STAT 33A Workbook 10

CJ HINES (3034590053)

Nov 5, 2020

This workbook is due **Oct Nov 5, 2020** by 11:59pm PT.

The workbook is organized into sections that correspond to the lecture videos for the week. Watch a video, then do the corresponding exercises *before* moving on to the next video.

Workbooks are graded for completeness, so as long as you make a clear effort to solve each problem, you'll get full credit. That said, make sure you understand the concepts here, because they're likely to reappear in homeworks, quizzes, and later lectures.

As you work, write your answers in this notebook. Answer questions with complete sentences, and put code in code chunks. You can make as many new code chunks as you like.

In the notebook, you can run the line of code where the cursor is by pressing `Ctrl` + `Enter` on Windows or `Cmd` + `Enter` on Mac OS X. You can run an entire code chunk by clicking on the green arrow in the upper right corner of the code chunk.

Please do not delete the exercises already in this notebook, because it may interfere with our grading tools.

You need to submit your work in two places:

- Submit this Rmd file with your edits on bCourses.
- Knit and submit the generated PDF file on Gradescope.

If you have any last-minute trouble knitting, **DON'T PANIC**. Submit your Rmd file on time and follow up in office hours or on Piazza to sort out the PDF.

Printing Output

Watch the "Printing Output" lecture video.

Exercise 1

The `readline` function provides a way to collect *input* from the user. Read the documentation for `readline` to get an idea of how it works.

Write code that:

1. Calls `readline` to collect the user's name.
2. Prints `"Hello, NAME!"` where `NAME` is replaced by the collected name.

Write your code so that it works with any input name (that is, don't assume the same name will always be entered).

YOUR ANSWER GOES HERE:

```
x = function() {
  name = readline("What is your name? ")
  sprintf("My name is %s", name)
}

x()
```

```
## What is your name?

## [1] "My name is "
```

For-loops

Watch the "For-loops" lecture video.

Exercise 2

The Pell numbers are a sequence of numbers related to the Fibonacci numbers. Each Pell number is the previous number doubled plus the number before that.

For example, the first two Pell numbers are 0 and 1. The third Pell number is `2*1 + 0`, which is 2.

The first 10 Pell numbers are:

```
0 1 2 5 12 29 70 169 408 985
```

Write a loop that computes the first `n` Pell numbers.

Test your loop with `n = 30`.

YOUR ANSWER GOES HERE:

```
n = 30
pell = c(0, 1, numeric(n - 2))

for (i in 2:n) {
  pell[i + 1] = (pell[i] * 2) + pell[i - 1]
}

pell
```

```
##  [1]            0            1            2            5           12
##  [6]           29           70          169          408          985
## [11]         2378         5741        13860        33461        80782
## [16]       195025       470832      1136689      2744210      6625109
## [21]     15994428     38613965     93222358    225058681    543339720
## [26]   1311738121   3166815962   7645370045  18457556052  44560482149
## [31] 107578520350
```

Exercise 3

The Pell-Lucas numbers are computed the same way as the Pell numbers, but the first two numbers are 2 and 2.

So the first 5 Pell-Lucas numbers are:

```
2 2 6 14 34
```

Write a function `compute_pell` that can compute Pell numbers or Pell-Lucas numbers. Your function should have a parameter `n` that controls how many numbers are computed, and a parameter `initial` that controls the first two numbers in the sequence.

For example, the call `compute_pell(10, c(0, 1))` should return the first 10 Pell numbers. The call `compute_pell(10, c(2, 2))` should return the first 10 Pell-Lucas numbers.

Test that your function can compute both Pell numbers and Pell-Lucas numbers for a few different values of `n`.

Hint: Reuse your code from Exercise 2. You should only need to make a few small changes to turn it into a function.

YOUR ANSWER GOES HERE:

```
compute_pell = function(n, initial) {
  pell = c(initial, numeric(n - 2))
  for (i in 2:n) {
  pell[i + 1] = (pell[i] * 2) + pell[i - 1]
  }
  pell
}

#Test Pell
compute_pell(10, c(0, 1))
```

```
##  [1]    0    1    2    5   12   29   70  169  408  985 2378
```

```
compute_pell(20, c(0, 1))
```

```
##  [1]        0        1        2        5       12       29       70      169
##  [9]      408      985     2378     5741    13860    33461    80782   195025
## [17]   470832  1136689  2744210  6625109 15994428
```

```
compute_pell(30, c(0, 1))
```

```
##  [1]            0            1            2            5           12
##  [6]           29           70          169          408          985
## [11]         2378         5741        13860        33461        80782
## [16]       195025       470832      1136689      2744210      6625109
## [21]     15994428     38613965     93222358    225058681    543339720
## [26]   1311738121   3166815962   7645370045  18457556052  44560482149
## [31] 107578520350
```

```r
#Test Pell-Lucas
compute_pell(10, c(2, 2))
```

```
## [1]    2    2    6   14   34   82  198  478 1154 2786 6726
```

```r
compute_pell(20, c(2, 2))
```

```
##  [1]        2        2        6       14       34       82      198      478
##  [9]     1154     2786     6726    16238    39202    94642   228486   551614
## [17]  1331714  3215042  7761798 18738638 45239074
```

```r
compute_pell(30, c(2, 2))
```

```
##  [1]            2            2            6           14           34
##  [6]           82          198          478         1154         2786
## [11]         6726        16238        39202        94642       228486
## [16]       551614      1331714      3215042      7761798     18738638
## [21]     45239074    109216786    263672646    636562078   1536796802
## [26]   3710155682   8957108166  21624372014  52205852194 126036076402
## [31] 304278004998
```

Exercise 4

The Pell-Lucas numbers and Pell numbers are interesting because you can use them to approximate the square root of 2. The approximation is a Pell-Lucas number divided by two times the corresponding Pell number.

For example, the fourth Pell-Lucas number is 14, and the fourth Pell number is 5, so `14 / (2 * 5)`, or 1.4, is an approximation for the square root of 2.

Use your function `compute_pell` to compute the first 100 Pell-Lucas numbers and Pell numbers. Then use vectorized operations to divide the Pell-Lucas numbers by two times the corresponding Pell numbers.

Does the approximation get better or worse for larger Pell/Pell-Lucas numbers?

Historical Note: This approximation was first discovered by Indian mathematicians around 300 BC.

YOUR ANSWER GOES HERE:

```r
compute_pell = function(n, initial) {
  pell = c(initial, numeric(n - 2))
  for (i in 2:n) {
  pell[i + 1] = (pell[i] * 2) + pell[i - 1]
  }
  pell
}

compute_pell(100, c(2, 2)) / (2 * compute_pell(100, c(0, 1)))
```

```
##   [1]      Inf 1.000000 1.500000 1.400000 1.416667 1.413793 1.414286 1.414201
##   [9] 1.414216 1.414213 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [17] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [25] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
```

```
##  [33] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [41] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [49] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [57] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [65] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [73] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [81] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [89] 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214 1.414214
##  [97] 1.414214 1.414214 1.414214 1.414214 1.414214
```

```r
#The approimation gets better for larger Pell/Pell-Lucas numbers.
```

Loop Indices

Watch the “Loop Indices” lecture video.

No exercises for this section.

Preallocation

Watch the “Preallocation” lecture video.

Exercise 5

In the context of computer programming, *benchmarking* means timing code to see how long it takes to run.

The R package microbenchmark provides a function `microbenchmark` that can benchmark R code.

For example, to benchmark `(1:100) * 4`, you can run:

```r
# Don’t forget to install microbenchmark first!

library(microbenchmark)

microbenchmark(a = {
  (1:100) * 4
})
```

```
## Unit: nanoseconds
##  expr min  lq    mean median  uq   max neval
##     a 603 654 1364.66  792.5 989 13999   100
```

As usual, the curly braces `{` are optional if the code you want to benchmark is only one line long.

You can also use `microbenchmark` to benchmark multiple expressions at once, for comparison:

```r
microbenchmark(a = {
  (1:100) * 4
}, b = {
  (1:100) + 4
})
```

```
## Unit: nanoseconds
##  expr min    lq    mean median   uq   max neval
##     a 604 713.5 2090.01  878.5 2718 26227   100
##     b 601 679.0 1723.55  803.5 1116 36975   100
```

Use the microbenchmark package to benchmark the "BAD" and "GOOD" example from the lecture video.

Benchmark with three different values of `n` (testing both the "BAD" and "GOOD" example for each value). About how much faster is the "GOOD" example?

Hint 1: The `microbenchmark` function tries to automatically select appropriate units for the timings. Make sure to pay attention to the units so that your comparisons are valid! You can also use the `unit` parameter to override the automatic unit selection.

Hint 2: For accuracy, the `microbenchmark` function runs the code multiple times in order to compute timing statistics. The default is 100 times. This may be too many times for very slow code; you can use the `times` parameter to adjust how many runs are used. To avoid inaccurate statistics, make sure `times` is at least 30.

YOUR ANSWER GOES HERE:

```
microbenchmark(BAD = {
  x = c()
for (i in 1:1e5) {
  x = c(x, i * 2)
  }
x
}, GOOD = {
  n = 1e5
  x = numeric(n)
  for (i in seq_len(n)) {
    x[i] = i * 2
  }
  },
times = 30L, unit = "s"
)
```

```
## Unit: seconds
##  expr         min          lq        mean      median          uq
##   BAD 29.955203707 31.423540973 31.906535459 32.042355781 32.606020827
##  GOOD  0.005543751  0.005823785  0.006209223  0.006060927  0.006591345
##          max neval
##  33.194024404    30
##   0.007639373    30
```

Loops Example

Watch the "Loops Example" lecture video.

No exercises for this section.