# Pospesitev R-a

*Lara Lusa in Nina Ruzic Gorenjec*

*31 maj 2019*

Hadley Wickham, Advanced R: *R is not a fast language. This is not an accident. R was purposely designed to make data analysis and statistics easier for you to do. It was not designed to make life easier for your computer. While R is slow compared to other programming languages, for most purposes, it's fast enough.*

## Kako merimo cas, ki ga R uporabi za izvedbo kode

Uporabimo knjiznico `microbenchmark`, ki zelo natancno meri trajanje izvedbe zelo majhnih delckov kode.

```
#install.packages("microbenchmark")
require(microbenchmark)
```

Funkcija `microbenchmark` izvede neko funkcijo 100 krat (lahko spremenimo stevilo s parametrom `times`) in oceni hitrost vsake ponovitve. Rezultat je povzetek casa izvedbe funkcije (min, max, 1. in 3. kvartil, povprecje, mediana).

Na primer, lahko primerjamo hitrost izvedbe funkcije `mean()` in hitrost nase funkcije, ki rocno izracuna povrecno vrednost. Katera mislite, da bo hitrejsa?

```
f.my.mean <- function(x)
  sum(x)/length(x)
```

```
x <- runif(100)

microbenchmark(
  mean(x),
  f.my.mean(x)
)
```

V tem primeru so enote s katerimi je povzeta hitrost nanosekunde. Spomnimo se: ce za izvedbo potrebujemo 1 ms, potem 1.000 ponovitev traja 1s;

1 micros, potem 1.000.000 ponovitev traja 1s;

1 ns, potem 1.000.000.000 ponovitev traja 1s.

- Izracunajte kolikokrat lahko izvedemo funkcijo `mean` za 100 podatkov v eni sekundi (osredotocite se ne mediano). Kaj pa funkcijo `f.my.mean`?

```
#mean
1/(3162*10^(-9))
```

```
## [1] 316255.5
```

```
#f.my.mean
1/(396*10^(-9))
```

```
## [1] 2525253
```

- Odgovorite na prejsnje vprasanje brez izracunov (Namig: nastavite parameter `unit="eps"`- evaluations per second v funkciji `microbenchmark`, rezultat bo podan v stevilu ocenjenih funkcijah v eni sekundi). Kateri izpis lazje interpretirate? (Glede hitrosti funkcij?)

```
x <- runif(100)

microbenchmark(
  mean(x),
  f.my.mean(x),
  unit="eps"
)
```

```
## Unit: evaluations per second
##          expr        min        lq     mean    median        uq      max
##       mean(x) 10054.6976  302480.3 316325.9  332778.7  332778.7 369822.5
##  f.my.mean(x)   805.8414 1663893.5      Inf 3322259.1 3322259.1      Inf
##  neval
##    100
##    100
```

- Komentirajte variabilnost hitrosti.
- Uporabite tudi funkcijo `system.time()`. Ta funkcija je manj natancna, zato boste morali izvesti funkcijo bistevno veckrat kot z `microbenchmark`. Morali boste sami nastaviti stevilo ponovitev.Za ta primer boste morali ponoviti izvedbo funkcije vsaj 100.000 krat (na primer s `for` zanko). Namig: poglejte spodnjo kodo, kjer izvedemo funkcijo 1.000.000-krat.

```
n <- 1 : 1e6
system.time(for (i in n) mean(x)) / length(n)
```

```
##     user   system  elapsed
## 3.56e-06 0.00e+00 3.56e-06
```

```
system.time(for (i in n) f.my.mean(x)) / length(n)
```

```
##    user  system elapsed
## 6.6e-07 0.0e+00 6.6e-07
```

- Primerjajte hitrost funkcije `sqrt` s hitrostjo funkcije, ki izracuna koren stevilke kot `x^0.5` (s pomocjo funkcije `microbechmark`). Uporabite tudi funkcijo `system.time()`

```
x <- runif(100)

microbenchmark(
  sqrt(x),
  x ^ 0.5
)
```

## Ali hitreje sestevamo, mnozimo ali delimo stevilke v R-u?

```
x <- sample(1:1000,1000,replace=T)
y <- sample(1:1000,1000,replace=T)

microbenchmark(
  x+y,
  x*y,
  x/y)
```

## Ali so funkcije, ki imajo vec parametrov, bolj pocasne?

Namig: Uporabite prazno funkcijo `f<-function() NULL`, in druge prazne funkcije, katerim boste dodali parametre s privzetimi vrednostimi parametrov `(a=1, b=1, ...)`. Uporabite 10.000 ponovitev za oceno hitrosti. Primerjajte funkcijo, ki nima parametrov s funkcijami, ki imajo do 5 parametrov.

```
f<- function() NULL
f.1<- function(a=1) NULL
f.2<- function(a=1, b=1) NULL
f.3<- function(a=1, b=1, c=1) NULL
f.4<- function(a=1, b=1, c=1, d=1) NULL
f.5<- function(a=1, b=1, c=1, d=1, e=1) NULL

microbenchmark(
  f(),
  f.1(),
  f.2(),
  f.3(),
  f.4(),
  f.5(),
  times=10000)
```

## Naloga: Katera funkcija je hitrejsa?

Pohitrite spodnjo funkcijo (x je lahko vektor!). Preverite, kako je razlika v hitrosti odvisna od velikosti x-a.

Poskusite izracunati zeljeno drugace. Napisite dve alternativni funkciji: eno s pomocjo funkcije `p.min`, za drugo pa uporabite vektorsko notacijo. Preverite, ali funkcije res racunajo isto. Katera je najhitrejsa in zakaj?

```
f <- function(x, a=10){
  ifelse(x<a, x, a)
}
```

```
x <- 1:2000

f <- function(x, a=10){
  ifelse(x<a, x, a)
}

f.2 <- function(x, a=10){
  pmin(x, a)
}

f.3 <- function(x, a=10){
  x[x<a] <-a
  x
}

RES1=f(x)
RES2=f.2(x)
RES3=f.3(x)

all.equal(RES1,RES2,RES3)
```

```
microbenchmark(f(x), f.2(x), f.3(x), unit="us")
```

Ko primerjate dve funkciji:

- preveri, ali z obema funkcijama dobite isti rezultat (na vecih kljucnih primerih)
- testiraj hitrost na primerno velikih vhodnih podatkih – dovolj veliki, da vidite kljucno razliko, vendar izvedba ne traja predolgo

## Hitrost izpisa elementov

Oglejte si podatke mtcars. Izpisite 32 element spremenljivke carb (11. spremenljivka po vrsti.)

```
microbenchmark(
  mtcars[32, 11],
  mtcars$carb[32],
  mtcars[[c(11, 32)]],
  mtcars[[11]][32],
  .subset2(mtcars, 11)[32]
)
```

```
## Warning in microbenchmark(mtcars[32, 11], mtcars$carb[32], mtcars[[c(11, :
## Could not measure a positive execution time for one evaluation.
```

```
## Unit: nanoseconds
##                       expr  min   lq    mean median   uq    max neval cld
##             mtcars[32, 11] 7812 8113 8668.99 8413.5 8714  28245   100   b
##            mtcars$carb[32] 4207 4508 9080.64 4808.0 5108 423362   100   b
##       mtcars[[c(11, 32)]] 3606 3906 4089.84 3907.0 4207  10517   100  ab
##          mtcars[[11]][32] 3305 3606 3705.26 3606.0 3907   4808   100  ab
##  .subset2(mtcars, 11)[32]    0    0   63.60    1.0    1    902   100  a
```

H. Wickham: *R is over 20 years old. It contains nearly 800,000 lines of code. Changes to base R can only be made by members of the R Core Team (or R-core for short). Currently R-core has twenty members (http://www.r-project.org/contributors.html), but only six are active in day-to-day development. No one on R-core works full time on R. Most are statistics professors who can only spend a relatively small amount of their time on R. Because of the care that must be taken to avoid breaking existing code, R-core tends to be very conservative about accepting new code. However, the overriding concern for R-core is not to make R fast, but to build a stable platform for data analysis and statistics.*

*. . . There's no reason that there has to be such a huge difference in performance. It's simply that no one has had the time to fix it.*

## Naloga: Primerjajte hitrost `for` zanke, `apply` in `lapply` za izracun povprecne vrednosti stoplcev matrike

Spodaj je podana funkcija, ki uporabi for zanko.

```
my.mat <- matrix(rnorm(1000*10), ncol=1000)

f.for <- function(my.mat){
  my.mean <- NULL
  for(i in 1:ncol(my.mat)) {
    my.mean <- c(my.mean, mean(my.mat[,i]))
  }
  return(my.mean)
```

```
}
```

- Spremenite kodo funkcije `f.for` tako, da bodo rezultati shranjeni v vektor, za katerega je ze od zacetka izvedbe funkcije podana velikost (dolzina) vektorja, ki vsebuje rezultate. Primerjajte hitrost nove in stare funkcije. S pomocjo nekaj primerov preverite, ali so rezultati pridobljeni z novo funkcijo natancno enaki tistim, ki ste jih dobili s funkcijo `f.for`.

```
my.mat <- matrix(rnorm(1000*10), ncol=1000)

f.for <- function(my.mat){
  my.mean <- NULL
  for(i in 1:ncol(my.mat)) {
    my.mean <- c(my.mean, mean(my.mat[,i]))
  }
  return(my.mean)
}

res.for <- f.for(my.mat)

f.for2 <- function(my.mat){
  my.mean <- numeric(ncol(my.mat))
  for(i in 1:ncol(my.mat))  {
    my.mean[i] <- mean(my.mat[,i])
  }
  return(my.mean)
}

res.for2 <- f.for2(my.mat)

all.equal(res.for, res.for2)

microbenchmark(
  f.for(my.mat),
  f.for2(my.mat))
```

- Uporabite funkcijo `apply` za isti namen.

```
f.apply <- function(my.mat){
  apply(my.mat, 2, mean)
}

microbenchmark(
  f.for(my.mat),
  f.for2(my.mat),
  f.apply(my.mat))
```

- Uporabite funkcijo `apply` in `mean.default` za isti namen.

```
f.apply.md <- function(my.mat){
  apply(my.mat, 2, mean.default)
}

microbenchmark(
  f.for(my.mat),
  f.for2(my.mat),
  f.apply(my.mat),
```

```
  f.apply.md(my.mat))
```

- Uporabite funkcijo `apply` in rocno izracunajte povrecje (vsota elementov/velikost vzorca) za isti namen.

```
f.apply.manual <- function(my.mat){
  apply(my.mat, 2, function(x) sum(x)/length(x))
}

microbenchmark(
  f.for(my.mat),
  f.for2(my.mat),
  f.apply(my.mat),
  f.apply.md(my.mat),
  f.apply.manual(my.mat))
```

- Uporabite funkcijo `lapply` in `mean.default` za isti namen.

```
f.lapply.md <- function(my.mat){
  lapply(1:ncol(my.mat), function(i) mean.default(my.mat[,i]))
}

f.lapply.dataframe <- function(my.mat)
  lapply(as.data.frame(my.mat), mean.default)

microbenchmark(
  f.for(my.mat),
  f.for2(my.mat),
  f.apply(my.mat),
  f.apply.md(my.mat),
  f.apply.manual(my.mat),
  f.lapply.md(my.mat),
  f.lapply.dataframe(my.mat)
  )
```

- Uporabite funkcijo `colMeans` za isti namen.

```
f.colMeans <- function(my.mat){
  colMeans(my.mat)
}

microbenchmark(
  f.for(my.mat),
  f.for2(my.mat),
  f.apply(my.mat),
  f.apply.md(my.mat),
  f.apply.manual(my.mat),
  f.colMeans(my.mat),
  colMeans(my.mat))
```

## Using different data structures

Oceni korelacijo med dvema stolpcema v data frame s pomocjo bootstrapa.

```
boot_cor1 <- function(df, i) {
  sub <- df[sample(nrow(df), i, replace = TRUE),]
```

```
  cor(sub$x, sub$y)
}

boot_cor2 <- function(df, i ) {
  idx <- sample(nrow(df), i, replace = TRUE)
  cor(df$x[idx], df$y[idx])
}

df <- data.frame(x = runif(100), y = runif(100))

microbenchmark(
  boot_cor1(df, 100),
  boot_cor2(df, 100))
```

### Koliko casa prihranimo pri uporabi funkcije na manjsem vektorju?

```
lookup <- setNames(as.list(sample(100, 26)), letters)
x1 <- "j"
x10 <- sample(letters, 10)
x100 <- sample(letters, 100, replace = TRUE)

microbenchmark(
  lookup[x1],
  lookup[x10],
  lookup[x100]
)

with.for <- function(x){
  for (i in 1:length(x)) {
    lookup[x[i]]
  }
}

microbenchmark(
  lookup[x1],
  lookup[x10],
  lookup[x100],
  with.for(x1),
  with.for(x10),
  with.for(x100)
)
```

## Nasveti za optimizacijo kode

1. Najdi najbolj pocasen del kode (bottleneck) – **profiling**
2. Optimiziraj ta del kode:

- Poisci ze obstojece resitve: Google, stackoverflow.
- Uporabi funkcijo, ki naredi cim manj: izracuna le to kar rabis, je prilagojena tvojemu tipu podatkov
- Vektoriziraj: izogibaj se for zankam, uporabi funkcije, ki racunajo v C (npr. `colmeans`)
- Izogibaj se nepotrebnim kopijam: rezerviraj si prostor za rezultat in ne uporabljaj `c(), cbind(), rbind()`
- Byte-code compile: `compiler::cmpfun` (ponavadi manjse izboljsave)
- (C++)

3. Ponavljaj dokler koda ni dovolj hitra **za tvoje potrebe**...

cas, ki ga bos porabil za evalvacijo kode za svoj namen **VS** cas, ki ga porabis za optimizacijo kode

4. Paraleliziraj

## Profiling

Ko napisete daljse funkcije, lahko preverite, katere funkcije so najpocasnejse oziroma potrebujejo najvec casa za izvedbo (bottlenecks) s profilingom. R Studio vsebuje orodja za profiling.

V spodnjem primeru (https://support.rstudio.com/hc/en-us/articles/218221837-Profiling-with-RStudio) si lahko ogledate, kako uporabiti Profiling. Uporabili bomo funkcijo `profvis` iz knjiznice `profvis`.

Namen naloge: standardizirati stolpce ogromne matrike (glede na povprecje), pri tem upostevati, da eden od stolpec ne vsebuje stevilk (id) - zato ga ne bomo standardizirali.

V praksi: izracunati povrecno vrednost za vsak stolpec matrike data - razen id - (ki ima 400000 vrstic! in 151 stolpcev) in ga odsteti od vrednosti po stolpcih.

Uporabite spodnjo kodo: lahko si ogledate interaktivno, katere so funkcije, ki potrebujejo najvec casa in najvec spomina.

```
library(profvis)

times <- 4e5
cols <- 150
data <- as.data.frame(x = matrix(rnorm(times * cols, mean = 5), ncol = cols))
data <- cbind(id = paste0("g", seq_len(times)), data)

profvis({
 # Store in another variable for this run
 data1 <- data

 # Get column means
 means <- apply(data1[, names(data1) != "id"], 2, mean)

 # Subtract mean from each column
 for (i in seq_along(means)) {
   data1[, names(data1) != "id"][, i] <-
     data1[, names(data1) != "id"][, i] - means[i]
 }
})
```

- Katera funkcija uporabi najvec casa?

- Oglejte si spodnje okno v profilerju. Katere dodatne funkcije dejansko uporabi funkcija `apply`? Zakaj?
- Kako bi lahko pohitrili funkcijo? (Namig: za izracun povprecij lahko uporabite `colMeans`,`lapply` ali `vapply`, kot v prejsnjih primerih!). Preverite ucinkovitost sprememb s funkcijo `profvis`.

```
profvis({
  data1 <- data
  # Four different ways of getting column means
  means <- apply(data1[, names(data1) != "id"], 2, mean)
  means <- colMeans(data1[, names(data1) != "id"])
  means <- lapply(data1[, names(data1) != "id"], mean)
  means <- vapply(data1[, names(data1) != "id"], mean, numeric(1))
})
```

- Izberite funkcijo, ki je bila najhitrejsa! Se enkrat preverite splosno hitrost funkcije, ki normalizira stolpce glede povprecja.

```
profvis({
 # Store in another variable for this run
 data1 <- data

 # Get column means
 means <- vapply(data1[, names(data1) != "id"], mean, numeric(1))

 # Subtract mean from each column
 for (i in seq_along(means)) {
   data1[, names(data1) != "id"][, i] <-
     data1[, names(data1) != "id"][, i] - means[i]
 }
})
```

- Naredite se koncno spremembo kode, ki pohitri tudi drugi del kode oziroma si oglejte spodnji namig, ki poda resitev (in interpretirajte kodo!)

Namig: spodnja koda pohitri funkcijo za 8x!

```
profvis({
 data1 <- data

 # Given a column, normalize values and return them
 col_norm <- function(col) {
   col - mean(col)
 }

 # Apply the normalizer function over all columns except id
 data1[, names(data1) != "id"] <-
   lapply(data1[, names(data1) != "id"], col_norm)
})
```

## Paralelizacija - hitri uvod

Ce moramo izvesti neodvisne funkcije velikokrat, lahko paraleliziramo nase delo v R-u. Ogledali si bomo zelo preprost primer, ki uporabi funkcijo `lapply` in jo paralelizira. Opomba: ukazi in postopki so razlicni glede na operativni sistem racunalnika.

1. Primer: zelimo generirati 10.000 vzorcev iz standardne normalne porazdelitve (velikost 1000) in si za vsakega shraniti povprecno vrednost.

```r
f <- function(i, n=1000){
  mean(rnorm(n))
}

set.seed(1234)
my.means <- lapply(1:10000, f)
```

```r
library(parallel)
cores <- detectCores()
cores
```

```
## [1] 8
```

Unix/Mac

```r
set.seed(1234)
my.means.par<-mclapply(1:10000, f,  mc.cores = cores)
```

Windows

```r
cluster <- makePSOCKcluster(cores)

set.seed(1234)
my.means.par <- parLapply(cluster, 1:10000, f)
```

- Primerjate cas izvedbe z `lapply` in paralelno funkcijo (uporabite `system.time()`).

```r
system.time(lapply(1:10000, f))
system.time(parLapply(cluster, 1:10000, f))
```

2. Generirajte veliko matriko (10.000 stolpcev) in za vsak stolpec izracunajte standardni odklon. Uporabite paralelizacijo!

```r
times <- 1000
cols <- 10000
data <- as.data.frame(x = matrix(rnorm(times * cols, mean = 5), ncol = cols))

res = apply(data, 2, sd)
res.par = parApply(cluster, data, 2, sd)

all.equal(res,res.par)

system.time(apply(data, 2, sd))
system.time(parApply(cluster, data, 2, sd))
```