EFFICIENT R CODE

R Ladies 4th December 2018

Marian Keane

WHY WRITE EFFICIENT CODE?

- Code will run faster
- Algorithm can scale up to larger datasets
- Less frustrating all round

SAMPLE DATASET

```
rm(list=ls())
n <-10^5
col1 <- runif (n, 0, 2)
col2 <- rnorm (n, 0, 2)
col3 <- rpois (n, 3)
col4 <- rchisq (n, 2)
df <- data.frame (col1, col2, col3, col4)
```

col1	col2	col3	col4
1.788725	2.559988	2	1.210794
1.444972	2.01943	4	2.913491
1.398471	-0.45867	2	3.438176
0.874435	-0.19789	2	1.999501
0.489978	0.787998	3	0.214151
0.942418	-0.02275	2	0.67963
0.119262	0.920319	3	1.639454
0.475953	0.741586	4	0.774466
1.9303	0.569817	4	6.814686
0.333456	-2.99974	1	6.674212

Source: https://www.r-bloggers.com/strategies-to-speedup-r-code/ or https://datascienceplus.com/strategies-to-speedup-r-code/

SAMPLE ALGORITHM

```
 for (i in 1:nrow(df)) { # for every row
    if ((df[i, 'col1'] + df[i, 'col2'] + df[i, 'col3'] +
df[i, 'col4']) > 4) { # check if > 4
      df[i, 5] <- "greater_than_4" # assign 5th column
    } else {
      df[i, 5] <- "less_than_or_equal_to_4" # assign 5th
column
    }
  }
```

Time it using system.time()

WHY DO WE WRITE INEFFICIENT CODE?

- Copying old code that was for another purpose
- Poor understanding of how memory is used (in R)
- Poor understanding of algorithm complexity

FIX #1: VECTORISE AND PRE-ALLOCATE DATA STRUCTURES

```
output <- character (nrow(df)) #Vectorise and pre-allocate
system.time({
  for (i in 1:nrow(df)) {
    if ((df[i, 'col1'] + df[i, 'col2'] + df[i, 'col3'] + df[i,
'col4']) > 4) {
      output[i] <- "greater_than_4"
    } else {
      output[i] <- " less_than_or_equal_to_4"
    }
  }
df$output <-output})
```

FIX #1: VECTORISE AND PRE-ALLOCATE DATA STRUCTURES

```
output <- character(nrow(df))    #Vectorise and pre-allocate
system.time({
  for (i in 1:nrow(df)) {
    if ((df[i, 'col1'] + df[i, 'col2'] + df[i, 'col3'] + df[i,
'col4']) > 4) {
      output[i] <- "greater_than_4"
    } else {
      output[i] <- " less_than_or_equal_to_4"
    }
  }
df$output <-output})
```

Pre-allocate

Vectorise

FIX #2: TAKE CONDITION CHECKING OUTSIDE THE LOOP

```
# after vectorization and pre-allocation, taking the condition checking
outside the loop.

output <- character (nrow(df))

condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4  # condition
check outside the loop

system.time({
  for (i in 1:nrow(df)) {
    if (condition[i]) {
      output[i] <- "greater_than_4"
    } else {
      output[i] <- " less_than_or_equal_to_4"
    }
  }
  df$output <- output
})
```

FIX #3: RUN THE LOOP ONLY FOR TRUE CASES

```
output <- c(rep("less_than_or_equal_to_4",nrow(df)))
#vectorise, pre-allocate and set default value to False
condition output


condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4

system.time({

  for (i in (1:nrow(df))[condition]) {  # run loop only for
true conditions

      output[i] <- "greater_than_4"

  }

  df$output <-output })
```

FIX #4: USE IFELSE() WHENEVER POSSIBLE

```
output <- character (nrow(df))

system.time({

  output <- ifelse ((df$col1 + df$col2 + df$col3 + df$col4)
> 4, "greater_than_4", "less_than_or_equal_to_4")

  df$output <- output

})
```

FIX #5: USE WHICH()

```
system.time({
  want = which(rowSums(df) > 4)
  output = rep("less_than_or_equal_to_4", times =
nrow(df))
  output[want] = "greater than 4“
  df$output <-output
})
```

LETS TRY A NEW PROBLEM…

Problem type:

Given a set of ENTRY DATES and EXIT DATES, count the number of people IN a given state on each date.

Example – let's say these are check in / check out dates from a hotel.

Person	Date of Entry	Date of Exit
0001	1st January 2018	10th January 2018
0002	3rd January 2018	8th January 2018
0003	3rd January 2018	9th January 2018

If these are the only guests, how many are in the hotel each night?

1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
1	1	3	3	3	3	3	2	1	0

CREATE SOME DUMMY DATA

```
start <-as.Date("2008-01-01")
end <-Sys.Date()
timeseq <-seq(start,end,by=1)

n <-100
entrydate <-sample(timeseq,n,replace=TRUE)
length_of_stay <-rpois(n,5) *365
exitdate <-entrydate+length_of_stay


inforce <-data.frame(date=timeseq,
       headcount=rep(0,length(timeseq)))
```

ATTEMPT #1: ITERATE OVER DATES

```
inforce1 <-
data.frame(date=timeseq,headcount=rep(0,length(timeseq)))

count_inforce <-function(entry,exit,mydate){ #entry and
exit are vectors of dates, mydate is a single date
  inforce <-0
  for (i in 1:length(entry)) {
       inforce <-inforce + ifelse(mydate %in%
              seq(entry[i],exit[i],by=1),1,0)
              }
  return(inforce)
}
system.time(
       for (i in 1:nrow(inforce1)) {
              inforce1$headcount[i] <-
       count_inforce(entrydate,exitdate,inforce1$date[i])
       }
       )
```

ATTEMPT #2: ITERATE OVER PEOPLE

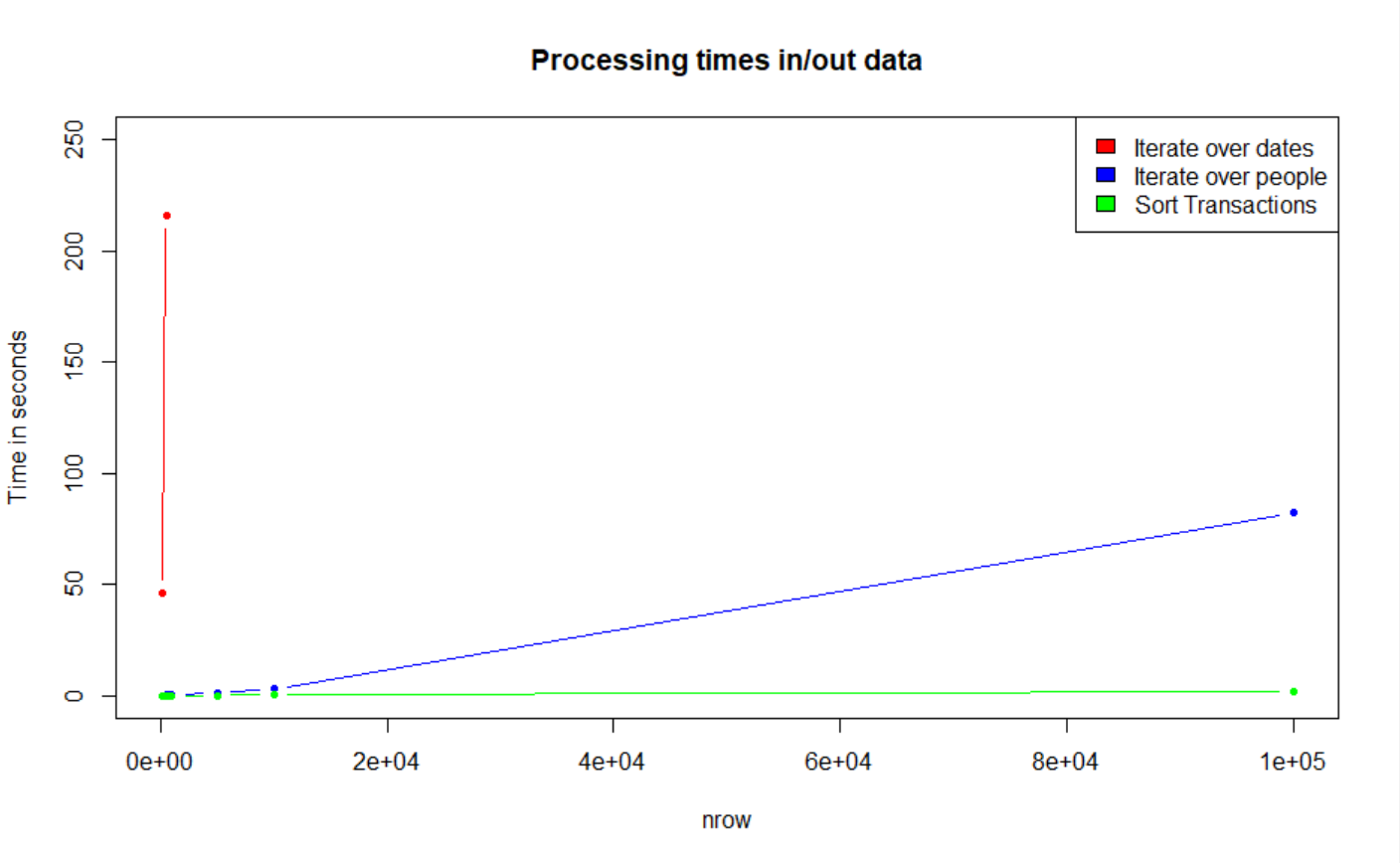
```
inforce2 <-data.frame(date=timeseq,
       headcount=rep(0,length(timeseq)))

system.time(
       for (i in 1:length(entrydate)){
              indates <-which(inforce2$date %in%
                     seq(entrydate[i],exitdate[i],by=1))
              inforce2$headcount[indates] <-
                     inforce2$headcount[indates]+1
              }
       )
```

ATTEMPT #3: ARRANGE DATA AS TRANSACTIONS

```
#https://www.geeksforgeeks.org/find-the-point-where-maximum-intervals-overlap/
inforce3 <-data.frame(date=timeseq)
system.time({
        transactions_in <-data.frame(date=entrydate,
                change=rep(1,length(entrydate)))
        transactions_out <-data.frame(date=exitdate+1,
                change=rep(-1,length(exitdate)))
        transactions <-rbind(transactions_in,transactions_out)
        transactions <-transactions[order(transactions$date),]
        transactions <-aggregate(change~date,
                data=transactions,FUN=sum)
        transactions$headcount <-cumsum(transactions$change)
        inforce3 <-merge(inforce3,
                transactions[,c("date","headcount")],
                by="date",all.x=TRUE)
        #Fill in NA values with last non-NA value
        inforce3$headcount <-repeat.before(inforce3$headcount)
        })
```

PROCESSING TIMES



CONCLUSION

- Vectorise and pre-allocate.
- Avoid for() loops if possible.
- Ifelse() usually faster than if().
- Sometimes the whole algorithm needs to be reconsidered.
- Other options not discussed – apply() functions, data tables, etc.

APPENDIX

```
#The repeat.before function
#This was used as a final clean-up step in the sorted
transactions algorithm.

repeat.before = function(x) {   # repeats the last non NA value. Keeps
leading NA
    ind = which(!is.na(x))      # get positions of nonmissing values
    if(is.na(x[1]))             # if it begins with a missing, add the
          ind = c(1,ind)        # first position to the indices
    rep(x[ind], times = diff(   # repeat the values at these indices
       c(ind, length(x) + 1) )) # diffing the indices + length yields
how often
}
```