

Solutions to Selected Exercises
for
Braun and Murdoch's
A First Course in Statistical Programming with R

Kristy Alexander, Yiwen Diao, Qiang Fu, and Yu Han
W. John Braun and Duncan J. Murdoch

November 1, 2007

Chapter 4

Programming with R

4.1 Flow Control

4.1.1 The for() loop

1. (a)

```
> Fibonacci <- numeric(12)
> Fibonacci[1] <- 2
> Fibonacci[2] <- 2
> for (i in 3:12) Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]
> Fibonacci
[1] 2 2 4 6 10 16 26 42 68 110 178 288
```
- (b)

```
> Fibonacci <- numeric(12)
> Fibonacci[1] <- 3
> Fibonacci[2] <- 2
> for (i in 3:12) Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]
> Fibonacci
[1] 3 2 5 7 12 19 31 50 81 131 212 343
```
- (c)

```
> Fibonacci <- numeric(12)
> Fibonacci[1] <- Fibonacci[2] <- 1
> for (i in 3:12) Fibonacci[i] <- Fibonacci[i-1] - Fibonacci[i-2]
> Fibonacci
[1] 1 1 0 -1 -1 0 1 1 0 -1 -1 0
```
- (d)

```
> Fibonacci <- numeric(12)
> Fibonacci[1] <- Fibonacci[2] <- Fibonacci[3] <- 1
> for (i in 4:12) Fibonacci[i] <- Fibonacci[i-1] + Fibonacci[i-2] + Fibonacci[i-3]
> Fibonacci
[1] 1 1 1 3 5 9 17 31 57 105 193 355
```
3. (a)

```
> answer <- 0
> for (j in 1:5) answer <- answer + j
> answer
[1] 15
```
- (b)

```
> answer <- NULL
> for (j in 1:5) answer <- answer + j
> answer
numeric(0)
```
- (c)

```
> answer <- 0
> for (j in 1:5) answer <- c(answer, j)
> answer
```

```

[1] 0 1 2 3 4 5
(d) > answer <- 1
    > for (j in 1:5) answer <- answer*j
    > answer
[1] 120
(e) > answer <- 3
    > for (j in 1:15) answer <- c(answer, (7*answer[j])%%31)
    > answer
[1] 3 21 23 6 11 15 12 22 30 24 13 29 17 26 27 3
5. > x <- 0.5
    > count <- 0
    > while(abs(x-cos(x))>0.01){
+     x <- cos(x)
+     count <- count+1
+ }
    > count
[1] 10
    > x <- 0.5
    > count <- 0
    > while(abs(x-cos(x))>0.001){
+     x <- cos(x)
+     count <- count+1
+ }
    > count
[1] 15
    > x <- 0.5
    > count <- 0
    > while(abs(x-cos(x))>0.0001){
+     x <- cos(x)
+     count <- count+1
+ }
    > count
[1] 21
    > x <- 0.7
    > count <- 0
    > while(abs(x-cos(x))>0.0001){
+     x <- cos(x)
+     count <- count+1
+ }
    > count
[1] 17
    > x <- 0.0
    > count <- 0
    > while(abs(x-cos(x))>0.0001){
+     x <- cos(x)
+     count <- count+1
+ }
    > count
[1] 23

```

4.1.2 The if() statement

1. Yes, the function will work properly when n is not an integer.

```
3. > GIC <- function(P, n){
+   if (n<=3) i <- 0.04 else i <- 0.05
+   return(P*((1+i)^n -1))
+ }
```

4.1.3 The while() loop

```
1. > Fib1 <- 1
> Fib2 <- 1
> Fibonacci <- c(Fib1, Fib2)
> while (Fib1 < 300){
+   Fibonacci <- c(Fibonacci, Fib2)
+   Fib2 <- Fib1 + Fib2
+   Fib1 <- max(Fibonacci)
+ }
> print(Fibonacci)

[1] 1 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
3. > i <- 0
> i0 <- 0.006
> while (abs(i - i0)>= 0.000001){
+   i <- i0
+   i0 <- (1-(1+i)^(-20))/19
+ }
> print(i)

[1] 0.004955135
```

```
5. > Fibonacci <- c(1, 1, 1)
> while (max(Fibonacci)<1000000){
+   Fibonacci <- c(Fibonacci, Fibonacci[length(Fibonacci)]+
+                 Fibonacci[length(Fibonacci)-1])
+ }
> print(length(Fibonacci))

[1] 32
```

4.1.4 Newton's method for root finding

```
1. > x <- 0
> f <- x^7 + 10000*x^6 + 1.06 * x^5 + 10600*x^4 + 0.0605 * x^3 + 605 * x^2 + 0.0005 *x +5
> tolerance <- 0.000001
> count <- 0
> while (abs(f) > tolerance){
+   f.prime <- 7*x^6+6*10000*x^5 +5*1.06*x^4 + 4*10600*x^3+3*0.0605*x^2 +2*605 *x +0.0005
+   x <- x - f/f.prime
+   f <- x^7 + 10000*x^6 + 1.06*x^5 + 10600*x^4 + 0.0605*x^3 + 605*x^2 + 0.0005*x +5
+   count <- count+1
+ }
> count

[1] 1
```

```

3. > x <- -1.5
   > f <- cos(x) + exp(x)
   > tolerance <- 0.000001
   > while (abs(f) > tolerance){
+     f.prime <- -sin(x) - 1/x
+     x <- x - f/f.prime
+     f <- cos(x) + exp(x)
+   }
   > x

[1] -1.746139

```

5. The function has only one zero which is $x = 0.6$. For initial guess 0.5, 0.75 and 0.2, Newton's method gives a solution that approaches $x=1$. For initial guess 1.25, the method does not converge.

```

7. > i <- 0.006
   > count <- 0
   > f <- (1- (1+i)^(-20))/19 - i
   > tolerance <- 0.000001
   > while (abs(f) > tolerance){
+     f.prime <- (20/19)*((1+i)^(-21))-1
+     i <- i - f/f.prime
+     f <- (1- (1+i)^(-20))/19 - i
+     count <- count+1
+   }
   > i

[1] 0.004939979

> count

[1] 2

```

4.1.5 The repeat loop, and the break and next statements

```

1. > x1 <- 0
   > x2 <- 2
   > repeat{
+     f1 <- x1^3+2*x1^2-7
+     f2 <- x2^3+2*x2^2-7
+     x3 <- (x1+x2)/2
+     f3 <- x3^3+2*x3^2-7
+     if(f3==0){
+       print(x3)
+       break
+     }else{
+       if(f3*f1>0){
+         x1 <- x3
+       }else{
+         x2 <- x3
+       }
+     }
+     if(abs(x1-x2)<0.000001){
+       print((x1+x2)/2)
+       break
+     }
+   }

```

```
[1] 1.428817
```

4.2 Managing Complexity through Functions

4.2.1 What are functions?

1. Type in the function names to return the function. i.e.:

```
> var
function (x, y = NULL, na.rm = FALSE, use)
{
  if (missing(use))
    use <- if (na.rm)
      "complete.obs"
    else "all.obs"
  na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs"))
  if (is.data.frame(x))
    x <- as.matrix(x)
  else stopifnot(is.atomic(x))
  if (is.data.frame(y))
    y <- as.matrix(y)
  else stopifnot(is.atomic(y))
  .Internal(cov(x, y, na.method, FALSE))
}
<environment: namespace:stats>
```

3.

```
> bisection <- function(f, x1, x2){
+   repeat{
+     f1 <- f(x1)
+     f2 <- f(x2)
+     x3 <- (x1+x2)/2
+     f3 <- f(x3)
+     if(f3==0){
+       print(x3)
+       break
+     }else{
+       if(f3*f1>0){
+         x1 <- x3
+       }else{
+         x2 <- x3
+       }
+     }
+   }
+   if(abs(x1-x2)<0.000001){
+     print((x1+x2)/2)
+     break
+   }
+ }
+ }
```

4.3 Miscellaneous Programming tips

1.

```
> factorial(10)
```

```
[1] 3628800
```

```

> factorial(50)
[1] 3.041409e+64
> factorial(100)
[1] 9.332622e+157
> factorial(1000)
[1] Inf

Warning message:
value out of range in 'gammafn'

```

```

3. > fix(compound.interest)
> #change the function to:
> function(P, j, m, n){
+   i.r <- j/m
+   return( P*(1+i.r)^(n*m))
+ }

5. > mortgage.payment <- function(P, i.r, n){
+   R <- P*i.r/(1-(1+i.r)^(-n))
+   return(R)
+ }

```

4.4 Some General Programming Guidelines

4.4.1 Top-down design

```

1. > mergesort <- function(x, decreasing=FALSE){
+   if(decreasing==FALSE){
+     len <- length(x)
+     if (len<2) result <- x
+     else{
+       y <- x[1:(len/2)]
+       z <- x[((len/2)+1):len]
+       y <- mergesort(y)
+       z <- mergesort(z)
+       result <- c()
+       while(min(length(y), length(z))>0){
+         if(y[1]<z[1]){
+           result <- c(result, y[1])
+           y <- y[-1]
+         }else{
+           result <- c(result, z[1])
+           z <- z[-1]
+         }
+       }
+       if(length(y)>0)
+         result <- c(result, y)
+       else
+         result <- c(result, z)
+     }
+   }
+   return(result)
+ }else{

```

```

+     len <- length(x)
+     if (len<2) result <- x
+     else{
+       y <- x[1:(len/2)]
+       z <- x[((len/2)+1):len]
+       y <- mergesort(y, decreasing=TRUE)
+       z <- mergesort(z, decreasing=TRUE)
+       result <- c()
+       while(min(length(y), length(z))>0){
+         if(y[1]>z[1]){
+           result <- c(result, y[1])
+           y <- y[-1]
+         }else{
+           result <- c(result, z[1])
+           z <- z[-1]
+         }
+       }
+       if(length(y)>0)
+         result <- c(result, y)
+       else
+         result <- c(result, z)
+     }
+   return(result)
+ }

```

4.7 Chapter Exercises

1.

```
> directpoly <- function(x, coef){
+   n <- length(coef)
+   result <- 0
+   for(i in 1:n){
+     result <- result+coef[i]*x^(n-i)
+   }
+   return(result)
+ }
```
2.

```
> hornerpoly <- function(x, coef){
+   n <- length(coef)
+   a <- matrix(0 ,length(x), n)
+   a[,n] <- coef[1]
+   for(i in (n-1):1){
+     a[,i] <- a[,i+1]*x+coef[n-i+1]
+   }
+   return(a[, 1])
+ }
```
5. (a)

```
> x1 <- 2.1
> x2 <- 3.1
> count <- 1
> repeat{
+   count <- count+1
+   f1 <- (x1-3)*exp(-x1)
+   f2 <- (x2-3)*exp(-x2)
```

```

+     x3 <- (x1+x2)/2
+     f3 <- (x3-3)*exp(-x3)
+     if(f3==0){
+         break
+     }else{
+         if(f3*f1>0){
+             x1 <- x3
+         }else{
+             x2 <- x3
+         }
+     }
+     if(abs(x1-x2)<0.0000001){
+         break
+     }
+ }
> count
[1] 25

```

```

(b) > x1 <- 2.1
> x2 <- 3.1
> count <- 1
> repeat{
+     count <- count+1
+     f1 <- (x1^2 -6*x1 +9)*exp(-x1)
+     f2 <- (x2^2 -6*x2 +9)*exp(-x2)
+     x3 <- (x1+x2)/2
+     f3 <- (x3^2 -6*x3 +9)*exp(-x3)
+     if(f3==0){
+         break
+     }else{
+         if(f3*f1>0){
+             x1 <- x3
+         }else{
+             x2 <- x3
+         }
+     }
+     if(abs(x1-x2)<0.0000001){
+         break
+     }
+ }

```