Stat 243: Problem Set 4

Eugene Yedvabny

10/15/2014

1

The appearance of the `3` over a `2` is due to lazy evaluation. The value for `x` is only evaluated when `x` is called by the `+` operator within `f1`. Since `x` assigns `y` as part of its evaluation block, lazy eval ensures that default `y=0` is overwritten and thus `x + y = 2 + 1 = 3`.

2

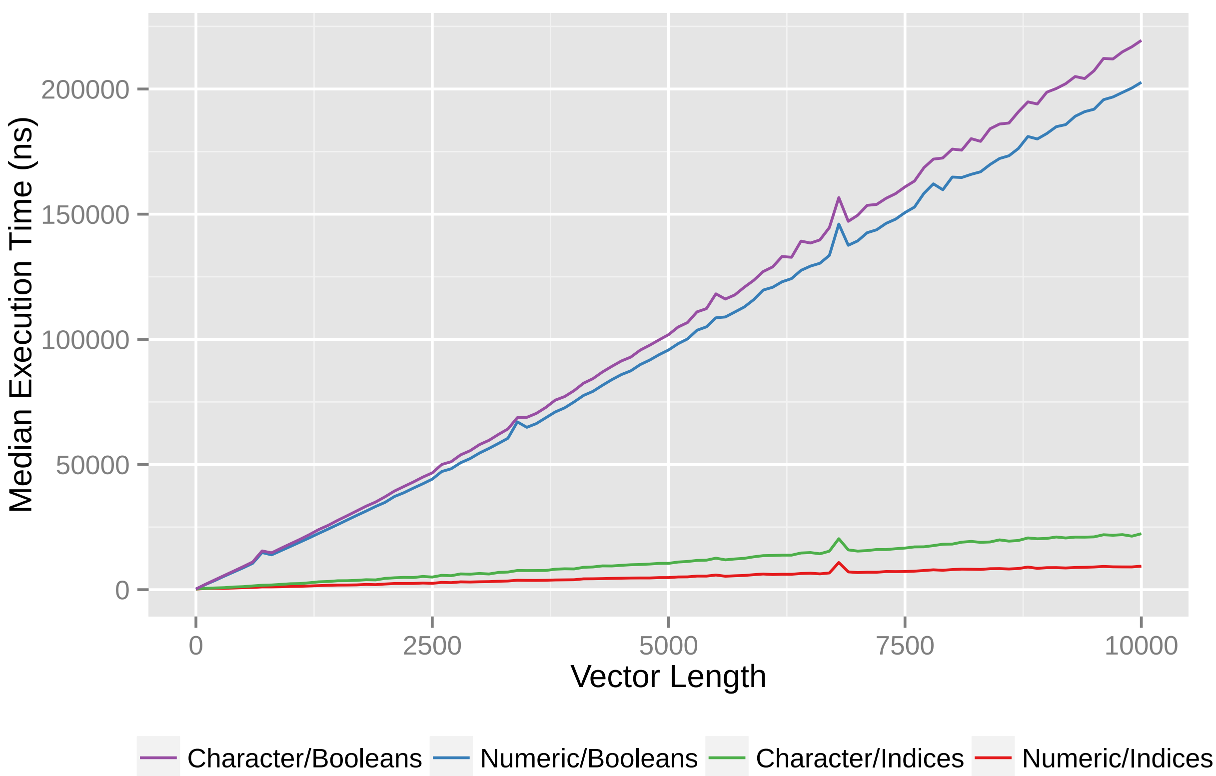
The answer to this question will assume that the time it takes to *create* the index vector or the boolean vector is irrelevant. As such I am going to create vectors of varying sizes and types, as well as their indexing vectors, and only benchmark the subsetting operation itself. I've created the following helper function to generate a dataframe for the each length of the vector I want to subset.

```
library(dplyr)
library(microbenchmark)

# The following function will benchmark list subsetting for numeric and string lists
# The input parameter is the length of the vector to be subset
benchSubsetting <- function(length){
  # Create a random vector of numbers
  test.nums <- runif(length)

  # Create a random vector of strings (of 10 characters)
  test.strs <- replicate(length,
                         paste(sample(letters, 10, replace = T),
                               collapse=''))

  # Create an accessing boolean vector (skewed probs favor smaller subsets)
  test.bool <- sample(c(T,F), length, replace=T, prob=c(0.3,0.7))

  # Create an interger index corresponding to the same items as above
  test.index <- which(test.bool)

  # Run the test
  test.timing <- microbenchmark(
    nums.ints = test.nums[test.index],
    nums.bools = test.nums[test.bool],
    strs.ints = test.strs[test.index],
    strs.bools = test.strs[test.bool]
  )

  # Summarize on the mean timing?
  test.timing %>%
    group_by(expr) %>%
    summarize(median.time = median(time)) %>%
```

```r
    mutate(length = length)
}
```

The **microbenchmark** package executes each expression 100 times and returns the range of timings. Since there are often high-exec-time outliers, best metric to look at is the median of the timing distribution.

```r
source("benchSubsetting.R")

# Enough lengths to get a good plot going
bench.data <- seq(0,10000,100) %>% lapply(benchSubsetting) %>% rbind_all

ggplot(bench.data,aes(x=length, y=median.time, color=expr)) +
  geom_line() +
  scale_color_brewer(type="qual", palette=6, name="",
                     breaks=c("strs.bools","nums.bools",
                              "strs.ints","nums.ints"),
                     labels=c("Character/Booleans","Numeric/Booleans",
                              "Character/Indices","Numeric/Indices")) +
  xlab("Vector Length") +
  ylab("Median Execution Time (ns)") +
  theme(legend.position = "bottom")
```



As is evident from the figure above, subsetting by index is *a lot* faster than subsetting by boolean. The perfomance costs are linear with vector length and are slightly higher for character vectors than numeric vectors. Since numeric vectors are stored continuosly while character vectors are essentially arrays of pointers to individual strings, this likely yields the observed difference in subsetting time.

The difference between numeric and character vectors, however, pales in comparison to the gap between index-subsetting and boolean-subsetting. I've generated indexing boolean vectors of the same size as the data vector, so recycling should not be a factor in the timing. The issue really comes down to the number of elements to scan. I've set the parameters to recall about 30% of a vector's elements. In the case of a 100-element data vector, index subsetting would pass in 30 indices and underlying code would know exactly which memory addresses to compile into the new vector. In the case of boolean values, an entire 100-element vector is passed in, and the underlying code has to scan it all to check which of the data elements should be returned. So while the boolean subsetting is *convenient*, as it allows the use of vectorized logic expressions to generate the keys, it scales very poorly with vector size.

What's more telling is that this performance difference holds even when the number of passed-in elements and returned elements is the same for both methods. In the following example I am requesting all 1000 of the vector elements, in one instance by passing a 1000-entry logical vector of T, and in another a 1000-entry vector of indices.

```
test.data <- runif(1000)
test.bool <- rep(T,1000)
test.ind <- c(1:1000)

microbenchmark(
  with.bool = test.data[test.bool],
  with.ind = test.data[test.ind]
)
```

```
## Unit: microseconds
##       expr    min      lq     mean  median      uq    max neval
##  with.bool 18.751 18.9135 24.33168 19.1575 28.1205 63.297   100
##   with.ind  3.069  3.2095  6.70716  3.4400  6.4150 35.958   100
```

Based off the median timing, indexing with booleans is six times slower than indexing with indices, regardless of the vector length or number of elements requested.

3

For the purposes of answering this question, I am going to rely on the `pryr` package as it makes checking memory locations a lot easier. The package is up on CRAN now, so I figure it is fair game. **dplyr** also has convenience functions `location` and `changes` for tracking data frame memory addresses and they are a lot cleaner in presentation than `.Internal(inspect())`.

```
# Let's start of by creating heterogeneous data
strings <- replicate(100, paste(sample(letters, 10, replace = T), collapse=''))
numbers <- runif(100)
booleans <- sample(c(T,F),100,replace=T)
factors <- factor(sample(c("one","two","three"),100,replace=T))
raw.addr <- c(address(strings),address(numbers),address(booleans),address(factors))

# Merge the data into a dataframe
data.f <- data_frame(strings,numbers,booleans,factors)
frame.addr <- location(data.f)[[2]]

# Merge the data into a list
data.l <- list(strings,numbers,booleans,factors)
list.addr <- sapply(data.l,address)
```

```r
# We can verify that the data frame is indeed just
# metadata over the underlying vectors
addresses <- data_frame(raw = raw.addr, data.frame = frame.addr, list = list.addr)
knitr::kable(addresses)
```

raw	data.frame	list
0x3d48810	0x3d48810	0x3d48810
0x33b5ac0	0x33b5ac0	0x33b5ac0
0x316a8d0	0x316a8d0	0x316a8d0
0x55af6b0	0x55af6b0	0x55af6b0

Ok, so far so good. The data frame and list are just metadata over the underlying vectors.

```r
# Metadata prior to the change
location(data.f)
```

```
## <0x53e1d40>
## Variables:
##  * strings:   <0x3d48810>
##  * numbers:   <0x33b5ac0>
##  * booleans:  <0x316a8d0>
##  * factors:   <0x55af6b0>
## Attributes:
##  * names:     <0x53e1c20>
##  * row.names: <0x51ca380>
##  * class:     <0x53e1d88>
```

```r
# Change a value in one of the underlying vectors
data.f[10,1]<-"Hello"

location(data.f)
```

```
## <0x4f94408>
## Variables:
##  * strings:   <0x53c64a0>
##  * numbers:   <0x33b5ac0>
##  * booleans:  <0x316a8d0>
##  * factors:   <0x55af6b0>
## Attributes:
##  * names:     <0x53e1c20>
##  * row.names: <0x549f5f0>
##  * class:     <0x53e1d88>
```

The modification has replaced the entire column of the data frame. The address of the data frame itself has changed too, but since a data frame is just a wrapper around lists, that's not a costly operation. The untouched columns have remained the same. Furthermore the change is local to the data-frame; the raw string vector and the list are still pointing into the same original location. Changing one of the two makes another copy and now the raw vector, the list, and the data frame all point in different locations.

```r
c(address(strings),address(data.f$strings),address(data.l[[1]]))
```

```
## [1] "0x3d48810" "0x53c64a0" "0x3d48810"
```

```r
strings[10] <- "Hello"
c(address(strings),address(data.f$strings),address(data.l[[1]]))
```

```
## [1] "0x4f56fe0" "0x53c64a0" "0x3d48810"
```

R always makes a copy if the underlying data is referred from other locations. But now that we have three different string vectors, does the copying still take place?

```r
strings[11] <- "Goodbye"
data.f$strings[11] <- "Goodbye"
data.l[[1]][11]<-"Goodbye"
c(address(strings),address(data.f$strings),address(data.l[[1]]))
```

```
## [1] "0x5839be0" "0x5db6fa0" "0x4c99b70"
```

It appears the answer is *yes*. Even though each object has their own version of the string vector, an element change results in an entirely new vector being created. The unmodified elements, however, remain the same across all three objects as they haven't been modified since creation.

```r
c(address(numbers),address(data.f$numbers),address(data.l[[2]]))
```

```
## [1] "0x33b5ac0" "0x33b5ac0" "0x33b5ac0"
```

```r
numbers[2]<-10;
data.f$numbers[2]<-10;
data.l[[2]][2]<-10;
c(address(numbers),address(data.f$numbers),address(data.l[[2]]))
```

```
## [1] "0x51ae770" "0x53a2bf0" "0x4e8ab60"
```

```r
numbers[2]<-15;
data.f$numbers[2]<-15;
data.l[[2]][2]<-15;
c(address(numbers),address(data.f$numbers),address(data.l[[2]]))
```

```
## [1] "0x4e6b560" "0x4e82de0" "0x5789860"
```

Same issue as before: even when only one pointer is accessing the vector, a new copy is still being made. I know for a fact that this should not be the case in newer versions of R, hence the following disclaimer.

Disclaimer: I've ran these codes through RStudio and knitr, which are known to yield different results than raw R. Since I cannot verify 100% what's going in the background with these packages, I am going to default to Hadley Wickham's extensive explanation of R's memory management, available (http://adv-r.had.co.nz/memory.html)[here].

4

I am still a bit confused on how to have the S3 object maintain a list of attributes, such as the starting point, but only pass back a single value. Right now the printing of the result is well-formatted, but the actual returned object has access to all atributes of the rw, including the raw path. In any case, the function is fully vectorized and accomplished the random-walk task as an S3 class

```
# Compute a 2D random walk
# Input: number of steps, whether to return the full path
# Output: an (x,y) vector or list of such vector if full path

rw <- function(num.steps = 0, full.path = F){

  # Sanity check on the input
  if(!is.numeric(num.steps) | num.steps <= 0 ){
    stop("Invalid number of steps")
  }

  # Since this is a random walk, just generate the moves
  moves <- list(c(1,0),c(-1,0),c(0,1),c(0,-1))
  steps <- sample(moves,num.steps,replace=T)
  steps.matrix <- matrix(unlist(steps),ncol=2,byrow=T)

  # Sum up the moves and add them
  steps.matrix[,1] <- cumsum(steps.matrix[,1])
  steps.matrix[,2] <- cumsum(steps.matrix[,2])
  colnames(steps.matrix) <- c("x","y")

  walk <- list(origin = c(0,0), full.path = full.path)
  class(walk)<-"rw"
  walk$path <- steps.matrix

  return(walk)
}

# Print a well-formatted representation
print.rw <- function(walk, ...){
  if(walk$full.path){
    print(walk$path)
  }else{
    print(walk$path[length(walk$path)/2,])
  }
}

# Plot the walk on a 2D chart
plot.rw <- function(walk, ...){
  data <- data_frame(x=walk$path[,1],y=walk$path[,2])
  ggplot(data,aes(x=x,y=y))+
    geom_path()
}

# Return the ith step of the random walk
`[.rw` <- function(walk,i,...){
  if(!is.numeric(i) || i <= 0){
```

```r
    return(walk$origin)
  }else{
    return(walk$path[i,])
  }
}

# Return the starting coordinate
start <- function(object, ...) UseMethod("start")
start.rw <- function(object){
  return(object$origin)
}

# Set the starting coordinate
`start<-` <- function(object, ...) UseMethod("start<-")
`start<-.rw` <- function(object,value){
  object$origin <- value
  return(object)
}
```

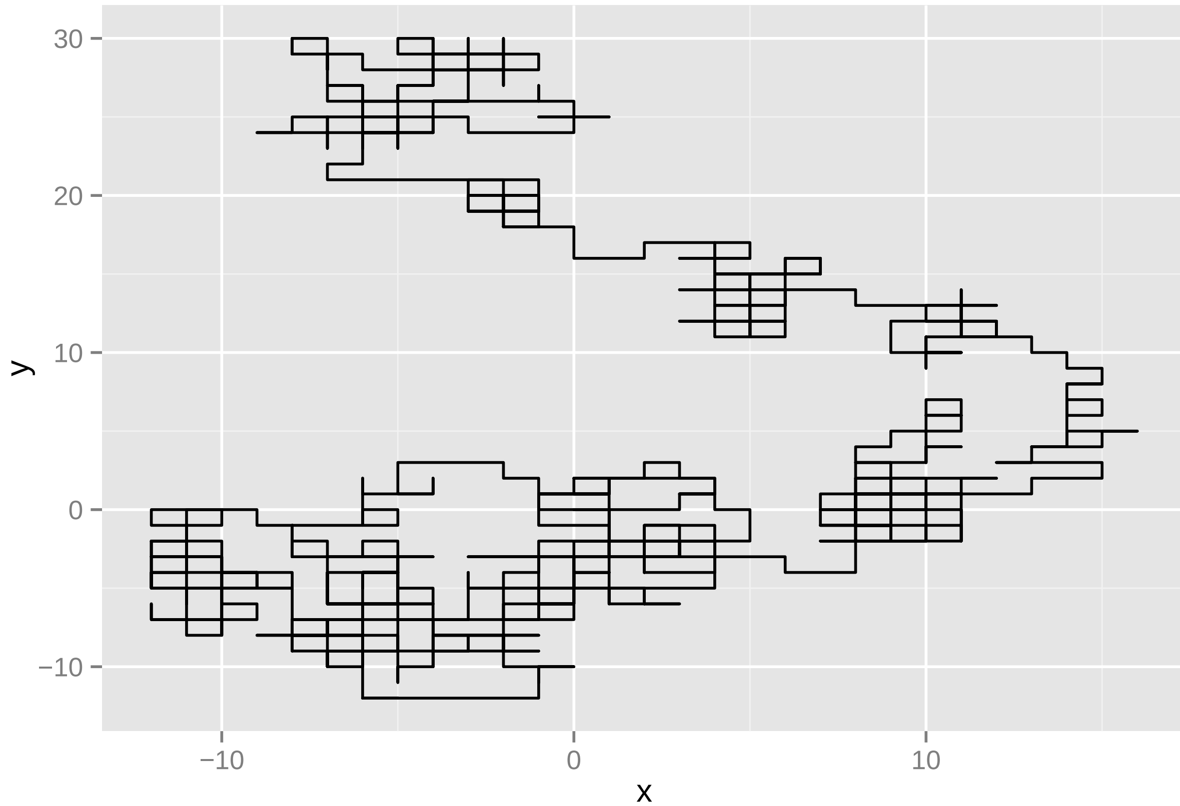
```r
source("randomWalk.R")

walker <- rw(1000,T)
walker2 <- rw(1000,F)

# First walker will only return the end
# Second walker will print the fill path
walker2
```

```
##    x   y
##   -3 -29
```

```r
head(walker)
```

```
##      x y
## [1,] 1 0
## [2,] 1 1
## [3,] 0 1
```

```r
# Use ggplot2 path geom to chart out the walk
plot(walker)
```




```
# The origin is set to 0,0
# Calling start() allows resetting it
start(walker)
```

```
## [1] 0 0
```

```
start(walker) <- c(5,7)
start(walker)
```

```
## [1] 5 7
```

```
# We can of course get random elements of the walk
walker[10]
```

```
## x y
## 2 2
```

```
walker[25]
```

```
## x y
## 1 0
```