Efficiency in R
Simple rules, Garbage Collection, Profiling & algorithms

Duncan Temple Lang
Dept. of Statistics
UC Davis

- R and S-Plus primarily designed for EDA (Exploratory Data Analysis)
- Easy to express commands to look at aspects of data, fit models, simulate.
- Gradually, we have been increasingly using it for
  - large data,
  - programming, and
  - software development.

- Over 1000 publicly available add-on packages.
- So code is run not just by author but by numerous other users.
- Not just interested in saving human programming time but also run-time.
- And we want big data problems that wouldn't ordinarily be feasible in R to actually be feasible.

- R uses pass-by-value or copy rather than references.
- e.g. any changes that the lm() function makes to the data frame in
  lm(y ~ x, data = myData)
  do not change the original contents of the myData variable.
- This is a good thing - for users!
  Don't corrupt data and leave it in inconsistent state.
- And makes debugging easier.
  Can go up to earlier call frame to see original value of a n argument in subsequent call.

- But of course, making copies is very expensive for large datasets.
- So R tries to be clever internally and avoid copies where possible.
- Copy on change, i.e.
  data$y doesn't create a copy,
  but
  data$y[data$y == 0] = .00001 does.
- And R uses lazy evaluation to avoid processing an argument until it is needed.

- We often create an R script/"program" one line at a time
  e.g. type command at prompt, get it to work, add it to a file, and do the next step.
- The result can be an obvious approach that is easy to read (good)
- but that may not be the most efficient to compute.
- For example, we may end up recomputing the same thing, e.g. indices of interest, numerous times rather than computing once and assigning to a variable.
  Y = data[data$gender == "Female", "Weight"]
  X = data[data$gender == "Female", "Height"]

- So we end up with unnecessary computations that slow things down.
- Of course, if we assign intermediate values to variables, we are consuming more memory. Standard trade-off.
- But these issues can matter when we are dealing with large'ish amounts of data or computationally intensive methods.
- So we then sometimes need to write code in a slightly more intelligent or less natural way to make things go faster.

- There are some simple rules of thumb to remember.
- And there are tools to help identify where code is inefficient so that one can focus on improving just those bits.
- The typical process is
  - do the naive, simple thing
  - if and only if it turns out to be too slow, find out which part is causing the biggest slowdown.
  - Then improve just that bit.
  - If can't improve sufficiently in R code, use compiled code via .C()/.Call()/.Fortran().

- Use initial implementation to verify that subsequent, smarter versions give the same & correct results.
- Refine code to make it better.
- Sometimes have to think quite differently from obvious, mathematical formulation.
- "Premature optimization is the root of all evil" Donald Knuth
- Or, optimization -
  - Don't do it
  - Don't do it until you really have to.

Profiling

- So you write some code, and it runs slowly.
  i.e. it won't complete in time to hand in homework!
- Need to make it faster, but how?
- Use rules from earlier.
- But what if still not fast enough?
- Find out what bit is taking the most time and see if you can improve its performance.

Example - MCMC

- You've seen how to generate random numbers for a PDF f(x) using a few techniques.
- Convenient if you can find majorizing function or inverse of CDF.
- A general approach is Markov Chain Monte Carlo - MCMC
- Create a mechanism for generating a new random number based on current value, i.e. get X(t+1) from X(t).
- Subject to quite general conditions, can sample from any f using this technique.

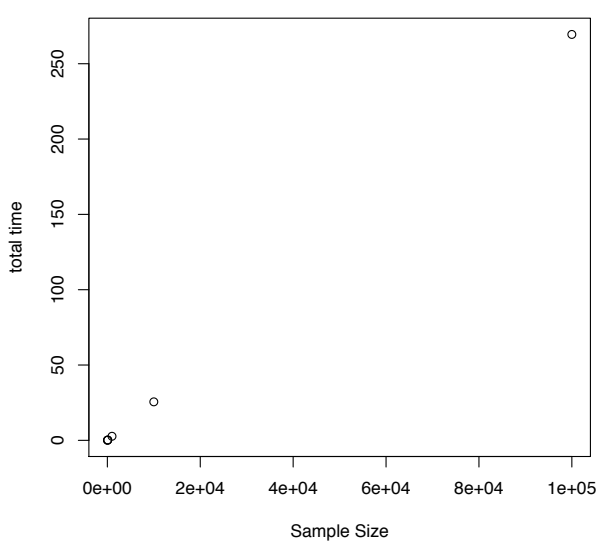
MCMC

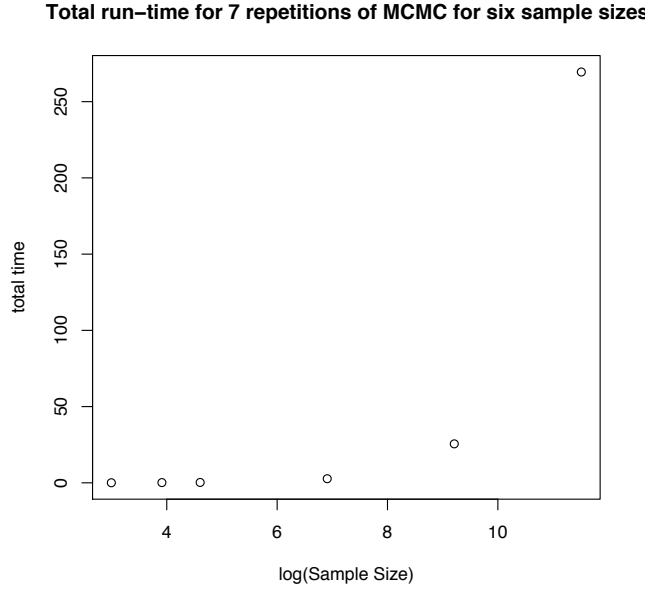
- R function mcmc() to do this.
- Give X(0) (starting value), target distribution (stationary), and sample size (n)
  And proposal density function and random number generator for that proposal density.
- Generate potential new value Y from proposal distn. centered at X(t). Then toss a biased coin and if heads, accept Y as X(t+1); o.w. X(t+1) = X(t)
- Brilliance is in determining the bias of the coin generally.
- Generate n values and then take all but the first 10,000 say to avoid dependencies on initial value.

How fast is this?

```
sizes = c(20, 50, 100, 1000, 10000, 100000)
times =
  sapply(sizes,
         function(n)
          system.time(replicate(7,
                        mcmc(-5, r, q,
                              stationary = dnorm,
                              n = n, alg = hastings))))
```

- Get user, system and total time for 7 runs for each sample size.





- How do we make this go faster?
- Find the parts of the code that take the most time
- And see if we can make them faster.

Rprof()

- Could use system.time() to time particular pieces, but that is tedious
  Need to modify existing code, store times, etc.
- The function Rprof() will gather data about how code in R runs as it is being evaluated.
- Call Rprof() to start the collection and tell it to write the information to a file,
  Rprof("profileData.prof")
- Run the code
  mcmc(-10, r, q, sta = dnorm, n = 1000)
- Stop the profiling with Rprof(NULL)

Analyzing Profile Data

- How do we access the profiling data?
- Call summaryRprof() with the name of the file containing the data.
  p = summaryRprof("profileData.prof")
- Now, p contains summary of time spent in each R function that was invoked.
- It arranges by total time in function and all functions it called, and so on.
- Or by "self" - time spent in that function alone

- Doesn't tell us the number of times a function was called, just the total amount of time spent in that function.

```
> p$by.self
                self.time self.pct total.time total.pct
"ifelse"             0.24     12.9       1.14      61.3
"rnorm"              0.24     12.9       0.24      12.9
"stationary"         0.18      9.7       0.18       9.7
"rep.default"        0.16      8.6       0.34      18.3
"rep"                0.12      6.5       0.46      24.7
"storage.mode"       0.12      6.5       0.18       9.7
"any"                0.10      5.4       0.10       5.4
```


- So the use of ifelse() is consuming the most time.
- Is that in our code, or the functions it calls.
- Look at the function mcmc to see.

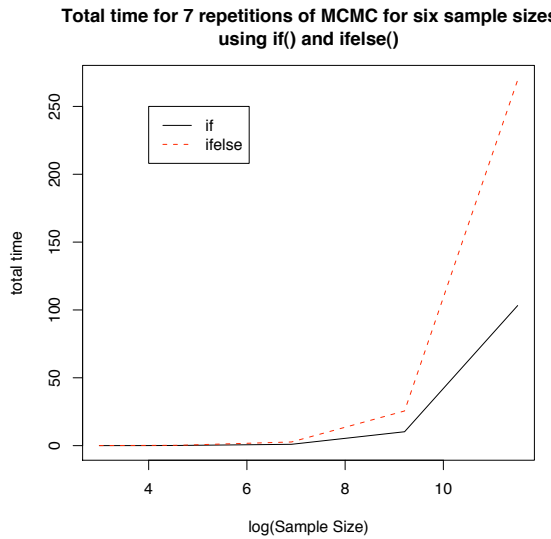
```
mcmc =
function(x.0 = 0, r, q, stationary, n = 1000,
         algorithm = metropolis)
{
     xs = numeric(n+1)
     xs[1] = x.0
     for(i in 1:n) {
       y = r(xs[i])
       k = algorithm(xs[i], y, stationary, q)
       xs[i+1] = ifelse(runif(1) <= k, y, xs[i])
       if(is.na(xs[i+1]))
        stop("Problems with missing value")
     }

     class(xs) <- "mcmc"
     xs
}
```


- So it is possibly our use of ifelse()
- Is there an alternative that we can try to see if it improves matters?
- Can use
  xs[i + 1] = if(runif(1) <= k) y else xs[i]
- Change the code, and re-run the profiling.

	self.time	self.pct	total.time	total.pct
"runif"	0.16	20.0	0.16	20.0
"rnorm"	0.14	17.5	0.14	17.5
"dnorm"	0.10	12.5	0.10	12.5
"mcmc"	0.08	10.0	0.74	92.5
"algorithm"	0.08	10.0	0.34	42.5
"min"	0.06	7.5	0.16	20.0
"gc"	0.06	7.5	0.06	7.5
"stationary"	0.06	7.5	0.06	7.5

- So now it is the low-level internal functions runif, rnorm, dnorm that take the time.
- Mcmc and algorithm which is the hastings() function might be improved.
- But let's compare the times with ifelse and if



Vectorize

- Pass vector arguments rather than single values to R functions.
- E.g. sum(x) rather than
  ans = 0
  for(i in x) ans = ans + x[i]
- E.g. grep("abc", lines)
- And write functions you create to accept vectors.

Avoid concatenation!

- When looping and creating the answer such as a matrix or vector/list,
  - preallocate the result as an empty data structure with the correct size and
  - then fill in the elements.
- Do not concatenate the i-th result to the previous ones.

Preallocate Space for the Result

- Consider the following code
  ```
  ans = numeric()
  for(i in 1:n)
     ans = cbind(ans, mean(rnorm(1000)))
  ```
- In each step, we combine the new result with the previous ones via cbind.

- Consider the last iteration, i.e. i == n
- The result from the previous iteration is a vector with n-1 elements.
- We then create a new result with n elements.
- So before we a… …ve have approximately 2…
- And we have to … …riginal to the new result.
- This is bad news.
  Some computations will not be feasible.

Introduces fragmented memory with small allocations in different place

Alternative

- We know the result is a numeric vector with n elements,
  so allocate it first and then assign each iteration's result into the corresponding column.
- ans = numeric(n)
  for(i in 1:n)
    ans[i] = mean(rnorm(1000))
- This does the allocation (for the result) just once and doesn't create new objects, just modifies the existing one.
- The key thing is that ans[i] doesn't create a new copy of ans, but writes the values into the appropriate

Time Comparisons

- system.time({ans = numeric() ; for(i in 1:10000) ans = cbind(ans, rnorm(10))})
  [1] 14.57 4.72 19.62 0.00 0.00
- system.time({ans = matrix(NA, 10, 10000) ; for(i in 1:10000) ans[,i] = rnorm(10)})
  [1] 0.32 0.01 0.34 0.00 0.00
- Of course, need to have multiple measurements to get better estimates.
- And the characteristics of the machine, etc. matter, but still can compare the two meaningfully.

- We could use apply() to make this read more easily and be more efficient
  sapply(1:n, function(i) rnorm(10))
  or replicate(n, rnorm(10))
- The apply functions allocate the result space for us.
- Note that we can define an "anonymous" function in the call to sapply().
  functions are first class objects in R.
- But when we can't use an apply function, making space and writing into that existing space is much faster.

Why do we need to know about memory?

- Because, when you run simulations as for your current project, you may run into memory problems.
  It then helps to be able to reason about them.
- It is good to be able to determine approximately how much memory you will need in a computation. Then you can determine if it is feasible or not.
- And it can also allow you to specify hints to R for how much space it will need and can reserve.

Garbage Collection

- Note that you never have to tidy up and remove values when you no longer need them.
- R does it for you, but at a small cost.
- When R starts, it allocates a pool of memory it can use for vectors, etc.
- When R needs to allocate space, it see if it has enough and if not, reclaims no-longer used memory via garbage collection.
- Then allocates the needed space.
- The reclaiming of space can take time.
  Also, has to grow the pool in certain cases.

- If we know that we will need a certain amount of space (how?), then we can tell R to preallocate a big enough pool.
- Then the garbage collection won't occur, or at least as often.
- We can ask for a large amount of memory when starting R using, e.g.
  R --min-vsize=.5G
- See help for Startup, Memory

- Start R with default settings, i.e. just run R with no extra command line arguments.
- Now, let's create a large matrix - 1000 x 1000
- Before we do, ask R to tell us when it does garbage collection/resizing of the available space.
  Use gcinfo(TRUE).

- gcinfo(TRUE)
- m = matrix(rnorm(1000 * 1000), 1000, 1000)
- ```
  Garbage collection 4 = 1+0+3 (level 2) ...
  180323 cons cells free (51%)
  9.6 Mbytes of heap free (95%)
  Garbage collection 5 = 1+0+4 (level 2) ...
  180330 cons cells free (51%)
  9.6 Mbytes of heap free (54%)
  Garbage collection 6 = 1+0+5 (level 2) ...
  180333 cons cells free (51%)
  9.6 Mbytes of heap free (37%)
  ```
- ```
  object.size(m)
  [1] 8000120
  ```

- Now, let's try that again, but this time start R with 2Gb of memory.
  Don't do this unless you know you need it!
- Start R and tell it to use 2Gb of space for data objects
  R --min-vsize=2G

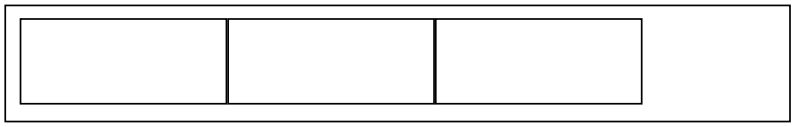
- Again, turn on reporting of garbage collection
  gcinfo(TRUE)
- Now, allocate the same matrix.
  m = matrix(rnorm(1000 * 1000), 1000, 1000)
- Note, there was no garbage collection.

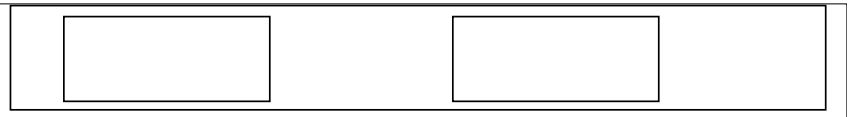
General Lesson

- Don't normally have to tell R about the memory you will need
- But, if you have knowledge about the application, you can provide it and often get some improvement.
- And, when working with large data and complex tasks, it is important to be able to know how much memory an object will consume and whether you can handle 2, 3 or 4 copies of it in memory.

Fragmentation

- Fragmentation happens when we create numerous objects and then remove some and leave holes in the allocated memory.
- x1 = rnorm(10000)
  x2 = rnorm(10000)
  y = 10 * x1 + 20 * x2
  rm(x2)





- When we remove x2, we are left with a big hole.
- If we go to allocate space for say 10001 elements, we cannot use this space.
- We may have lots of little pieces of space which cumulatively total more than the desired amount of new space.
- But since they are not contiguous, we cannot use them and so we cannot satisfy the new request.
- We don't have much control over this in R, but it is good to know about it.

Recursion

- Recursion is a technique that is often used to program certain tasks.
- It is essentially a way to loop or iterate over different states.
- It can be very natural and greatly simplify certain problems.
- It can also be quite inefficient and more clumsy iterative techniques can be more efficient.

Fibonacci Numbers

- Fibonacci introduced a sequence of numbers defined by the n-the element F_n
  $F_o = 0$,
  $F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$, $n > 1$
- It is a sequence that arises in many different contexts and has amazing mathematical properties.
- For our purposes, note that the value for n is computed from previously computed values, i.e. for n - 1 and n - 2.

Fibonacci function

- Let's write an R function to calculate the value of the Fibonacci sequence for a given n.
- ```
  fibonacci =
  function(n) {
    if(n == 0 || n == 1)
      return(n)

    fibonacci(n - 1) + fibonacci(n - 2)
  }
  ```
- This is nice and simple.
  The function calls itself - recursion.
  ```

Basics of Recursion

- The function
  - calls itself
    with a different argument!
  - does some computations to solve the simple or special cases on the original argument, e.g. n = 0, 1.
- Any recursive algorithm can be written in an iterative manner - i.e. using loops.

Iterative Fibonacci

```
fib2 =
function(n)
{
  if(n == 0 || n == 1) return(n)
  if(n == 2)  return(1)

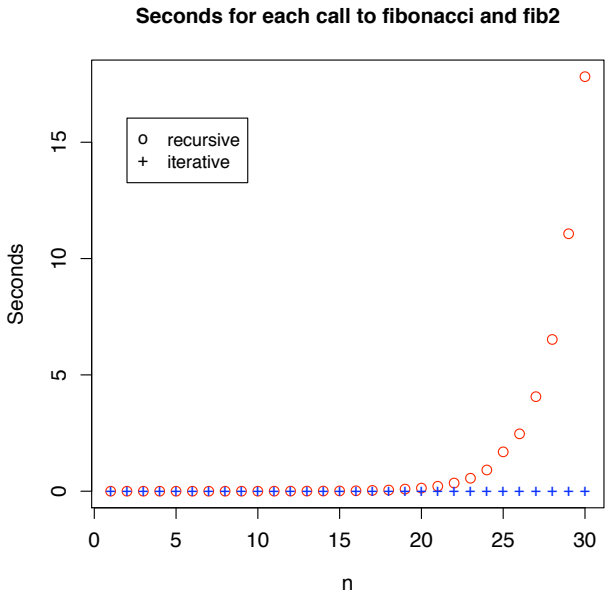
  f1 = f2 = 1
  for(i in seq(2, n-1)) {
    f = f1 + f2
    f2 = f1
    f1 = f
  }
  f
}
```

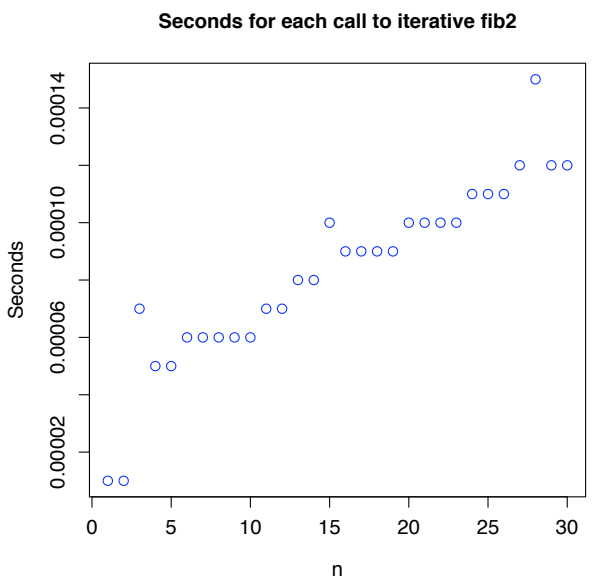

- Let's compare these in terms of speed
  Which one will be faster?
- Can you compare them in your head or on paper?
- Or empirically
  Create a simple experiment to measure the time
  do 20 repetitions each calculating F_{20}.

- How do we measure time for a computation:
  system.time(command)
- Get back a vector with 5 elements:
  user time, system time, cpu time (and sub-processes)

- system.time(fib2(20))
  [1] 0 0 0 0 0
- Finite resolution that depends on the operating system. (usually 1/1000 second)
- So repeat the calculations many times to get longer times. Then divide by the number of times performed.
- fib2.times =
  system.time(sapply(1:1000, function(x) fib2(20)))
  [1] 0.10 0.00 0.11 0.00 0.00
- So total time of .11 seconds, per call .11/1000

- The iterative version is much, much faster as n gets big.
- Repeat it for various values of n, e.g. 1, 2, …, 30
- Then plot n & time taken and see if you see any relationship.
- fib2.times =
  sapply(1:30,
  function(i)
  system.time(sapply(1:1000,
  function(x) fib2(i))))
- Same for fibonacci, and dynFibonacci.





When plotted on the appropriate scale, it is approximately linear.

Dynamic Programming

- Another approach is to use the simple recursive algorithm but to store the values we have previously computed.
  Access these in subsequent calls.

Dynamic Fibonacci

```
.Fibonacci<- c(1, 1)
dynFibonacci <-
function(n)
{
  top = length(.Fibonacci)
  if(top >= n)
      return(.Fibonacci[n])

 for(i in seq(top + 1, n)) {
     .Fibonacci[i] <<- .Fibonacci[i - 1]  + .Fibonacci[i - 2]
  }
  .Fibonacci[n]
}
```


- Call dynFibonacci(10), then dynFibonacci(6) and it is already calculated and stored.
- dynFibonacci(20) can start from the previous highest element, i.e. F_{10}.
- This is called memoization and is essentially Dynamic Programming.
  Solve a problem by solving smaller problems and store the results for these smaller problems for repeated reuse.