Input, Output and Other Programming Issues

STAT2005

Chapter 5

Error handling

A well-written function or program should have good error handling ability.

Let us modify the temperature conversion function with error handling.

```
f2c_err <- function(low,up,step) {
      # f2c with error handling
      # error checking
      if (low>up) stop ("Error: first arg. > second arg.")
      # check if low<=up
      if (step<=0) stop ("Error: last arg. <= 0")
      # check if step>0
      f<-seq(low,up,step)        # create vector f
      c<-(5/9)*(f-32)       # compute vector c from f
      return(cbind(f,c))  # combine f and c
}
```

Obviously we have assumed that `low<=up` and `step>0`. Therefore we may need to check these conditions before carry out the computation.

Let us try to test this function with errors:

```
> f2c_err(200,0,20)
Error in f2c_err(200, 0, 20) : Error: first arg. > second arg.

> f2c_err(0,200,-20)
Error in f2c_err(0, 200, -20) : Error: last arg. <= 0
```

Warnings

Warnings are weaker than errors; they signal that something has gone wrong, but the code has been able to recover and continue.

Unlike errors, you can have multiple warmings from a function call.

```
f <- function() {
    ...
    warning("W1")
    ...
    warning("W2")
}
```

User defined operator

We can define our own operator as well.

For example, we define the operator `%+-%` such that `x %+-% s` will return `(x-s,x+s)`.

```
> "%+-%" <- function(x,s) { c(x-s,x+s) }
> 3 %+-% 5
[1] -2  8
```

Replacement functions

Replacement functions act like they are modifiying the argument in place. For example, we can use `names()` to modify the name attribute of a list.

```
> x<-list(a=1,b=2)
> x
$a
[1] 1

$b
[1] 2
```

```
> names(x)<-c("c","d")
> x
$c
[1] 1

$d
[1] 2
```

Replacement functions

We can write our own replacement functions. It's name takes the form XXX<-. For example,

```
"modify<-" <- function(x, position, value) {
    x[position] <- value
    x
}
x <- 1:10
modify(x,2) <- 5L
x
 [1]  1  5  3  4  5  6  7  8  9 10
```

Flexible number of arguments

- We can have flexible number of arguments as demonstrated in the following example.
- The `maxlen()` function will return the maximum length of the input vectors. However, the number of inputted argument is undecided, it is only available upon execution.

```
maxlen <- function (...) {  # allow flexible arguments
      arg <- list(...)      # save the argument list to arg
      mx <- 0               # initialize mx
      for (x in arg) mx <-max(mx,length(x))  # find max length
      return(mx)
}
> maxlen(1:5,1:8,1:3,1:10)  # no. of argument = 4
[1] 10
> maxlen(1:4,runif(20),1:15)# no. of argument = 3
[1] 20
```

Saving multiple outputs from function

Sometimes we may want to save several outputs from a function.

For example, consider a function `stat.v1` to compute the mean, standard deviation, median, minimum and maximum of an input vector.

```
stat.v1<-function (x) {    # input vector x
      m<-mean(x)           # mean
      s<-sd(x)             # stdev
      med<-median(x)       # median
      min<-min(x)          # min
      max<-max(x)          # max
      return(c(mean=m,stdev=s,median=med,min=min,max=max))
      # output as a vector
}
> out<-stat.v1(1:12)       # save the output as vector
> out
     mean      stdev     median        min        max
 6.500000  3.605551  6.500000  1.000000 12.000000
```

We can also save and label the outputs using `list()` as follow:

```
stat.v2<-function (x) {  # input vector x
     m<-mean(x)          # mean
     s<-sd(x)            # stdev
     med<-median(x)      # median
     min<-min(x)         # min
     max<-max(x)         # max
     out<-list(m,s,med,min,max)
     # save items to out and apply names
     names(out)<-c("mean", "stdev", "median", "min", "max")
     return(out)         # display outputs
}

> result<-stat.v2(1:12)  # save the output to result
> names(result)     # to display the items in result
[1] "mean"   "stdev"  "median" "min"    "max"
```

Now the result has six components. To display them, use

```
> result
$mean
[1] 6.5
$stdev
[1] 3.605551
$median
[1] 6.5
$min
[1] 1
$max
[1] 12
```

We could then compute the range of the input vector (i.e. max – min) by

```
> result$max-result$min
[1] 11
```

Formatted input and output

- We can have a better control of our input, output and error checking in our own function.
- Let us begin by producing a CDF (left-tail probability) table of a standard normal distribution found in most statistical textbook:

```
y<-seq(0,3.4,0.1)
# define sequence of y from 0 to 3.4 with step 0.1
x<-seq(0,0.09,0.01)
# define sequence of x from 0 to 0.09 with step 0.01
z<-outer(y,x,"+")
# save the table to z, where z(i,j)=y(i)+x(j)
options(digits=4)      # specify output display to 4 decimal place
t<-pnorm(z)            # compute the left tail and save them to t
t<-rbind(x,t)          # add the first row to t
y<-c(0,y)              # add a zero to y
cbind(y,t)             # output the table
```

x	y									
	0.0	0.0000	0.0100	0.0200	0.0300	0.0400	0.0500	0.0600	0.0700	0.0800	0.0900
	0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
	0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
	0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
	0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
	0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
	0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
	0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
	0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
	0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
	0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
	1.0	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
	1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
	1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
	1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9131	0.9147	0.9162	0.9177
	1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
	1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
	1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
	1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
	1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
	1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
	2.0	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817

The built-in function `sprintf()` can be used to control the format of outputting numbers.

```
> sprintf("Pi is %f", pi)
# output real number with default option = 6 decimal places
[1] “Pi is 3.141593"
> sprintf("%.3f", pi)     # with 3 decimal places
[1] "3.142"
> sprintf("%5.1f", pi)    # fixed width=5 with 1 decimal places
[1] "  3.1"
> sprintf("%-10f", pi)    # left justified with fixed width=10
[1] "3.141593  "
> sprintf("%e", pi) # scientific notation
[1] "3.141593e+00“
```

The first argument (begins with %) is the formatting string. f stands for fixed point.

See `help(sprintf)` for more details.

The `cat` function

- `cat` function is very useful in displaying strings, numbers or variables inside the user defined function.
- Suppose we want to print a symmetric matrix in lower triangular form instead of printing it in full mode. We can write a function `prtsym()` to do the job:

```
> x1<-matrix(1:16,ncol=4)
> x2<-matrix(1:16,ncol=4,byrow=T)
> y<-x1+x2    # y is a symmetric matrix
> y
     [,1] [,2] [,3] [,4]
[1,]    2    7   12   17
[2,]    7   12   17   22
[3,]   12   17   22   27
[4,]   17   22   27   32
```

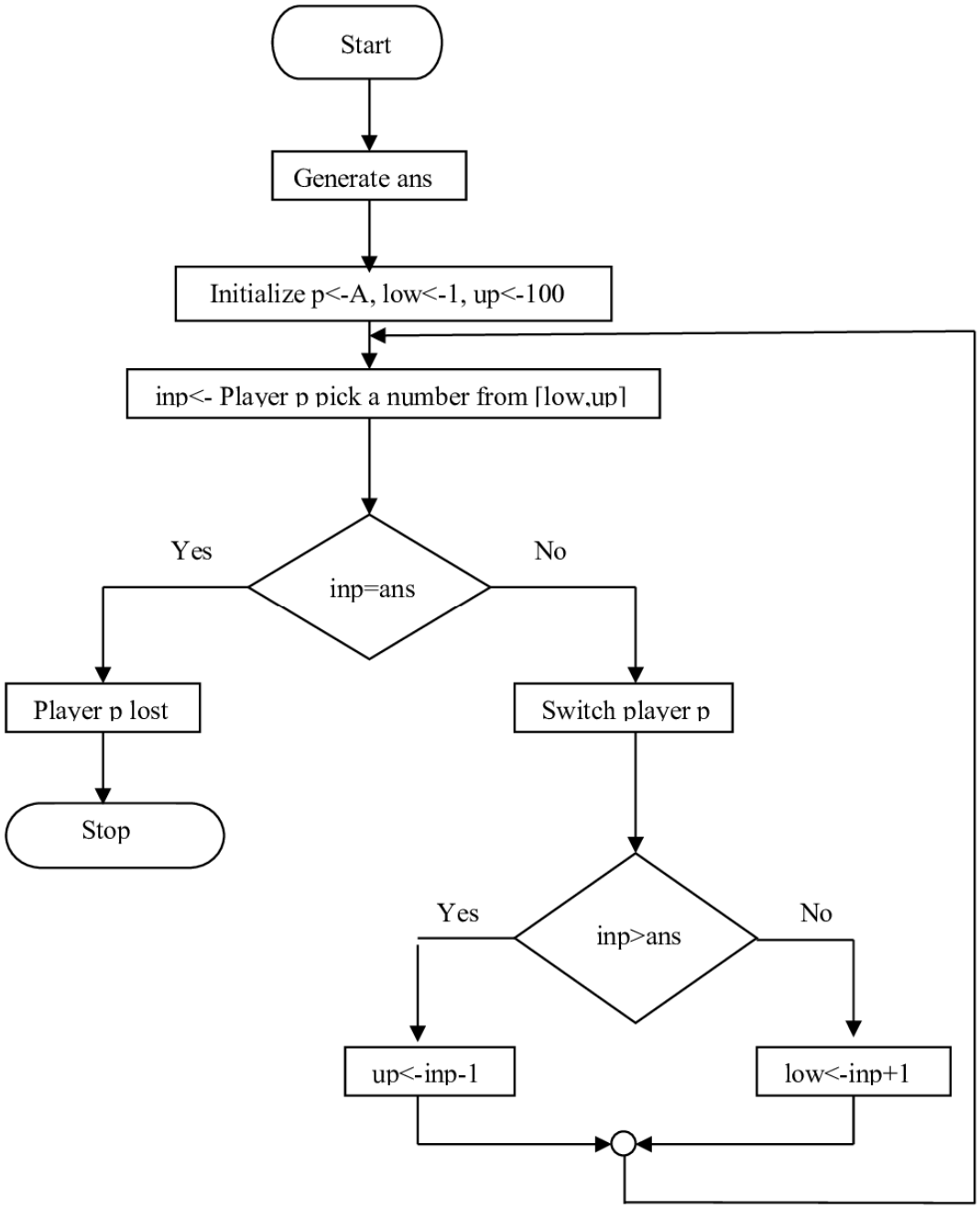
```
prtsym<-function(x) {
# print a symmetric matrix in lower triangular form
      n1<-dim(x)[1]    # get the row dimension of x
      n2<-dim(x)[2]    # get the column dimension of x
      if (n1!=n2) stop("Input matrix is not a square matrix")
      for (i in 1:n1) {      # check if x is symmetric
            for (j in 1:i) {
                  if (x[i,j]!=x[j,i])
                        stop("Input matrix is not a symmetric matrix")
            }
      }
      for (i in 1:n1) {      # loop for i from 1 to n1
            for (j in 1:i) { # nested loop for j from 1 up to i
                  cat(sprintf("%2.0f", x[i,j]), "")
                  # print x[i,j] with an extra space after x[i,j]
            }
            cat("\n")  # print a carriage return, i.e. new line
      }
}
```

- To demonstrate the error checking techniques, we check whether the input matrix is a square matrix and a symmetric matrix before we print it out.
- We use a nested for loop to print the matrix in lower triangular form.
- The character `\n` stands for newline. That means it will print on the next line.

```
> prtsym(y)
 2
 7 12
12 17 22
17 22 27 32
```

The `readline` function

- `readline()` function can be used to accept user's input from keyboard.
- Let us illustrate it by a simple guessing game. The program will randomly choose an hidden integer X between 1 and 100, and player A and B are going to pick an integer in turns.
- The program will indicate a range for each player to choose from. Player picks the hidden integer X will lose the game.
- When writing complicated program, it may be helpful to plan ahead by drawing flow chart and/or writing some pseudo-code first.


Start
Generate ans
Initialize p<-A, low<-1, up<-100
inp<- Player p pick a number from [low,up]
Yes
inp=ans
No
Player p lost
Stop
Switch player p
Yes
inp>ans
No
up<-inp-1
low<-inp+1

Pseudo-code

1. Generate ans
2. Initialize p <-A, low <-1, up <-100
3. inp <- Player p input a number from [low, up]
4. If (inp=ans) then display: player p lost, stop
5. if p=A then p <- B else p <- A
6. If (inp>ans) then up <- inp-1
   else low <- inp+1
7. Go to step [3]

```
guess<-function() {          # guessing game between player A and B
      cat("I will randomly choose an integer x from 1 to 100.\n\n")
      ans<-sample(1:100,size=1)# generate an integer between 0 and 100
      p<-"A"                 # initialize p, low and up
      low<-1
      up<-100
      repeat {
            repeat {
                  cat("Player",p,", please pick an integer from",low,"to",up,": ")
                  inp<-readline()                    # input
                  inp<-as.numeric(inp)         # change to numeric
                  if ((inp<=up)&(inp>=low)) break
                  else cat("Number out of range!\n")
            }
            if (inp==ans) break                # inp=ans, lose game
            if (p=="A") p<-"B" else p<-"A"     # switch player
            if (inp>ans) up<-inp-1                   # update up
            else low<-inp+1                          # update low
      }
      cat("Player",p,"lose the game.\n") # display end message
}
```

There are two points worth mentioning.

- First, `inp<-as.numeric(inp)` is needed to change `inp` to numeric mode. Otherwise, unexpected results will occur.
- Second, the second repeat loop is for validation of player's input. It is necessary to prevent cheating by entering invalid numbers.

Sierpinski triangle

Now we consider an interesting example which is related to fractals.

We construct a Sierpinski triangle named after the Polish mathematician Wacław Sierpiński who described it in 1915. The following pseudo code describes the construction:

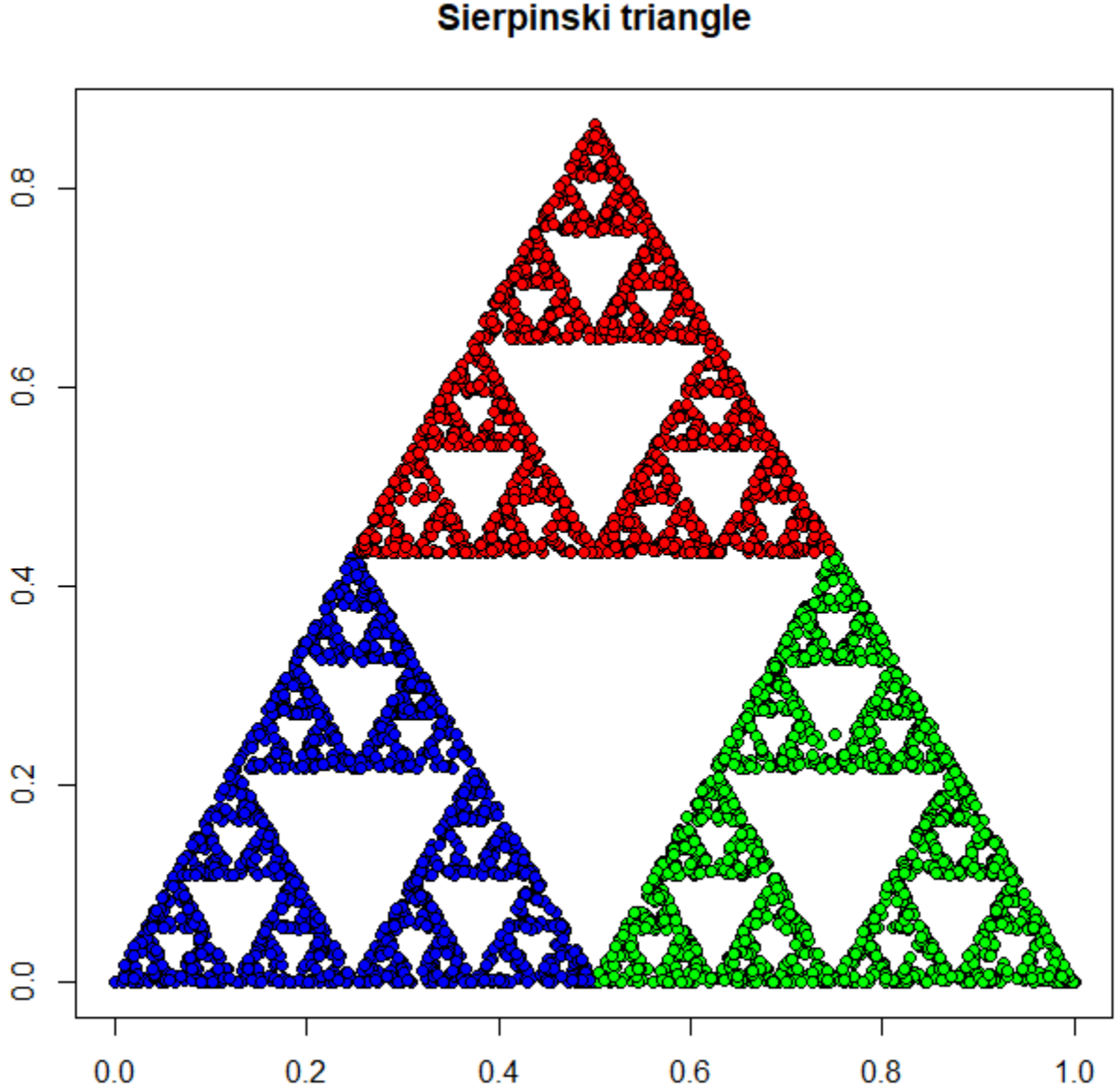
1. Plot the following blue, green and red point:
   $b = (b_1, b_2) = (0, 0)$, $g = (g_1, g_2) = (1, 0)$,
   $r = (r_1, r_2) = (\cos(\pi/3), \sin(\pi/3)$.
2. Initialize $p_0 = (x_0, y_0) = (0.5, 0.5)$.
3. Randomly choose a color: blue, green or red.

4. If color=blue, compute and plot the point
   p = mid-point of p_0 and b in blue;
   else if color=green, compute and plot
   p = mid-point of p_0 and g in green;
   else if color=red, compute and plot
   p = mid-point of p_0 and r in red.
5. Update p_0 to p.
6. Repeat step 3 to 5 for 5,000 times.

The following is the actual implementation of these in R.

```
# sierpinski triangle
b1<-0           # set (b1,b2) to (0,0)
b2<-0
g1<-1           # set (g1,g2) to (1,0)
g2<-0
r1<-cos(pi/3)
# set (r1,r2) to (cos(pi/3),sin(pi/3))
r2<-sin(pi/3)
xy<-rbind(c(b1,b2),c(g1,g2),c(r1,r2))
# form xy matrix for plotting
plot(xy,main="Sierpinski triangle",xlab="",ylab="")
# plot 3 points
points(b1,b2,pch=21,bg="blue")
points(g1,g2,pch=21,bg="green")
points(r1,r2,pch=21,bg="red")
x0<-0.5             # initialize (x0,y0) to (0,5)
y0<-0.5
```

```r
set.seed(1234)     # set random seed
n<-5000            # number of points
for (i in 1:n) {
      col<-sample(c("b","g","r"),prob=c(1/3,1/3,1/3),size=1)
      # randomly pick a color
      if (col=="b") {          # color=blue
            x<-(x0+b1)/2       # mid-point between x0 and b
            y<-(y0+b2)/2
            points(x,y,pch=21,bg="blue")  # plot this point in blue
      }
      if (col=="g") {          # color=green
            x<-(x0+g1)/2       # compute mid-point bewtten x0 and g
            y<-(y0+g2)/2
            points(x,y,pch=21,bg="green") # plot this point in green
      }
      if (col=="r") {          # color=red
            x<-(x0+r1)/2       # compute mid-point between x0 and r
            y<-(y0+r2)/2
            points(x,y,pch=21,bg="red")          # plot this point in red
      }
      x0<-x                    # update x0
      y0<-y                    # update y0
}
```


Sierpinski triangle
0.0
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0

Recursive function

One of the most powerful (and dangerous) programming ability in all high level programming language is the recursive function. Briefly speaking, recursive function is a function that calls itself. Here is a classical example:

```
fac<-function(n){
      # factorial function, assume n is an integer > 0
      if (n<=2) return(n)
      else return(n*fac(n-1))
      # fac calls itself; fac(n)=n*fac(n-1)
}
```

Note that when $n = 0$, the above function `fac(0)=0` is not correct.

Exercise

Modify this function to give correct output of `fac(0)`.

```
fac1<-function(n){
  if (n==0)
     return(1)
  else if (n<=2)
    return(n)
  else
    return(n*fac1(n-1))
}
```

Exercise

Rewrite this factorial function without using recursion and gives correct answer for $n = 0$.

```
fac2<-function(n){
     if ((n==0)|(n==1)) {
          return(1)
     } else {
          result <- 1
          for (i in 2:n)
               result <- result * i
          return(result)
     }
}
```

Efficient programming

- In order to write efficient code, you need to understand the platform you are working with.
- For example, R is designed to work with vectors. Operations on whole vectors are usually much faster than working one element at a time.
- This is why vector-based programming is better than using loops.

Measure the time your program takes

- When performing large scale calculations, speed consideration is important.
- The built-in function `proc.time()` enables us to calculate the exact computing time of a program.
- Consider the following two versions of program.

Version 1: Building x one-by-one

```
pt0<-proc.time()          # save initial time
n<-50000                  # define length of vector
m<-n/2
x<-NULL                   # initialize x
for (i in 1:m) {
     x<-c(x,i)            # combine x to i
     x<-c(x,n-i+1)        # combine x to n-i+1
}
proc.time()-pt0           # compute process time
```

Version 2: Create a vector of length n first

```
pt0<-proc.time()  # save initial time
n<-50000          # define length of vector
m<-1:(n/2)
x<-0*(1:n)        # create a vector of length n
for (i in m) {
     x[2*i-1]<-i  # save i to x[2*i-1]
     x[2*i]<-n-i+1# save n-i+1 to x[2*i]
}
proc.time()-pt0   # compute process time
```

Version 1 gives:

```
   user  system elapsed
   9.46    2.02   11.47
```

Version 2 gives:

```
   user  system elapsed
   0.65    0.03    0.69
```

It is clear that version 2 is much faster than version 1. However, an even faster version is to rewrite the code without using loop.

Version 3: Without using loop

```
pt0<-proc.time()  # save initial time
n<-50000          # define length of vector
m<-1:(n/2)
x<-0*(1:n)        # create a vector of length n
x[2*m-1]<-m       # save the odd index in x
x[2*m]<-n-m+1     # save the even index in x
proc.time()-pt0   # compute process time
```

Version 3 gives:

```
   user  system elapsed
   0.28    0.00    0.29
```

How to make a function vectorized?

Consider the following function.

```
is.positive <- function(x) {
  if (x>0) 1 else 0
}
```

This function does not support vector input because of the if statement.

```
x <- c(-1,0,1)
is.positive(x)
```

```
[1] 0
Warning message:
In if (x > 0) 1 else 0 :
  the condition has length > 1 and only the first
element will be used
```

We know that we can use `ifelse()` to make it vectorized.

In general, we can use the `vapply()` function to vectorize the operation of a function that does not support vectorization. It's syntax takes the form

```
vapply(X, FUN, FUN.VALUE, ...)
```

where `FUN.VALUE` specifies the output format of the function `FUN`. For example, we can vectorise the operation of `is.positive()` as follows.

```
vapply(x,is.positive,FUN.VALUE=numeric(1))
```

with `numeric(1)` indicating floating point format.

Other possible output format includes

```
logical(1)
integer(1)
character(1)
```

We have used `outer()` to create table. It can also be used to perform many operations on matrix without using loop and hence speed up the execution time.

For example, we can change all the upper triangular elements in a square matrix to zero without using loop.

```
A<-matrix(1:16, nrow=4)
# create a square matrix A
outer(1:nrow(A),1:nrow(A),'>=')*A
# change upper triangular elements to zero
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    2    6    0    0
[3,]    3    7   11    0
[4,]    4    8   12   16
```

Exercise

Without using loops, write a function named `slash()` that generates the following results.

```
> slash(matrix(1:20,ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    0    0    0   16
[2,]    0    0   12    0
[3,]    0    8    0    0
[4,]    4    0    0    0
[5,]    0    0    0    0
> slash(matrix(1:20,ncol=5))
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0   13    0
[2,]    0    0   10    0    0
[3,]    0    7    0    0    0
[4,]    4    0    0    0    0
```

Some general programming guidelines

Writing a computer program to solve a problem can usually be reduced to the following sequence of steps:

1. Understand the problem.
2. Work out a general idea for how to solve it.
3. Translate your general idea into a detailed implementation.
4. Check: does it work?
   - Is it good enough?
   - If yes, you are done!
   - If no, go back to step 2.

Example: Sorting

We wish to write a program which will sort a vector of integers into increasing order.

Understand the problem

A good way to understand your programming problem is to try a specific case. You will often need to consider a simple case, but take care not to make it too trivial.

For the sorting problem, we might consider sorting the vector consisting of the elements 3, 5, 24, 6, 2, 4, 13, 1. We will write a function to be called `sort()` for which the output should be the numbers in increasing order:

1, 2, 3, 4, 5, 6, 13, 24.

Work out a general idea

- A first idea might be to find where the smallest value is, and put it aside. Then repeat, with the remaining values, setting aside the smallest value each time.
- An alternative idea: compare successive pairs of values, starting at the beginning of the vector, and running through to the end. Swap pairs if they are out of order.
- After checking, you will find that the alternative idea doesn't work. Try using this idea on 2, 1, 4, 3, 0, for example. After running through it, you should end up with 1, 2, 3, 0, 4.
- In your check of this alternate idea, you may notice that the largest value always lands at the end of the new vector. This means that we can sort the vector by starting at the beginning of the vector, going through all adjacent pairs, then repeating this procedure for all but the last value, and so on.

Detailed implementation

- At the implementation stage, we need to address specific coding questions. In this sorting problem, one question to be addressed is as follows: how do we swap `x[i]` and `x[i+1]`?
- Here is a way to swap the value of `x[3]` with that of `x[4]`:

```
save <- x[3]
x[3] <- x[4]
x[4] <- save
```

- Alternatively, we may write

```
x[3] <- x[3] + x[4]
x[4] <- x[3] - x[4]
x[3] <- x[3] - x[4]
```

```r
sort <- function(x) {
    # x is initially the input vector and will be
    # modified to form the output
    for(last in length(x):2) {
        for(first in 1:(last - 1)) {
            if(x[first] > x[first + 1]) {
                # swap the pair
                save <- x[first]
                x[first] <- x[first + 1]
                x[first + 1] <- save
            }
        }
    }
    return (x)
}
```

Check

Always begin testing your code on simple examples to identify obvious bugs.

```
sort(c(2, 1))
[1] 1 2
sort(c(2, 24, 3, 4, 5, 13, 6, 1))
[1] 1 2 3 4 5 6 13 24
sort(1)
Error in if (x[first] > x[first + 1]) {: missing value
where TRUE/FALSE needed
```

The problem here is that when `length(x) == 1`, the value of `last` will take on the values `1:2`, rather than no values at all. This doesn't require a redesign of the function, since we can fix it by handling this as a special case at the beginning of our function:

```
sort <- function(x) {
     if (length(x) < 2) return (x)
     # last is the last element to compare with
     for(last in length(x):2) {
          for(first in 1:(last - 1)) {
               if(x[first] > x[first + 1]) {
                    # swap the pair
                    save <- x[first]
                    x[first] <- x[first + 1]
                    x[first + 1] <- save
               }
          }
     }
     return (x)
}
```

Good programming habits

- Good programming is clear rather than clever.
- You may find that even programs you write yourself can be very difficult to understand after only a few weeks have passed if things are not clear enough.
- We find the following to be useful guidelines:

1. Start each program with some comments giving the name of the program, the author, and what the program does.
2. A description of what a program does should explain what all the inputs and outputs are.

3. Variable names should be descriptive, that is, they should give a clue as to what the value of the variable represents.
4. You can find out whether or not your preferred name for an object is already in use by the `exists` function: `exists("name")`.
5. Use blank lines to separate sections of code into related parts, and use indenting to distinguish the inside part of an if statement or a for or while loop.

Debugging and maintenance

- Computer errors are called bugs. Removing these errors from a program is called debugging.
- Debugging is difficult, and one of our goals is to write programs that don't have bugs in them: but sometimes we make mistakes.
- We have found that the following five steps help us to find and fix bugs in our own programs.

1. Recognize that a bug exists.
2. Make the bug reproducible.
3. Identify the cause of the bug.
4. Fix the error and test.
5. Look for similar errors.

Recognizing that a bug exists

- If the program doesn't work, there is a bug.
- However, in some cases the program seems to work, but the output is incorrect, or the program works for some inputs, but not for others.
- Strategies to recognize a bug:
  - Break up your program into simple, self-contained functions.
  - Document their inputs and outputs.
  - Within the function, test that the inputs obey your assumptions about them, and think of test inputs where you can see at a glance whether the outputs match your expectations.

Make the bug reproducible

- Before you can fix a bug, you need to know where things are going wrong.
- This is much easier if you know how to trigger the bug. Bugs that appear unpredictably are extremely difficult to fix.
- For example, a common mistake in programming is to misspell the name of a variable. Normally this results in an immediate error message, but sometimes you accidentally choose a variable that actually does exist.
- Then you'll probably get the wrong answer, and the answer you get may appear to be random, because it depends on the value in some unrelated variable.

Identify the cause of the bug

- If your program has stopped with an error, read the error messages. Try to understand them as well as you can.
- If you are using a loop, you may try using the `print()` or `cat()` functions to show the intermediate results in each iteration. The `readline()` function would also be useful to pause the program before the error occur.
- In R, you can obtain extra information about an error message using the `traceback()` function. When an error occurs, R saves information about the current stack of active functions, and `traceback()` prints this list.

Example: Debugging

In this function we calculate the coefficient of variation as the standard deviation of a variable, after dividing by its mean. However, our test case gives an error:

```
cv <- function(x) {
     sd(x / mean(x))
}
xl <- rnorm(10)
cv(x1)
Error in is.data.frame(x): object 'x1' not
found
```

The error message talks about the function `is.data.frame()`, which we didn't use. To find out where it was called from, we use `traceback()`:

```
traceback()
4: is.data.frame(x)
3: var(if (is.vector(x) || is.factor(x)) x else
as.double(x), na.rm = na.rm)
2: sd(x/mean(x)) at #2
1: cv(x1)
```

This shows that our `cv()` function called the standard function `sd()`, and it called `var()`, and it ended up calling `is.data.frame()`.

But notice that the only place `x1` is mentioned is in our original call, so we need to look at it more closely.

When we do, we discover that our original variable was named xl (with an "L," not a "one"), and that's the cause of the error.

Fixing errors and testing

- Once you have identified the bug in your program, you need to fix it.
- Try to fix it in such a way that you don't cause a different problem. Then test what you've done.

Look for similar errors elsewhere

- Often when you have found and fixed a bug, you can recognize the kind of mistake you made.
- It is worthwhile looking through the rest of your program for similar errors, because if you made the mistake once, you may have made it twice.