Loops

Ken Wu

6/24/2021

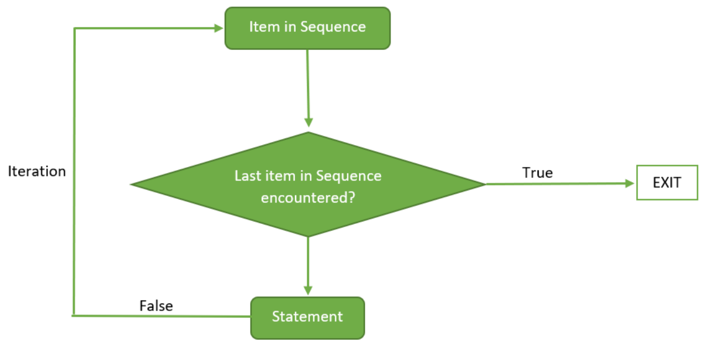
Contents

Control Flow

Loops are an important programming concept, allowing programmers to execute blocks of code repeatedly, usually with varying options. This post will cover the three types of loops— for, while, and repeat. We will then solve some problems using loops to demonstrate the power of iteration in programming. Whenever possible, we attempt to solve problems using different methods, including different types of loops as well as parallel processing. Many of R's functions are vectorized, meaning that the function will operate on all elements of a vector without needing to loop through and act on each element one at a time. We leverage this unique feature of R to show that many problems that seem to involve loops can actually be solved differently in R, although the resulting programs may be harder to intuit.

for loop

```
# Import image
knitr::include_graphics("for loop.png")
```



Basic syntax:

```
for (item in vector) perform_action
```

For each item in vector, perform_action is called once; updating the value of item each time. There are two ways to terminate a for loop early:

- next exits the current iteration
- break exits the entire for loop

```
for (i in 1:10) {
  if (i < 3) {
    next
  }

  print(i)

  if (i > 5) {
    break
  }
}
[1] 3
[1] 4
[1] 5
[1] 6
```

Common pitfalls

1. Use `seq_along(x)` as the vector in `for()` since it always returns a value the same length as x, even when x is a length zero vector:

```
# Declare variables
means <- c()
out <- vector("list", length(means))
# For loop
for (i in seq_along(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

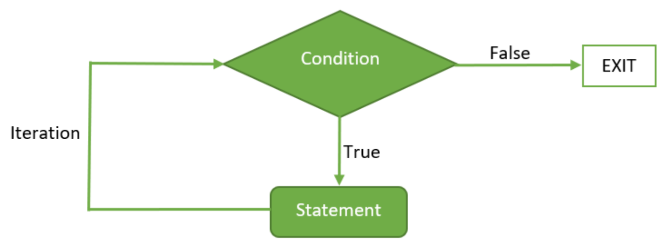
2. When iterating over S3 vectors, loops typically strip the attributes. Use [[to work around this:

```
# Date
xs <- as.Date(c("2020-01-01", "2010-01-01"))
# Loop
for (i in seq_along(xs)) {
  print(xs[[i]] + 10)
}
[1] "2020-01-11"
[1] "2010-01-11"
```

---

while loop

```
# Import image
knitr::include_graphics("while loop.png")
```



Basic syntax:

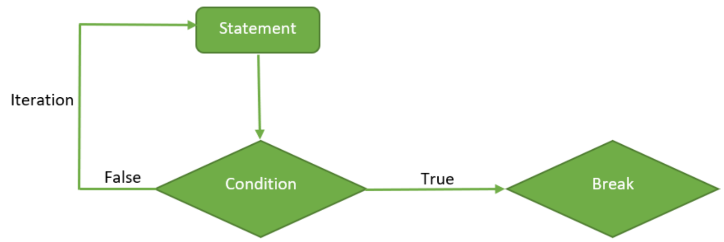
```
while (condition) {
  expression_1
  ...
}
```

When a while command is executed, logical_expression is evaluated first. If it is true, then the group expressions in {} is executed. Control is then passed back to the start of the command: if logical_expression is still TRUE then the grouped expressions are executed again, and so on. For the loop to stop, logical_expression must eventually be FALSE. To achieve this, logical_expression usually depends on a variable that is altered within the grouped expressions.

---

repeat loop

```
# Import image
knitr::include_graphics("repeat loop.png")
```



Basic syntax:

```
repeat{
  expression_1
  ...

  if (condition) {
    break
  }
}
```

It is a simple loop that will run the same statement or a group of statements repeatedly until the stop condition has been encountered. Repeat loop does not have any condition to terminate the loop, a programmer must specifically place a condition within the loop's body and use the declaration of a break statement to terminate this loop. If no condition is present in the body of the repeat loop then it will iterate infinitely.

Examples

Forecast pension growth under compounding interest (for loop)

- Inputs

```
# Annual interest rate
r <- 0.11
# Forecast duration (in years)
term <- 10
# Time between payments (in years)
period <- 1 / 12
# Amount deposited each period
payments <- 100
```

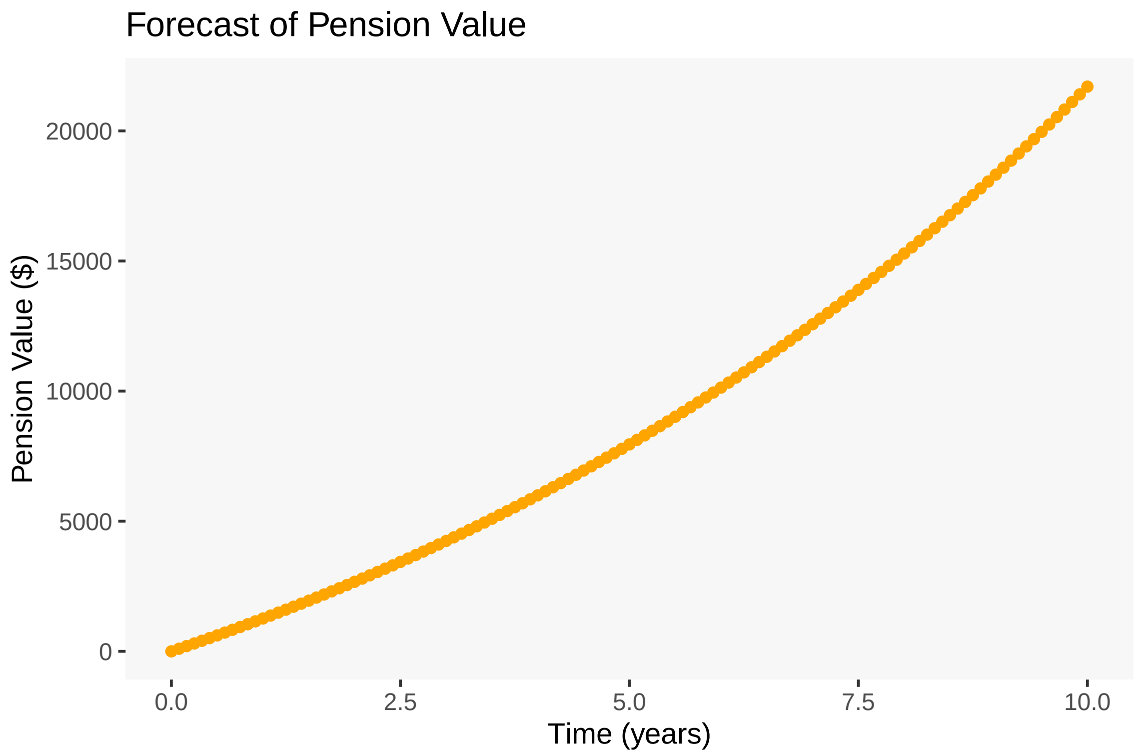
- Calculations. The function `ceiling()` takes a single numeric argument x and returns a numeric vector containing the *smallest integers not less than the corresponding elements of x*. On the other hand, `floor()` takes a single numeric argument x and returns a numeric vector containing the *largest integers not greater than the corresponding elements of x*.

```
# Number of payments
n <- floor(term / period)
# Pre-allocate pension container
pension <- vector(mode = "double", length = n)
```

```r
# Object size
lobstr::obj_size(pension)
1,008 B
# Use seq_along
seq_along(pension)
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120
# For loop (compounded monthly)
for (i in seq_along(pension)) {
  pension[[i + 1]] <- pension[[i]] * (1 + r * period) + payments
}
# New object size
lobstr::obj_size(pension)
1,016 B
```

- Graph the output

```r
# Time
time <- (0:n) * period
# Plot
ggplot(data = tibble(time, pension), mapping = aes(x = time, y = pension)) +
  geom_point(color = "orange") +
  labs(
    title = "Forecast of Pension Value",
    x = "Time (years)", y = "Pension Value ($)"
  ) +
  theme(
    panel.background = element_rect(fill = "grey97"),
    panel.grid = element_blank()
  )
```



Duration of a fixed-payment loan under monthly compounding interest (while loop)

- Inputs

```
# Annual interest rate
r <- 0.11
# Time between repayments (in years)
period <- 1 / 12
# Initial principal
initial_principal <- 1000
# Fixed payment amount
payments <- 12
```

- Calculations

```
# Initialize variables
time <- 0
principal <- initial_principal
# While loop
while (principal > 0) {
  # Time (in years)
  time <- time + period
  # Principal payments
  principal <- principal * (1 + r * period) - payments
}
```

- Output

```
cat("Fixed-payment loan will be repaid in", time, "years\n")
Fixed-payment loan will be repaid in 13.25 years
```

Piece-wise function (Loop and vectorization)

Consider the function $y = f(x)$ defined by

x	≤ 0	$\in (0, 1]$	> 1
$f(x)$	$-x^3$	x^2	$\sqrt{x}$

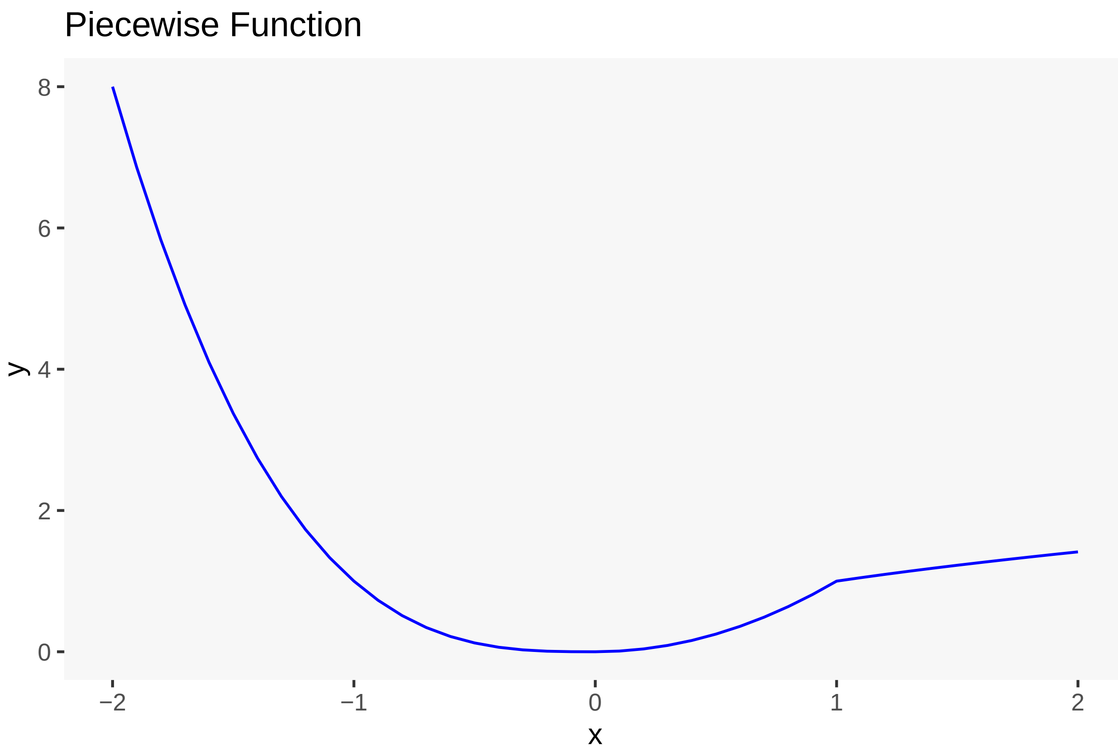
- Implement the function using for loop:

```
# Define x
x_vals <- seq.int(from = -2, to = 2, by = 0.1)
# Initialize sequence
seq <- seq_along(x_vals)
# Pre-allocate container for y values
y_vals <- vector(mode = "double", length = length(x_vals))
# For loop
for (i in seq) {

  # Set x values
  x <- x_vals[[i]]

  if (x <= 0) {
    y <- -x^3
  } else if (x > 0 & x <= 1) {
    y <- x^2
  } else if (x > 1) {
    y <- sqrt(x)
  }

  # Compute y values and store in the container vector
  y_vals[[i]] <- y
}
# Plot the function
ggplot(data = tibble(x_vals, y_vals)) +
  geom_line(mapping = aes(x = x_vals, y = y_vals), color = "blue") +
  labs(
    title = "Piecewise Function",
    x = "x", y = "y"
  ) +
  theme(
    panel.background = element_rect(fill = "grey97"),
    panel.grid = element_blank()
  )
```



- Implement the function using `case_when()` (Note that the function is $-x^3$ when $x \leq 0$; hence the negative sign in front of x)

```
# Vectorization
y_vals_vectorized <- case_when(
  x_vals <= 0 ~ -x_vals^3,
  x_vals > 0 & x_vals <= 1 ~ x_vals^2,
  x_vals > 1 ~ sqrt(x_vals)
)
y_vals_vectorized
 [1] 8.000000 6.859000 5.832000 4.913000 4.096000 3.375000 2.744000 2.197000
 [9] 1.728000 1.331000 1.000000 0.729000 0.512000 0.343000 0.216000 0.125000
[17] 0.064000 0.027000 0.008000 0.001000 0.000000 0.010000 0.040000 0.090000
[25] 0.160000 0.250000 0.360000 0.490000 0.640000 0.810000 1.000000 1.048809
[33] 1.095445 1.140175 1.183216 1.224745 1.264911 1.303840 1.341641 1.378405
[41] 1.414214
```

Sequence (for loop and while loop)

- Let $h(x, n) = 1 + x + x^2 + \cdots + x^n = \sum_{i=0}^{n} x^i$. Let us implement this sum of a geometric sequence using a for loop:

```
# Function
sum_of_sequence_for_loop <- function(x, n) {

  # Initialize sequence
```

```
  seq <- 0:n
  # Pre-allocate container
  terms <- vector(mode = "double", length = (n + 1))
  # Loop
  for (i in seq) {
    terms[[i + 1]] <- x^i
  }

  # Sum
  sum(terms)
}
# Test
sum_of_sequence_for_loop(x = 0.3, n = 55)
[1] 1.428571
sum_of_sequence_for_loop(x = 6.6, n = 8)
[1] 4243336
sum_of_sequence_for_loop(x = 1, n = 8)
[1] 9
```

- Using a while loop:

```
# Function
sum_of_sequence_while_loop <- function(x, n) {

  # Initialize i
  i <- 0
  # Pre-allocate container
  terms <- vector(mode = "double", length = (n + 1))
  # Loop
  while (i <= n) {
    terms[[i + 1]] <- x^i
    i <- i + 1
  }

  # Sum
  sum(terms)
}
# Test
sum_of_sequence_while_loop(x = 0.3, n = 55)
[1] 1.428571
sum_of_sequence_while_loop(x = 6.6, n = 8)
[1] 4243336
sum_of_sequence_while_loop(x = 1, n = 46)
[1] 47
```

- Implement using parallel processing— vectorization

```
# Function
sum_of_sequence_vectorized <- function(x, n) {

  # Create vector of x
  vector_of_x <- rep(x = x, times = n + 1)
```

```
  # Create vector of exponents
  vector_of_exponents <- seq.int(from = 0, to = n, by = 1)

  # Create vector of terms in the sequence
  vector_of_terms <- vector_of_x^vector_of_exponents

  # Find the sum
  sum(vector_of_terms)
}
# Test
sum_of_sequence_vectorized(x = 0.3, n = 55)
[1] 1.428571
sum_of_sequence_vectorized(x = 6.6, n = 8)
[1] 4243336
sum_of_sequence_vectorized(x = 1, n = 46)
[1] 47
```

Geometric and Harmonic Means (for loop and vectorization)

The geometric mean of a vector is defined as follows:

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 x_2 \cdots x_n}$$

- Geometric mean (for loop)

```
geometric_for_loop <- function(x) {

  # Length of vector
  n <- length(x)

  # Warning
  if (is.numeric(x) == FALSE) {
    rlang::abort("Vector is of the wrong type; input must be numeric")
  } else if (n < 2) {
    rlang::abort("Input vector must contain more than 1 element")
  }

  # Initialize first term (as.double() ensures no integer overflow)
  x_val <- as.double(x[[1]])
  # Iterate over the sequence 1:(n - 1)
  # The algorithm involves multiplying the current element i by the next (i + 1) element in x
  # Setting (n - 1) as the last item safeguards against out-of-bounds subsetting of "x"
  seq <- 1:(n - 1)
  # Iterate
  for (i in seq) {
    x_val <- x_val * x[[i + 1]]
```

```r
  }


  # Geometric mean
  (x_val)^(1 / n)
}
# Test
# Create a random vector
x <- sample(x = 1:45, size = 200, replace = TRUE)
# A function from the psych package
psych::geometric.mean(x)
[1] 16.4906
# Our custom function
geometric_for_loop(x)
[1] 16.4906
```

- Geometric mean (vectorization)

```r
geometric_vectorization <- function(x) {

  # Length of vector
  n <- length(x)

  # Warning
  if (is.numeric(x) == FALSE) {
    rlang::abort("Vector is of the wrong type; input must be numeric")
  } else if (n < 2) {
    rlang::abort("Input vector must contain more than 1 element")
  }

  # Product of vector elements
  # The function prod() is primitive
  prod <- prod(x)
  # Geometric mean
  prod^(1 / n)
}
# Test
geometric_vectorization(x)
[1] 16.4906
```

- Harmonic mean (for loop)

```r
harmonic_for_loop <- function(x) {

  # Length of vector
  n <- length(x)

  # Warning
  if (is.numeric(x) == FALSE) {
    rlang::abort("Vector is of the wrong type; input must be numeric")
  } else if (n < 2) {
    rlang::abort("Input vector must contain more than 1 element")
```

```r
  }

  # Initialize x value
  x_val <- as.double(1 / x[[1]])
  # Create sequence
  seq <- 1:(n - 1)
  # Iterate
  for (i in seq) {
    x_val <- x_val + (1 / x[[i + 1]])
  }

  # Harmonic mean
  n / x_val
}
# Test
# A function from the psych package
psych::harmonic.mean(x)
[1] 8.144297
# Our custom function
harmonic_for_loop(x)
[1] 8.144297
```

- Harmonic mean (vectorization)

```r
harmonic_vectorization <- function(x) {

  # Length of vector
  n <- length(x)

  # Warning
  if (is.numeric(x) == FALSE) {
    rlang::abort("Vector is of the wrong type; input must be numeric")
  } else if (n < 2) {
    rlang::abort("Input vector must contain more than 1 element")
  }

  # Find element-wise reciprocals
  x_reciprical <- 1 / x
  # Sum the reciprocals
  sum <- sum(x_reciprical)
  # Harmonic mean
  n / sum
}
# Test
harmonic_vectorization(x)
[1] 8.144297
```

Find the Sum of Every nth Element of a Vector

- Using for loop

```r
# Function
every_nth_element_for_loop <- function(x, n) {

  # Define the nth term
  n <- n
  # Initialize sequence
  seq <- seq_along(x)
  # Initialize counter
  counter <- 0
  # Pre-allocate container
  new_x <- vector(mode = "double", length = length(x))
  # Loop
  for (i in seq) {

    # Count the term
    counter <- counter + 1

    # If counter gets to n, copy that term to the container
    if (counter == n) {
      new_x[[i]] <- x[[i]]

      # Reinitialize counter to zero
      counter <- 0
    }
  }

  # Sum
  new_x
}
# Test vector
x <- sample(x = 1:203, size = 100, replace = TRUE)
x
  [1] 130  57   9  54 154  80 189 148  64  77   8 119  99  72 154 111 144  24
 [19]  39  61  65  16  62  44 135  49 165  99 199 166 111  69 171  27 200 144
 [37]  46  44  80 128 203 158  83 135 152  94  42  61 196 101  66 139 195  76
 [55] 142 101  55   9  82  66  44 158  84  49  82  56  84 161 201   9  39   3
 [73]  44 197  75  93  29 200 114 124  18  57 184 151 108 103 156  76 117 175
 [91]  85 105 114 140  53  23  50 195  41 135
# A vector that contains every thirteenth element of a vector
every_nth_element_for_loop(x = x, n = 13)
  [1]   0   0   0   0   0   0   0   0   0   0   0   0  99   0   0   0   0   0
 [19]   0   0   0   0   0   0   0  49   0   0   0   0   0   0   0   0   0   0
 [37]   0   0  80   0   0   0   0   0   0   0   0   0   0   0   0 139   0   0
 [55]   0   0   0   0   0   0   0   0   0   0  82   0   0   0   0   0   0   0
 [73]   0   0   0   0   0 200   0   0   0   0   0   0   0   0   0   0   0   0
 [91]  85   0   0   0   0   0   0   0   0   0
# Find sum
sum(every_nth_element_for_loop(x = x, n = 13))
[1] 734
```

- Using while loop

```r
# Function
every_nth_element_while_loop <- function(x, n) {

  # Length of vector
  length <- length(x)
  # Initial value
  value <- 0
  # Initialize counter
  counter <- n
  # Loop
  # Use modulo to ensure that, whenver the counter gets to the nth element, the logical evaluates to tr
  while (counter %% n == 0) {

    # Extract the element from x using the index "counter"
    # This counter is every nth element in the vector or the logical above wouldn't have evaluated to t
    # Alter the value by add the nth term
    value <- value + x[[counter]]

    # Increase counter
    counter <- counter + n

    # Exit condition
    if (counter > length) {
      break
    }
  }

  # Sum
  value
}
# Test (This result should corroborate with that of the function above)
every_nth_element_while_loop(x = x, n = 13)
[1] 734
```

- Using sub-setting and seq()

```r
# Function
every_nth_element_subsetting <- function(x, n) {

  # Define the nth term
  n <- n
  # Create a sequence of indices
  seq <- seq.int(from = n, to = length(x), by = n)
  # Sum
  sum(x[seq])
}
# Test
every_nth_element_subsetting(x = x, n = 13)
[1] 734
```

Chart the flow

- Program 1

```
x <- 3 # line 1
for (i in 1:3) { # line 2
  show(x) # line 3
  if (x[[i]] %% 2 == 0) { # line 4
    x[[i + 1]] <- x[[i]] / 2 # line 5
  } else { # line 6
    x[[i + 1]] <- 3 * x[[i]] + 1 # line 7
  } # line 8
} # line 9
[1] 3
[1]  3 10
[1]  3 10  5
show(x) # line 10
[1]  3 10  5 16
```

- line 1: Set x equal to 3, which is a double vector of length 1.
- line 2: Set i to 1.
- line 3: Show x to the screen.
- line 4: Take the first element of x and divide by 2; by default, r returns the dividend if the divisor is larger than the dividend (i.e. $2 > 1$ and so 1 %% 2 is 1). Therefore, (x[[i]] %% 2 == 0) evaluates to FALSE. Proceed to line 7.
- line 7: Carry out the sub-assignment by setting the second element of x to 10.
- line 8: End of else action.
- line 9: End of for loop and return to line 2.
- line 2: Set i to 2.
- line 3: Show x, which is now a length-2 vector $\langle 3, 10\rangle$.
- line 4: The expression (x[[2]] %% 2 == 0) evaluates to TRUE, since 10 divided by 2 is 5 remainder 0. Proceed to line 5.
- line 5: Sub-assign the third element in x as 5; x[[2]] is 10 and 10 divided by 2 is 5.
- line 6: End of if statement and return to line 2.
- line 2: Set i to 3.
- line 3: Show x, which is now a length-3 vector $\langle 3, 10, 5\rangle$.
- line 4: (x[[3]] %% 2 == 0) evaluates to FALSE since x[[3]] is 5 and 5 %% 2 is 2 remainder 1. Proceed to line 7
- line 7: Sub-assign the fourth element of x to 16 since $(3 \times 5) + 1 = 16$.
- line 8: End of else action.
- line 9: End of for loop. The sequence is exhausted. Proceed to line 10.
- line 10: Show x, which is now a length-4 vector $\langle 3, 10, 5, 16\rangle$.

- Program 2

The second program implements the Lotka-Volterra model for a 'predator-prey' system. We suppose that $x(t)$ is the number of prey animals at the start of a year t (rabbits) and $y(t)$ is the number of predators (foxes), then the Lotka-Volterra model is:

$$x(t+1) = x(t) + b_r \cdot x(t) - d_r \cdot x(t) \cdot y(t)$$
$$y(t+1) = y(t) + b_f \cdot d_r \cdot x(t) \cdot y(t) - d_f \cdot y(t)$$

where the parameters are defined by:

- b_r is the natural birth rate of rabbits in the absence of predation;
- d_r is the death rate per encounter of rabbits due to predation;
- d_f is the natural death rate of foxes in the absence of food (rabbits);
- b_f is the efficiency of turning predated rabbits into foxes.

```
# Growth rate of rabbits
br <- 0.04
# Death rate of rabbits due to predation
dr <- 0.0005
# Death rate of foxes in the absence of of food
df <- 0.2
# Efficiency of turning predated rabbits into foxes
bf <- 0.1
# Initial predator/prey populations
x <- 4200
y <- 100
# Model output
while (x > 3900) { # line 1
  cat("x =", x, " y =", y, "\n") # line 2
  x.new <- (1 + br) * x - dr * x * y # line 3
  y.new <- (1 - df) * y + bf * dr * x * y # line 4
  x <- x.new # line 5
  y <- y.new # line 6
} # line 7
x = 4200  y = 100
x = 4158  y = 101
x = 4114.341  y = 101.7979
x = 4069.499  y = 102.3799
x = 4023.962  y = 102.7356
x = 3978.218  y = 102.8587
x = 3932.749  y = 102.7467
```

- line 1: The initial population of rabbits is $x = 4000$. Therefore $(x > 3900)$ evaluates to TRUE. Proceed to line 2.
- line 2: Concatenate and print the populations of predator and prey at state one. The "dash n" in `cat` means start a new line, ensuring that the printed output are printed lines by line successively instead of just one line.
- line 3: Compute the new population of rabbits and bind that object value to the name x.new.
- line 4: Compute the new population foxes and bind that object value to the name y.new.
- line 5: Bind x.new to x.
- line 6: Bind y.new to y.
- line 7: End of while loop. Return to line 1.
- line 1: If (x > 3900) still evaluates to TRUE, repeat as above for state two, three, and so on. If not, end of program.

Find the Minimum of a Vector

```r
find_min_max <- function(x, summary_stat) {

  # Find minimum or maximum
  if (summary_stat == "min") {

    # Initialize minimum value
    x_min <- x[[1]]
    # Loop
    for (i in 2:length(x)) {
      if (x_min > x[[i]]) {
        x_min <- x[[i]]
      }
    }
    # Output
    x_min
  } else if (summary_stat == "max") {

    # Initialize minimum value
    x_max <- x[[1]]
    # Loop
    for (i in 2:length(x)) {
      if (x_max < x[[i]]) {
        x_max <- x[[i]]
      }
    }
    # Output
    x_max
  } else {

    # Warning
    rlang::abort(message = "summary_stat must either be min or max")
  }
}
```

The function above uses if statements and for loops; I need to benchmark to test for performance.

```r
# Test vector
x <- sample(x = 20:1923, size = 1000, replace = FALSE)
# Find min and max
find_min_max(x, summary_stat = "min")
[1] 22
find_min_max(x, summary_stat = "max")
[1] 1923
# Confirm using base R functions
min(x)
[1] 22
max(x)
[1] 1923
```