,      # the unimodal function
                a,      # the left end point of the interval
                b,      # the right end point of the interval
                delta    # error tolerance
                )
  local tau, alpha, beta, ml, mr, vl, vr;

  if delta <= 0.0 then          # avoid a negative delta
    return "error tolerance must be positive";

```

```

end if;

tau := evalf( 0.5*(-1.0+sqrt(5.0)) ); # golden section ratio
alpha, beta := a, b; # working interval
mr := (1-tau)*alpha + tau*beta; # the mid-right point
ml := tau*alpha + (1-tau)*beta; # the mid-left point
vl, vr := f(ml), f(mr); # function values

while abs(beta-alpha) >= delta do # the main loop
  if vl > vr then
    beta := mr; # update working interval
    mr, vr := ml, vl; # update mr and vr
    ml := tau*alpha + (1-tau)*beta; # the new mid-left
    vl := f(ml); # the new vl
  else
    alpha := ml; # update working interval
    ml, vl := mr, vr; # update ml and vl
    mr := (1-tau)*alpha + tau*beta; # the new mid-right
    vr := f(mr); # the new vr
  end if;
end do;

return 0.5*(alpha+beta); # output
end proc;

```

A test execution of the program using the unimodal function in Figure 3.1:

```

> f := x -> -x^2 + 3*x -2:
> a, b, tol := 0, 2, 0.001:
> goldsec(f, a, b, tol);

```

1.500193498

Remark: Variables `alpha` and `beta` are initially set to be a and b respectively to form the necessary working interval $[\alpha, \beta]$, since `a` and `b` are input arguments that are not allowed to alter inside the program.

3.3 Vectors

3.3.1 Generating and accessing a vector

For Maple a vector is an ordered set of Maple objects, so a vector can be used to store a sequence of data. For example, to divide the interval $[0,1]$ into 100 subintervals of equal length 0.01, the subinterval endpoints (often called nodes) are:

$$x_1 = 0, x_2 = .01, x_3 = .02, \dots, x_{100} = .99, x_{101} = 1.00$$

To generate a vector of nodes, open an (empty) vector first:

```
> node := Vector(101):
```

Then generate the nodes with a loop as follows. Notice that the index starts at 1.

```
> for i from 1 to 101 do
  node[i] := (i-1)*0.01
end do:
```

The value of x_i for each i is stored in `node[i]`. The values can be shown by using the sequence command (`seq`):

```
0, .01, .02, .03, .04, .05, .06, .07, .08, .09, .10, .11, .12, .13, .14, .15, .16, .17, .18, .19, .20,
.21, .22, .23, .24, .25, .26, .27, .28, .29, .30, .31, .32, .33, .34, .35, .36, .37, .38, .39,
.40, .41, .42, .43, .44, .45, .46, .47, .48, .49, .50, .51, .52, .53, .54, .55, .56, .57, .58,
.59, .60, .61, .62, .63, .64, .65, .66, .67, .68, .69, .70, .71, .72, .73, .74, .75, .76, .77,
.78, .79, .80, .81, .82, .83, .84, .85, .86, .87, .88, .89, .90, .91, .92, .93, .94, .95, .96,
.97, .98, .99, 1.00
```

Each entry of the vector is referenced by `vector_name[index]` (notice the square brackets). In this example,

```
> node[19];
```

```
.18
```

```
> node[87];
```

```
.86
```

Vectors may contain numbers, strings, and other data types.

3.3.2 Vectors as input

When a vector (say \mathbf{x}) is used as an input of a program, its dimension can be retrieved using the statement

```
n := LinearAlgebra[Dimension](x);
```

as shown in the example below. The function `Dimension` is in the package `LinearAlgebra`, which will be discussed extensively in later chapters.

Example 3.2 The following program calculates the mean value of the values stored in a vector:

```

mean := proc(x)
  local mn, k;

  n := LinearAlgebra[Dimension](x); # get dimension

  mn := 0; # calculate the sum
  for k from 1 to n do
    mn := mn + x[k];
  end do;

  return mn/n; # output the mean
end proc;

```

Test run:

```
> a := Vector([3.7,5.4,1.2,7.9,8.3]);
```

$$a := \begin{bmatrix} 3.7 \\ 5.4 \\ 1.2 \\ 7.9 \\ 8.3 \end{bmatrix}$$

```
> mean(a):
```

5.3

3.3.3 Vectors as output

For a program to generate and output a vector, there must be a statement to define a vector and subsequent statements to calculate/assign its entries, as shown in the following example.

Example 3.3 Fibonacci again. Suppose the values of the Fibonacci sequence are needed for later use. An easy way to do this is to output a vector. The following is a modification of the previous Fibonacci program.

```

# program to generate the first n terms of the Fibonacci sequence.
FibonacciVector := proc(n)
  local F,k;

```

```

F := Vector[row](n);           # define an empty vector
F[1] := 0; F[2] := 1;         # the first two terms

for k from 3 to n do          # calculate/assign entries iteratively
    F[k] := F[k-1]+F[k-2];
end do;

return F;                     # output
end proc;

```

Running the program:

```
> G := FibonacciVector(20);
```

$$G := \begin{bmatrix} 20 \text{ Element Row Vector} \\ \text{Data Type : anything} \\ \text{Storage : rectangular} \\ \text{Order : Fortran_order} \end{bmatrix}$$

By default, Maple does not show vectors of dimensions larger than 10. Functions `evalm` and `seq` can be used to view the entries of a vector:

```
> evalm(G);
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

```
> seq(G[j], j=1..20);
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181
```

```
> H := FibonacciVector(50): H[50]; # access a specific entry, say H[50]
```

```
7778742049
```

Besides the `Dimension` command in Example 3.2, another handy function in the package `LinearAlgebra` is `RandomVector`. It is useful for generating vectors for computing experiment. For example, to generate an *integer* vector of dimension 10 with random entries spread from -9 to 9:

```
> a := LinearAlgebra[RandomVector][row](10, generator=-9..9);
```

```
[-1 -3 -4 -6 5 -6 5 -8 -5 -1]
```

In contrast, to generate a *real* vector of the same dimension with random entries spread from -9.0 to 9.0:

```
> a := LinearAlgebra[RandomVector][row](10,generator=-9..9);

[-7.25162793775474412, - 8.16891496863922838, - 4.01538627070398046,
 -8.42700876520642694, 3.70882958435295862, - 5.91863961939188776,
 2.79860202319602002, - 1.93991364838497304, 4.37638442624849056,
 4.63932235041000141]
```

Obviously, the option [row] can be deleted if column vectors are needed.

Example 3.4 Maximum entry of a vector. The program for finding the value and location of the largest entry of an input vector:

```
findmax := proc( s                               # input vector
)
local i, n, value_max, index_max;

n := LinearAlgebra[Dimension](s); # get dimension

value_max := evalf(s[1]); # assume the first one is max
index_max := 1;
for i from 2 to n do # check the remaining entries
  if evalf(s[i]) > value_max then # if a bigger value is found ...
    value_max := s[i]; # update the maximum value
    index_max := i; # update the index of the max
  end if;
end do;

return value_max, index_max;
end proc;

> # generate a test vector
a := LinearAlgebra[RandomVector][row](10,generator=0..9);

a := [7, 3, 5, 7, 2, 8, 1, 3, 1, 4]

> vmax, imax := findmax(a);
```

```
vmax, imax := 8, 6
```

3.3.4 Sorting (*Computer Science*)

Sorting is the problem of rearranging a sequence of numbers x_1, x_2, \dots, x_n in ascending order. The following steps constitute the most basic sorting method:

step 1: find the smallest entry from x_1, x_2, \dots, x_n and swap it to x_1 .

step 2: find the smallest entry from x_2, x_3, \dots, x_n and swap it to x_2 .

step 3: find the smallest entry from x_3, x_4, \dots, x_n and swap it to x_3 .

.....

step n-1: find the smaller entry between x_{n-1} and x_n and swap it to x_{n-1}

(Question to ponder: is there a step n?)

In summary, each loop for $k = 1, 2, 3, \dots, n - 1$, carries out the following step:

step k: find the smallest entry from x_k, x_{k+1}, \dots, x_n and swap it to x_k .

There must be another loop nested inside to find the smallest entry in a subsequence.

```

sort_ascend := proc( x # vector to be sorted
)
  local n, k, j, tmp, value_min, index_min;

  n := LinearAlgebra[Dimension](x);      # get dimension
  y := x[1..n];                          # the working vector

  for k from 1 to n-1 do

    # find the value_min and index_min of y[k],y[k+1],...,y[n]
    value_min := y[k];    index_min := k;
    for j from k+1 to n do
      if evalf(value_min) > evalf(y[j]) then
        value_min := evalf(y[j]);    index_min := j;
      end if;
    end do;

    # swap to k-th entry if necessary
    if index_min > k then
      y[index_min] := y[k];    y[k] := value_min;
    end if;

  end do;

  return y;
end proc;

```

```
>a := Vector[row]([3,-4,9,8,-Pi,exp(1),sqrt(12),1/3,0]):
```

$$a := \left[3, -4, 9, 8, -\pi, e, 2\sqrt{3}, \frac{1}{3}, 0 \right]$$

```
> y := sort_ascend(a);
```

$$y := \left[-4, -\pi, 0, \frac{1}{3}, e, 3, 2\sqrt{3}, 8, 9 \right]$$

3.4 Exploring scientific computing

3.4.1 The bisection method (*Numerical Analysis*)

The Intermediate Value Theorem covered in algebra and calculus courses states that if $f(x)$ is a continuous function on the interval $[a, b]$ such that $f(a)$ and $f(b)$ have different signs, then there exists a number $x_* \in [a, b]$ such that $f(x_*) = 0$. A sketch makes this theorem obvious (although there may be more than one root!).

Assuming a continuous function and two points a and b that satisfy the conditions of the theorem, the theorem can be exploited to approximate a root by cutting the interval in half once it is determined which half contains a root. The mid-point is $mp = (a + b)/2$ and the signs of $f(mp)$ and $f(a)$ can be compared to determine if they are the same or opposite (an arbitrary choice - comparing $f(mp)$ with $f(b)$ is equivalent), since the signs of these two numbers will determine whether we choose the interval $[a, mp]$ or the interval $[mp, b]$. The sign of $f(mp) * f(a)$ is positive if they have the same sign and negative otherwise, so

$$\text{the solution } x_* \text{ is in or equal to } \begin{cases} [a, mp] & f(a)f(mp) < 0 \\ mp & f(mp) = 0 \\ [mp, b] & \textit{otherwise} \end{cases}$$

At each step the interval $[a, b]$ is replaced with the new subinterval, either $[a, mp]$ or $[mp, b]$, of half length. This process, called the **bisection method**, can be repeated until the length of the final interval is less than the error tolerance. At the end, the approximate solution can be given as the midpoint of the final interval. The implementation of the bisection method has much in common with the program for the method of the golden section, and is left to the reader.

How can one stop the loop, given a tolerance requirement? An option would be to calculate the number of steps: $n \geq \frac{(b-a)}{tol}$, and make sure you take at least that many steps. Another is to let Maple do it using the `while .. do` loop.

Two notes:

The expression $mp = (a + b)/2$ creates a potential roundoff problem and should be written $mp = a + \frac{b-a}{2}$. (Maple will do with this what it wishes, since it tends to simplify expressions, despite the user's best intentions!)

The `while` loop can include a complicated logical statement such as: `while (b-a) > tol and f(mp)<>0 do` (The '`<>`' symbol means 'not equal to'.)

3.4.2 The greatest common divisor (*Number Theory*)

The greatest common divisor (GCD) of integers m and n , denoted by $d = \gcd(m, n)$, is defined as the largest positive integer that divides both m and n . For computational purpose, only the case $m > n > 0$ needs to be considered (why?). A classical method for computing GCD is the Euclidean Algorithm, which is one of the oldest algorithms in the history. The algorithm appeared in *Euclid's Elements* around 300 BC., but it was probably discovered 200 years earlier.

The Euclidean Algorithm can be understood from the the Euclid Division Lemma, which asserts that there is a unique pair of quotient q and remainder r such that

$$m = n \cdot q + r \quad \text{with } 0 \leq r < |n|,$$

implying the GCD d divides r since the d divides both m and n . Furthermore, there is a decreasing sequence of remainders r_j for $j = 0, 1, 2, \dots$ satisfying

$$r_{j-1} = r_j \cdot q_j + r_{j+1} \quad \text{for } j = 1, 2, \dots \quad (3.2)$$

with $r_0 = m$ and $r_1 = n$. The GCD d must divide every member of the remainder sequence. Therefore, the last nonzero remainder in the sequence must be the GCD. For example, the Euclidean Algorithm for computing $\gcd(13578, 9198)$ generates the remainder sequence

$$r_0 = 13578, \quad r_1 = 9198, \quad r_3 = 4380, \quad r_4 = 438, \quad r_5 = 0$$

leading to the GCD 438.

The implementation of the Euclidean Algorithm will be an exercise. The Maple function `mod` can be conveniently used to find each remainder (c.f. `?mod` in Maple).

3.4.3 The greatest common divisor (*Computer Algebra*)

The greatest common divisor (GCD) of polynomials $f(x)$ and $g(x)$, denoted by $d = \gcd(f, g)$ is, similar to the GCD of integers, defined as the monic polynomial $d(x)$ of the highest degree that divides both $f(x)$ and $g(x)$. Here a polynomial is monic if its leading coefficient is one. Computing polynomial GCD is one of the fundamental problems in Computer Algebra.

For example, the GCD of

$$\begin{aligned} f(x) &= (x-1)^3(x-2)^5(x-3) & \text{and} \\ g(x) &= (x-1)^4(x-2)^2(x+3)^2 \end{aligned} \quad (3.3)$$

is $d(x) = (x - 1)^3(x - 2)^2$.

The Euclidean Algorithm can also be applied to compute the GCD of two polynomials. Without loss of generality again, assume the degree of $f(x)$ is no less than that of $g(x)$. Let $r_0(x) = f(x)$ and $r_1(x) = g(x)$. There is a remainder sequence $r_j(x)$ for $j = 1, 2, \dots$ with decreasing degrees such that

$$r_{j-1} = r_j(x) \cdot q_j(x) + r_{j+1}(x).$$

The last nonzero remainder in the sequence is the GCD after making it monic.

Maple commands `rem`, `lcoeff`, and `degree` are needed to compute remainders and leading coefficients. Syntaxes and examples of these commands can be accessed by `?rem` and `?lcoeff`. The Euclidean Algorithm for finding GCDs can be illustrated using the polynomial pairs in (3.3):

```
> f := expand( (x-1)^3*(x-2)^5*(x-3));   g := expand( (x-1)^4*(x-2)^2*(x+3)^2 );
```

$$\begin{aligned} f &:= x^9 - 16x^8 + 112x^7 - 450x^6 + 1143x^5 - 1902x^4 + 2072x^3 - 1424x^2 + 560x - 96 \\ g &:= x^8 - 2x^7 - 13x^6 + 40x^5 + 11x^4 - 170x^3 + 253x^2 - 156x + 36 \end{aligned}$$

```
> degree(f,x), degree(g,x);   # verify the degrees
```

9, 8

```
> r[0], r[1] := f, g;           # initialize r[0] and r[1]
> r[2] := rem( r[0], r[1], x);  # finding the remainder r[2]
```

$$r_2 := 408 + 97x^7 - 672x^6 + 1692x^5 - 1578x^4 - 561x^3 + 2274x^2 - 1660x$$

```
> r[3] := rem( r[1], r[2], x);  # finding the remainder r[3]
> r[4] := rem( r[2], r[3], x);  # finding the remainder r[4]
```

$$r_4 := -\frac{67744800}{1934881} + \frac{16936200}{1934881}x^5 - \frac{118553400}{1934881}x^4 + \frac{321787800}{1934881}x^3 - \frac{423405000}{1934881}x^2 + \frac{270979200}{1934881}x$$

```
> r[5] := rem( r[3], r[4], x);  # finding the remainder r[5]
```

$$r_5 := 0$$

```
> r[4]/lcoeff(r[4],x);         # scale r[4] to make it monic, obtaining the GCD
```

$$-4 + x^5 - 7x^4 + 19x^3 - 25x^2 + 16x$$

Implementing the Euclidean Algorithm for computing polynomial GCD will be an exercise (Problem 6)

3.4.4 Rational approximation (*Number Theory*)

As legend goes, Pythagoras held an unshakable belief that all numbers were rational and thus every number could be written as $\frac{p}{q}$ with integers p and q . One of Pythagoras's students, Hippasus, was trying to find a rational representation for $\sqrt{2}$ while traveling at sea but made an astonishing discovery: It is impossible to write $\sqrt{2}$ as a fraction $\frac{p}{q}$ using any pair of integers p and q . Instead of celebrating the discovery of irrational numbers, however, Pythagoras could not accept the demise of his belief and he could not disprove Hippasus argument either. As a result, fellow Pythagoreans angrily threw Hippasus overboard and drowned him.

It is now known that it is impossible to find a rational number to equal a irrational number such as $\sqrt{2}$, π , e , etc. However, rational numbers are “dense” on the number line in the sense that there are rational numbers within any neighborhood of any irrational number. Consequently, an irrational numbers can be approximated by infinitely many rational numbers. For example, two fractions

$$\frac{22}{7} = \mathbf{3.14285714\dots}, \quad \frac{355}{113} = \mathbf{3.14159292035\dots}$$

are the most famous rational approximations of π . The question becomes how to find the most accurate rational approximation p/q to an irrational number, using smallest possible denominator q .

Perhaps the simplest algorithm for computing rational approximations is as follows, with input r to be a real number that is to be approximated and N to be the upper bound on the denominator. Starting with $p = \lfloor r \rfloor$, $q = 1$ and the initial error $\varepsilon_0 = |\frac{p}{q} - r|$, repeat the following as long as $q \leq N$:

- If $\frac{p}{q} < r$, increase p by 1.
- Otherwise, increase q by 1.
- With the new pair of p and q , calculate the current error $\varepsilon_1 = |\frac{p}{q} - r|$
- If the current error ε_1 is smaller than the previous error ε_0 , record the current rational approximation p/q and reset $\varepsilon_0 := \varepsilon_1$.

This process generates a sequence $\{\frac{p_1}{q_1}, \frac{p_2}{q_2}, \frac{p_3}{q_3}, \dots, \frac{p_k}{q_k}\}$ with increasing accuracy approximating r . Implementation of this method will be an exercise. Some sample results are shown below.

Approximating π with denominator bound 16900:

```
> a, k := RationalApproximate(Pi,16900):
> seq(a[j],j=1..k);
```

$$3, \frac{13}{4}, \frac{16}{5}, \frac{19}{6}, \frac{22}{7}, \frac{179}{57}, \frac{201}{64}, \frac{223}{71}, \frac{245}{78}, \frac{267}{85}, \frac{289}{92}, \frac{311}{99}, \frac{333}{106}, \frac{355}{113}, \frac{52518}{16717}, \frac{52873}{16830}$$

(One can see the beauty of $\frac{355}{113}$ from this sequence. A better approximation requires a denominator larger than 15,000.)

Approximating $\sqrt{2}$ with denominator bound 20000:

```
> b, n := RationalApproximate(sqrt(2),20000):
> seq(b[j],j=1..n);
```

$$1, \frac{3}{2}, \frac{4}{3}, \frac{7}{5}, \frac{17}{12}, \frac{24}{17}, \frac{41}{29}, \frac{99}{70}, \frac{140}{99}, \frac{239}{169}, \frac{577}{408}, \frac{1393}{985}, \frac{3363}{2378}, \frac{4756}{3363}, \frac{8119}{5741}, \frac{19601}{13860}, \frac{27720}{19601}$$

3.4.5 Rational approximation revisited (*Number Theory*)

The Euclidean Algorithm can be extended to generating a continuous fraction for approximating an irrational number r . Assuming $r > 0$ without loss of generality, initialize $r_0 = r$, $r_1 = 1$, and the integer “quotient” $q_1 = \lfloor r \rfloor$ such that

$$r_0 = r_1 \cdot q_1 + r_2$$

where the “remainder” r_1 satisfies $0 < r_2 < r_1$. Equivalently

$$\frac{r_0}{r_1} = q_1 + \frac{1}{\frac{r_1}{r_2}}.$$

Using the integer “quotient” $q_2 = \lfloor \frac{r_1}{r_2} \rfloor$ yields

$$r_1 = r_2 \cdot q_2 + r_3 \quad \text{or} \quad \frac{r_1}{r_2} = q_2 + \frac{1}{\frac{r_2}{r_3}},$$

leading to

$$\frac{r_0}{r_1} = q_1 + \frac{1}{q_2 + \frac{1}{\frac{r_2}{r_3}}}.$$

This process can be continued infinitely and produces

$$r = \frac{r_0}{r_1} = q_1 + \frac{1}{q_2 + \frac{1}{q_3 + \frac{1}{q_4 + \frac{1}{\ddots}}}}.$$

Each truncation of the continuous fraction is a rational approximation to the irrational number r . For instance, the Euclidean Algorithm produces the continuous fraction

$$\pi = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{\ddots}}}}$$

and rational approximations

$$3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103993}{33102}, \dots$$

Computing rational approximations by the Euclidean Algorithm will be an exercise. The method for evaluating truncated continuous fractions described in §2.2.4 is recommended.

3.4.6 The sieve of Eratosthenes

A classical algorithm for finding prime numbers was the sieve method that was devised by Eratosthenes of Alexandria in the third century BC. The sieve method strikes out non-prime numbers and, at the end, leaves only primes in the “sieve”.

Consider integers $1, 2, \dots, n$. The sieve method can be described as the following process for $k = 1, 2, 3, \dots, \lfloor \sqrt{n} \rfloor$

$k = 1$: Strike out 1 since it is neither prime nor composite.

$k = 2$: Knowing $k = 2$ is in the sieve, strike out $4, 6, 8, \dots$ since they are multiples of 2.

$k = 3$: Knowing $k = 3$ is still in the sieve, strike out its multiples $6, 9, 12, \dots$

$k = 4$: Since $k = 4$ has been struck out, do nothing.

...

Generally at step k , strike out its multiples $2k, 3k, \dots$ up to n if k is still in the sieve, otherwise do nothing.

The sieve method can be programmed using a shadow vector s of dimension n . Initially, every entry of s is set to be a string "in", meaning all numbers from 1 to n are initially in the sieve. If any number k is to be struck out, reassign s_k as "out". After finishing the sieving, collect those integer k 's where s_k still carries the string "in" in a vector for output. A sample program and test run are as follows:

```

PrimeSieve := proc(n)
  local p, s, k, i, count;

  s := Vector(n);          # initialize the shadow vector
  s[1] := "out";          # 1 is not a prime
  for k from 2 to n do
    s[k] := "in"          # numbers 2, 3, ..., n are in the sieve initially
  end do;

  for k from 2 to floor(sqrt(n)) do # the loop of sieving
    if s[k]="in" then
      for i from k to n/k do
        s[i*k] := "out";      # mark multiples of k as "out"
      end do;
    end if;
  end do;

  # count the number of primes found
  count := 0;
  for i from 2 to n do
    if s[i]="in" then
      count := count+1;
    end if;
  end do;

  p := Vector[row](count); # open a vector to store prime numbers
  k := 0;                  # initialize counter
  for i from 2 to n do
    if s[i]="in" then      # when s[i] is "in", i is the k-th prime
      k := k+1; p[k] := i; # store this prime
    end if;
  end do;

  return p[1..count];
end proc;

> p := PrimeSieve(100);

      [ 25 Element Row Vector ]
      [ Data type: anything   ]
      [ Storage: rectangular  ]
      [ Order: Fortran_order  ]

> seq(p[j],j=1..25)

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

```

3.5 Exercise

1. A division algorithm (*Number Theory*). The Euclid Division Lemma in Number

Theory states that for any given integers m and n , there is a unique pair of integers q and r called the quotient and the remainder respectively, such that

$$m = nq + r, \quad 0 \leq r < |n|.$$

There is a simple algorithm for computing q and r assuming both m and n are nonnegative: For $k = 0, 1, \dots$, compute the decreasing sequence

$$r_k = m - n \cdot k$$

until r_k falls in the range $0 \leq r_k < n$. Then output the quotient $q = k$ and the remainder $r = r_k$. Use a while-do for the program.

2. **A division algorithm, revisited** (*Number Theory*). The division algorithm in Problem 1 above assumes both m and n be nonnegative. Analyze the three other cases $\{m > 0, n < 0\}$, $\{m < 0, n > 0\}$ and $\{m < 0, n < 0\}$. Then improve the algorithm and the program so that all four cases can be handled.
3. **Harmonic series** (*Calculus*). The harmonic series $\sum_{j=1}^{\infty} \frac{1}{j} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$ is a classical example of divergent series. Namely, for any given number $n > 0$, there is a $k_n > 0$ such that

$$\sum_{j=1}^{k_n} \frac{1}{j} \geq n$$

Write a program that, for input integer $n > 0$, output the row vector $[k_1, k_2, \dots, k_n]$. For example, when input $n = 8$, the output is $[1, 4, 11, 31, 83, 227, 616, 1674]$.

4. **Longevity of a retirement fund** (*Finance*). A retiree wants to withdraw a fixed amount $\$x$ per month from a fund of initial balance $\$B$ paying $r\%$ annual interest rate compounded monthly. He decided to write a Maple program for figuring out how long will the fund last. First of all, if the the withdraw x is no more than the interest of the first month, the fund will last forever. Otherwise, the monthly balance will decrease according to

$$\left\{ \begin{array}{c} \text{this month's} \\ \text{balance} \end{array} \right\} = \left\{ \begin{array}{c} \text{last month's} \\ \text{balance} \end{array} \right\} + \left\{ \begin{array}{c} \text{this month's} \\ \text{interest} \end{array} \right\} - \left\{ \begin{array}{c} \text{withdraw} \\ \text{amount} \end{array} \right\}.$$

The withdraw would continue as long as the monthly balance stay positive. Write a program that, for input B , r and x , outputs the number of months the fund would last.

5. **Integer GCD: The Euclidean Algorithm** (*Number Theory*). Implement the Euclidean Algorithm of §3.4.2 by writing a program that, for input integers m and n , output $\text{gcd}(m, n)$.
6. **Polynomial GCD: The Euclidean Algorithm** (*Computer Algebra*). Implement the Euclidean Algorithm in §3.4.3 by writing a program to compute the GCD of polynomials $f(x)$ and $g(x)$. The program should accept input f , g and x and output the (monic) GCD.

7. **Rational approximation: Simple method** (*Number Theory*). Implement the simple rational approximation method described in §3.4.4 that, for input irrational number r and denominator bound N , outputs a sequence of rational approximations with decreasing errors.
8. **Rational approximation: The Euclidean Algorithm** (*Number Theory*). Implement the Euclidean Algorithm described in §3.4.5 for computing a sequence of n rational approximations to an input irrational number r .
9. **Angle between two vectors** (*Linear Algebra*). Given two vectors $x = [x_1, \dots, x_n]$ and $y = [y_1, \dots, y_n]$, the angle θ between x and y is defined as

$$\theta(x, y) = \arccos \frac{\sum_{j=1}^n x_j y_j}{\sqrt{\sum_{j=1}^n x_j^2} \sqrt{\sum_{j=1}^n y_j^2}}$$

Write a program that, for input vectors x and y , output the angle $\theta(x, y)$.

10. **Vector projection** (*Linear Algebra*). Given two vectors $x = [x_1, \dots, x_n]$ and $y = [y_1, \dots, y_n]$, the projection of x along the direction of y is a vector z defined by

$$z = [\alpha y_1, \dots, \alpha y_n], \quad \text{where } \alpha = \frac{\sum_{j=1}^n x_j y_j}{\sum_{j=1}^n y_j^2}.$$

Write a program that, for input vectors x and y , output the projection z of x along the direction of y .

11. **Nearest component** (*Statistics*). Write a program that, for an onput vector $x = [x_1, \dots, x_n]$ and a number s , output the index i and distance d such that

$$d = |s - x_i| = \min_{1 \leq j \leq n} |s - x_j|$$

. Here x_i is the component of x that is nearest to s .

12. **Standard deviation** (*Statistics*). Write a program that, for an input Vector $x = [x_1, \dots, x_n]$, calculates the standard deviation

$$\sigma = \sqrt{\sum_{i=1}^n \frac{(x_i - \mu)^2}{n}}$$

where μ is the arithmetic mean of the vector x_1, x_2, \dots, x_n , i.e.:

$$\mu = \sum_{i=1}^n \frac{x_i}{n}.$$

13. **Pearson's sample correlation coefficient, I (Statistics).** Let $(x_1, y_1), \dots, (x_n, y_n)$ be a sequence of data pairs. The Pearson's sample correlation coefficient r is defined as

$$r = \frac{1}{(n-1) s_x s_y} \sum_{j=1}^n (x_j - X)(y_j - Y)$$

where X and Y are arithmetic means of $x = [x_1, \dots, x_n]$ and $y = [y_1, \dots, y_n]$ respectively, while s_x and s_y are the standard deviations of x and y respectively. Write a program that, for input vectors x and y , outputs the the Pearson's sample correlation coefficient r .

14. **Pearson's sample correlation coefficient, II (Statistics).** Let $(x_1, y_1), \dots, (x_n, y_n)$ be a sequence of data pairs. The Pearson's sample correlation coefficient r can also be equivalently defined as

$$r = \frac{\sum_{j=1}^n x_j y_j - \frac{1}{n} \left(\sum_{j=1}^n x_j \right) \left(\sum_{j=1}^n y_j \right)}{\sqrt{\sum_{j=1}^n x_j^2 - \frac{1}{n} \left(\sum_{j=1}^n x_j \right)^2} \sqrt{\sum_{j=1}^n y_j^2 - \frac{1}{n} \left(\sum_{j=1}^n y_j \right)^2}}$$

Write a program that, for input vectors $x = [x_1, \dots, x_n]$ and $y = [y_1, \dots, y_n]$, outputs the the Pearson's sample correlation coefficient r using this formula.

15. **Sorting by insertion (Computer Science)** For an input vector $x = [x_1, \dots, x_n]$, it can be sorted into ascending order by the following insertion method: First of all, pass the input data to a working vector $y = [y_1, \dots, y_n]$. Consider initially y_1 is sorted. Generally, assume y_1, \dots, y_k are sorted into ascending order, then y_{k+1} will be compared with y_1, y_2, \dots, y_k one by one and inserted into the proper place so that the new entries y_1, \dots, y_{k+1} are in ascending order. This process is continued for $k = 1, 2, \dots, n-1$ and end up with a sorted vector y . Write a program to implement this sorting method.
16. **Fibonacci primes (Number Theory).** Many Fibonacci numbers are primes that can be identified by the Maple function `isprime`. Write a program that, for input n , identifies all the primes among the Fibonacci numbers F_0, F_1, \dots, F_n and outputs the them in a row vector. For example, when $n = 20$, the output should be $[2, 3, 5, 13, 89, 233, 1597]$.
17. **Ulam's lucky numbers¹ (Number Theory).** From the list of positive integers $1, 2, 3, 4, \dots, n$, remove every second number, leaving $1, 3, 5, 7, 9, \dots$. The first surviving number is 3 and now remove every 3rd number from the *remaining* numbers, yielding $1, 3, 7, 9, 13, 15, 19, 21, \dots$. The next surviving number is 7 so every 7th surviving number is removed, leaving $1, 3, 7, 9, 13, 15, 21, \dots$. Continue this sieving process until all the surviving numbers are used. Numbers that are not removed are

¹M. C. Wunderlich, *Sieving procedures on a digital computer*, J. of ACM, Vol. 14, pp 10-19, 1967

considered “lucky”. Write a program which outputs a vector of the lucky numbers between 1 and n using the sieve method described in §3.4.6. Sample result:

```
> u := UlamLuckyNumbers(100):
> seq(u[j], j=1..LinearAlgebra[Dimension](u));
```

1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69, 73, 75, 79, 87, 93, 99

18. **The Josephus problem²(Number Theory)**. During the Jewish rebellion against Rome (A.D. 70) 40 Jews were holding out in a cave and facing certain defeat. They would rather die than be slaves. Therefore they agreed upon a process of mutual destruction by standing in a circle and kill every seventh person until only one was left. That last one would commit suicide. The only survivor, Flavius Josephus, who quickly figured out the spot to become the last man standing, did not carry out the last step. Instead he lived to tell the story.

The generalized Josephus problem: Let n persons be arranged in a circle and numbered from 1 to n . Starting from 1, remove every k -th standing person around the circle continuously until only one person (Josephus) remains standing. What is the Josephus’ number?

Write a program that, for input integer $n > 0$ and k , output the vector containing the numbers in the order they would be removed. Josephus’ number is the last entry of the vector. Sample results

```
> p := Josephus(40,7):
> seq(p[j], j=1..40);
```

7, 14, 21, 28, 35, 2, 10, 18, 26, 34, 3, 12, 22, 31, 40, 11, 23, 33, 5, 17, 30, 4, 19,
36, 9, 27, 6, 25, 8, 32, 16, 1, 38, 37, 39, 15, 29, 13, 20, 24

The result shows that Josephus was at the number 24 spot.

²see Chris Groër, *The Mathematics of Survival: From Antiquity to the Playground*, The American Mathematical Monthly, Vol. 110, pp. 812–825, 2003.

