Lab8

Irlanda Ayon-Moreno

4/18/2018

Lab 8: Simple Loops

Gaston Sanchez

> ## Learning Objectives
>
> - Forget about vectorized code (pretend it doesn't exist)
> - Practice writing simple loops
> - Get familiar with the syntax of a `for` loop
> - Get familiar with the syntax of a `while` loop
> - Get familiar with the syntax of a `repeat` loop
> - Encapsulate loops inside a function call

Introduction

The majority of functions that work with vectors are vectorized. Remember that vectorized operations are calculations that are applied to all the elements in a vector (element-wise operations).

In order to learn about loops and iterations, it's good to forget about vectorized operations in R. This means that you will be asked to write code, using some sort of loop structure, to perform tasks for which there is already a vectorized implementation. For example, in this lab you will have to write code with various types of loops to calculate the mean of a numeric vector. This can easily be done using the function `mean()`. But we don't want you to use `mean()`. We want you to think about control-flow structures, which are essential in any programming activity.

For loops

Let's start with a super simple example. Consider a vector `vec <- c(3, 1, 4)`. And suppose you want to add 1 to every element of `vec`. You know that this can easily be achieved using vectorized code:

```
vec <- c(3, 1, 4)

vec + 1
```

```
## [1] 4 2 5
```

In order to learn about loops, I'm going to ask you to forget about the notion of vectorized code in R. That is, pretend that R does not have vectorized functions.

Think about what you would manually need to do in order to add 1 to the elements in `vec`. This addition would involve taking the first element in `vec` and add 1, then taking the second element in `vec` and add 1, and finally the third element in `vec` and add 1, something like this:

```
vec[1] + 1
vec[2] + 1
vec[3] + 1
```

The code above does the job. From a purely arithmetic standpoint, the three lines of code reflect the operation that you would need to carry out to add 1 to all the elements in `vec`.

From a programming point of view, you are performing the same type of operation three times: selecting an element in `vec` and adding 1 to it. But there's a lot of (unnecessary) repetition.

This is where loops come very handy. Here's how to use a `for ()` loop to add 1 to each element in `vec`:

```
vec <- c(3, 1, 4)

for (j in 1:3) {
  print(vec[j] + 1)
}
```

```
## [1] 4
## [1] 2
## [1] 5
```

In the code above we are taking each `vec` element `vec[j]`, adding 1 to it, and printing the outcome with `print()` so you can visualize the additions at each iteration of the loop.

Your turn: rewrite the `for` loop in order to triple every element in `vec`, and printing the output at each step of the loop:

```
vec <- c(3, 1, 4)

for (j in c(1:3)) {
  print(vec[j] * 3)
}
```

```
## [1] 9
## [1] 3
## [1] 12
```

What if you want to create a vector `vec2`, in which you store the values produced at each iteration of the loop? Here's one possibility:

```
vec <- c(3, 1, 4)
vec2 <- rep(0, length(vec))  # "empty" of zeros vector to be filled in the loop

for (i in c(1:3)) {
  vec2 <- vec[i] * 3
}
```

```
vec2
```

```
## [1] 12
```

```
vec <- c(3, 1, 4)
vec2 <- rep(0, length(vec))  # "empty" of zeros vector to be filled in the loop

for (i in c(1:3)) {
  vec2 <- c(vec2, vec[i] * 3)
}
```

```
vec2
```

```
## [1]  0  0  0  9  3 12
```

```
vec <- c(3, 1, 4)
vec2 <- rep(0, length(vec))  # "empty" of zeros vector to be filled in the loop

for (i in c(1:3)) {
  vec2[i] <- vec[i] * 3
}
```

```
vec2
```

```
## [1]  9  3 12
```

Summation Series

Write a for loop to compute the following two series. Your loop should start at step $k = 0$ and stop at step n. Test your code with different values for n. And store each k-th term at each iteration. Does the series converge as n increase?

$$sum\_k = 0^{n}$$
$$frac12^{k} = 1 +$$
$$frac12 +$$
$$frac14 +$$
$$frac18 +$$
$$dots +$$
$$frac12^{n}$$

```
k <- 0
n <- 10
terms1 <- rep(0, n)
series1 <- 0
for (i in k:n) {
  terms1[i + 1] <- (1/2)^i
  series1 <- series1 + terms1[i + 1]
}
terms1
```

```
##  [1] 1.0000000000 0.5000000000 0.2500000000 0.1250000000 0.0625000000
##  [6] 0.0312500000 0.0156250000 0.0078125000 0.0039062500 0.0019531250
## [11] 0.0009765625
```

```
series1
```

```
## [1] 1.999023
```

$$sum\_k = 0^{n}$$
$$frac19^{k} = 1 +$$
$$frac19 +$$
$$frac181 +$$
$$dots +$$
$$frac19^{n}$$

```
k <- 0
n <- 10
terms2 <- rep(0, n)
series2 <- 0
for (i in k:n) {
  terms2[i + 1] <- (1/9)^i
  series2 <- series2 + terms2[i + 1]
}
terms2
```

```
##  [1] 1.000000e+00 1.111111e-01 1.234568e-02 1.371742e-03 1.524158e-04
##  [6] 1.693509e-05 1.881676e-06 2.090752e-07 2.323057e-08 2.581175e-09
## [11] 2.867972e-10
```

```
series2
```

```
## [1] 1.125
```

Arithmetic Series

Write a for loop to compute the following arithmetic series $a_n = a_1 + (n - 1)d$ when $a_1 = 3$, and $d = 3$. For instance: 3 + 6 + 9 + 12 + ….

$a_n = a_1 + (n - 1)d$

Test your code with different values for n. And store each n-th term at each iteration. Does the series converge as n increase?

```
a1 <- 3
d <- 3
k <- 1
n <- 10
series3 <- 0
for (i in k:n) {
  print(a1 + (i -1) * d)
  series3 <- series3 + (a1 + (i -1) * d)
}
```

```
## [1] 3
## [1] 6
## [1] 9
## [1] 12
## [1] 15
## [1] 18
## [1] 21
## [1] 24
## [1] 27
## [1] 30
```

```
series3
```

```
## [1] 165
```

Geometric Sequence

A sequence such as 3, 6, 12, 24, 48 is an example of a geometric sequence. In this type of sequence, the n-th term is obtained as:

$a_n = a_1 \times r^{n-1}$

where: a_1 is the first term, r is the common ratio, and n is the number of terms.

Write a for loop to compute the sum of the first n terms of: 3 + 6 + 12 + 24 + …. Test your code with different values for n. Does the series converge as n increase?

```
a1 <- 3
r <- 2
n <- 10
terms4 <- rep(0, n)
sequence <- 0
for (i in 1:n) {
  terms4[i] <- a1 * r^(i - 1)
  sequence <- sequence + terms4[i]
}
terms4
```

```
##  [1]    3    6   12   24   48   96  192  384  768 1536
```

```
sequence
```

```
## [1] 3069
```

Sine Approximation

Consider the following series that is used to approximate the function *sin*(*x*):

$$\begin{gathered} sin(x) \\ approxx - \\ fracx^3 3! + \\ fracx^5 5! - \\ fracx^7 7! + \\ dots \end{gathered}$$

Write a `for` loop to approximate *sin*(*x*). Try different number of terms, $n = 5, 10, 50, 100$. Compare your loop with the `sin()` function.

```
sign <- 1
for (i in 1:5) {
  sign <- -1*sign
  print(sign)
}
```

```
## [1] -1
## [1] 1
## [1] -1
## [1] 1
## [1] -1
```

```
x <- 1
n <- 5
term <- rep(0, n)
for (i in 1:n) {
  term[i] <- x^(i + 2)/factorial(i + 2)
}
term
```

```
## [1] 0.1666666667 0.0416666667 0.0083333333 0.0013888889 0.0001984127
```

```
x <- 1
n <- 5
sign <- 1
term <- rep(0, n)
sine <- 0
for (i in 1:n) {
  sign <- -1 * sign
  term[i] <- (x^(i + 2))/factorial(i + 2)
  sine <- sine + sign * term
  print(sine)
}
```

```
## [1] -0.1666667  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 0.00000000 0.04166667 0.00000000 0.00000000 0.00000000
## [1] -0.166666667  0.000000000 -0.008333333  0.000000000  0.000000000
## [1] 0.000000000 0.041666667 0.000000000 0.001388889 0.000000000
## [1] -0.1666666667  0.0000000000 -0.0083333333  0.0000000000 -0.0001984127
```

```
x <- 1
n <- 5
sign <- 1
pow_factorial <- 1
term <- rep(0, n)
sine <- 0
for (i in 1:n) {
  term[i] <- (x^pow_factorial)/factorial(pow_factorial)
  sign <- -1 * sign
  pow_factorial <- i + 2
  sine <- sine + sign * term[i]
  print(sine)
}
```

```
## [1] -1
## [1] -0.8333333
## [1] -0.875
## [1] -0.8666667
## [1] -0.8680556
```

```
term
```

```
## [1] 1.000000000 0.166666667 0.041666667 0.008333333 0.001388889
```

```
x <- 1
n <- 5
sign <- 1
pow_factorial <- 1
term <- rep(0, n)
sine <- 0
for (i in 1:n) {
  term[i] <- sign * (x^pow_factorial)/factorial(pow_factorial)
  sign <- -1 * sign
  pow_factorial <- i + 2
  sine <- sine + term[i]
  print(sine)
}
```

```
## [1] 1
## [1] 0.8333333
## [1] 0.875
## [1] 0.8666667
## [1] 0.8680556
```

```
term
```

```
## [1]  1.000000000 -0.166666667  0.041666667 -0.008333333  0.001388889
```

```
x <- 1
n <- 5
sign <- 1
pow <- 1
sin_sum <- 0

for (k in 1:n) {
  term <- sign * (x^pow) / factorial(pow)
  pow <- k + 2
  sign <- -1 * sign
  sin_sum <- sin_sum + term
  print(sin_sum)
}
```

```
## [1] 1
## [1] 0.8333333
## [1] 0.875
## [1] 0.8666667
## [1] 0.8680556
```

For loop with a matrix

Consider the following matrix `A`:

```
A <- matrix(1:20, nrow = 5, ncol = 4)
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

Say we want to add 1 to all elements in row 1, add 2 to all elements in row 2, add 3 to all elements in row 3, and so on. To do this without using vectorized coe, you need to work with two nested `for()` loops. One loop will control how you traverse the matrix by rows, the other loop will control how you traverse the matrix by columns. Here's how:

```
# empty matrix B
B <- matrix(NA, nrow = 5, ncol = 4)

# for loop to get matrix B
for (i in 1:nrow(A)) {
  for (j in 1:ncol(A)) {
    B[i,j] <- A[i,j] + i
  }
}

B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    7   12   17
## [2,]    4    9   14   19
## [3,]    6   11   16   21
## [4,]    8   13   18   23
## [5,]   10   15   20   25
```

Your turn

Consider the following matrix `X`:

```
set.seed(123)
X <- matrix(rnorm(12), nrow = 4, ncol = 3)
X
```

```
##             [,1]       [,2]       [,3]
## [1,] -0.56047565  0.1292877 -0.6868529
## [2,] -0.23017749  1.7150650 -0.4456620
## [3,]  1.55870831  0.4609162  1.2240818
## [4,]  0.07050839 -1.2650612  0.3598138
```

Write code in R, using loops, to get a matrix `Y` such that the negative numbers in `X` are transformed into squared values, while the positive numbers in `X` are transformed into square root values

```
#empty maxtrix Y
Y <- matrix(NA, nrow(X), ncol(X))

for (i in 1:nrow(X)) {
  for (j in 1:ncol(X)) {
    if (X[i, j] < 0) {
      Y[i, j] <- X[i, j] **2
    }
    else {
      Y[i, j] <- sqrt(X[i, j])
    }
  }
}

Y
```

```
##            [,1]      [,2]      [,3]
## [1,] 0.31413295 0.3595660 0.4717668
## [2,] 0.05298168 1.3096049 0.1986146
## [3,] 1.24848240 0.6789081 1.1063823
## [4,] 0.26553416 1.6003799 0.5998448
```

Dividing a number by 2 multiple times

The following examples involve dividing a number by 2 until it becomes odd.

Using a `repeat` loop

```
# Divide a number by 2 until it becomes odd.
val_rep <- 898128000 # Change this value!

repeat {
  print(val_rep)
  if (val_rep %% 2 == 1) { # If val_rep is odd,
    break                  # end the loop.
  }
  val_rep <- val_rep / 2 # Divide val_rep by 2 since val_rep was even.
  # When the end of the loop is reached, return to the beginning of the loop.
}
```

```
## [1] 898128000
## [1] 449064000
## [1] 224532000
## [1] 112266000
## [1] 56133000
## [1] 28066500
## [1] 14033250
## [1] 7016625
```

Using a `while` Loop

```
# Divide a number by 2 until it becomes odd.
val_while <- 898128000 # Change this value!

while (val_while %% 2 == 0) { # Continue the loop as long as val_while is even.
  print(val_while)
  val_while <- val_while / 2
}
```

```
## [1] 898128000
## [1] 449064000
## [1] 224532000
## [1] 112266000
## [1] 56133000
## [1] 28066500
## [1] 14033250
```

```
print(val_while)
```

```
## [1] 7016625
```

Make a `reduce()` function

Now generalize the above code to create a function `reduce()` which performs the same operation. (You should change very little.)

```
# your reduce() function
reduce <- function(x) {
  while (x %% 2 == 0) { # Continue the loop as long as val_while is even.
  x <- x / 2
  }
  return(x)
}

reduce(898128000)
```

```
## [1] 7016625
```

Average

The average of n numbers $x_1, x_2, \ldots, x_n$ is given by the following formula:

$$barx = frac1n sum_i = 1^n x_i = fracx_1 + x_2 + n dots + x_n$$

Write R code, using each type of loop (e.g. `for` , `while` , `repeat`) to implement the arithmetic mean of the vector `x = 1:100`

```
# avg with for loop

x <- 1:100
n <- length(x)
avg <- 0

for (i in 1:n) {
  avg <- avg + x[i]/n
}
avg
```

```
## [1] 50.5
```

```
# avg with a while loop

x <- 1:100
avg <- 0
i <- 1

while (i <= length(x)) {
  avg <- avg + x[i]/length(x)
  i <- i + 1
}
avg
```

```
## [1] 50.5
```

```
# avg with a repeat loop

x <- 1:100
avg <- 0
i <- 1

repeat {
  avg <- avg + x[i]/length(x)
  i <- i + 1
  if (i == length(x)) {
    break
  }
}
avg
```

```
## [1] 49.5
```

```
# avg with a repeat loop

x <- 1:100
avg <- 0
i <- 1

repeat {
  avg <- avg + x[i]/length(x)
  i <- i + 1
  if (i > length(x)) {
    break
  }
}
avg
```

```
## [1] 50.5
```

Standard Deviation

The sample standard deviation of a list of n numbers $x_1, x_2, \ldots, x_n$ is given by the following formula:

$$SD = sqrtfrac1n - 1 sum_i = 1^n(x_i - barx)^2$$

Write R code, using each type of loop (e.g. `for` , `while` , `repeat`) to implement the sample standard deviation of the vector `x = 1:100`

```
# sd with a for loop

x <- 1:100
n <- length(x)
sum <- 0

for (i in 1:n) {
  sum <- sum + (x[i] - avg)^2
}
sd <- sqrt((1/(n-1)) * sum)
sd
```

```
## [1] 29.01149
```

```
# sd with while loop

x <- 1:100
n <- length(x)
i <- 1
sum <- 0

while (i <= n) {
  sum <- sum + (x[i] - avg)^2
  i <- i + 1
}
sd <- sqrt((1/(n-1)) * sum)
sd
```

```
## [1] 29.01149
```

```
# sd with a repeat loop

x <- 1:100
n <- length(x)
i <- 1
sum <- 0

repeat {
  sum <- sum + (x[i] - avg)^2
  i <- i + 1
  if (i > n) break
}
sd <- sqrt((1/(n-1)) * sum)
sd
```

```
## [1] 29.01149
```

Geometric Mean

The geometric mean of n numbers $x_1, x_2, \ldots, x_n$ is given by the following formula:

$$barx = left(prod_i = 1^n x_i right)^{1/n}$$

Write R code, using each type of loop (e.g. `for` , `while` , `repeat`) to implement the geometric mean of the vector `x = 1:50`

```
# geometric mean with for loop

x <- 1:50
n <- length(x)
product <- 1

for (i in 1:n) {
  product <- product * x[i]
}
(product)^(1/n)
```

```
## [1] 19.48325
```

```
# geometric mean with while loop

x <- 1:50
n <- length(x)
i <- 1
product <- 1

while (i <= n) {
  product <- product * x[i]
  i <- i + 1
}
(product)^(1/n)
```

```
## [1] 19.48325
```

```
# geometric mean with repeat loop

x <- 1:50
n <- length(x)
i <- 1
product <- 1

repeat {
  product <- product * x[i]
  i <- i + 1
  if (i > n) break
}
(product)^(1/n)
```

```
## [1] 19.48325
```

Distance Matrix of Letters

The following code generates a random matrix `distances` with arbitrary distance values among letters in English:

```
# random distance matrix
num_letters <- length(LETTERS) # num_letters = 26
set.seed(123)
values <- sample.int(num_letters) # this is a vector of 26 random numbers
distances <- values %*% t(values) # t() returns the transpose.
                                  # vectors are treated as columns so t() will return a
 1-row matrix
                                  # %*% is matrix multiplication
diag(distances) <- 0 # places 0's in the diagnal
dimnames(distances) <- list(LETTERS, LETTERS) # names the rows and columns
```

The first 5 rows and columns of `distances` are:

```
distances[1:5, 1:5]
```

```
##     A   B   C   D   E
## A   0 160  80 168 184
## B 160   0 200 420 460
## C  80 200   0 210 230
## D 168 420 210   0 483
## E 184 460 230 483   0
```

Consider the following character vector `vec <- c('E', 'D', 'A')`. The idea is to use the values in matrix `distances` to compute the total distance between the letters: that is from `E` to `D`, and then from `D` to `A`:

```
# (E to D) + (D to A)
483 + 168
```

```
## [1] 651
```

Hence, you can say that the word `'E' 'D' 'A'` has a value of 651.

Your Turn: Write a function `get_dist()` that takes two inputs:

- `distances` = the matrix of distance among letters.
- `ltrs` = a character vector of upper case letters.

The function must return a numeric value with the total distance. Also, include a stopping condition—via `stop()`—for when a value in `ltrs` does not match any capital letter. The error message should be `"Unrecognized character"`

Here's an example of how you should be able to invoke `get_dist()`:

```
vec <- c('E', 'D', 'A')
get_dist(distances, vec)
```

And here's an example that should raise an error:

```
err <- c('E', 'D', ')')
get_dist(distances, err)
```

```
A <- matrix(1:12, nrow = 3, ncol = 4)
dimnames(A) <- list(c("A", "B", "C"), c("A", "B", "C", "D")) # list(row names, column na
mes)
A
```

```
##   A B C  D
## A 1 4 7 10
## B 2 5 8 11
## C 3 6 9 12
```

```
dimnames(A)
```

```
## [[1]]
## [1] "A" "B" "C"
##
## [[2]]
## [1] "A" "B" "C" "D"
```

```
# will give the names of rows first
# will give the names of columns second
```

```
dimnames(A)[[1]][1] # row
```

```
## [1] "A"
```

```
dimnames(A)[[2]][3] # column
```

```
## [1] "C"
```

```
A[dimnames(A)[[1]][1], dimnames(A)[[2]][3]]
```

```
## [1] 7
```

```
dimnames(A)[[1]][1] %in% c("A", "B", "C")
```

```
## [1] TRUE
```

```
get_rows <- function(matrix, ltrs) {
  row_index <- c()
  for (i in 1:nrow(matrix)) {
    if (dimnames(matrix)[[1]][i] %in% ltrs) {
      row_index <- c(row_index, dimnames(matrix)[[1]][i])
    }
  }
  return(matrix[row_index, ])
}
```

```
get_rows(distances, c('E', 'D', 'A'))
```

```
##     A   B   C   D   E  F   G   H   I   J   K   L   M   N  O   P  Q   R   S
## A   0 160  80 168 184  8  88 136 192 208 128  56 144 152 16 112 24 176 200
## D 168 420 210   0 483 21 231 357 504 546 336 147 378 399 42 294 63 462 525
## E 184 460 230 483   0 23 253 391 552 598 368 161 414 437 46 322 69 506 575
##     T   U  V   W   X   Y   Z
## A 120  48 32 104  40  96  72
## D 315 126 84 273 105 252 189
## E 345 138 92 299 115 276 207
```

```
simplifymatrix <- function(matrix, ltrs) {
  rows_of_interest <- get_rows(matrix, ltrs)
  col_index <- c()
  for (j in 1:ncol(matrix)) {
    if (dimnames(matrix)[[2]][j] %in% ltrs) {
     col_index <- c(col_index, dimnames(matrix)[[2]][j])
    }
  }
  return(rows_of_interest[ , col_index])
}
```

```
B <- simplifymatrix(distances, c('E', 'D', 'A'))
B
```

```
##     A   D   E
## A   0 168 184
## D 168   0 483
## E 184 483   0
```

```
sum <- 0
for (i in 1:2) {
  sum <- sum + B[c('E', 'D', 'A')[i], c('E', 'D', 'A')[i + 1]]
}
sum
```

```
## [1] 651
```

```
# Putting it ALL together
get_dist <- function(matrix, ltrs) {
  if (any((ltrs %in% LETTERS) == FALSE)) {
    stop("Unrecognized character")
  }
  row_index <- c()
  col_index <- c()
  sum_dist <- 0
  for (i in 1:nrow(matrix)) {
    if (dimnames(matrix)[[1]][i] %in% ltrs) {
      row_index <- c(row_index, dimnames(matrix)[[1]][i])
    }
  }
  for (j in 1:ncol(matrix)) {
    if (dimnames(matrix)[[2]][j] %in% ltrs) {
     col_index <- c(col_index, dimnames(matrix)[[2]][j])
    }
  }
  simplified_matrix <- matrix[row_index, col_index]
  for (k in 1:(length(ltrs) - 1)) {
    sum_dist <- sum_dist + simplified_matrix[ltrs[k], ltrs[k + 1]]
  }
  return(sum_dist)
}
```

```
get_dist2 <- function(matrix, ltrs) {
  if (any((ltrs %in% LETTERS) == FALSE)) {
    stop("Unrecognized character")
  }
  sum_dist <- 0
  for (k in 1:(length(ltrs) - 1)) {
    sum_dist <- sum_dist + matrix[ltrs[k], ltrs[k + 1]]
  }
  return(sum_dist)
}
```

```
get_dist(B, c('E', 'D', 'A'))
```

```
## [1] 651
```

```
get_dist2(B, c('E', 'D', 'A'))
```

```
## [1] 651
```

Test your function with the following character vectors:

- `cal <- c('C', 'A', 'L')`

```
cal <- c('C', 'A', 'L')
get_dist2(distances, cal)
```

```
## [1] 136
```

- `stats <- c('S', 'T', 'A', 'T', 'S')`

```
stats <- c('S', 'T', 'A', 'T', 'S')
get_dist2(distances, stats)
```

```
## [1] 990
```

- `oski <- c('O', 'S', 'K', 'I')`

```
oski <- c('O', 'S', 'K', 'I')
get_dist2(distances, oski)
```

```
## [1] 834
```

- `zzz <- rep('Z', 3)`

```
zzz <- rep('Z', 3)
get_dist2(distances, zzz)
```

```
## [1] 0
```

- `lets <- LETTERS`

```
lets <- LETTERS
get_dist2(distances, lets)
```

```
## [1] 4800
```

- a vector `first` with letters for your first name, e.g. `c('G', 'A', 'S', 'T', 'O', 'N')`

```
first <- c("I", "R", "L", "A", "N", "D", "A")
get_dist2(distances, first)
```

```
## [1] 1457
```

- a vector `last` for your last name, e.g. `c('S', 'A', 'N', 'C', 'H', 'E', 'Z')`

```
last <- c("M", "O", "R", "E", "N", "O")
get_dist2(distances, last)
```

```
## [1] 1061
```

Your turn: Assuming that you already created the objects listed above, now create an R list `strings` like this:

```
# use your own 'first' and 'last' objects
strings <- list(
  cal = cal,
  stats = stats,
  oski = oski,
  zzz = zzz,
  lets = lets,
  first = first,
  last = last
)
```

Write a `for()` loop to iterate over the elements in `strings`, and compute their distances. At each iteration, store the calculated distances in a list called `strings_dists`; this list should have the same names as `strings`.

```
strings[[7]]
```

```
## [1] "M" "O" "R" "E" "N" "O"
```

```
strings_dists <- rep(0, 7)
for (i in 1:7) {
  strings_dists[i] <- get_dist2(distances, strings[[i]])
}
```

How does your list `strings_dists` look like?

```
strings_dists
```

```
## [1]  136  990  834    0 4800 1457 1061
```