In this class we’re going to take a step back from statistical analysis and instead take a closer look at what’s going on under the hood when you run a program in R. This is incredibly important, because when you work with large real-world datasets, even small inefficiencies can lead to large costs.

We will cover:

- vectorization
- how to time functions
- a closer look at function calls in R
- parallelization
- some additional information about data structures and the `apply` class of functions

### Constructing squares

Suppose we need to construct a vector of perfect squares. More specifically, let’s say we want to create a function `my_squares` that takes an integer `n` and returns a vector containing all of the squares of the numbers between 1 and `n`. Eventually we may want to call this function on a large value of `n`, say `n = 100000`.

**Problem:**

Think of at least three different ways to implement the function. In other words, complete the following code in at least three ways:

```r
my_squares <- function(n) {
  # insert your code here
}
```

(Hint: At first, you may want to test your function with a value of `n` smaller than 100000. Try to write your function using (a) a for loop, (b) `sapply` or a related function, (c) neither of the above.)

### Which solution is best?

Which of these six (or more!) functions is best to use? In general there are several ways we may want to optimize our code, but for now let’s just focus on speed.

R provides some simple tools that we can use to test which of these methods is the fastest.

**Problem:**

Try to figure out how to use either `proc.time` or `system.time` to evaluate how long each method takes to run, and write some code to perform those tests. (You may need to look at some documentation.) Then try to think about why each method takes as long as it does (relative to the others).
Some details on R and vectorization

To understand why the functions perform so differently, we have to delve a bit deeper into the way that R actually runs code. R is a “high-level programming language.” This means that it is quite human-readable, but does not necessarily correspond to exactly what a computer does. In contrast, a lower-level language like C or Fortran is written in a way that is more difficult for a human to read, but more directly translates to exactly what the computer will do. (C is essentially one level of abstraction above assembly, which is machine code.)

So, what actually happens when we run a function in R? Well, typically R wraps up the requisite information needed to execute the function and then ships it off to a faster language to actually run. Compare the printout for the functions `nrow` and `ndim`.

```r
dim
## function (x) .Primitive("dim")
nrow
## function (x)
## dim(x)[1L]
## <bytecode: 0x24ef4a0>
## <environment: namespace:base>
```

The `.Primitive("dim")` means that `dim` is a function that isn’t written in R. On the other hand, `nrow` is written in R, and actually makes a call to `dim` and then takes the first element. When you evaluate a `dim`, R calls some compiled C code. This is good because C is a much faster language, but bad because there is some overhead in the communication. Ideally, you want to write code that passes back and forth between R and C as few times as possible.

We can see that the power operator is also a C primitive:

```r
^ ^
## function (e1, e2) .Primitive("^")
(1:n)^2
```

`(1:n)^2` only needs to make one function call regardless of the size of `n` and is thus much faster at large values. In contrast, the other methods, including the for loops and `sapply`, make at least `n` function calls. The principle of writing code that can make a constant number of function calls independent of the length of the input is typically referred to as vectorization, and it is crucial to optimizing R code.

Parallelization

Almost all modern personal computers have multiple CPU cores. This means that properly written code can sometimes utilize more than one of these cores at a time, speeding up the time it takes to run. Note that this is not possible for all tasks; some tasks are more easily parallelized than others, and here we will only discuss embarrassingly parallel problems. A task is embarrassingly parallel if it can be easily divided into many small task that can each be completed without speaking to any of the other tasks.

Some embarrassingly parallel problems:

- Doing a group project by having each person independently complete a section.
• Calculating all the perfect squares between 1 and 100000.

Some problems that are not embarrassingly parallel:

• Doing a group project where each person gets feedback on their section
• Multiple regression

Two other embarrassingly parallel problems that are very common in statistics are cross-validation and the bootstrap! Each of these methods can be thought of as small problems that are done totally independently. One easy way to do multicore processing in R is via the `mclapply` function in the `parallel` library.

Problem:

Write a parallel solution to the perfect squares problem. How does the runtime compare to other solutions?

Parallel bootstrap

Here is a normal implementation of the bootstrap, taken from the Chapter 5 R session video:

```
library(boot) # for the boot function
library(ISLR) # contains the data

alpha <- function(x, y) {
  vx <- var(x)
  vy <- var(y)
  cxy <- cov(x,y)
  (vy - cxy) / (vx + vy - 2*cxy)
}

alpha.fn <- function(data, index) {
  with(data[index,], alpha(X,Y))
}

# calculate alpha on the dataset
alpha.fn(Portfolio, 1:100)

# generate one bootstrap replica of alpha
set.seed(1)
alpha.fn(Portfolio, sample(1:100, 100, replace=TRUE))

# generate 1000 replicas
boot.out <- boot(Portfolio, alpha.fn, R=1000)

boot.out
plot(boot.out)
```

Problem:

Write a new parallel way of computing `boot.out` (without using the `boot` function). Compare your results with the `boot` function’s multicore option.
Some additional practice

The apply functions

Problem:

1. What function would you use to calculate the row means of a matrix? What about the column means? Verify your answer using the following matrix.

\[ X \leftarrow \text{matrix}(1:15, \text{nrow}=3) \]

2. What is the difference between `lapply`, `sapply`, and `vapply`? Run the following lines to find out.

\[
\begin{align*}
\text{l} & \leftarrow \text{lapply}(1:10, \text{function}(x) \{x^2\}) \\
\text{s} & \leftarrow \text{sapply}(1:10, \text{function}(x) \{x^2\}) \\
\text{v} & \leftarrow \text{vapply}(1:10, \text{function}(x) \{x^2\}, 1)
\end{align*}
\]

Data structures

Problem:

Let’s explore the relationship between lists and data frames. First create a data frame and a list:

\[
\begin{align*}
\text{my\_df} & \leftarrow \text{data.frame}(a = 1:5, b = 6:10) \\
\text{my\_list} & \leftarrow \text{list}(a = 1:5, b = 6:10)
\end{align*}
\]

1. Try and access the first column of `my\_df` and `my\_list` by using the dollar sign notation (`my\_df\$a`) and the double bracket notation (`my\_df[[\text{a}]]` or `my\_df[[1]]`). Now try it using the row-column index notation (`my\_df[,\text{a}]` or `my\_df[,1]`). What happens?

2. Try and access the first row of `my\_df` and `my\_list` by using the row-column index notation (`my\_df[1,\text{a}]`). What happens?

3. Try and create another data frame and another list, again with two columns but where each column has a differing number of elements. What happens?

What do the above exercises tell you about the relationship between data frames and lists? Which is a larger class of objects?

Now explore how atomic vectors (those formed with the combine function `c()`) differ from lists. Try the following and see what happens:

1. Combine values of two different types into a vector. Types include booleans (`TRUE`, `FALSE`), integers (`1`), numerics (`1.5`), and strings (`stanford`).

2. Combine several vectors into one, or combine a vector with a single value (which is really a vector of length 1).