Comparing Four Methods for Findng Factorial

Sheila Braun

July 16, 2018

Contents

Part 1

The objective is to write a function that computes the factorial of an integer greater than or equal to 0. The factorial of 0 is defined to be 1. Here are four different versions of the Factorial function:

Factorial_loop()

```
Factorial_loop <- function(x) {
      result <- x
      if (x < 0) (return(paste('You entered a negative number: not gonna work.')))
      if (x < 1) (return(1))
      for (i in (x - 1):1) {
            result <- result * i
            }
      result
}
x = 5
Factorial_loop(x)
```

```
## [1] 120
```

Factorial_reduce()

```
library(purrr)
```

```
## Warning: package 'purrr' was built under R version 3.5.1
```

```
Factorial_reduce <- function(x) {
      if (x < 0) (return(paste('You entered a negative number: not gonna work.')))
      if (x < 1) (return(1))
      reduce(x:1, function(x, y) {
            x * y
      })
}
```

```
x = 5
Factorial_reduce(x)
```

```
## [1] 120
```

Factorial_func()

```
Factorial_func <- function(x) {
      if (x == 0) (1)
      else if (x < 0) {
            paste('You entered a negative number: not gonna work.')
      } else {
            return(Factorial_func(x - 1) * x)
      }
}
x = 5
Factorial_func(x)
```

```
## [1] 120
```

Factorial_mem()

```
previous_factorials <- 1
Factorial_mem <- function(n) {
      if (x == 0) (return(1))
      else if (x < 0) return(paste('You entered a negative number: not gonna work.'))

      #grow previous_factorials if necessary
      if (length(previous_factorials) < n) previous_factorials <<- `length<-`(previous_factorials, n)

      #return pre-calculated value
      if (!is.na(previous_factorials[n])) return(previous_factorials[n])

      #calculate new values
      previous_factorials[n] <<- n * Factorial_mem(n - 1)
      previous_factorials[n]
}

x = 5
Factorial_mem(x)
```

```
## [1] 120
```

Benchmarks

The next code chunk uses a range of inputs to time the operation of the four functions above. It also provides a visual summary of their performance.

```
library(dplyr)
library(purrr)
library(magrittr)
```

```r
library(tidyr)
library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 3.5.1
```

```r
library(ggplot2)
```

```r
#Function to run a benchmark on one of the routines
benchmark_it <- function(fun = Factorial_loop) {
      bench_data <- map(1:10, function(x) {microbenchmark(fun(x),
                                                         times = 1000)$time})
      names(bench_data) <- paste(1:10)
      bench_data <- as_tibble(bench_data)
      bench_data %<>%
            gather(num, time) %>%
            group_by(num) %>%
            summarise(median_times = median(time))
      bench_data$num <- as.numeric(bench_data$num)
      bench_data[order(bench_data$num),]
}
```

Now run the benchmark_it() function on each of the four factorial-calculating functions.

```r
loop_data <- benchmark_it(Factorial_loop)
reduce_data <- benchmark_it(Factorial_reduce)
recursion_data <- benchmark_it(Factorial_func)
memo_data <- benchmark_it(Factorial_mem)

#Combine them in a single df

dt <- cbind(loop_data[,1:2],
             reduce_data[,2],
             recursion_data[,2],
             memo_data[,2])
colnames(dt) <- c('Number', 'Median Loop Times', 'Median Reduce Times',
                           'Median Recursion Times', 'Median Memoization Times')
library("reshape2")
```

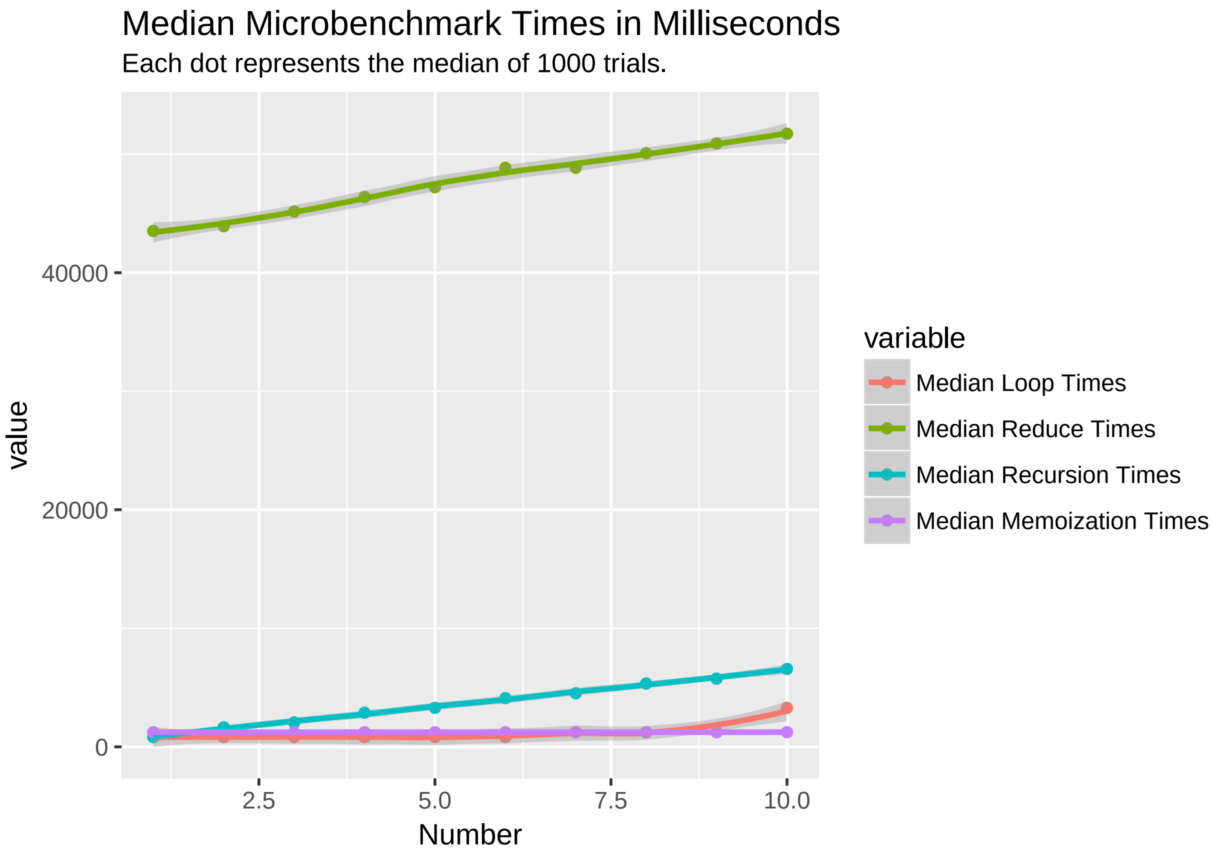
```
##
## Attaching package: 'reshape2'

## The following object is masked from 'package:tidyr':
##
##     smiths
```

```r
test_data_long <- melt(dt, id = "Number")  # convert to long format
```

Next, plot the times.

```r
suppressWarnings(print(
      ggplot(test_data_long, aes(Number,value,
                           col = variable)) +
            geom_point() +
            geom_smooth() +
            labs(title = "Median Microbenchmark Times in Milliseconds",
                 subtitle = "Each dot represents the median of 1000 trials.")
))
```



The loop times seem surprisingly low. They were done with a backwards loop multiplying the previous result by the index. That saved a lot of time. Recursion with memoization also performed well, as expected.