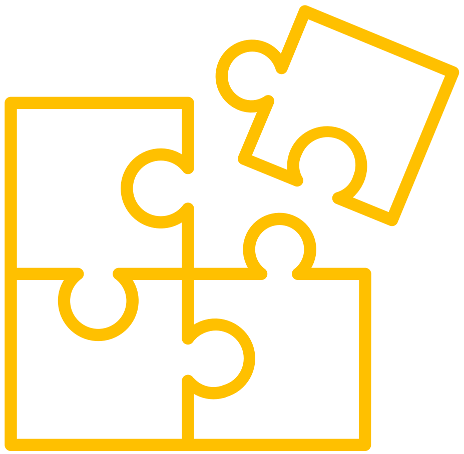
Measuring and Improving Performance

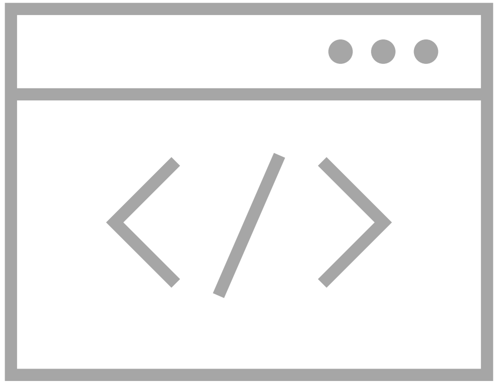
Carmen Garro
Luke Smith

What is performance?

How quickly can the computer undertake a particular task given a code



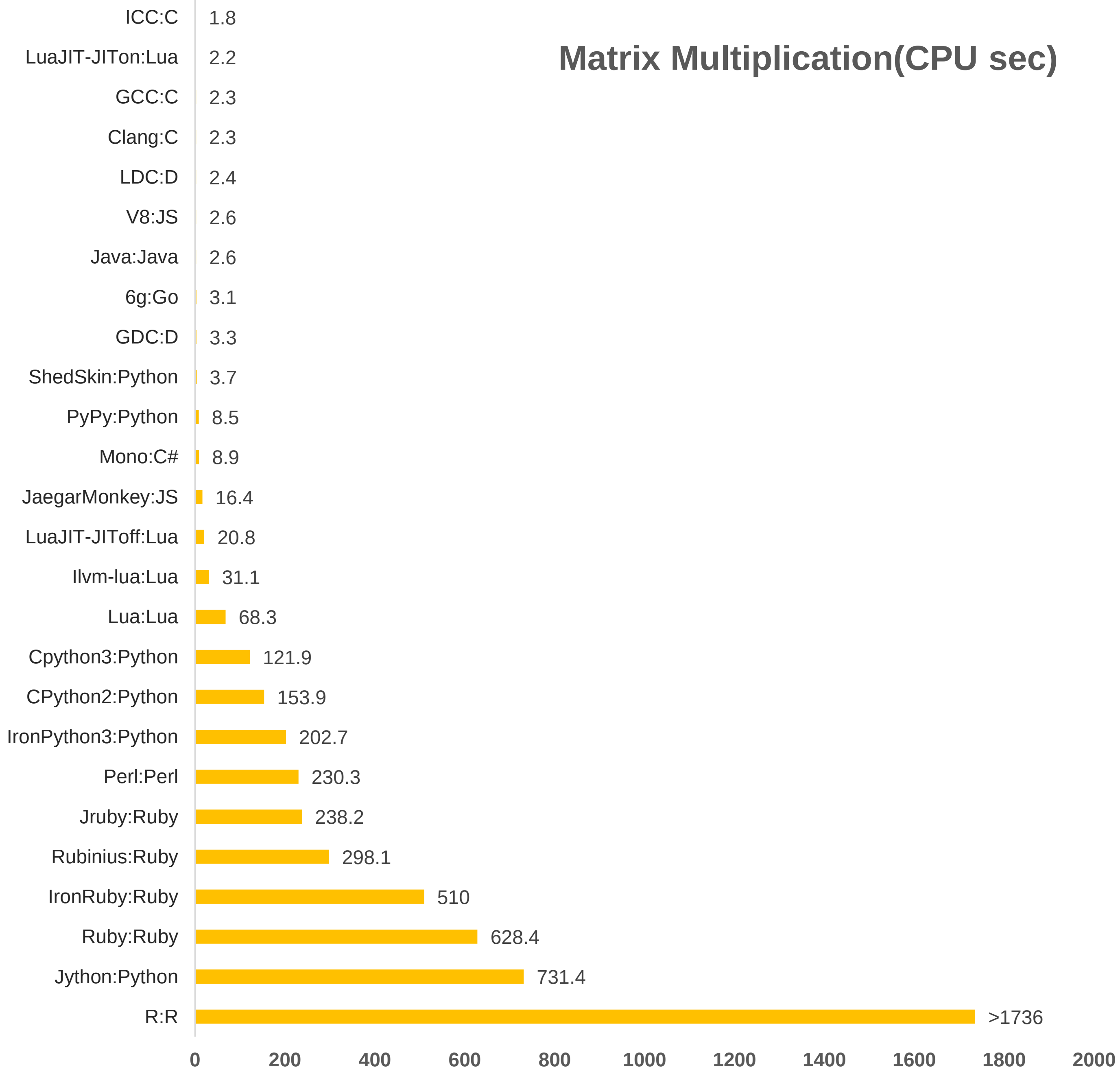
**Balance between code efficiency and programmer productivity*



Is R an efficient coding language?

☑ Language designed to make data analysis and statistics easier for people.

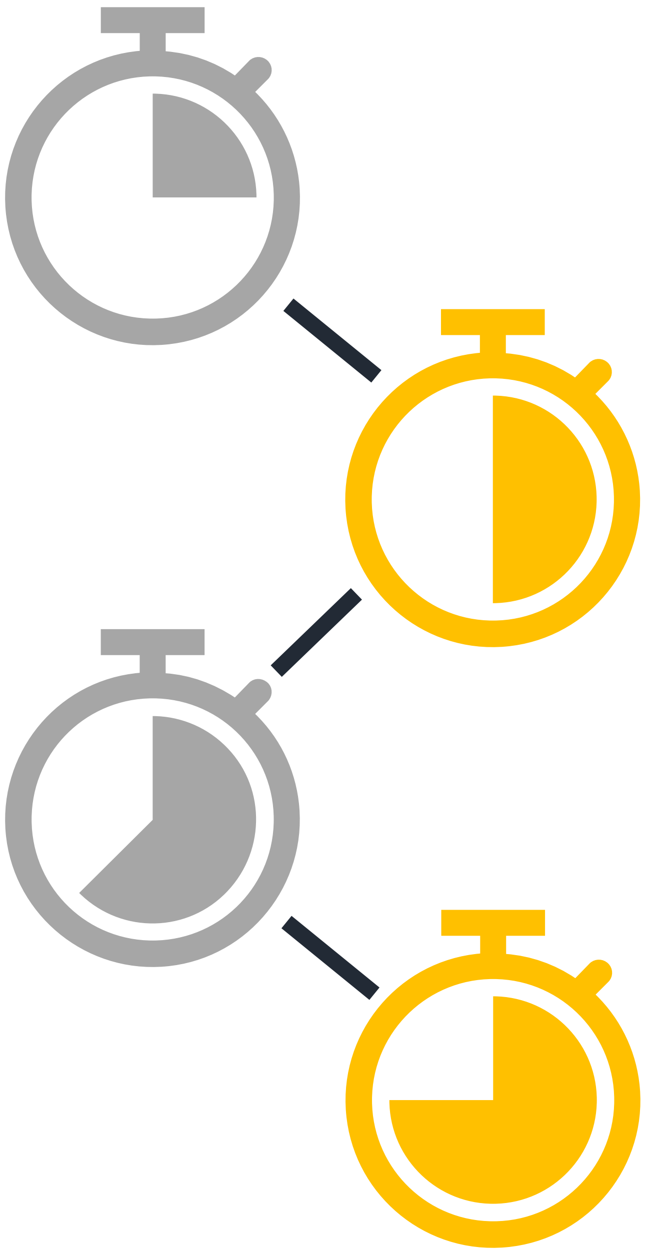
☒ It is not a language designed with an efficient performance in mind.



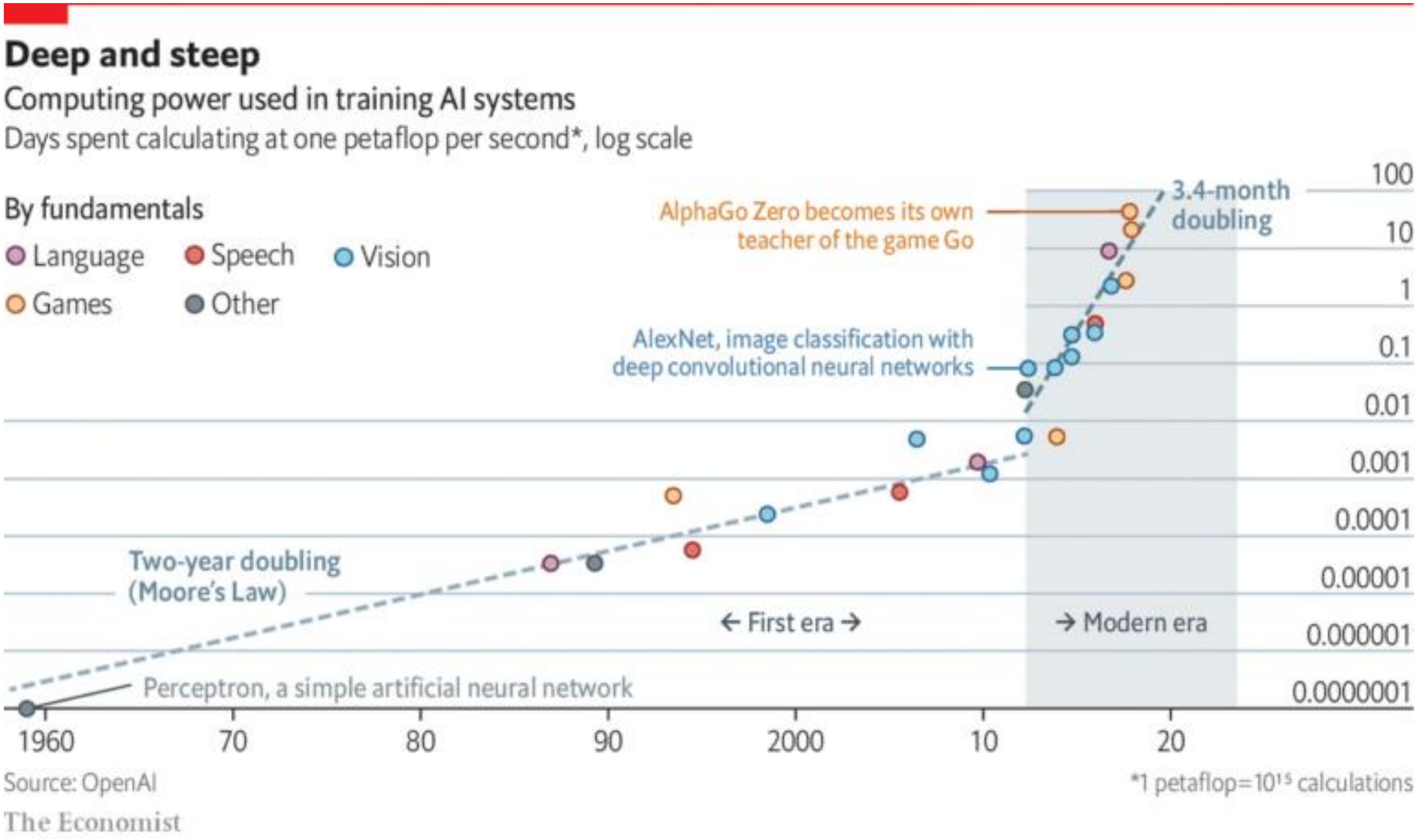
Source: Programming Languages Benchmark (2011) Available at: https://attractivechaos.github.io/plb/

Why is improving performance important?

- The less efficient the code, the more time it takes processing data
  - i.e., reduce bottlenecks in your code
- Less memory usage
- CO_2 emissions from energy usage with high intensity computing



Environmental Considerations



Environmental Considerations

Training models is much more energy intensive than **using** them

"Aligning artificial intelligence with climate change mitigation" from Professor Lynn Kaack

Relevant at **intense** levels of computing, think Facebook and Google

Unlikely to affect your locally produced code for class

Increasingly important as AI and machine learning continue to **increase** in prevalence and usage

Google's machine translation system may process more than 100 billion words per day
Facebook's datacenters are re-trained anywhere from hourly to multi-monthly

Important to keep in mind for your **future** career

How to make your code more efficient?

1. Start by identifying “bottle necks”
2. Experiment with alternatives to find faster code

1. Profiling
2. Benchmark

```
```{r, message=FALSE,warning=FALSE}
library(profvis)
library(bench)
```
```

Profiles

Your code profile will measure key factors in <u>each</u> line of code

- Run-time
- Memory
- Garbage collector

```{r}
library(unvotes)
```

Percentage rate of equal votes of two countries starting a specific year

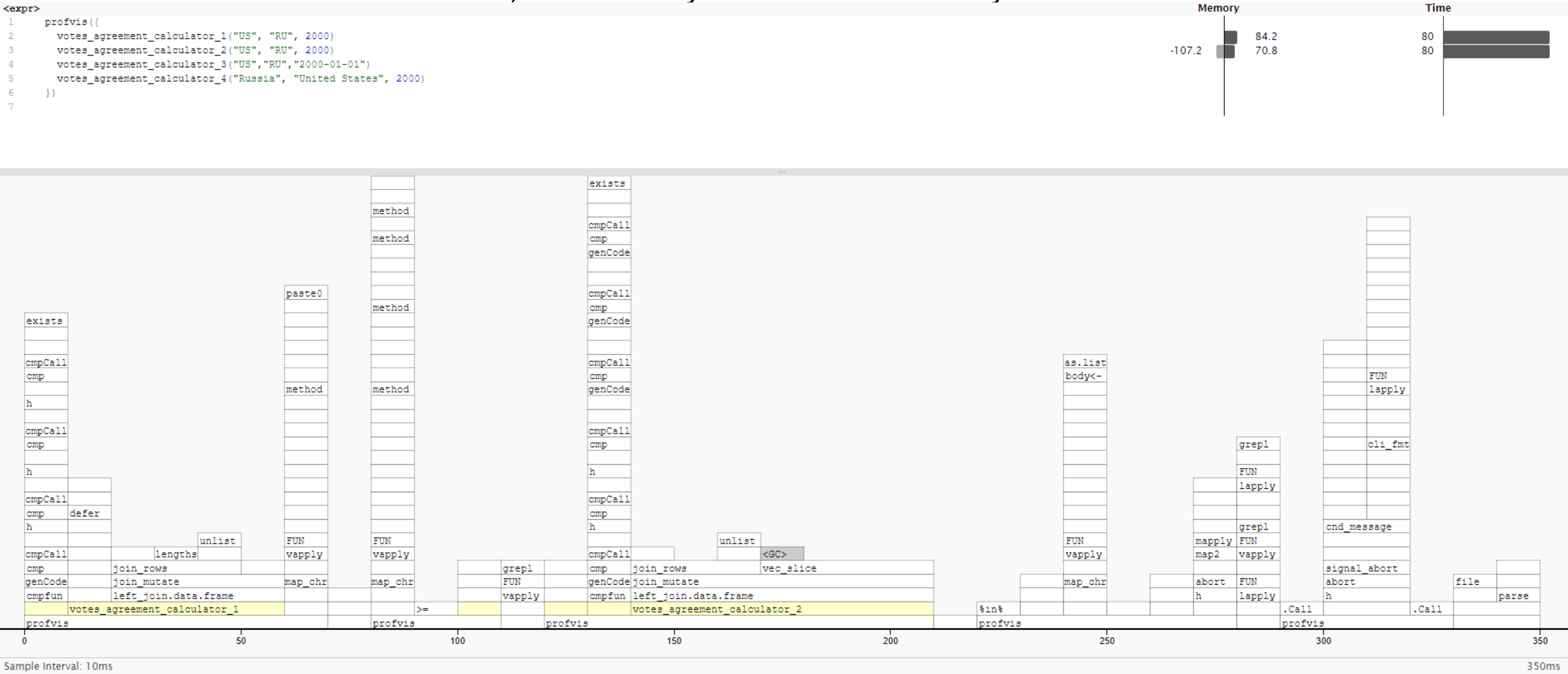
```{r, message=FALSE, warning=FALSE}

profvis({
  votes_agreement_calculator_1("US", "RU", 2000)
  votes_agreement_calculator_2("US", "RU", 2000)
  votes_agreement_calculator_3("US","RU","2000-01-01")
  votes_agreement_calculator_4("Russia", "United States", 2000)
})
```

Visualizing Profiles

Vis output shows the source code, overlaid with bar graphs for memory and execution time for each line of code.

The bottom pane displays a flame graph showing the full call stack. It allows to see when objects are called more than one time. This is also displayed in the data tab, which let's you zoom interactively.



Benchmark

```{r, message=FALSE,warning=FALSE}
library(bench)
```

Always uses the highest precision APIs available for each operating systems (often nanoseconds)

Tracks memory allocations for each expression

Tracks the number and type of R garbage collections per expression iteration

Verifies equality of expression results by default, to avoid accidentally benchmarking inequivalent code

Uses adaptive stopping by default, running each expression for a set amount of time rather than for a specific number of iterations

Expressions are run in batches and summary statistics are calculated after filtering out iterations with garbage collections, to isolate the effects of garbage collection on running time.

The time and memory usage are returned as custom objects which have human readable formatting for display and comparisons.

There is also support for plotting with ggplot2, including custom scales.

Benchmark output

returns the results as a tibble

Pay attention to the units. It is useful to know how many times a function needs to run before it takes a second. If a microbenchmark takes:

* 1 ms, then one thousand calls take a second
* 1 micros, then one million calls take a second
* 1 nanos, then one billion calls take a second

min, mean, median, mex, and itr/sec. These summarise the time taken by the expression. Focus on the minimum (best possible running time) and the median (the typical time).

```
## # A tibble: 2 × 6
##   expression                                          min    median itr/se…¹ mem_a…²
##   <bch:expr>                                     <bch:tm> <bch:tm>    <dbl> <bch:b>
## 1 function_1("US", "RU", 2000)                      2.31s    2.31s    0.433   331MB
## 2 function_2("Russia", "United States", 2000)     458ms 480.35ms    2.08    243MB
## # … with 1 more variable: `gc/sec` <dbl>, and abbreviated variable names
## #   ¹`itr/sec`, ²mem_alloc
```

mem_alloc tells you the amount of memory allocated by the first run, and n_gc() tells you the total number of garbage collections over all runs. These are useful for assessing the memory usage of the expression.

What’s next? Improving Performance

- Approaches to improve code
  - Code organization
  - Checking for existing solutions
  - Functions do as little work as possible
  - Vectorize code
- Balanced approach to code optimization

Code Organization

~~~~
```{r, message=FALSE,warning=FALSE}
library(bench)
```
~~~~

Bench” package

- Compare function speeds

    Which executes more quickly? How much memory is utilized?

Function 1

Function 2

- Example:
    - MDS Students’ functions for Problem Set #2 for IDS

~~~~
```{r, message=FALSE, warning=FALSE}

#Packages specifically for the example functions to work properly
library(tidyverse)
library(knitr)
library(dplyr)
library(countrycode)
library(tidyr)
library(lubridate)
library(rmarkdown)
library(unvotes)

```
~~~~

Code Organization – Function #1

```
#Function #1
# Create votes_agreement_calculator function
function_1 <- function(ccode_a, ccode_b, year_min){

  # Create joined data set with necessary columns; create year column from date column; select relevant v
  un <- un_votes %>%
    left_join(un_roll_calls, by = "rcid") %>%
    mutate(year = format(date,"%Y"), date = NULL) %>%
    select(rcid, country_code, vote, year) %>%
    filter(!is.na(country_code)) %>%
    filter(country_code %in% c(ccode_a, ccode_b))

  # Pivot countries as columns and take values from the respective voting decision
  un_votes_wide <- un %>%
    pivot_wider(names_from = "country_code", values_from = "vote")

  # Calculate agreement_share between ccode_a and ccode_b b since year_min
  agreement_share <- un_votes_wide %>%
    mutate(agreement = un_votes_wide[ccode_a] == un_votes_wide[ccode_b]) %>% #add agreement column
    filter(year >= year_min, !is.na(agreement)) %>% #discard NAs in agreement variable
    summarise(agreement_share = mean(agreement))%>% #calculate mean() of agreement variable
    as.numeric() #convert to numeric

  return(round(agreement_share, digits = 3))
}
```

Code Organization – Function #2

```
#Function #2
function_2 <- function(country_1, country_2, year_min) {
  #Here we pull in the necessary data frames directly from the 'unvotes' package, assuming that the above
  un_joined_func <- un_votes %>%
    inner_join(un_roll_calls, by = "rcid") %>%
    group_by(year = year(date), country)
  #Here we create a data frame specifically for country_1 and rename their "vote" column accordingly
  df_1 <- un_joined_func %>%
    filter(country %in% country_1, year >= year_min) %>%
    rename(country_1_vote = vote)
  #We create a second data frame here for country_2 that does the same as above
  df_2 <- un_joined_func %>%
    filter(country %in% country_2, year >= year_min) %>%
    rename(country_2_vote = vote)
  #Now, we merge these two new data frames into one so that we can compare their newly named vote columns
    joined_dfs <- df_1 %>%
    inner_join(df_2, by = "rcid")
  #Below, we count the number of times that the two countries agree with each other in the UN
  country_vote_agreement <- count(joined_dfs, country_1_vote == country_2_vote)
  #Now we take the average and round it down to three decimal places
  average_country_vote_agreement <-   round((country_vote_agreement$n[2]/(country_vote_agreement$n[1]+cou
  #Lastly, we create a list including the two countries' names and how often they voted with each other
  returned_list <- c(country_1, country_2, average_country_vote_agreement)
  return(average_country_vote_agreement)
}
```

Code Organization – Function Comparison

- Utilize “bench” to compare the two functions:

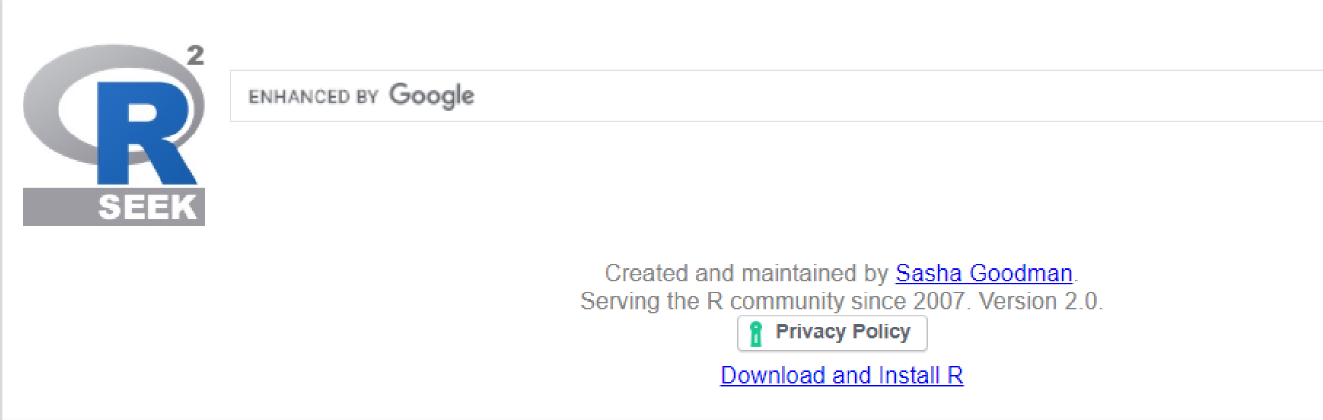
```
bench::mark(
  function_1("US", "RU", 2000),
  function_2("Russia", "United States", 2000)
)
```

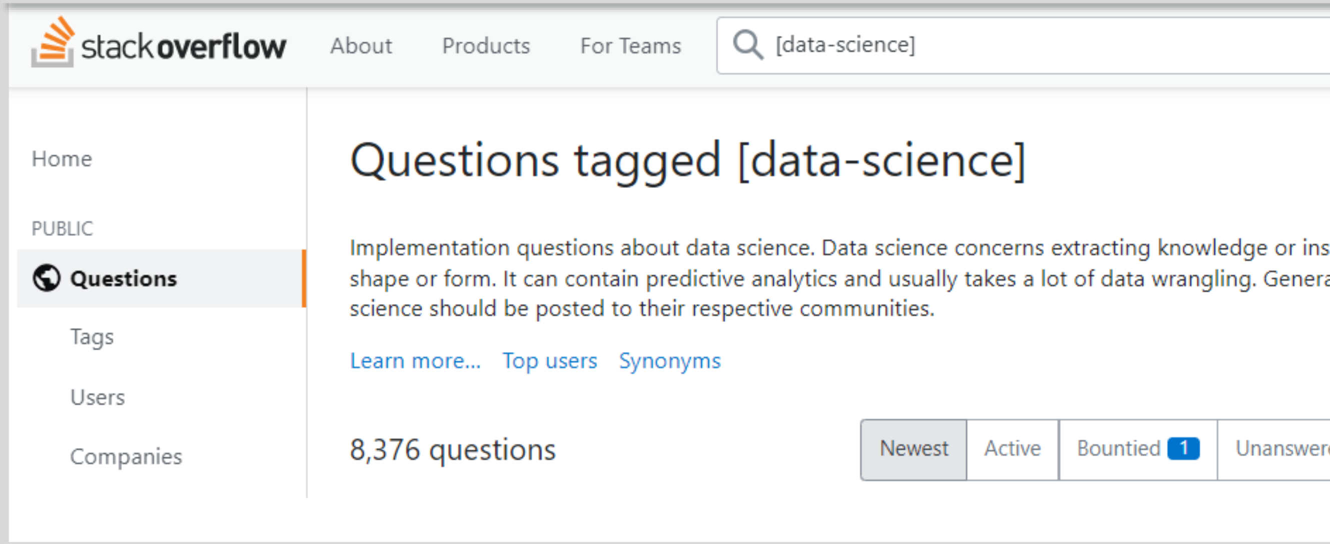
- Observe the results:

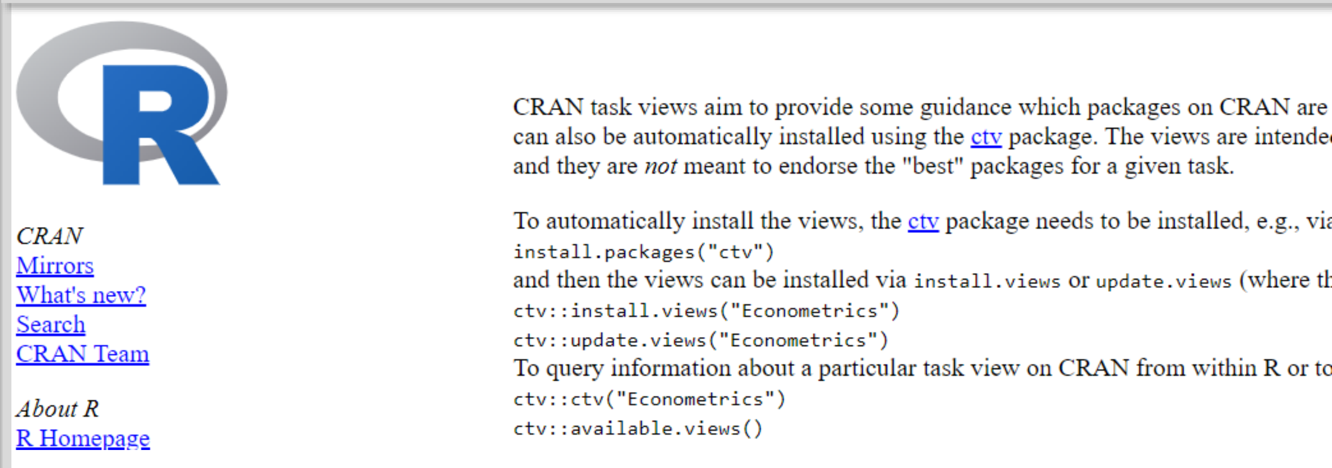
```
## # A tibble: 2 × 6
##   expression                                      min   median itr/se…¹ mem_a…²
##   <bch:expr>                                 <bch:tm> <bch:tm>    <dbl> <bch:b>
## 1 function_1("US", "RU", 2000)                  2.31s    2.31s    0.433   331MB
## 2 function_2("Russia", "United States", 2000)   458ms 480.35ms    2.08    243MB
## # … with 1 more variable: `gc/sec` <dbl>, and abbreviated variable names
## #   ¹`itr/sec`, ²mem_alloc
```

Checking for Existing Solutions

- Outside resources:
  - Online resources:
    - Rseek at rseek.org
    - Stackoverflow at stackoverflow.com
    - CRAN Task Views at cran. Rstudio.com/web/views/
  - In-person resources:
    - Colleagues
    - Professors
    - Students







Efficient Functions

Have your functions do as little work as possible

- Low runtime
- Low memory usage

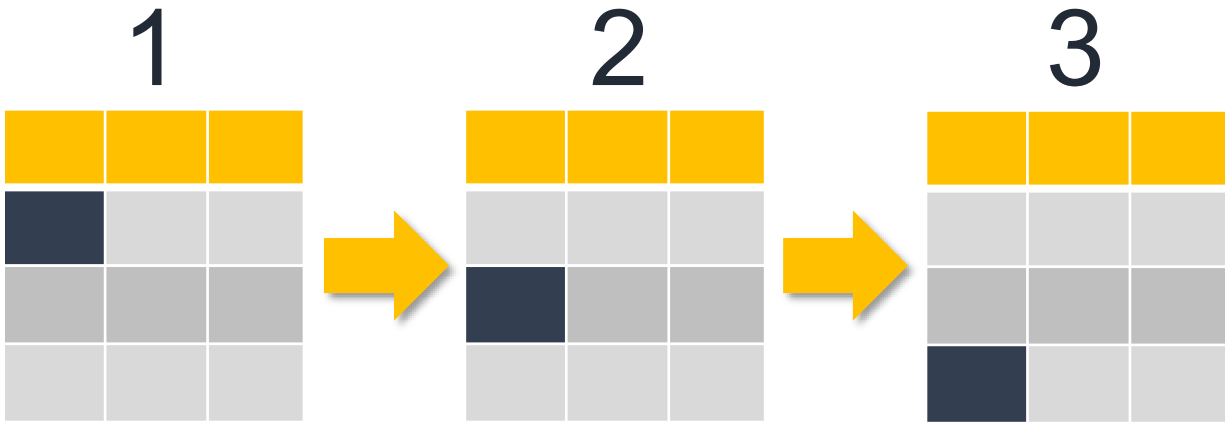
Some functions are faster than others

Requires experience and development of intuition

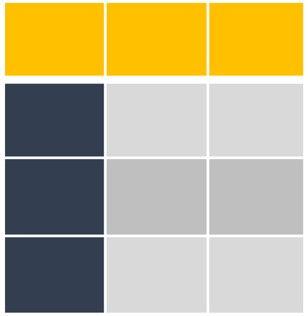
Examples:
* Readr::read_csv() is significantly faster than read.csv()
* rowSums(), colSums(), rowMeans(), and colMeans() are faster than apply() because they use vectors

Vectorize Your Code

For-loops iterate over a vector one at a time, increasing time and memory usage



Vectorized functions are applied to the entire vector at once rather than one at a time



Use "Microbenchmark" to compare simple functions like the following example

Vectorize Your Code - Example

```
#Function (vectorized) that multiplies two vectors.
vectorized_vector <- function(n) {
    x <- 1:n
    y <- x * x
    return(y)
}

#Function (non-vectorized) that multiplies two vectors
vector_for_loop <- function(n) {
    x <- 1:n
    y <- vector("numeric", length = n)

    for (i in 1:n) {
        y[i] <- x[i] * x[i]
    }
    return(y)
}


#Here we use the microbenchmark function to show how many nanoseconds are needed to run each function 100
microbenchmark(vectorized_vector(100),
               vector_for_loop(100),
               times = 100)
```

Vectorize Your Code – Example Output

- Observe how many nanoseconds are needed for each function
- Vectorized function is much more efficient and quicker

```
Unit: nanoseconds
                  expr  min   lq    mean median     uq   max neval
 vectorized_vector(100)  702  901 1227.05 1002.0 1301.0  8400   100
   vector_for_loop(100) 8401 8701 9104.88 8801.5 9001.5 23301   100
```

Thanks!