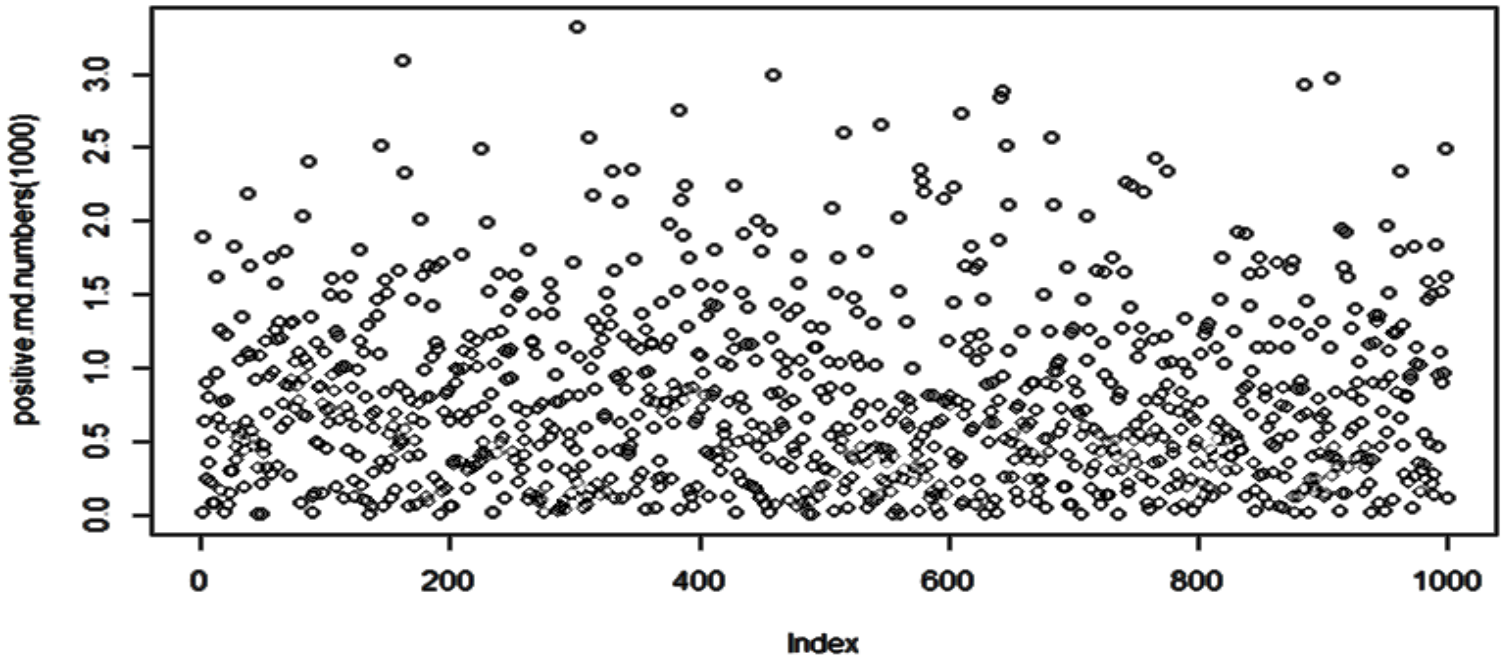
Introduction to Computing for Economics and Management

Lecture 8: Loops Continued



Previous lecture: looping

- The most frequently used looping construct is `for(x in vec) {expression}`
- The for-loop iterates through all elements of the vector `vec`
- For each element of the vector `vec` there will be one iteration of the loop and `expression` is executed
- At each iteration, the variable `x` takes the value of the current element of `vec`
  - First iteration: `x = vec[1]`
  - Second iteration: `x = vec[2]`
  - …

Previous lecture: print variable when iterating

- Let's print out the value of variable `x` when iterating through the vector `vec`

```
> vec <- c(1:5)

> vec
[1] 1 2 3 4 5

> for(x in vec) {print (x)}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Previous lecture: compute length of a vector

- Write our own function for computing the length of a vector

```
## function to compute length of vector vec
vec.length <- function(vec)
{
  # initialize counter
  counter <- 0

  # iterate through vec and increase counter
  for(x in vec) {counter <- counter + 1}

  # return counter
  return(counter)
}
```

Previous lecture: compute Euclidean norm of a vector

```
## compute Euclidean norm of a vector vec
Euclid.norm <- function(vec)
{
  # initialize norm
  norm <- 0

  # compute sum of squared vector elements
  for(x in vec) {norm <- norm + x^2}

  # sqrt of sum
  norm <- sqrt(norm)

  return(norm)
}
```

Previous lecture: square elements of a vector

- We can change the elements of the input vector and return a new vector, e.g. square the elements of a vector

```
## square elements of vector vec
square.vec <- function(vec)
{
  # initialize output vector vec.res
  vec.res <- vector()

  # fill vec.res with squared elements of vec
  for(x in vec) {vec.res <- c(vec.res, x^2)}

  return(vec.res)
}
```

Previous lecture: read data from file

- We read text data from a file into a vector

```
> word.vec <- scan("text.txt", "")
Read 15 items

> word.vec
 [1] "a"       "text"    "consists" "of"
 [5] "a"       "word"    "and"      "another"
 [9] "word"    "and"     "so"       "on"
[13] "and"     "so"      "forth"
```

Previous lecture: if-else

- In order to implement the sorting feature, we need a control flow construct with the following functionality:
  - Check the value of the variable `sort.by.freq`
  - In case the condition `sort.by.freq = TRUE` is satisfied, sort by word frequency else sort alphabetically
- A control flow construct which provide this functionality is the so called if-else statement

```
if (condition) {expression1} else {expression2}
```

- Depending on whether `condition` is true, the result is `expression1` or else `expression2`

Previous lecture: if-else

```
> x <- 2
> y <- if(x == 2) x else x+1
> y
[1] 2

> x <- 3
> y <- if(x == 2) x else x+1
> y
[1] 4

> x <- 3
> y <- if(x == 2){z<-5; x} else {x+1}
> y
[1] 4
> z
```

Previous lecture: Plotting word frequencies

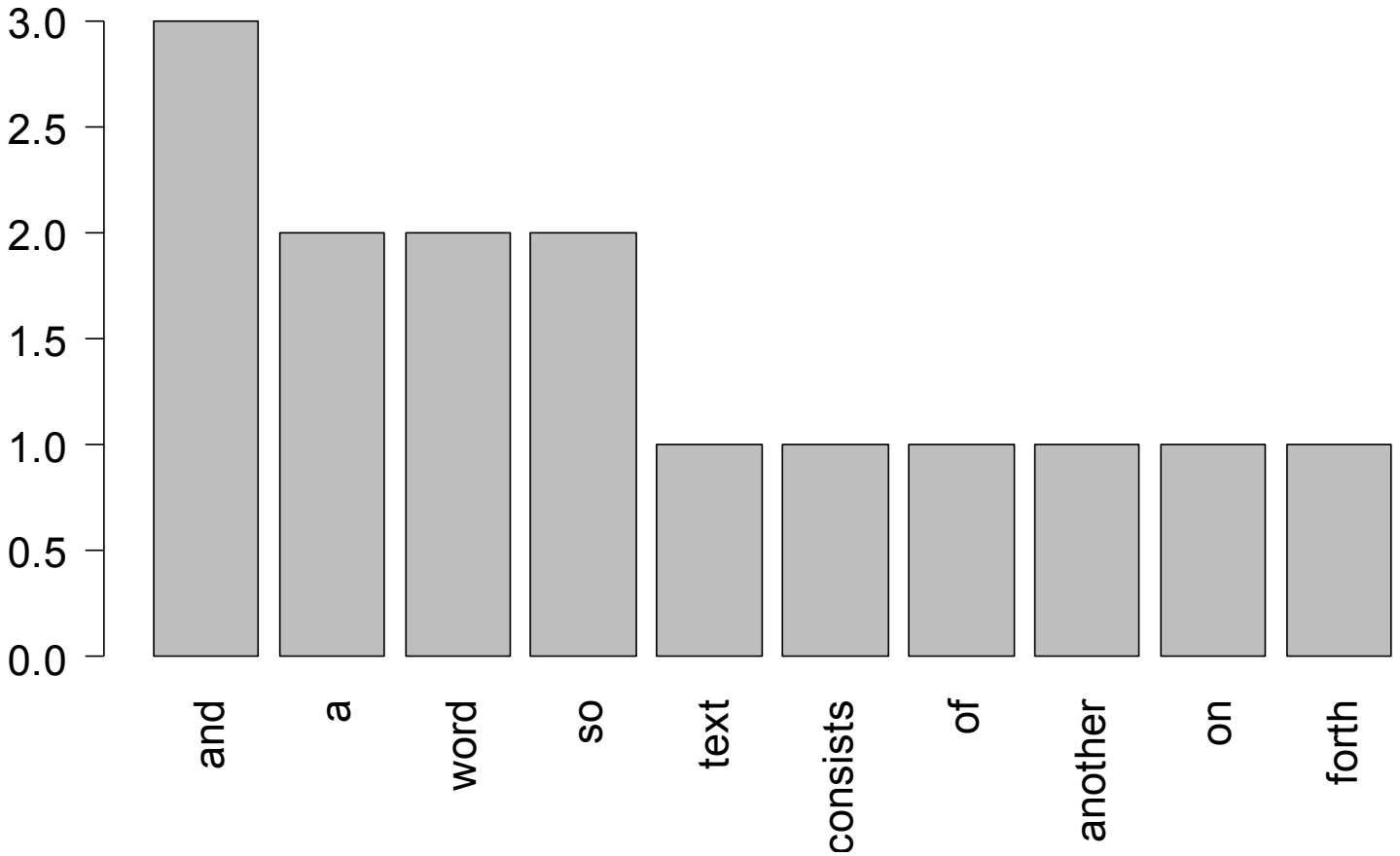
From the resulting list of word positions returned by `findwords` we can easily calculate the word frequencies using `sapply`

```
> word.list <- findwords("text.txt",
sort.by.freq=T)
Read 15 items

> word.freq <- sapply(word.list, length)

> word.freq
     and          a       word         so       text
       3          2          2          2          1
consists         of    another         on     forth
       1          1          1          1          1
```

Previous lecture: plotting word frequencies

- We create a barplot of the word frequencies

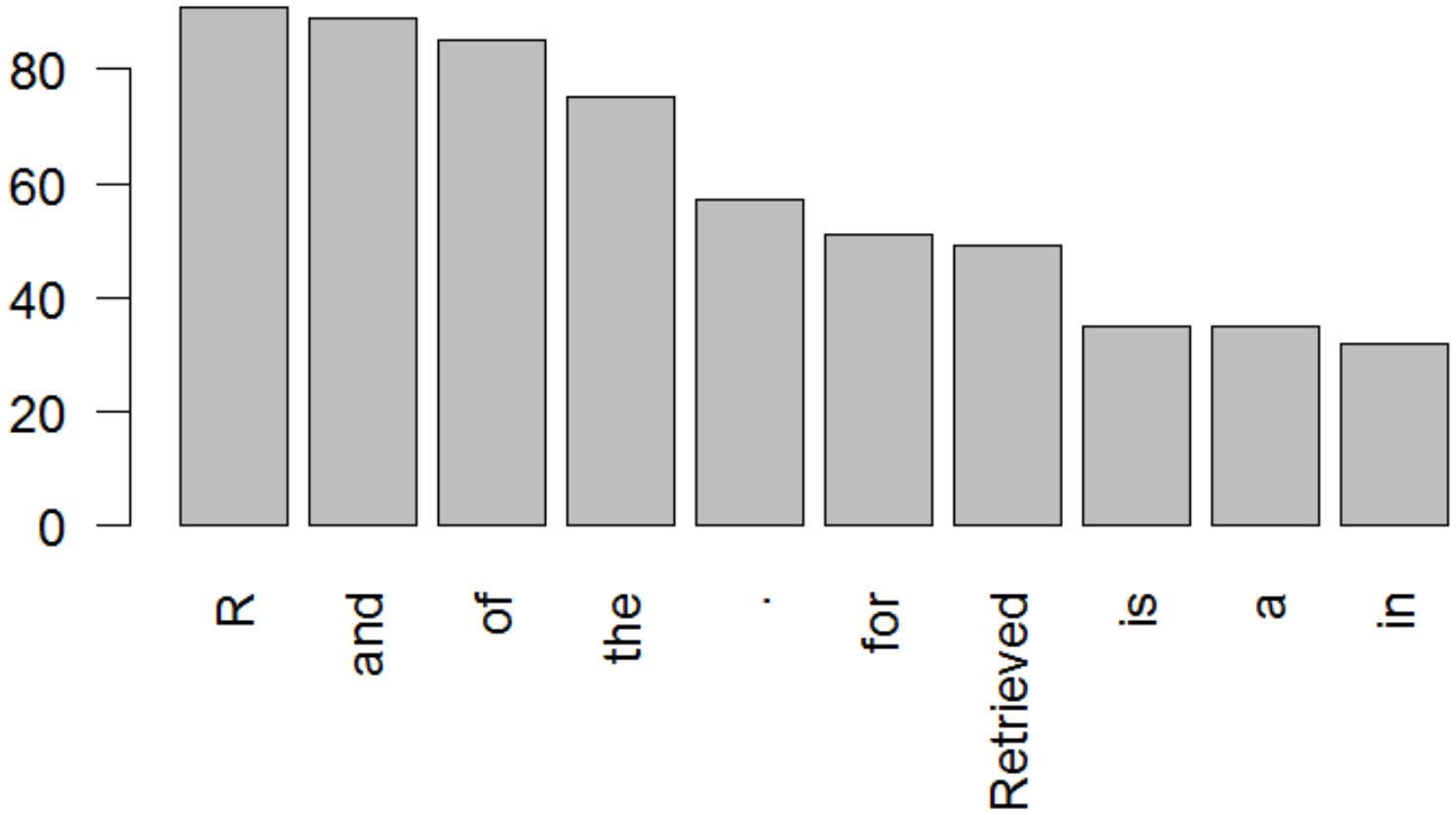
```
> barplot(word.freq, las=2)
```



Previous lecture: word frequency in Wikipedia

```
> word.list <- findwords("R_wikipedia.txt",
sort.by.freq=T)
Read 3395 items

> word.freq <- sapply(word.list, length)

> barplot(word.freq[1:10], las=2)
```

Previous lecture: word frequency in Wikipedia



Program today

- Nested loops
- Alternative loop constructs
  - While
  - Repeat
- Loop control
  - Break
  - Next

Nested loops

- So far we've used simple loops
- In nested loops, an inner loop is placed inside of another outer loop
- At each iteration of the outer loop, the inner loop is processed

Nested loops

- Let's print the values of `i` and `j` form the outer and inner loop respectively

```
for(i in 1:2)
{
  for(j in 1:3)
  {
    print(paste("outer", i, "inner", j))
  }
}
[1] "outer ? inner ?"
[1] "outer ? inner ?"
[1] "outer ? inner ?"
[1] "outer ? inner ?"
[1] "outer ? inner ?"
[1] "outer ? inner ?"
```

Nested loops

- Let's print the values of `i` and `j` form the outer and inner loop respectively

```
for(i in 1:2)
{
  for(j in 1:3)
  {
    print(paste("outer", i, "inner", j))
  }
}
[1] "outer 1 inner 1"
[1] "outer 1 inner 2"
[1] "outer 1 inner 3"
[1] "outer 2 inner 1"
[1] "outer 2 inner 2"
[1] "outer 2 inner 3"
```

Nested loops

We can use the outer counter `i` in the inner loop for adapting the number of iterations in the inner loop

```
string <- ""
for(i in 1:5)
{
  for(j in 1:i)
  {
    string <- paste(string, j)
  }
  print(string)
  string <- ""
}
[1] "?"
[1] "?"
[1] "?"
[1] "?"
[1] "?"
```

Nested loops

We can use the outer counter `i` in the inner loop for adapting the number of iterations in the inner loop

```
string <- ""
for(i in 1:5)
{
  for(j in 1:i)
  {
    string <- paste(string, j)
  }
  print(string)
  string <- ""
}
[1] " 1"
[1] " 1 2"
[1] " 1 2 3"
[1] " 1 2 3 4"
[1] " 1 2 3 4 5"
```

While loop

- A frequently used looping construct is
  `while(condition) {expression}`

- As long as the condition is satisfied, the expression is executed

- Example:

```
> i <- 1
> while(i<5) {i <- i+1}
> i
[1] 5
```

- In the example we observe that the while loop is executed 4 times

While loop

- Be aware that you can easily end up with a endless loop, e.g.

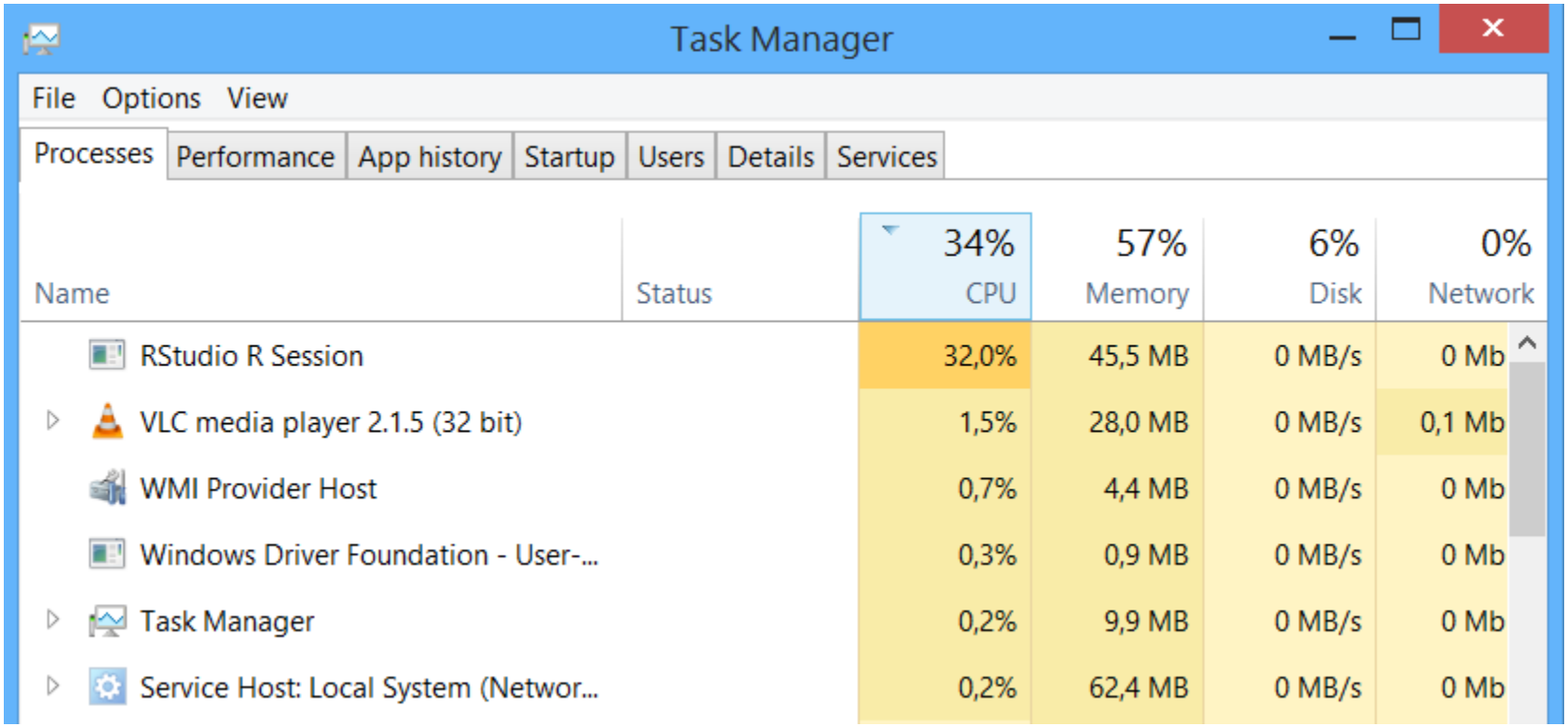
```
> i <- 1
> while(i<5) {i <- i-1}
```

- The condition above is always true and thus the loop will not terminate

- In a while-loop we always have to take care how to end the loop

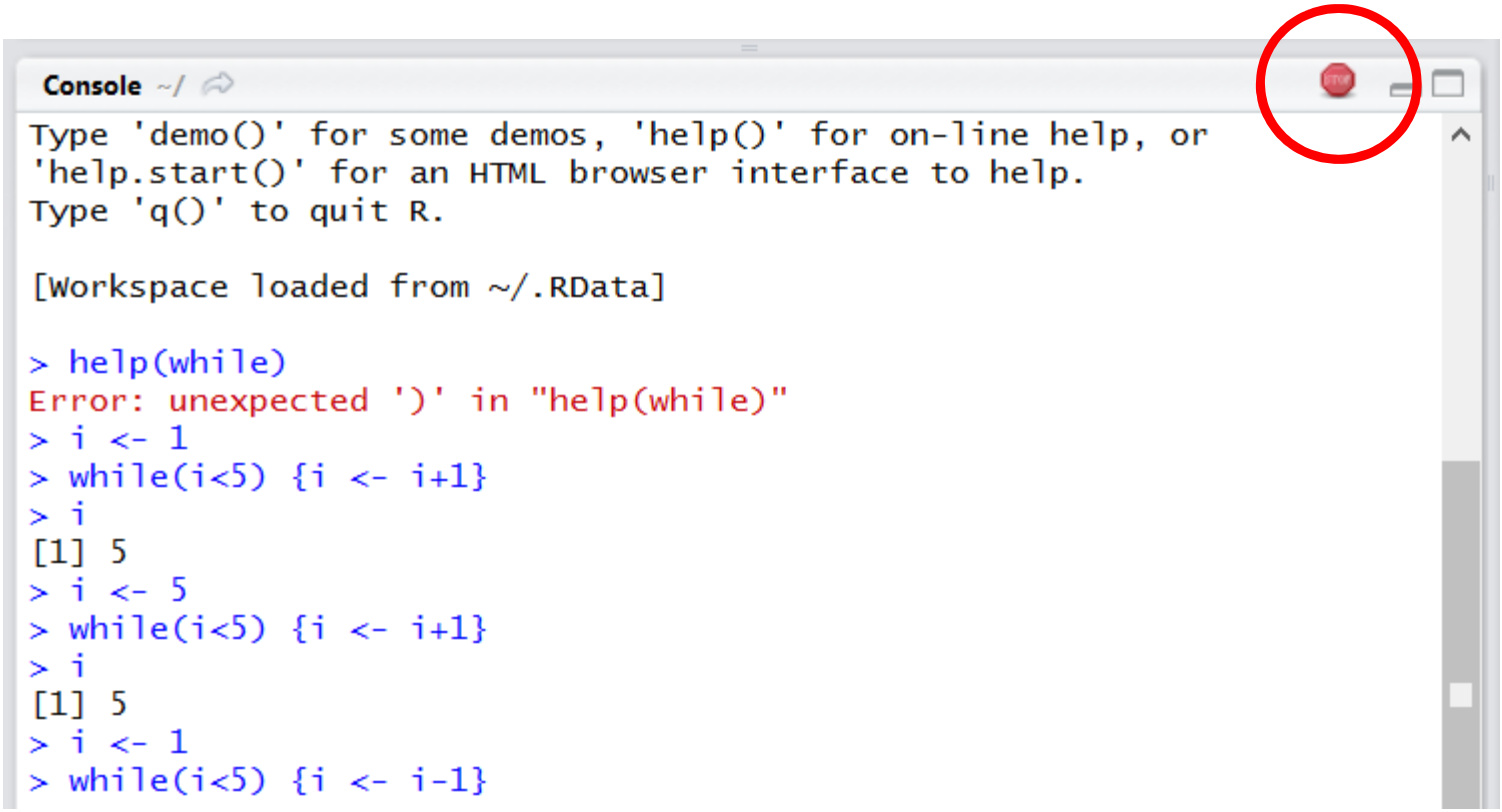
While loop

- Endless loops are a simple way to generate computational load



While loop

- In case of a long processing time, RStudio shows a stop symbol for terminating process



While vs. for loop

- In the previous lecture we've learned several implementations in which we used the for-loop
  - Print vector elements
  - Compute length of a vector
  - Compute Euclidian norm of a vector

- In the next slides, we will learn how to implement these functions with while-loops

- We will compare both implementations

Print vector elements when looping

- We iterate through vector `vec` and print vector's elements

```
vec <- c(7:11)
i <- 1
while(i <= length(vec))
{
  print (vec[i])
  i <- i+1
}
[1] ?
[1] ?
[1] ?
[1] ?
[1] ?
```

Print vector elements when looping

- We iterate through vector `vec` and print vector's elements

```
vec <- c(7:11)
i <- 1
while(i <= length(vec))
{
  print (vec[i])
  i <- i+1
}
[1] 7
[1] 8
[1] 9
[1] 10
[1] 11
```

Print vector elements when looping

Since we operate with the index `i` anyway we can print it together with the vector element using the `paste()` function

```
vec <- c(7:11)
i <- 1
while(i <= length(vec))
{
  print (paste("Element", i, "is", vec[i]))
  i <- i+1
}
[1] "Element ? is ?"
[1] "Element ? is ?"
[1] "Element ? is ?"
[1] "Element ? is ?"
[1] "Element ? is ?"
```

Print vector elements when looping

Since we operate with the index `i` anyway we can print it together with the vector element using the `paste()` function

```
vec <- c(7:11)
i <- 1
while(i <= length(vec))
{
  print (paste("Element", i, "is", vec[i]))
  i <- i+1
}
[1] "Element 1 is 7"
[1] "Element 2 is 8"
[1] "Element 3 is 9"
[1] "Element 4 is 10"
[1] "Element 5 is 11"
```

Compute length of a vector

- If we decrease the while condition by 1 we can use the resulting value of `i` as vector length

```
vec <- c(7:11)
i <- 1
while(i <= length(vec)-1)
{
  i <- i+1
}
i
[1] ?
```

- It is not a very useful implementation since we use `length` anyway in the while condition

Compute Euclidean norm of a vector

```
## compute Euclidean norm of a vector vec
Euclid.norm2 <- function(vec)
{
  # initialize norm
  norm <- 0

  # compute sum of squared vector elements
  i <- 1
  while(i <= length(vec))  # what is the for equivalent?
  {
    norm <- norm + vec[i]^2
    i <- i + 1
  }

  # sqrt of sum
  norm <- sqrt(norm)

  return(norm)
}
```

While vs. for loop

- We can check whether the new function for computing the Euclidean norm delivers the same results like the one from previous lecture where we used the for-loop

```
> Euclid.norm(c(1, 2, 3))
[1] 3.741657
> Euclid.norm2(c(1, 2, 3))
[1] 3.741657

> Euclid.norm(c(sqrt(1), sqrt(3)))
[1] 2
> Euclid.norm2(c(sqrt(1), sqrt(3)))
[1] 2
```

While vs. for loop

- We can compare both implementations in terms of running time
- We use the function `system.time` to measure CPU (and other) times that an expression used
- In order to see a real difference between both implementations, we compute the norm of a vector having 10 million dimensions

While vs. for loop

```
> system.time(Euclid.norm(c(1:10000000)))
   user  system elapsed
   4.87    0.03    4.92

> system.time(Euclid.norm2(c(1:10000000)))
   user  system elapsed
  12.38    0.03   12.41
```

- We observe that CPU time (called `user` time) is almost 3 times higher for the second implementation which uses the while-loop

While vs. for loop

- The CPU time for the while-loop is higher because we have to perform additional operations at each iteration
  - We have to check whether `i` is less than vector length
  - We have to increase `i` by one

```
# compute sum of squared vector elements
i <- 1
  while(i <= length(vec))
  {
    norm <- norm + vec[i]^2
    i <- i + 1
  }

# compute sum of squared vector elements
for(x in vec) {norm <- norm + x^2}
```

Break

- An alternative way to terminate a while-loop is `break`

```
i <- 1
while(i < 10)
{
  i <- i + 1
  break
}
i
[1] ?
```

- The break command causes a termination of the loop after the first iteration although the while-condition is still true

Break

- We can control when to exit the while-loop by using `break` in combination with an `if` statement

```
i <- 1
while(TRUE)
   {
     i <- i + 1
     if(i >= 10) {break}
   }
i
[1] ?
```

Quiz with the while loop

- We often use the while-loop when the number of iterations is not known beforehand

- For example, we want to implement a quiz: we ask the same question again and again until we get the right answer

```
quiz <- function()
{
  answer <- 0
  while(answer != 155)
  {
    answer <- readline("How many students are registered for
this course? ")
    answer <- as.numeric(answer)
  }
  print("Congratulations, 155 is the right number.")
}
```

Quiz with the while loop

```
> quiz()
How many students are registered for this course? 50
How many students are registered for this course? 100
How many students are registered for this course? 200
How many students are registered for this course?
```

- We better provide some help for solving the quiz …

Quiz with the while loop

```
quiz <- function()
{
  answer <- 0
  while(answer != 155)
  {
    answer <- readline("How many students are registered for
this course? ")
    answer <- as.numeric(answer)

    if(answer < 155) {print("No, more students.")}
    if(answer > 155) {print("No, less students.")}
  }
  print("Congratulations, 155 is the right number.")
}
```

Quiz with the while loop

```
> quiz()
How many students are registered for this course? 50
[1] "No, more students."
How many students are registered for this course? 100
[1] "No, more students."
How many students are registered for this course? 200
[1] "No, less students."
...
How many students are registered for this course? 155
[1] "Congratulations, 155 is the right number."
```

Random numbers with the while loop

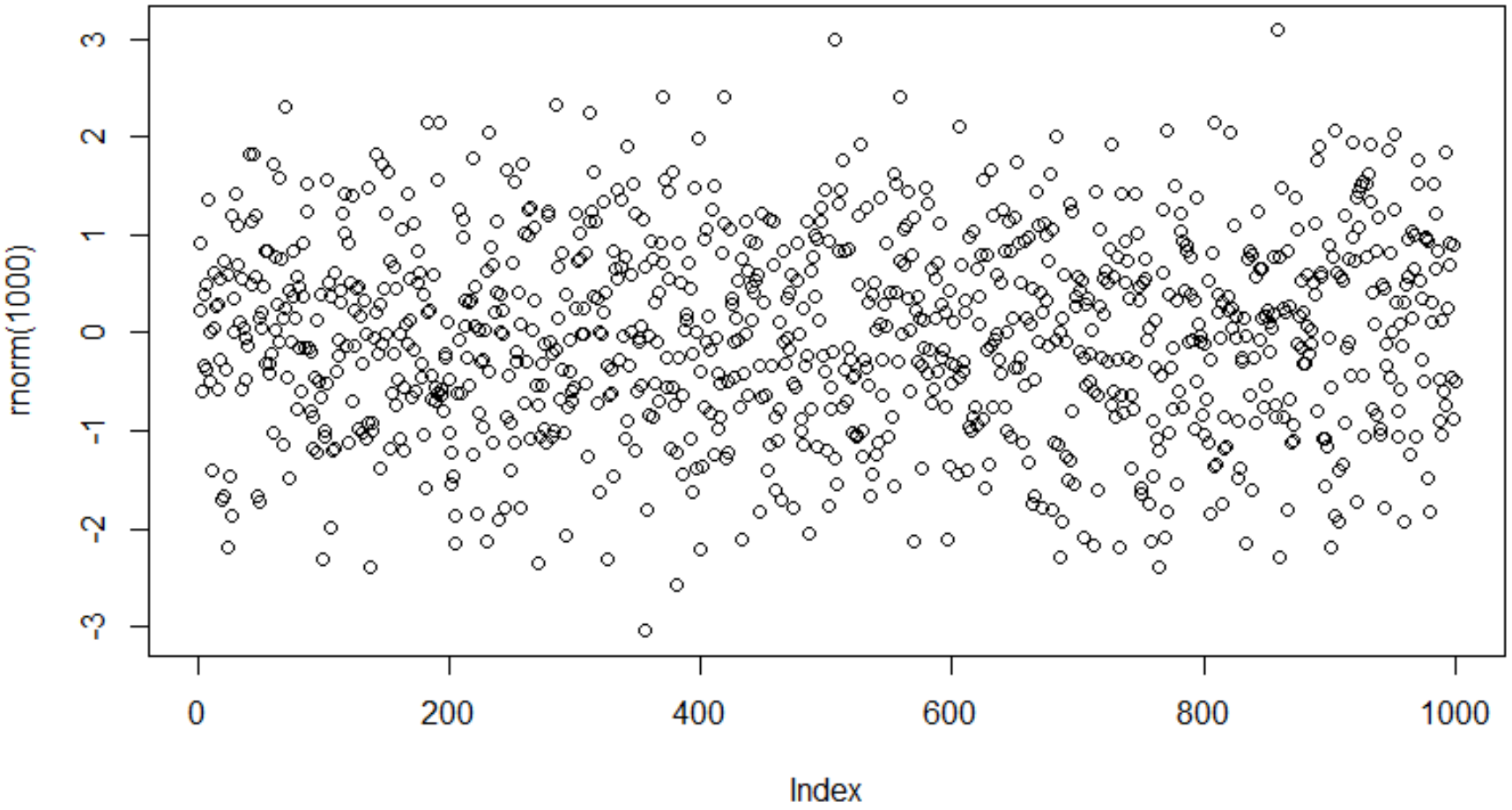
- Another example in which we don't know the number of iterations beforehand is when we want to generate 1000 positive random numbers

- We use the `rnorm` function which generates normal distributed random numbers with mean 0

- Let's start with generating 1000 normal distributed random numbers and plotting them

```
> plot(rnorm(1000))
```

Random numbers with the while loop



Random numbers with the while loop

- From the previous plot we observe that the generated random numbers are distributed around 0 as expected

- Let's check how many positive random numbers we get

```
> rnd.vec <- rnorm(1000)
> length(rnd.vec[rnd.vec>0])
[1] 506

> rnd.vec <- rnorm(1000)
> length(rnd.vec[rnd.vec>0])
[1] 518

> rnd.vec <- rnorm(1000)
> length(rnd.vec[rnd.vec>0])
[1] 493
```

Random numbers with the while loop

- Let's use the for-loop to estimate how many positive random number we get on average when generating 1000 random numbers with mean 0

```
rnd.numbers.above0 <- function(iterations=1000)
{
  nr.above0 <- vector()
  for(i in 1:iterations)
  {
    # generate 1000 normal distributed random numbers
    rnd.vec <- rnorm(1000)

    # save number of positive random numbers
    nr.above0 <- c(nr.above0, length(rnd.vec[rnd.vec>0]))
  }
  return(mean(nr.above0))
}
```

Random numbers with the while loop

- We call `rnd.numbers.above0` several times

```
> rnd.numbers.above0()
[1] 499.858
> rnd.numbers.above0()
[1] 499.137
> rnd.numbers.above0()
[1] 500.514
```

- We observe that mean number of random numbers above 0 is around 500 as expected

- We could generate 2000 random numbers in order to have around 1000 positive numbers but it usually does not give us exactly 1000 positive numbers

Random numbers with the while loop

- In order to generate a particular amount of positive random numbers we better follow a different strategy
- We generate random numbers one at a time and immediately check whether the current number is above 0
- In case we got a positive number we add it to a vector
- We proceed until we have reached our desired number
- Since we don't know beforehand how many iterations we need, we use the while-loop in combination with `break` and `if`

Random numbers with the while loop

```
i <- 1
rnd.vec <- vector()
while(TRUE)
{
  # generate a single random number
  rnd.number <- rnorm(1)
  # if random number is positive add it to vector rnd.vec
  if(rnd.number > 0)
  {
    rnd.vec <- c(rnd.vec, rnd.number)
    i <- i + 1
  }
  # exit after 10 positive random number
  if(i >= 10) {break}
}
```

Random numbers with the while loop

- We test our implementation

```
> rnd.vec
[1] 1.1025410 0.4441786 0.7766904 1.7748363
[5] 1.7230330 0.4275433 0.9531675 0.3045710
[9] 1.6230163

> rnd.vec
[1] 0.3344614 0.7984514 1.4530143 1.0746171
[5] 1.4693036 1.5364879 0.2782448 0.1011667
[9] 1.2492443
```

- As a next step, we transfer our implementation into a function and add a parameter to control the amount of positive random numbers

Random numbers with the while loop

```
### generate n positive random numbers
positive.rnd.numbers <- function(n=1000)
{
  i <- 1
  rnd.vec <- vector()
  while(TRUE)
  {
    # generate a single random number
    rnd.number <- rnorm(1)
    # if random number is positive add it to vector rnd.vec
    if(rnd.number > 0)
    {
      rnd.vec <- c(rnd.vec, rnd.number)
      i <- i + 1
    }
    # return rnd.vec after n positive random number
    if(i > n) {return(rnd.vec)}
  }
}
```

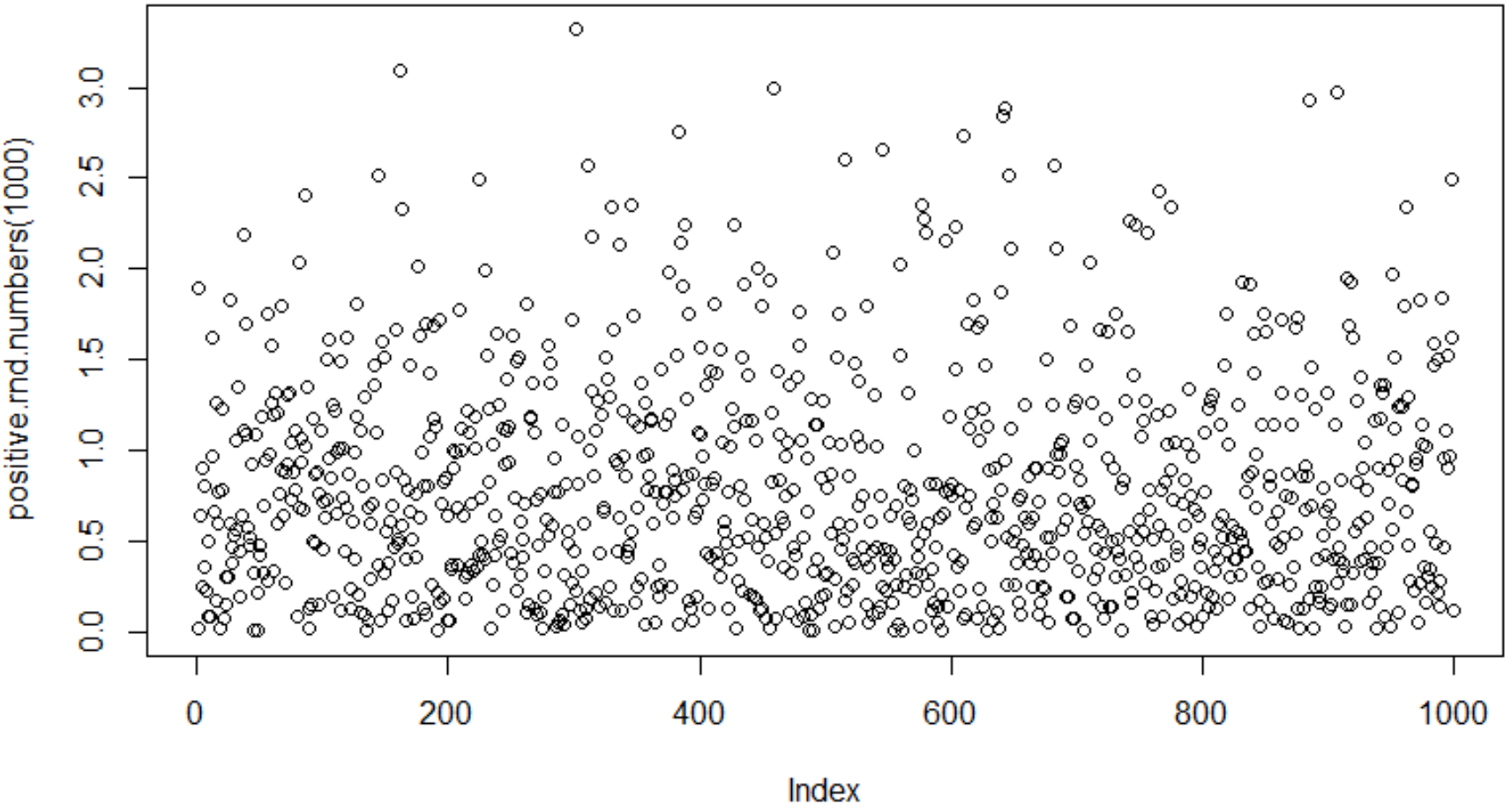
Random numbers with the while loop

- In the function we use `return` to exit the loop since we need the vector as result

- Let's test our function by plotting

```
> plot(positive.rnd.numbers(1000))
```

Random numbers with the while loop



Repeat loop

- Another looping construct is
  `repeat {expression}`
- Expression is executed until the loop is terminated with
  `break`
- In comparison to the while-loop there is no longer a condition test
- We can use it whenever we don't have a condition to test

Repeat loop

- Example in which we use `repeat` instead of `while(TRUE)`

```
i <- 1
repeat
   {
      i <- i + 1
      if(i >= 10) {break}
   }
i
[1] ?
```

Random numbers with the repeat loop

```
### generate n positive random numbers
positive.rnd.numbers <- function(n=1000)
{
  i <- 1
  rnd.vec <- vector()
  repeat
  {
    # generate a single random number
    rnd.number <- rnorm(1)
    # if random number is positive add it to vector rnd.vec
    if(rnd.number > 0)
    {
      rnd.vec <- c(rnd.vec, rnd.number)
      i <- i + 1
    }
    # return rnd.vec after n positive random number
    if(i > n) {return(rnd.vec)}
  }
}
```

Next

- Another useful statement is `next`, which skips the remainder of the current iteration of the loop and proceed directly to the next iteration

- We can use a `next` statement in while-loops, repeat-loops and for-loops as well

```
for(i in 1:3)
{
  print("a")
  next
  print("b")
}
[1] "?"
[1] "?"
[1] "?"
```

Random numbers with the repeat loop and next

- A next statement is useful in our previous function for generating positive random numbers
- We check whether the current random number is negative
- In case of a negative random number we proceed with the next iteration, otherwise we go on and add the current number to our vector

Random numbers with the repeat loop and next

```
### generate n positive random numbers
positive.rnd.numbers <- function(n=1000)
{
  i <- 1
  rnd.vec <- vector()
  repeat
  {
    # generate a single random number
    rnd.number <- rnorm(1)
    # if random number is negative proceed with next iteration
    if(rnd.number < 0) {next}

    rnd.vec <- c(rnd.vec, rnd.number)
    i <- i + 1

    # return rnd.vec after n positive random number
    if(i >= n) {return(rnd.vec)}
  }
}
```

Homework

1. Write a function that uses the while-loop for iterating through a vector and compute the sum of vector's elements

2. Replace the while-loop from the first task with a repeat-loop

3. Implement a quiz with the repeat-loop.

4. Write a function that generates random numbers below 0 by using the repeat-loop.