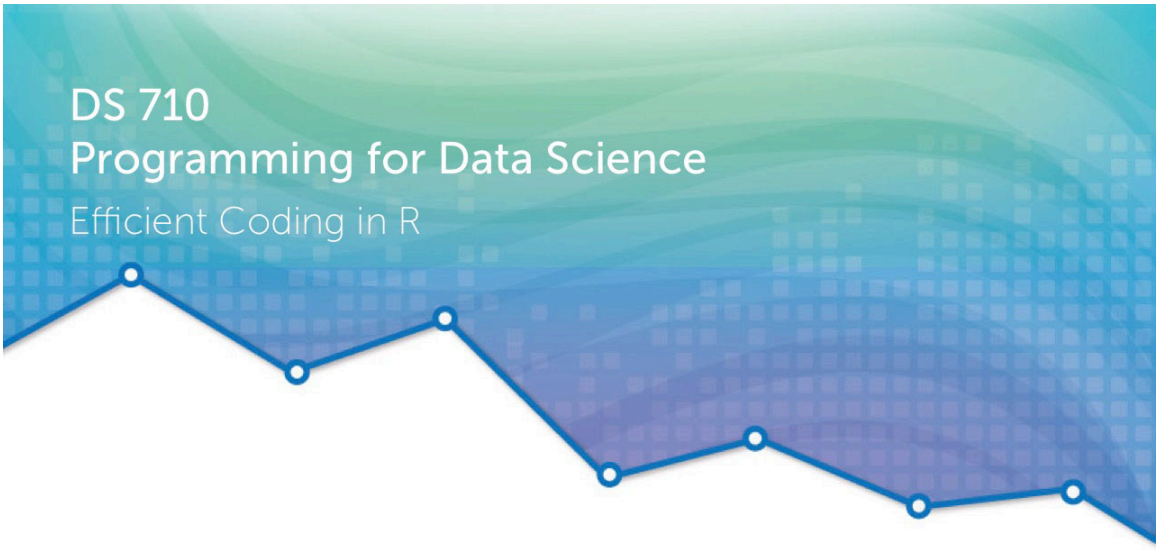
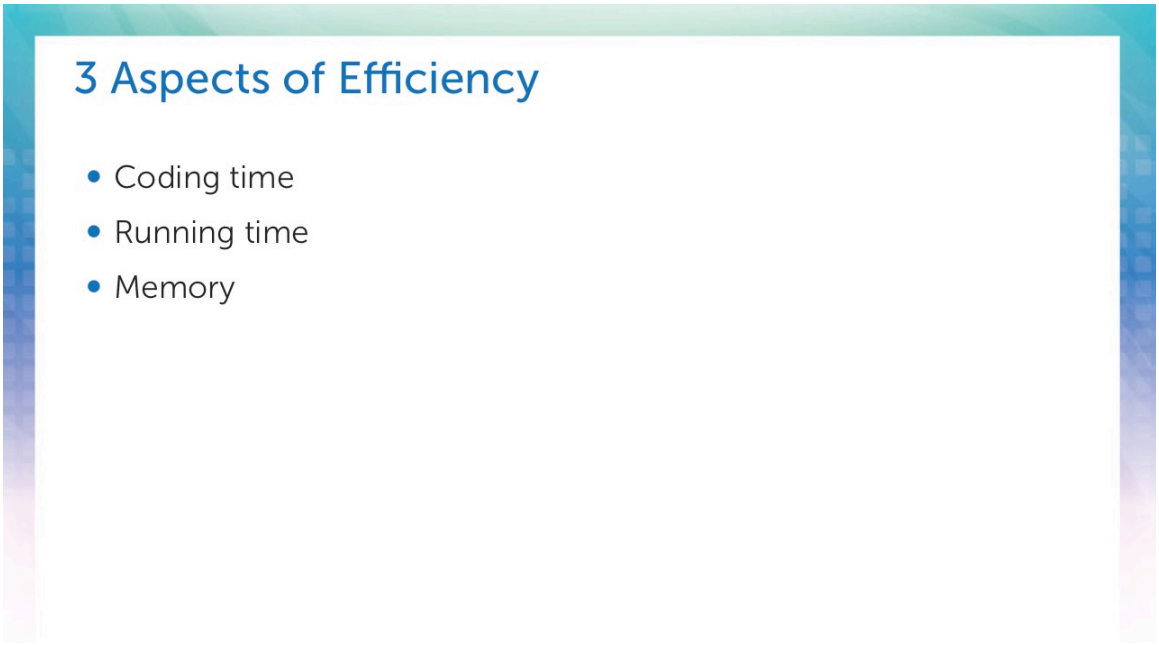
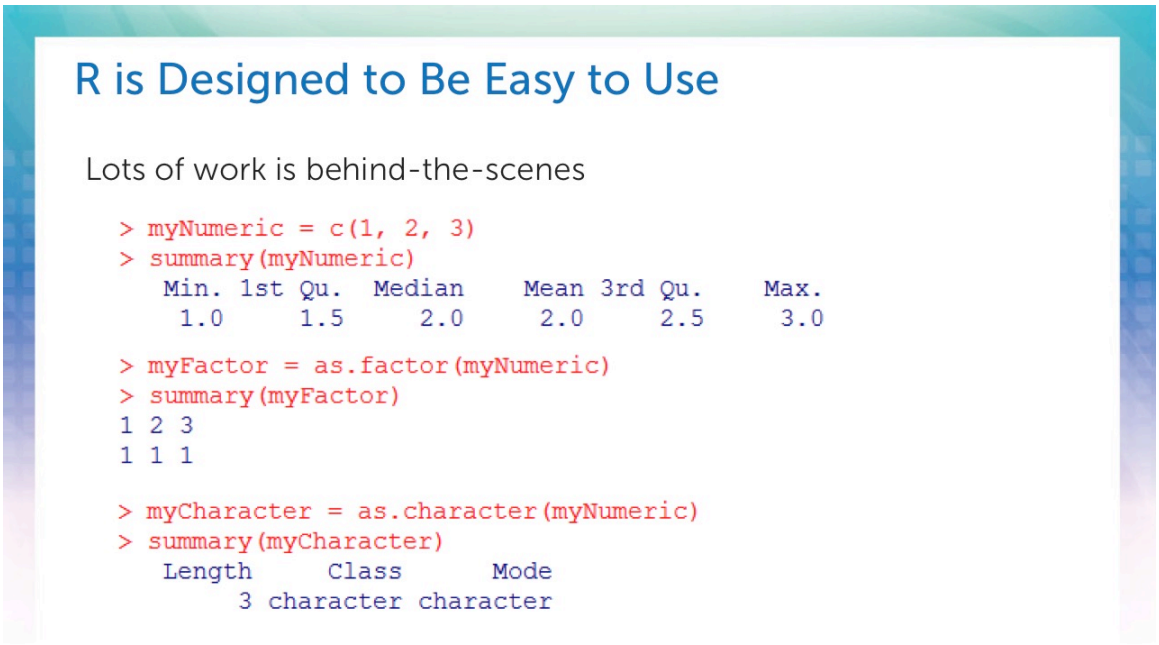




When you're thinking about the efficiency of your code, there are three aspects to consider-- the time it takes to write your code, the time it takes to run your code, and the amount of memory your code uses. Frequently, there are trade offs among these three aspects. For example, you can improve the running time of your code by spending a lot of time writing it, making sure that each line of code is optimized. But that's something you probably only want to do for code that you plan on running many, many times in the future. So in the long run, the running time will far outweigh the time you spend coding it.

Another example of a trade off is between memory and running time. One way to improve the running time of code is to store more information in memory so that you don't have to keep recomputing the same values. But that increases the amount of memory your code uses, so you would not want to use that strategy for code that you plan running on large data files that are hard to store in your computer's memory.



R is designed to be easy to use. This is an advantage for coding time efficiency, but it can be a disadvantage for running time efficiency. In R, a lot of work is done behind the scenes. For example, you're familiar with the function summary, which produces a summary of whatever variable you use as input.

But you also know that the format of that summary is different depending on what kind of variable you use as input, whether it's numeric, a factor, or a character variable. That means that behind the scenes R is figuring out what kind of variable you just used as input and then looking up the specific code for that kind of variable.

A lot of functions in R are like this. And this means that R is slower than some other languages, such as C. It's doing this work behind the scenes. Fortunately, in R there are also many different ways of doing the same computation. And that lets us take advantage of opportunities to improve the running time efficiency of our code.

Outline: Strategies for improving efficiency

1. Avoid repetition.
2. Vector calculations are better than for loops.
3. Use existing functions.
4. Plan ahead for memory use.

Avoid repetition

```
MileageLookup <- function(m){
    # Prints the price and engine size (in liters)
    # of any cars with Mileage == m.
    print( Price[ which(Mileage == m) ] )
    print( Liter[ which(Mileage == m) ] )
} # end function "MileageLookup"
```

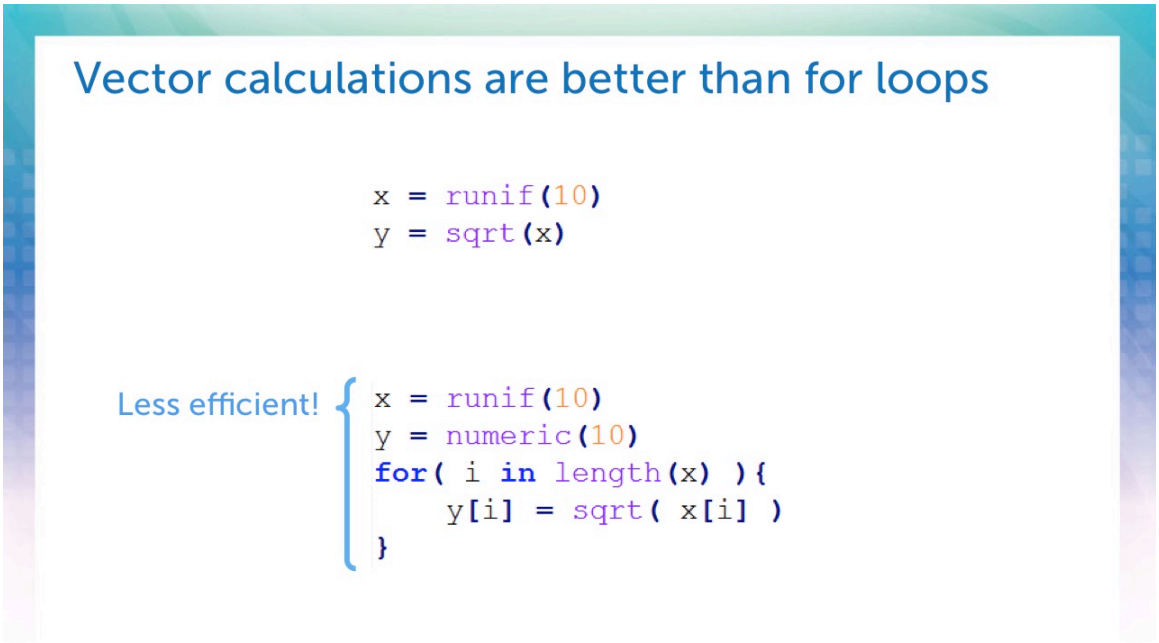
The first strategy to improve the running time efficiency of our code is to avoid repetition. For example, here's a function to print the price and engine size of any cars in a data set that have mileage equal to m. You'll notice that we're using the code which mileage double equal sign m twice in this function. That means we're repeating the process of looking through the whole data set to identify which cars have mileage equal to m, once for the price and once for the engine size in liters.

That means we're spending more time than we necessarily have to.

Avoid repetition

```
MileageLookup <- function(m){
    # Prints the price and engine size (in liters)
    # of any cars with Mileage == m.
    goodCar <- which(Mileage == m)
    print( Price[ goodCar ] )
    print( Liter[ goodCar ] )
} # end function "MileageLookup"
```

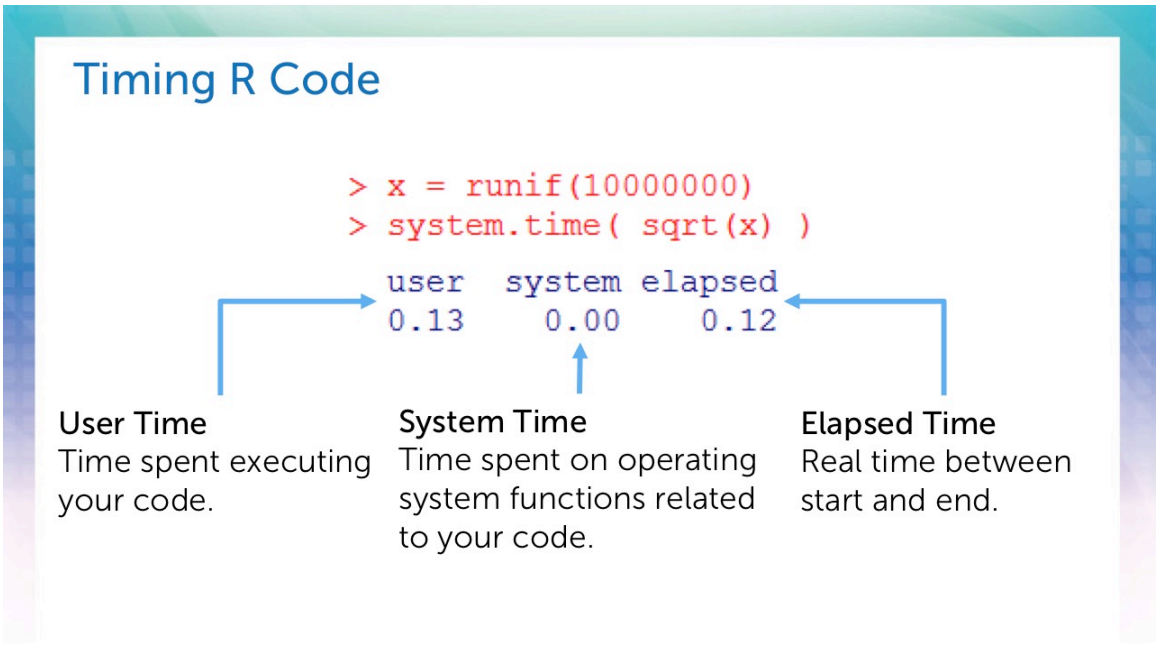
Here's a more efficient way to do the same function. We first assign a variable good car to be the indices of all the cars that have mileage equal to m, and then reuse the variable, good car, for the price and the liter calculation.



The second strategy for improving the running time efficiency of our code is that in R, vector calculations are better than for loops. Here's an example of that. Here we're defining x to be a vector of ten random numbers from a uniform distribution between 0 and 1. And y is equal to the square root of x. This is a vector calculation because we're applying the function square root to the entire vector x.

When we do this, R goes behind the scenes to figure out what kind of variable x is. It figures it out that it's a vector of real numbers. And then it takes the square root.

In contrast, we could do this same calculation in a different way using a for loop that would look like this. But in this case, we're taking the square root of one number at a time and each time we take a square root, R has to go behind the scenes and figure out what kind of variable this is. So in this case, R needs to do ten times as many behind the scene calculations as it did when we did the vector calculation. So the vector calculation is more efficient.

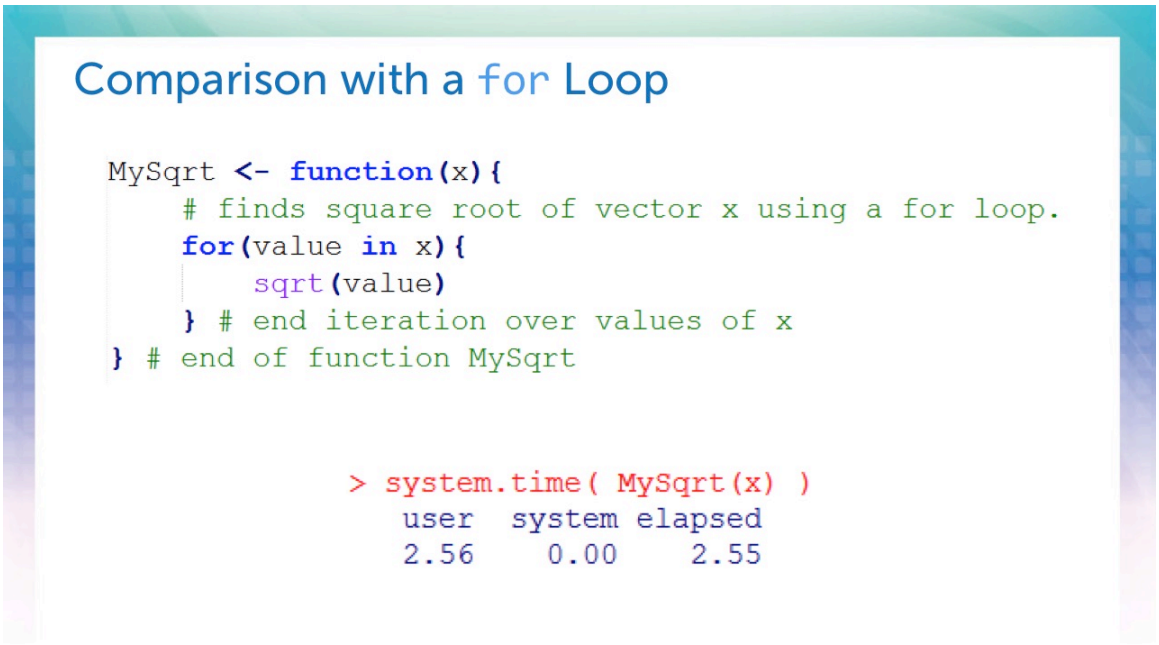


We can examine the running time efficiency of our code using the system dot time function, as shown here. If we actually tried this with the square root of ten random numbers, the running time would be so short it would round off to 0. So here I've used 10 million random numbers instead.

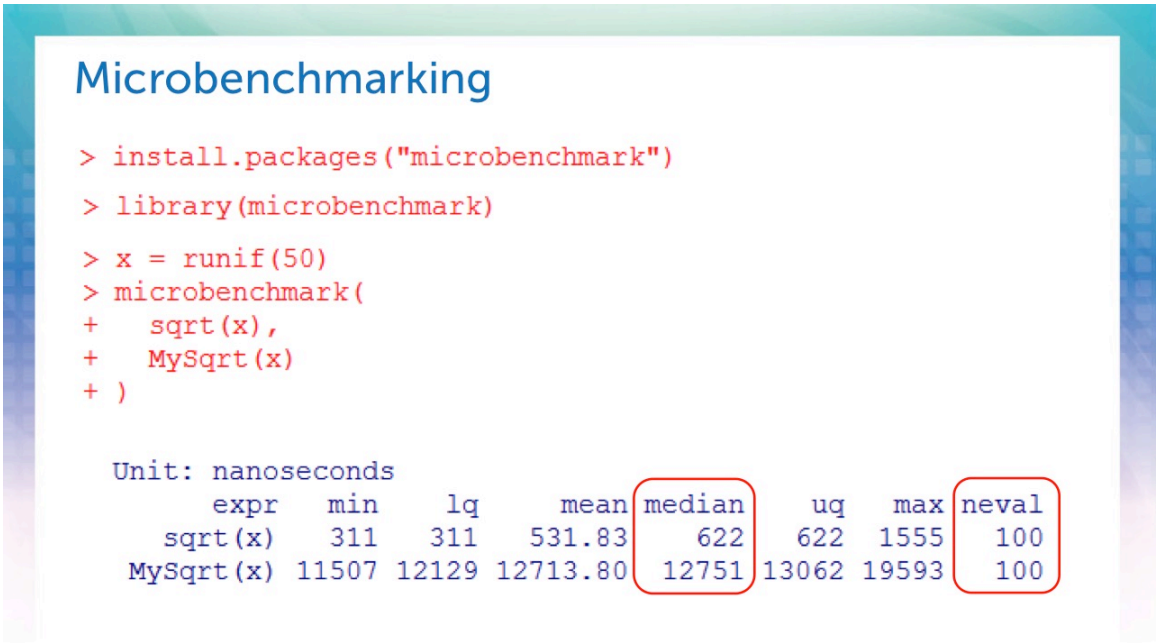
When we use system dot time, we get three pieces of output, as shown here in blue. The user time is the amount of time that your computer's CPU spent executing your code. In this case, 0.13 seconds. The system time is the amount of time your computer spent on operating system functions related to your code. Frequently, this will involve things like input and output if you're reading and writing files.

And finally, the elapsed time is the real time between the start and end of the code. That's the amount of time you spent sitting in your chair waiting for the code to run. The elapsed time is sensitive to what else is going on on your computer at the time you're running your code. So you might see the elapsed time being greater than the sum of the user time plus the system time if you were running lots of other programs on your computer at the same time.

Sometimes the elapsed time will also be less than the user time plus the system time. That can be due to your computer using multiple cores on your computer to run the code in parallel.



Let's compare the running time of the vector calculation of square root that we just looked at with the running time of a for loop. To use system dot time, it's most convenient to enclose that for loop inside a function. In this case, I called it my square root. Then we can apply system dot time to the function my square root of x. And here you see a user time of 2.56 seconds. That's much longer than the 0.13 seconds we saw for the vector calculation.



System dot time returns results in units of seconds, and the smallest difference that it can show us is differences of 1/100 of a second. In some cases, that's not sufficiently fine scale for what we're interested in. For example, in the square root example, we couldn't look at the difference between the vector calculation and the for loop on just ten numbers. We had to look at 10 million numbers.

For situations like this, there's another function that's helpful. It's called micro benchmarking. To use this, you'll need to install the micro benchmark package. Remember, you only need to install the package once for any implementation of R on a particular computer. And then every time you restart R when you want to use this function, you want to load the library that it's in. And that's just library micro benchmark.

Then we can set up R to be, say, 50 random numbers. And then use micro benchmark with the two functions we want to test inside it. We could list the functions all on one line inside the parentheses of micro benchmark. But sometimes, for readability purposes, it's nice to have them listed on separate lines, indented, as if they were inside a for loop.

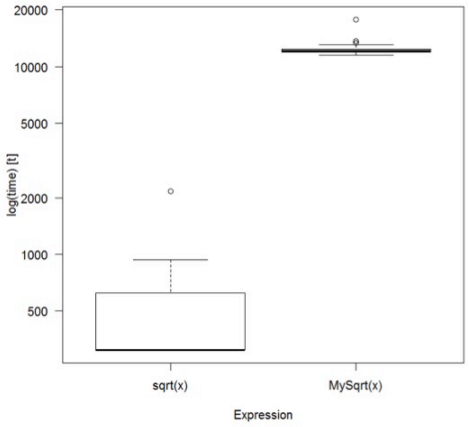
Here's what the output from micro benchmark looks like. Micro benchmark can return results in different units of time. Nanoseconds, microseconds, or

milliseconds, depending on how long your functions took. The last column of the output tells the number of repetitions of each function that it performed. The default is 100, but you can change this using an argument to micro benchmark, which you can see if you look at its manual page.

In addition to looking at the number of evaluations, the main thing you want to focus on is the median amount of time. In this case, for the vector calculation, square root of x, it was 622 nanoseconds. And for the my square root function, which use a for loop, it was 12,751 nanoseconds.

Timing Boxplot

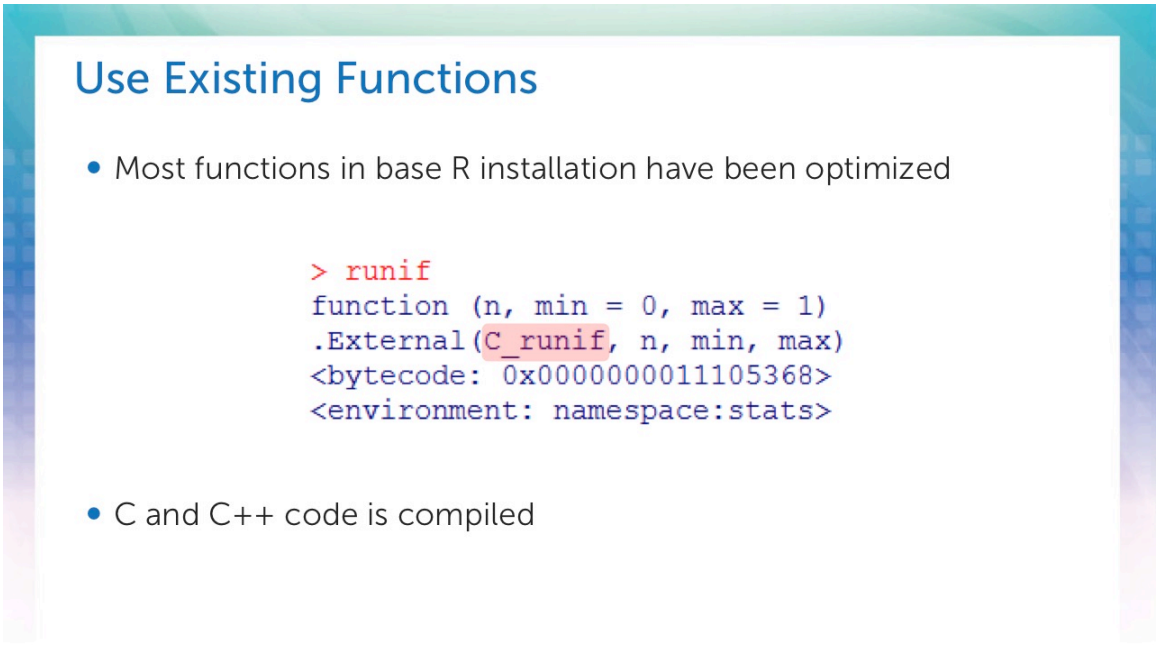
```
> timings = microbenchmark(sqrt(x),MySqrt(x))
> boxplot( timings, las=1 )
```



You can also see a box plot of the micro benchmark results by assigning some variable equal to those results. I called it timings. And then using the box plot function on that variable. Here I've used the argument LAS equals 1 to make sure that my axis labels are horizontal.

Here's what those results look like for the vector calculations square root and the for loop calculation my square root. You'll notice that the y-axis uses a log scale. That's because the micro benchmark timings are frequently right skewed. Remember that a right skewed distribution is very common for numerical values that have some lower limit, but no upper limit. So in extreme cases they can grow very large.

If you don't want to view the log scale, you can use the argument log equals false in the box plot function.



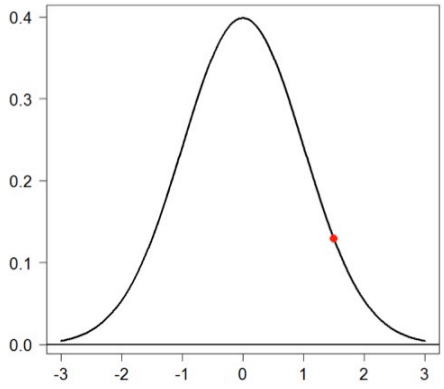
The third strategy for improving the efficiency of our code is to use existing functions. This is good for your coding time efficiency because you're not stopping to write new functions to do each thing you want to do. It's also good for your running time efficiency because most functions in the base R installation have been optimized to run quickly.

In fact, you'll notice that if you type the name of the function without any arguments or parentheses after it, many of them are not written in R at all. For example, the runif function is actually written in C. C and C++ code tends to be faster than R code because it's compiled. That means that after the code was written, a process was run on the code to figure out all of the behind the scenes details that R usually does on the fly.

So when you call that function, the computer already knows what kind of variable it is, where it's stored in memory, and so on.

Example

- Find the height of the probability density function at X = 1.5 for a Normal distribution with .



Two Approaches

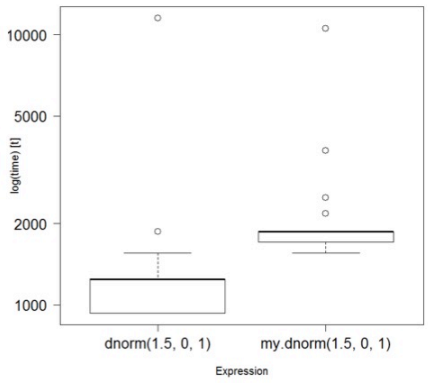
1. Existing function, written in C: **dnorm**
2. Use formula for PDF

```
my.dnorm <- function(x, mu, sigma){
    (1/(sigma*sqrt(2*pi))) * exp(-(x-mu)^2/(2*sigma^2))
} # end of function my.dnorm
```

There are two ways we could approach this function. One way would be to use an existing function that was written in C. And that's the denorm function. The second approach would be to use the formula for the probability density function of a normal distribution, which we can look up in a statistics textbook.

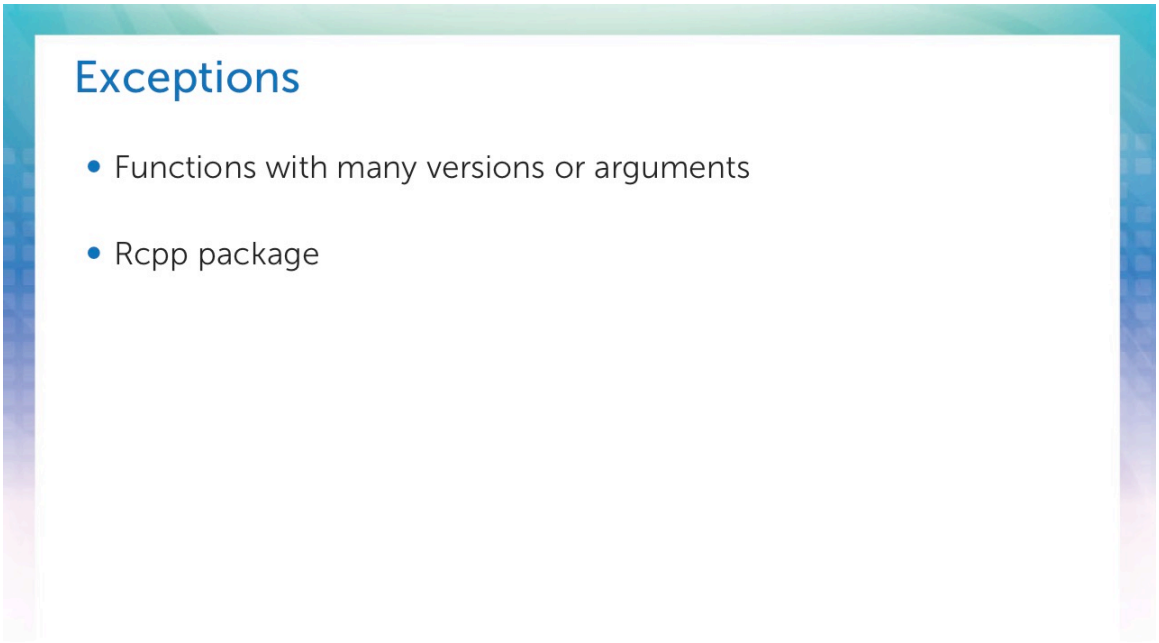
Timing comparison

```
> system.time( for(i in 1:100000) dnorm(1.5, 0, 1) )
   user  system elapsed
   0.15    0.00    0.15
> system.time( for(i in 1:100000) my.dnorm(1.5, 0, 1) )
   user  system elapsed
   0.23    0.00    0.23

> timings = microbenchmark(
+   dnorm(1.5, 0, 1),
+   my.dnorm(1.5, 0, 1)
+ )
> boxplot(timings)
```



We can compare these approaches using system dot time. Here I've used a for loop to iterate each approach 100,000 times. That lets us get a real comparison of these two where the difference is more than 1/100 of a second. So we can view the difference using system dot time.

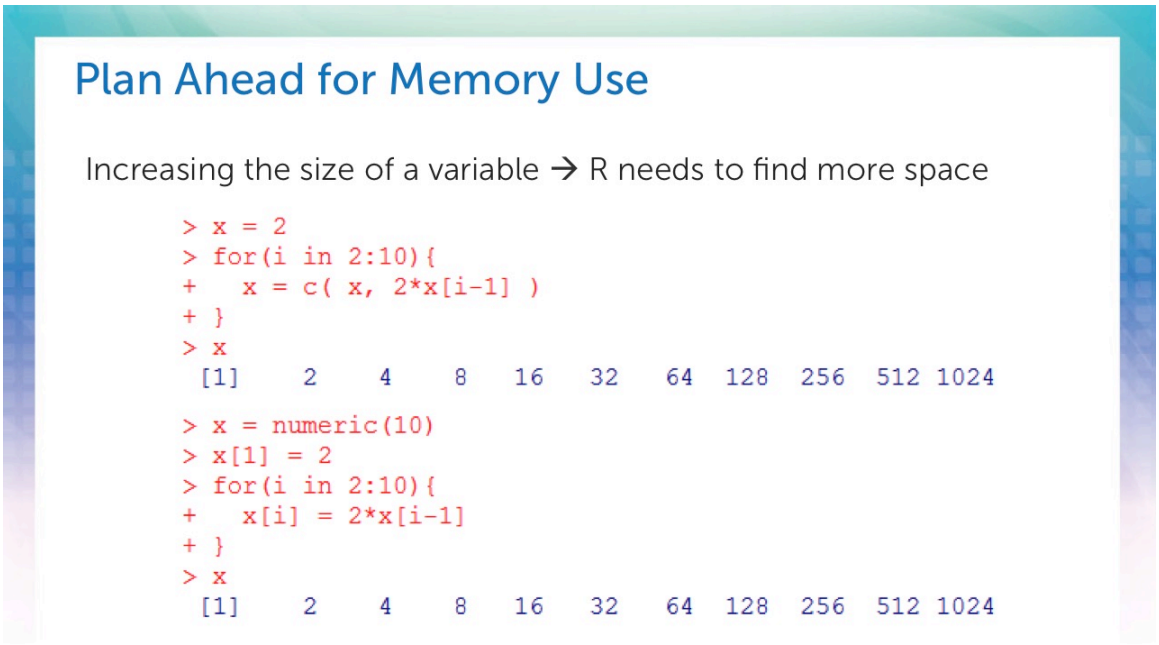
Here you'll notice that both the user time, the CPU time spent running the function, and the elapsed time, the amount of time I spent sitting in my chair waiting for the function, both were greater when using my dot denorm, the function that I wrote, compared to using the built in function denorm that was written in C. We can also compare these approaches using micro benchmark. Here you'll notice that the longest running time for denorm was higher than the longest running time for my dot denorm. However, the overall trend, if we compare it the median or the upper and lower quartiles or the upper and lower whiskers of the maximum and minimum non outliers, the trend is for my dot denorm to take longer than denorm.



There are some exceptions to the rule that existing functions are more efficient than functions you write yourself. Functions that have many different versions for different types of variables, like the summary function, can be less efficient than functions you write that are specific to a particular type of variable that you're working with in your code. That's because behind the scenes R has to go through all of the types of variables that could be used for a function and figure out which one applies for this time the function is being called.

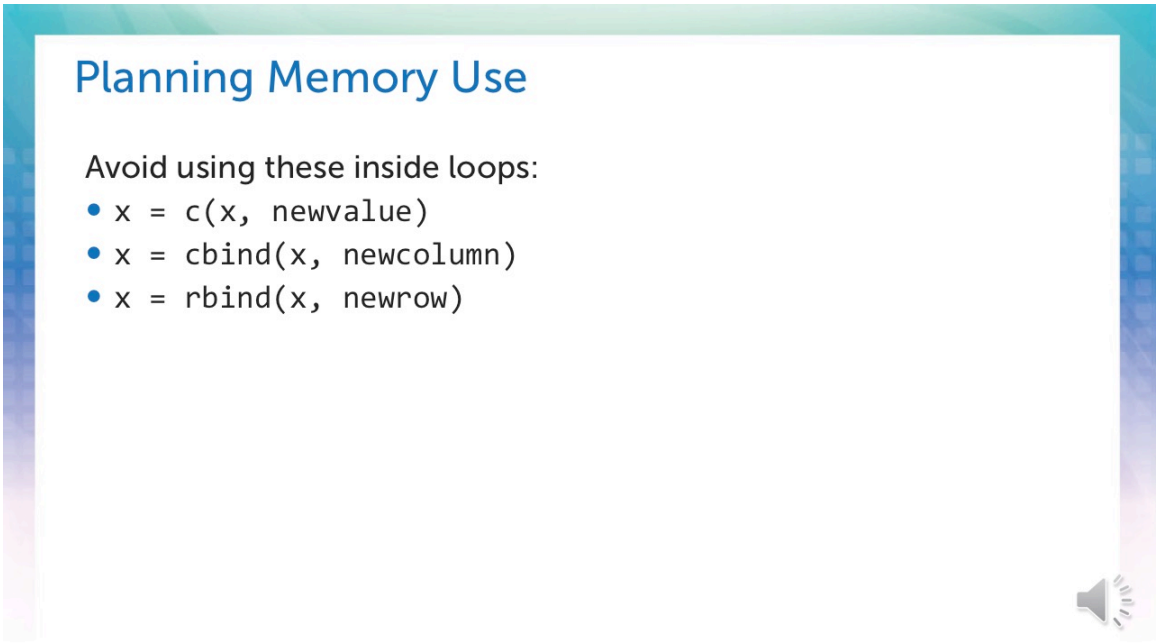
Similarly, functions that have many different arguments may be less efficient than a more streamlined function that you write that only uses the arguments that you care about. That's because, again, R has to go behind the scenes and figure out which arguments in the call to the function that are inside the parentheses are aligned with which values of the arguments in the behind the scenes code.

There is a package called RCPP that can serve as an interface for writing your own functions in C++. This is mainly useful for functions that you plan on using many, many times. So you really want to spend a lot of coding time optimizing the running time and making sure that it's as short as possible.

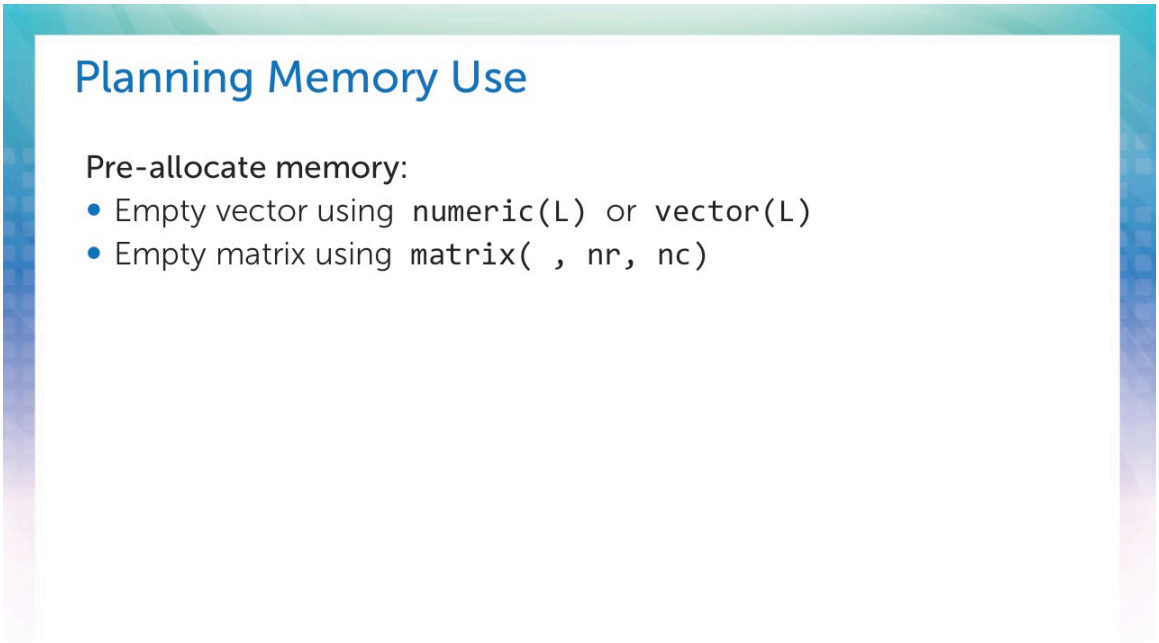


The final means strategies for improving the efficiency of our code is to plan ahead for memory use. Each time you increase the size of a variable, R needs to find more space to store that variable. And that takes time. For example, suppose we wanted to calculate the first 10 powers of 2. We could do that by setting x equal to 2 and then using a for loop to multiply each element of x by 2 and using the C function to create a new vector containing x and appending or tacking on the next value for the next power of two.

So here we've increased the size of the variable x nine times. In comparison, we could initialize x to be an empty vector with space for 10 values and set the first value of x equal to 2. Then, inside the for loop, we set the next value of x, x square bracket i, equal to 2 times x square bracket i minus 1. Here we only have to find space for the variable x one time, so this is more efficient.

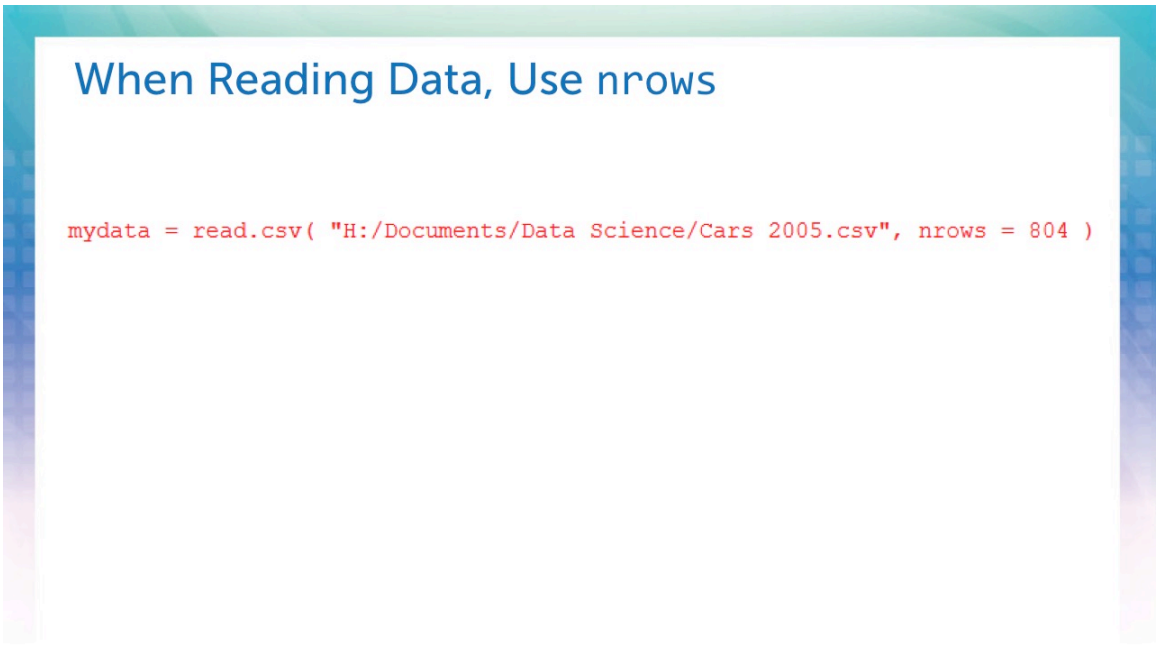


In general, you want to try to avoid using any of these three formats inside loops. x equals c of x comma a new value, thus updating the vector x to append a new value. Or x equals c bind or R bind x comma a new vector. This would be updating the matrix x to append a new column or a new row. But any of these three involve increasing the size of x, which tends to slow down the code.

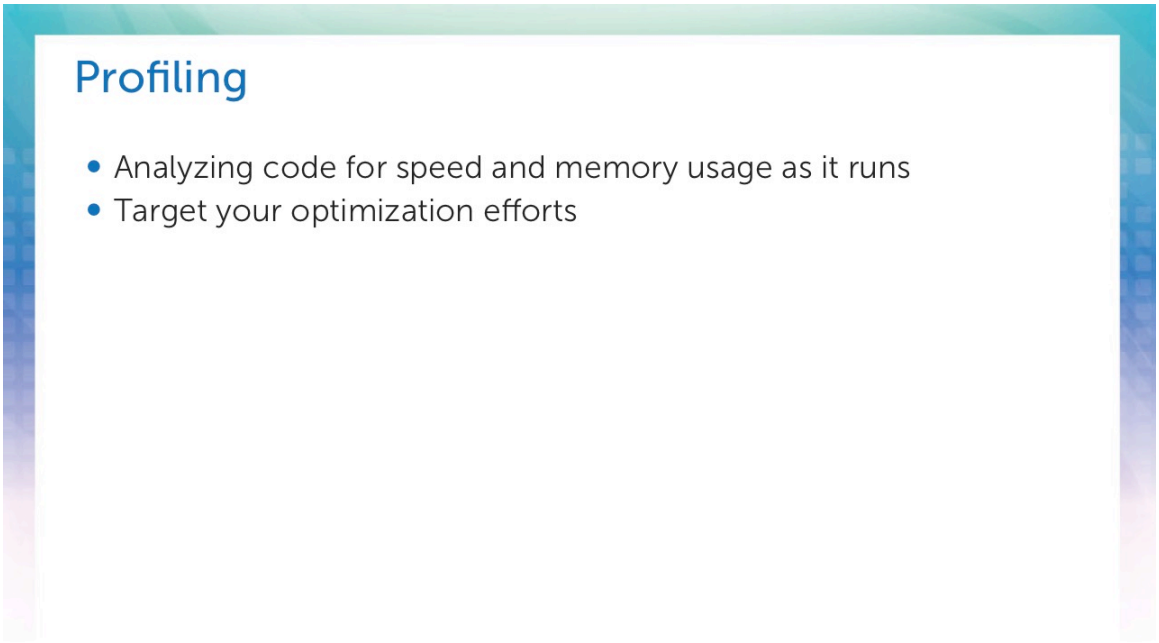


Instead, it's a good idea to preallocate the memory you plan to use. You can allocate an empty vector using the numeric function where the argument inside the parentheses is the length of the vector. You can also use vector to do the same thing, but vector is less specific about what kind of values are going to go inside that vector.

So if you know that the vector is going to store numbers, it's more efficient to use the numeric command. That tells R to expect numbers inside this vector so that future uses of that vector will be slightly more efficient. You can preallocate an empty matrix using the matrix function. If the argument before the first comma is blank, then the matrix will be empty or it will just contain n/a's. When you do this, though, you need to specify both the number of rows and the number of columns.



Another way to plan ahead for memory use is when you're reading data into R. You can use the n rows argument to tell R how many rows of data to expect. You can do this if you know how many rows are in the data set. For example, you've already looked at the data using Python. Or even if you just have an upper limit on how many rows to expect. A slight overestimate in the number of rows will still make the reading of data faster compared to giving no estimate at all.



Profiling code refers to analyzing it for speed and memory usage as it runs. This lets you target your optimization efforts to the sections of code that are taking the longest to run. So you're really spending your coding time as efficiently as possible to improve the running time.

I wouldn't profile every section of code that I wrote for any old homework assignment. I would only profile code that I was really planning to use many times in the future or code that seemed especially inefficient.

Example

- `Cars 2005.csv` contains data on whether the cars have upgrades: cruise control, leather, or an upgraded sound system.
- Write a function to make a subset of the data containing only cars with at least one upgrade.

FindOnes.r

```r
FindOnes <- function(x){
    # Extracts rows of x with 1's in columns 10-12
    y = NULL
    num.rows = dim(x)[1]
    for( i in 1:num.rows ){
        num.ones = length(which( x[i, 10:12] == 1 ))
        if(num.ones > 0){
            y = rbind( y, x[i,] )
        } # end "if there are ones"
    } # end iteration over rows of x
    return(y)
} # end function FindOnes
```

Here's an example of a function we could use to extract the rows of a matrix x that have 1's in columns 10 through 12. We're using the length and which functions to identify which of the columns in columns 10 through 12 have 1's. And if the number of 1;s is greater than 0, we append that row to our matrix y.

But you'll notice here that we have the line y equals R bind y and a new vector inside of a for loop. So we're breaking our rule of preallocating memory and planning ahead for memory use.

R Syntax for Profiling

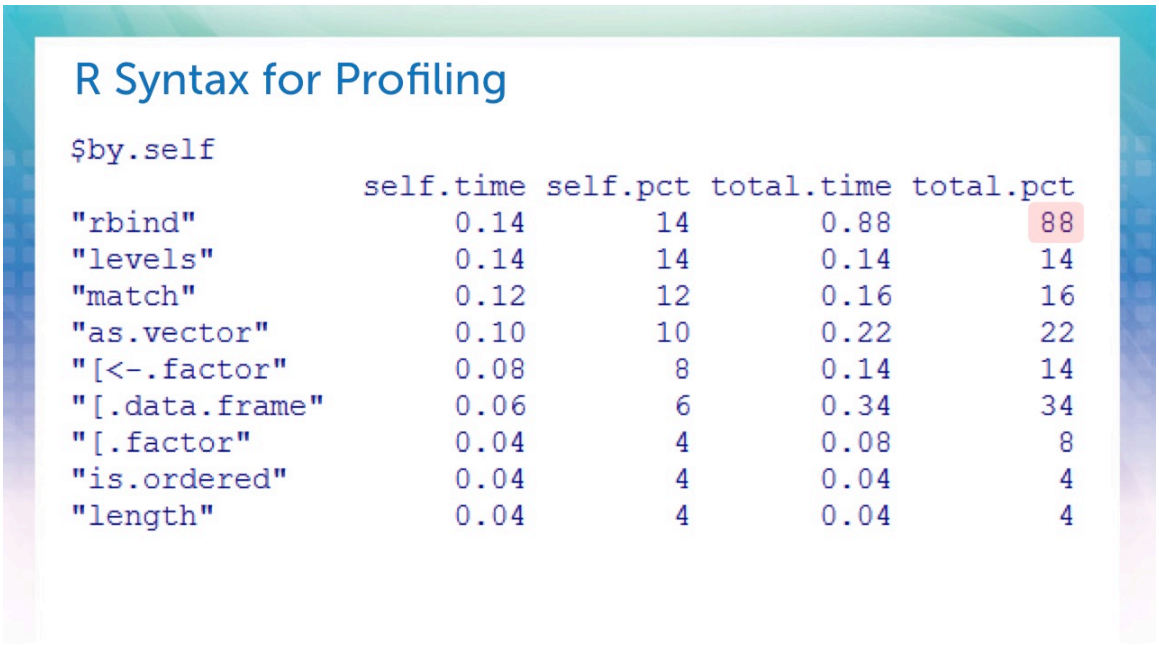
```
> gc()

> Rprof("H:/Documents/Data Science/FindOnes.txt")
> y = FindOnes(cars)
> Rprof(NULL)

> summaryRprof("H:/Documents/Data Science/FindOnes.txt")
```

Let's suppose that our function find 1's isn't working as fast as we think it should, but we're not sure why. We can use profiling to identify which sections of the code we're spending a lot of time in. To do this, we can start by using the function GC. That stands for garbage collection. And it just tells R to go through and find any variables that should've been deleted that are still hanging around in memory.

This is a nice thing to do before profiling because it helps prevent R from going into garbage collection mode automatically during the profiling. Because that can throw off our results.

Next we use the function R prof to do the profiling. And the argument for R prof is a file where the results are going to be stored. This can be any file you want, as long as it's a new file name. Because you wouldn't want to accidentally overwrite some other data with the results of your profiling.

Once you've given the R prof command, then you give whatever command or commands you need to run the function you want to analyze. And finally, you stop the profiling by using R prof null. To view the results you use the function summary R prof.



This will give a lot of output. But fortunately, we're mostly interested in what comes at the top of the output. The left-hand column here is a list of different functions that R was in when the R prof function randomly sampled what was going on behind the scenes. You'll notice that some of them are functions that we directly used in our code, like R bind and length. But others are functions that we didn't write in our code, such as levels or match. Those are functions that were called by other functions within our code.

The other four columns tell the amount of time and the percentage of time spent in each of these functions. The self dot time and self dot percent are the time and percentage of time directly in each of those functions. And the total dot time and total dot percent refer to the amount of time spent in those functions or any other functions that were called by the function in that row.

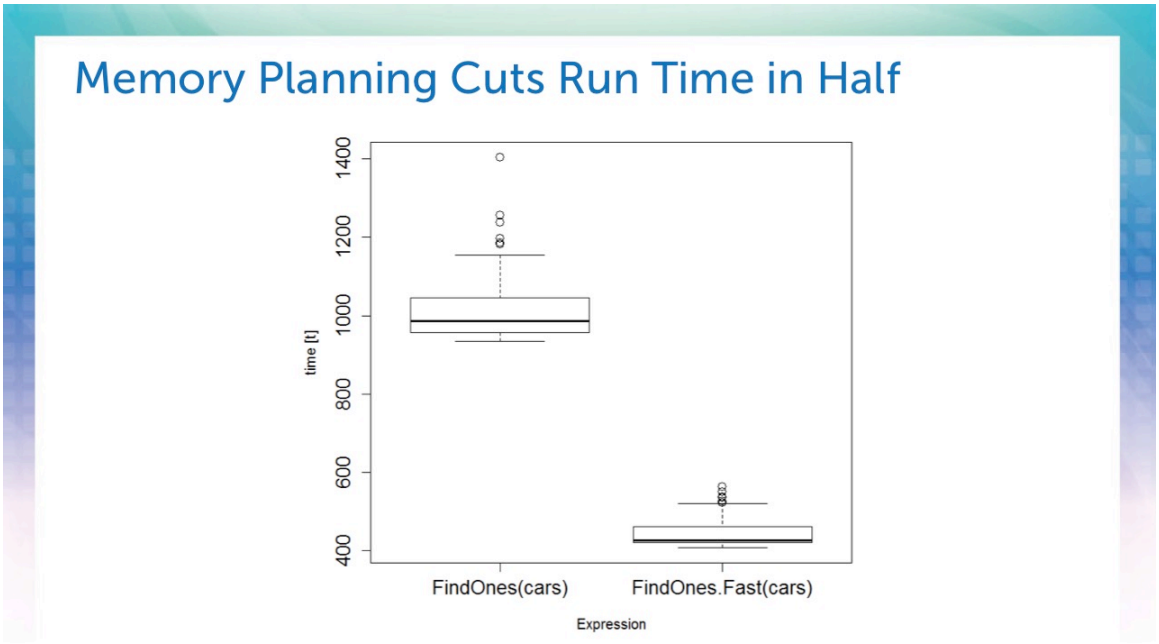
In this case, we notice that right at the top where the highest percentage is listed R bind is being used 88% of the time for R bind itself or for other things that were called as a result of R bind. That seems like a lot, and that could be used as a hint to help us recognize, oh, yeah. Maybe we could be doing our memory allocation more efficiently.

A More Efficient Version

```
FindOnes.Fast <- function(x){
    # Extracts rows of x with 1's in columns 10-12
    y = x
    j = 1
    num.rows = dim(x)[1]
    for( i in 1:num.rows ){
        num.ones = length(which( x[i, 10:12] == 1 ))
        if(num.ones > 0){
            y[j,] = x[i,]
            j = j + 1
        } # end "if there are ones"
    } # end iteration over rows of x
    return(y[1:(j-1),])
} # end function FindOnes.Fast
```

Here's a more efficient version of the same function that doesn't require R bind. You'll notice that we start by initializing y. Instead of being equal to null, it's equal to x. So it starts out the same size as our original data frame. We're still using length and which to pick out which columns of x are equal to 1. And we're still copying rows of x into y if the number of 1's is greater than 0.

But in this case, we're copying the i-th row of x into the existing j-th row of y rather than using R bind to tack on a new row. And we're updating j by using j equals j plus 1 each time we identify a new row that needs to be added to y.



Using micro benchmark, you can see that using this more efficient memory planning cuts the running time for this function in half.