

# Efficiency in R

Duncan Temple Lang

Code, etc @ <https://github.com/dsidavis/EfficientR>

# Approaches

- Avoid unnecessary computations in functions (e.g. colClasses, adding names)
- Find function(s) in existing packages (and hope faster)
- Byte-compile the code.
- Profiling: Time code & parts of code
  - Improve the bottlenecks
- Embarrassingly parallel code
  - use parallel package & others
- Use C code, but only for the slow parts.
  - .Call(), .C(), or Rcpp
- Different data structures and algorithm/computational approach.

- How much does it matter ? and to who?
- Will you or other people be running this code many times ?
- How much speedup do you need?
- Do some basic timings for different sizes of the inputs.
  - Plot the results for and see if the algorithmic complexity is linear, quadratic, worse!
  - This may give you a hint as to whether you need a different approach.
  - Use `system.time()` or `microbenchmark()`

- Remove objects in your work space when you no longer need them.
  - Frees memory
  - `save()/saveRDS()` them to disk and reload them latter.
- Avoid computing the same thing multiple times
  - do once, pass to functions that need it.

# Simple Things To Remember

- Use vectorized functions where you can
- Use `lapply()/sapply()` when you can rather than `for()` loops
- When looping, preallocate the answer to have the correct number of elements and fill them
  - Don't concatenate to the end.

# Concatenating Values to a Vector

N = 1e5

```
system.time({  
  ans = integer()  
  for(i in 1:N)  
    ans[i] = i  
})
```

19.774 seconds

N = 1e6    1814.2

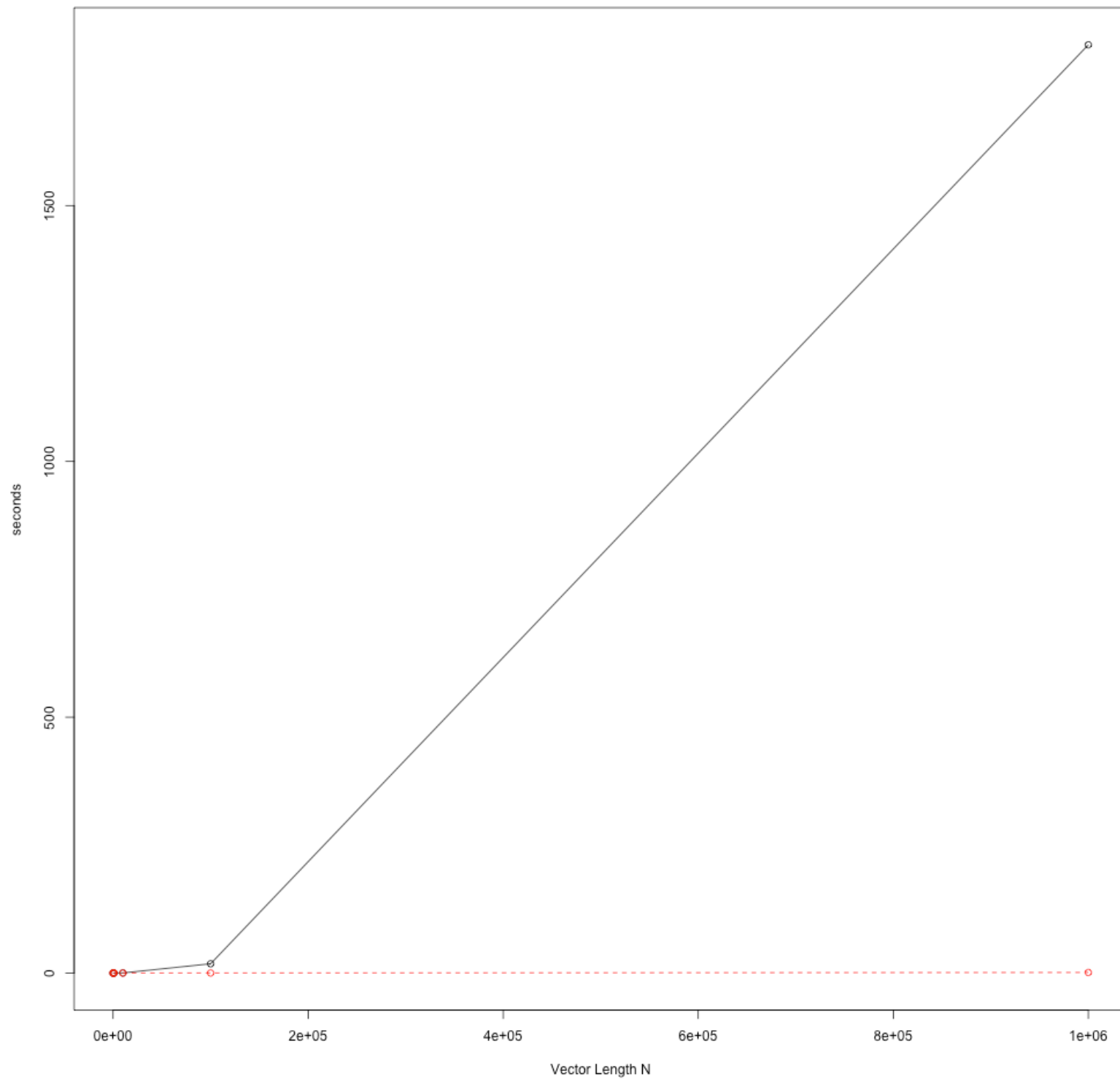
N = 1e5

```
system.time({  
  ans = integer(N)  
  for(i in 1:N)  
    ans[i] = i  
})
```

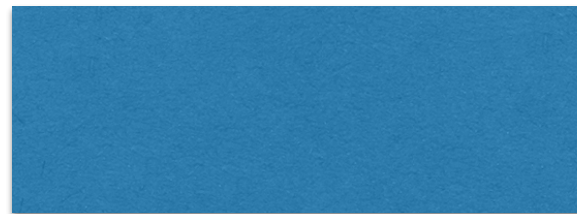
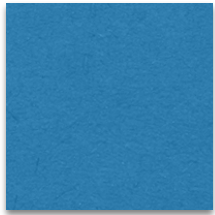
0.170 seconds

N = 1e6    1.1

Concatenation Slow Down



- Also leads to memory fragmentation.





# Why Concatenation is So Slow

- Think about the each iteration, e.g., 500th
- We add one element to the end of the 499-element vector
- R has to
  - allocate a new 500-element vector,
  - copy the 499 elements from the previous to the new vector
  - insert 500th value.
- Does this for each iteration.
- When we pre-allocate, just insert value. No allocation & copy.



# Premature Optimization

- “Premature optimization is the root of all evil” - Don Knuth.
- Get the wrong answer faster - Jon Bentley
- Write the obvious approach, and only try to get smarter after that.
- You have a correct version against which you can verify the results from the smarter version.
- You can focus on the parts that are slowest.



# Fibonacci Sequence

- Simple recurrence relationship

$$F(n) = F(n-1) + F(n-2)$$

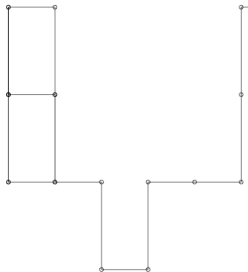
```
fib =  
function(n)  
{  
    if(n < 2)  
        n  
    else  
        fib(n - 1) + fib(n - 2)  
}
```

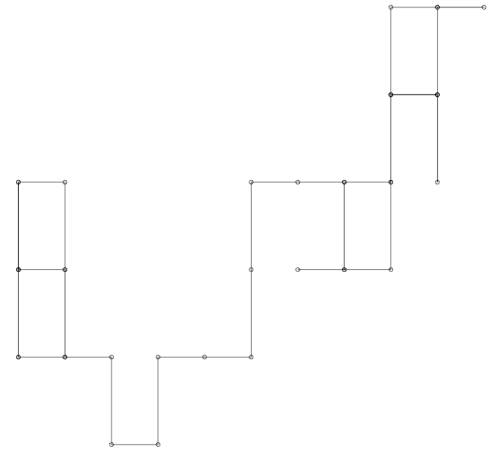
# Byte Compile

- Speed up the R implementation with the byte code compiler.
- `fib = compiler::cmpfun(fib)`
  - Note reassigning to `fib` since `fib` calls `fib`
- `system.time(fib(30))`                      1.878 seconds
- Almost 50% speedup on laptop, 23% on Linux server.

# Vectorization

# Different Computational Approach

- Consider a 2-D random walk
  - Naive Approach  
At each step,
    - Toss a coin for Horizontal or Vertical
    - Then for which direction (left/right or up/down)
    - Update current position
    - Repeat
- 



```
w2d1 = function(n) {  
  xpos = ypos = numeric(n)  
  truefalse = c(TRUE, FALSE)  
  plusminus1 = c(1, -1)  
  for(i in 2:n)  
    # Decide whether we are moving horizontally  
    # or vertically.  
    if (sample(truefalse, 1)) {  
      xpos[i] = xpos[i-1] + sample(plusminus1, 1)  
      ypos[i] = ypos[i-1]  
    }  
    else {  
      xpos[i] = xpos[i-1]  
      ypos[i] = ypos[i-1] + sample(plusminus1, 1)  
    }  
  list(x = xpos, y = ypos)  
}
```

- $N = 1e6$  steps
- Naive version: 26.5 seconds.
- Heavily vectorized version: .096 seconds
- Speedup Factor: 276
- On Linux machine, speedup x 88



- 4 possible directions at each step.  
 $(1, 0)$ ,  $(-1, 0)$ ,  $(0, 1)$ ,  $(0, -1)$
- Sample all  $N$  steps at once, i.e. in one call.
- Use a separate cumulative sum of the  $X$  and  $Y$  coordinates.

# Vectorized 2-D Walk

```
rw2d5 =
```

```
# Sample from 4 directions, not horizontally and vertically  
# then left/right or up/down.
```

```
function(n = 100000) {  
  xsteps = c(-1, 1, 0, 0)  
  ysteps = c(0, 0, -1, 1)  
  dir = sample(1:4, n - 1, replace = TRUE)  
  xpos = c(0, cumsum(xsteps[dir]))  
  ypos = c(0, cumsum(ysteps[dir]))  
  list(x = xpos, y = ypos)  
}
```

# Profiling

- What can you do to speed up your code?
- Profile it to identify how long each function takes.
- Count how many times those functions are called.
- `Rprof()`, `summaryRprof()`, `trace()`, `untrace()`
- `Rprof("nameOfOutputFile")`  
  `# run code`  
  `Rprof(NULL)`
- `summaryRprof("nameOfOutputFile")$by.self`

## Output from summaryRprof()\$by.self

	self.time	self.pct	total.time	total.pct
"match"	11.88	48.77	14.16	58.13
"%in%"	2.48	10.18	16.52	67.82
"FUN"	2.16	8.87	24.36	100.00
"rbinom"	1.62	6.65	1.62	6.65
"\$"	1.20	4.93	6.44	26.44
"[[.data.frame"	1.18	4.84	4.08	16.75
"<Anonymous>"	1.00	4.11	1.74	7.14
"\$.data.frame"	0.76	3.12	5.24	21.51
"["	0.40	1.64	4.48	18.39
"sys.call"	0.32	1.31	0.32	1.31
"lapply"	0.30	1.23	24.36	100.00
"nargs"	0.18	0.74	0.18	0.74
"all"	0.14	0.57	0.14	0.57
"cat"	0.12	0.49	0.12	0.49
"is.matrix"	0.12	0.49	0.12	0.49

- Used named matching, not `ids %in% names(obj)`
- Use vectorized versions of `rnorm`, `rbinom`, etc.  
Vectorized in the parameters.
- Use `sapply()` or better `vapply()` when function returns scalars.  
`do.call(rbind( , lapply()))`

# Infection Simulation



- See sim.R (after initializing different inputs to the simulation)

# How Many Calls ?

- Profiling indicates functions R spends more time in.
- But these may not be slow - just called a lot more times.
- So we want to know the number of calls to our bottleneck functions.
- Use `trace()` to monitor all calls to a function.
- `k = genCallCollector()`  
`trace(rnorm, quote(k("rnorm")), print = FALSE)`

# C/C++ Code

- Look for C/C++ code to do what you need or write it yourself.
- Translate only very small elements of the R code, mostly `for()` loops where the iterations depend on each other.
- Makes entire code a little more complicated
  - Need compiler installed, etc.
  - Can compile C code and manually load it into R session
  - R CMD SHLIB `foo.c bar.c`
  - `dyn.load("foo.so")` # or `foo.dll` on Windows.



# Fibonacci Sequence

- Simple recurrence relationship

$$F(n) = F(n-1) + F(n-2)$$

```
fib =  
function(n)  
{  
  if(n < 2)  
    n  
  else  
    fib(n - 1) + fib(n - 2)  
}
```

# Fibonacci Sequence

```
int fib(int n) {  
    if(n < 2)  
        return(n);  
    else  
        return(fib(n - 1) + fib(n - 2));  
}
```

C code  
can find online

```
void  
R_fib(int *rn, int *ans)  
{  
    *ans = fib( *rn );  
}
```

R wrapper routine  
call via .C() in R

- R CMD SHLIB fib.c
- dyn.load("fib.so")
- .C("R\_fib", 30L, ans = 0L)\$ans
- fib(30) times
  - R function : 2.791
  - C code: 0.006
  - Speedup factor: 465

# Use an R Package

- For simplicity, with C code, create an R package
  - Put R code in R/ directory, C/C++ code in src/
  - Add NAMESPACE file with  
`useDynLib(packageName)`
  - Copy and edit a DESCRIPTION file
  - R CMD INSTALL myPackage

- Long discussion of an example of writing code in R, C, data structures in Data Science in R: A Case Studies Approach  
<http://rdatasciencecases.org/> BML cars chapter.

# Appeal for Mutual Help

- You have to adapt your code to make it run faster.
- What if we could analyze your code before it is run and translate it to faster code!
- Nick, Clark and I are working on this.
  - E.g. we can take the naive 2-D random walk function as-is compiled it automatically to machine code.
  - Speedup of factor of 2 over best vectorized version.



# Appeal for Mutual Help

- We'd love to get your R code that is slow & that you care about as examples for our research.