Debugowanie i profilowanie kodu

Przemysław Biecek

6 XII 2015

Dziwne rzeczy się zdarzają

Jak to wyjaśnić?

```
(sekwencja <- seq(0.7, 0.9, by = 0.1))
```

```
## [1] 0.7 0.8 0.9
```

```
(indeksy <- sekwencja * 10 - 6)
```

```
## [1] 1 2 3
```

```
LETTERS[1:3]
```

```
## [1] "A" "B" "C"
```

```
# jakie litery się wyświetlą?
LETTERS[indeksy]
```

```
## [1] "A" "A" "C"
```

Debugowanie

Wykrywanie błędów i optymalizacja kodu jest przydatna, gdy pisze się programy działające w jednym procesie na jednej maszynie.

Gdy jednak zaczynamy pisać programy rozproszone, wykorzystujące kilka procesów, potencjalnie na różnych maszynach, bardzo często coś „idzie źle".

Znajomość podstawowych technik debugowania pozwala na identyfikacje i naprawienie problemu.

dump.frames()

Użyteczną funkcją w takiej pracy jest `dump.frames()`, zapisująca wszystkie otwarte ramki do pliku.

```
options(error = quote(dump.frames("errorDump",
    TRUE)))
options(error = NULL)
```

Przykład wywołania tej funkcji.

```
funkcja <- function(x) {
    log(x)
}
funkcja("napis")
## log: Using log base e.  Error in log(x) :
## Non-numeric argument to mathematical
## function Execution halted
(load("errorDump.rda"))
debugger(errorDump)
```

recover()

Jeżeli pracujemy w trybie interaktywnym i każdy błąd lubimy przeanalizować, to wygodne będzie ustawienie funkcji `recover()` jako funkcji do wywołania po wystąpieniu błędu.

```
options(error = recover)
```

traceback()

Jeżeli używamy programu RStudio, to po wystąpieniu błędu możemy zażądać wyświetlenia stosu wywołań funkcji w chwili napotkania błedu.

Jeżeli pracujemy w innym środowisku, to ten sam efekt możemy uzyskać stoując funkcję `traceback()`.

```
traceback()
```

debug() / undebug()

Jeżeli funkcja nie generuje błędu, ale zachowuje się inaczej niż byśmy chcieli, to do prześledzenia jej wykonania krok po kroku można wykorzystać funkcję `debug()`.

Debugowanie wyłącza się funkcją `undebug()`.

```
funkcja2 <- function(x, y) {
    funkcja(x)
    funkcja(y)
}
debug(funkcja2)
# wywołanie tej funkcji będzie realizowane
# linia po linii
funkcja2(1, "jeden")
```

try() / tryCatch()

Jeżeli uruchamiamy jakieś obliczenia na wielu rdzeniach i spodziewamy się, że gdzieś może pojawić się błąd ale nie chcemy by przerywał on całość obliczeń (błąd niezależny od nas, zależne od nas byśmy obsłużyli), to dobrym rozwiązaniem jest przechwycenie błędu.

Można do tego wykorzystać funkcja `try()` lub `tryCatch()`. Przykładowo, poniżej mamy bezpieczne wywołanie `funkcja2`.

```
try(funkcja2(1, "jeden"), silent = TRUE)
```

Profiler

Dużo ciekawych informacji pomocnych w debugowaniu i profilowaniu kodu przedstawia funkcja `Rprof()`.

Podstawowe statystyki wykonania przedstawia funkcja `summaryRprof()`.

```
generuj <- function() {
    runif(10^6)
    rexp(10^6)
    rnorm(10^6)
    1
}
wypisuj <- function() {
    replicate(10^5, rnorm(1))
}

Rprof("profiler.out", interval = 0.01, memory.profiling = TRUE)
for (i in 1:10) {
    generuj()
    wypisuj()
}
Rprof()

summaryRprof("profiler.out", memory = "both")
```

```
$by.self
          self.time self.pct total.time total.pct mem.total
"rnorm"        2.66    63.94       2.66     63.94    2626.7
"lapply"       0.52    12.50       2.99     71.88    3154.4
"FUN"          0.36     8.65       2.46     59.13    2751.5
"rexp"         0.33     7.93       0.33      7.93      76.3
"runif"        0.25     6.01       0.25      6.01     139.1

$by.total
                total.time total.pct mem.total self.time self.pct
```

```
"replicate"            3.02     72.60    3213.1      0.00     0.00
"sapply"               3.02     72.60    3213.1      0.00     0.00
"wypisuj"              3.02     72.60    3213.1      0.00     0.00
"lapply"               2.99     71.88    3154.4      0.52    12.50
"rnorm"                2.66     63.94    2626.7      2.66    63.94
"FUN"                  2.46     59.13    2751.5      0.36     8.65
"generuj"              1.14     27.40     291.8      0.00     0.00
"rexp"                 0.33      7.93      76.3      0.33     7.93
"runif"                0.25      6.01     139.1      0.25     6.01

$sample.interval
[1] 0.01

$sampling.time
[1] 4.16
```

Graficzne statystyki z profr

Wynik funkcji `summaryRprof` jest interesujący do analizy, ale często więcej możemy odczytać korzystając z graficznej prezentacji logów z profilowania.

```
library(profr)
library(PogromcyDanych)

Rprof("profiler2.out", interval = 0.01, memory.profiling = FALSE)
model <- lm(Cena.w.PLN ~ factor(Model), data = auta2012)
Rprof()

out <- parse_rprof("profiler2.out")
plot(out)
```

Mierzenie czasu

Czas wykonania operacji można mierzyć na kilka sposobów. Gdy są to długie operacje, liczone w sekundach, można wykorzystać `system.time()`

```
system.time({
    x = NULL
    for (i in 1:10^4) x = c(x, runif(1))
})
```

```
##    user  system elapsed
##   0.223   0.005   0.229
```

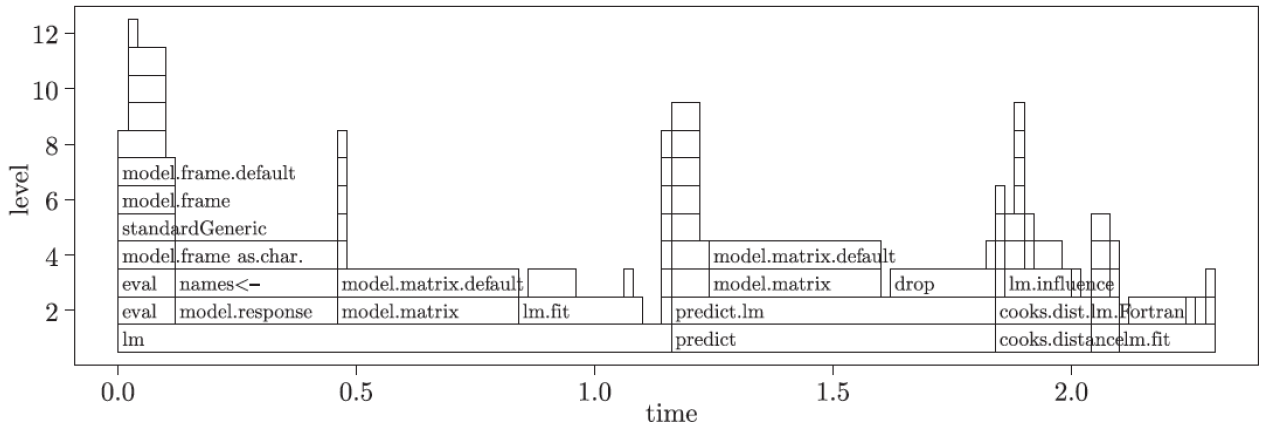


Figure 1: Graficzna prezentacja postaci stosu wywołań funkcji podczas wywołania instruckji lm.

```
system.time({
    x = numeric(10^4)
    for (i in 1:10^4) x[i] = runif(1)
})
```

```
##    user  system elapsed
##   0.025   0.000   0.026
```

```
system.time({
    x = NULL
    x = runif(10^4)
})
```

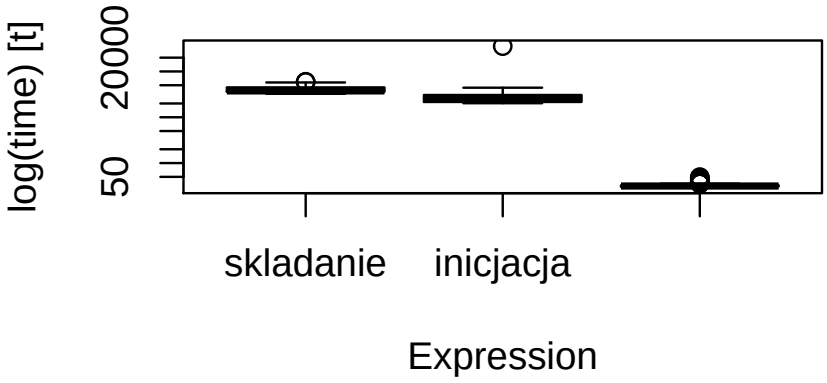
```
##    user  system elapsed
##       0       0       0
```

Dla krótszych operacji lepiej wykorzystać bibliotekę microbenchmark

```
library(microbenchmark)
res <- microbenchmark(skladanie = {
    x = NULL
    for (i in 1:10^3) x = c(x, runif(1))
}, inicjacja = {
    x = numeric(10^3)
    for (i in 1:10^3) x[i] = runif(1)
}, wektoryzacja = {
    x = NULL
    x = runif(10^3)
})
res
```

```
## Unit: microseconds
##          expr      min        lq       mean
##     skladanie 3253.380 3614.7955 4021.65037
##     inicjacja 2025.518 2155.3785 3008.28820
##  wektoryzacja   29.236   30.8465   32.59267
##     median       uq       max neval cld
##  3761.3950 4489.601  5961.611   100   c
##  2665.7835 3063.243 35663.892   100  b
##    31.5325   32.711    49.419   100 a
```

```
boxplot(res)
```



Przyśpieszanie obliczeń

W R, podobnie jak i w wielu innch językach dynamicznie interpretowanych, można znacznie przyśpieszyć fragmenty kodu starając się je wektoryzować.

```
N <- 200
system.time({
    wek <- c()
    z <- 0
    for (i in 1:N) for (j in 1:N) wek[z <- z +
        1] <- (i * j)%%10
    table(wek)
})
```

```
##    user  system elapsed
##   1.931   0.434   2.371
```

```
# równoważnie ale szybciej
system.time(tabulate(outer(1:N, 1:N, "*")%%10))
```

```
##    user  system elapsed
##   0.001   0.000   0.000
```

Materiały

Kilka bardzo ciekawych zadań do rozwiązania można znaleźć na stronach www.nimbios.org [1], [2].

[1] http://www.nimbios.org/ifiles/hpc/1_basics.pdf

[2] http://www.nimbios.org/ifiles/hpc/2_performance.pdf

Pakiet dplyr

Do typowych operacji na danych, można znacznie ułatwić operacje przetwarzania danych jak i znacznie przyśpieszyć operacje na danych poprzez skorzystnie z pakietu `dplyr`.

Aby ułatwić pracę na danych, `dplyr` tworzy abstrakcję źródła danych (`tbl_df`) i na niej wykonuje operacje. Sam pakiet tworzy przyjemną w użyciu warstwę anstrakcji, można go stosować na źródle dnych w postacie bazy danych czy pliku tekstowego.

```
library(PogromcyDanych)
library(dplyr)
(auta2012 <- tbl_df(auta2012[, c("Marka", "Model",
    "Wersja", "Przebieg.w.km", "Rok.produkcji",
    "Cena.w.PLN", "Brutto.netto", "KM", "Wyposazenie.dodatkowe",
    "Rodzaj.paliwa")]))
```

```
## Source: local data frame [207,602 x 10]
##
##            Marka     Model Wersja
##           (fctr)    (fctr) (fctr)
## 1             Kia    Carens
## 2      Mitsubishi Outlander
## 3       Chevrolet   Captiva
## 4           Volvo       S80
## 5  Mercedes-Benz  Sprinter
## 6  Mercedes-Benz     Viano
## 7         Renault    Trafic
## 8            Ford     Focus    Mk2
## 9            Ford    Fusion
## 10     Volkswagen  Multivan
## ..            ...       ...    ...
## Variables not shown: Przebieg.w.km (dbl),
##   Rok.produkcji (dbl), Cena.w.PLN (dbl),
##   Brutto.netto (fctr), KM (dbl),
##   Wyposazenie.dodatkowe (fctr),
##   Rodzaj.paliwa (fctr)
```

Filtry

Funkcja `filter()` pozwala na wybór tylko wierszy spełniających określony warunek

```
tmp <- filter(auta2012, Marka == "Porsche")
head(tmp, 3)
```

```
## Source: local data frame [3 x 10]
##
##     Marka   Model Wersja Przebieg.w.km
##    (fctr)  (fctr) (fctr)         (dbl)
## 1 Porsche     911                32350
## 2 Porsche     911                63000
## 3 Porsche Boxster                52849
## Variables not shown: Rok.produkcji (dbl),
##   Cena.w.PLN (dbl), Brutto.netto (fctr), KM
##   (dbl), Wyposazenie.dodatkowe (fctr),
##   Rodzaj.paliwa (fctr)
```

Możemy określać jednocześnie więcej warunków.

```
tylkoPorscheZDuzymSilnikiem <- filter(auta2012,
    Marka == "Porsche", KM > 300)
head(tylkoPorscheZDuzymSilnikiem, 3)
```

```
## Source: local data frame [3 x 10]
##
##     Marka         Model Wersja Przebieg.w.km
##    (fctr)        (fctr) (fctr)         (dbl)
## 1 Porsche           911                32350
## 2 Porsche           911                63000
## 3 Porsche Cayenne Turbo                57000
## Variables not shown: Rok.produkcji (dbl),
##   Cena.w.PLN (dbl), Brutto.netto (fctr), KM
##   (dbl), Wyposazenie.dodatkowe (fctr),
##   Rodzaj.paliwa (fctr)
```

Tworzenie nowych kolumn

Funkcja `mutate()` pozwala na stworzenie nowej zmiennej (jednej bądź wielu)

```
autaZWiekiem <- mutate(auta2012, Wiek.auta = 2013 -
    Rok.produkcji)
autaZCenaBrutto <- mutate(auta2012, Cena.brutto = Cena.w.PLN *
    ifelse(Brutto.netto == "brutto", 1, 1.23))
```

Sprawdźmy czy auto ma klimatyzacje Aby sprawdzić czy w kolumnie `Wyposazenie.dodatkowe` występuje określony element użyjemy funkcji `grepl()`

```
autaZWyposazeniem <- mutate(auta2012, Autoalarm = grepl(pattern = "autoalarm",
    Wyposazenie.dodatkowe), Centralny.zamek = grepl(pattern = "centralny zamek",
    Wyposazenie.dodatkowe), Klimatyzacja = grepl(pattern = "klimatyzacja",
    Wyposazenie.dodatkowe))
```

Wybór zmiennych

Funkcja `select()` pozwala na wybór jednej lub wielu zmiennych z ramki danych

```
dplyr:::select(autaZWiekiem, Wiek.auta, Rok.produkcji) %>%
    head(3)
```

```
## Source: local data frame [3 x 2]
##
##   Wiek.auta Rok.produkcji
##       (dbl)         (dbl)
## 1         5          2008
## 2         5          2008
## 3         4          2009
```

Sortowanie

Funkcją `arrange()` możęmy wykonać sortowanie po jednej lub większej liczbie zmiennych.

```
tylkoPorscheZDuzymSilnikiem <- filter(auta2012,
    Marka == "Porsche", KM > 300)
posortowanePorsche <- arrange(tylkoPorscheZDuzymSilnikiem,
    Cena.w.PLN)
head(posortowanePorsche, 3)
```

```
## Source: local data frame [3 x 10]
##
##     Marka   Model Wersja Przebieg.w.km
##    (fctr)  (fctr) (fctr)         (dbl)
## 1 Porsche Cayenne                   NA
## 2 Porsche Cayenne                   NA
## 3 Porsche     911                   NA
## Variables not shown: Rok.produkcji (dbl),
##   Cena.w.PLN (dbl), Brutto.netto (fctr), KM
##   (dbl), Wyposazenie.dodatkowe (fctr),
##   Rodzaj.paliwa (fctr)
```

Potoki

Rozważmy taki ciąg instrukcji

```
# tylko volkswagen
tylkoVolkswagen <- filter(auta2012, Marka == "Volkswagen")
# posortowane
posortowaneVolkswagen <- arrange(tylkoVolkswagen,
    Cena.w.PLN)
# tylko Golf VI
tylkoGolfIV <- filter(posortowaneVolkswagen, Model ==
    "Golf", Wersja == "IV")
# tylko z małym przebiegiem
tylkoMalyPrzebieg <- filter(tylkoGolfIV, Przebieg.w.km <
    50000)
```

Powyższe instrukcje można zamienić na jedno wywołanie, tak zwane przetwarzanie „na wielką cebulkę".

```
tylkoMalyPrzebieg <- filter(filter(arrange(filter(auta2012,
    Marka == "Volkswagen"), Cena.w.PLN), Model ==
    "Golf", Wersja == "IV"), Przebieg.w.km < 50000)
```

Mało czytelny, choć często spotykany zapis.

Rozwiązaniem problemu cebulki jest stosowanie specjalnego operatora do przetwarzania potokowego `%>%`. Ten operator pochodzi z pakietu `magrittr` i jest dostępny po włączeniu pakietu `dplyr`.

Cytując z dokumentacji: *to be pronounced with a sophisticated french accent*

Jak działa ten operator?

Przekazuje lewą stronę operatora jako pierwszy argument prawej strony tego operatora.

Instrukcja `a %>% f(b)` jest równoważna instrukcji `f(a, b)`.

Powyższa cebulka jest równoważna z poniższym potokiem.

```
auta2012 %>%                                    # weź dane o autach
  filter(Marka == "Volkswagen") %>%               # pozostaw tylko Volkswageny
  arrange(Cena.w.PLN) %>%                         # posortuj malejąco po cenie
  filter(Model == "Golf", Wersja == "IV") %>%     # pozostał tylko Golfy VI
  filter(Przebieg.w.km < 50000) %>%               # pozostał tylko auta o małym przebiegu
  head(3)

## Source: local data frame [3 x 10]
##
##        Marka  Model Wersja Przebieg.w.km
##       (fctr) (fctr) (fctr)         (dbl)
## 1 Volkswagen   Golf     IV           170
## 2 Volkswagen   Golf     IV           198
```

```
## 3 Volkswagen   Golf     IV           255
## Variables not shown: Rok.produkcji (dbl),
##   Cena.w.PLN (dbl), Brutto.netto (fctr), KM
##   (dbl), Wyposazenie.dodatkowe (fctr),
##   Rodzaj.paliwa (fctr)
```

Podsumowania / statystyki / agregaty grup

Funkcją `summarise()` można wyznaczyć agregaty w danych

```
auta2012 %>% summarise(sredniaCena = mean(Cena.w.PLN),
    sdCena = sqrt(var(Cena.w.PLN)), medianaPrzebiegu = median(Przebieg.w.km,
        na.rm = TRUE))
```

```
## Source: local data frame [1 x 3]
##
##   sredniaCena   sdCena medianaPrzebiegu
##         (dbl)    (dbl)            (dbl)
## 1    35755.11 70399.67           140000
```

Tworząc agregaty wygodnie jest korzystać z funkcji `n()`, której wynikiem jest liczba wierszy w zbiorze danych / grupie.

```
auta2012 %>% summarise(liczba.aut.z.klimatyzacja = sum(grepl("klimatyzacja",
    Wyposazenie.dodatkowe)), procent.aut.z.klimatyzacja = 100 *
    mean(grepl("klimatyzacja", Wyposazenie.dodatkowe)),
    liczba.aut = n())
```

```
## Source: local data frame [1 x 3]
##
##   liczba.aut.z.klimatyzacja
##                       (int)
## 1                    162960
## Variables not shown:
##   procent.aut.z.klimatyzacja (dbl),
##   liczba.aut (int)
```

Grupowanie

Funkcja `group_by()` pozwala na operacje na agregatach w grupach opisanych przez zmienną jakościową.

```
auta2012 %>% filter(Marka == "Volkswagen", Rok.produkcji ==
    2007) %>% group_by(Rodzaj.paliwa) %>% summarise(medianaCeny = median(Cena.w.PLN,
    na.rm = TRUE), liczba = n())
```

```
## Source: local data frame [3 x 3]
##
##            Rodzaj.paliwa medianaCeny liczba
##                   (fctr)       (dbl)  (int)
## 1               benzyna     33550.0    190
## 2           benzyna+LPG     34048.9      7
## 3 olej napedowy (diesel)     38900.0   1482
```

Agregaty są zwykłą ramką danych, można wykonywać na nich kolejne operacje, np sortowanie.

```
auta2012 %>% filter(Marka == "Volkswagen", Rok.produkcji ==
    2007) %>% group_by(Rodzaj.paliwa) %>% summarise(medianaCeny = median(Cena.w.PLN,
    na.rm = TRUE), liczba = n()) %>% arrange(liczba)
```

```
## Source: local data frame [3 x 3]
##
##            Rodzaj.paliwa medianaCeny liczba
##                   (fctr)       (dbl)  (int)
## 1           benzyna+LPG     34048.9      7
## 2               benzyna     33550.0    190
## 3 olej napedowy (diesel)     38900.0   1482
```

Grupowanie po dwóch zmiennych

Grupować można po kilku zmiennych, w tym przypadku agregaty liczone są w każdym podzbiorze zmiennych.

```
auta2012 %>% filter(Rok.produkcji == 2007, Marka ==
    "Volkswagen") %>% group_by(Model, Rodzaj.paliwa) %>%
    summarise(medianaCeny = median(Cena.w.PLN,
        na.rm = TRUE), medianaPrzebieg = median(Przebieg.w.km,
        na.rm = TRUE), liczba = n()) %>% head(3)
```

```
## Source: local data frame [3 x 5]
## Groups: Model [2]
##
##    Model          Rodzaj.paliwa medianaCeny
##   (fctr)                 (fctr)       (dbl)
## 1 Beetle               benzyna       39000
## 2  Caddy               benzyna       27900
## 3  Caddy olej napedowy (diesel)       30813
## Variables not shown: medianaPrzebieg (dbl),
##   liczba (int)
```

Challenge

Poniższe operacje wykonaj na bazie zbioru danych `auta2012`

1. Która Marka występuje najczęściej w zbiorze danych `auta2012`?
2. Spośród aut marki Toyota, który model występuje najczęściej.
3. Sprawdź ile jest aut z silnikiem diesla wyprodukowanych w 2007 roku?
4. Jakiego koloru auta mają najmniejszy medianowy przebieg?
5. Gdy ograniczyć się tylko do aut wyprodukowanych w 2007, która Marka występuje najczęściej w zbiorze danych `auta2012`?
6. Spośród aut marki Toyota, który model najbardziej stracił na cenie pomiędzy rokiem produkcji 2007 a 2008.
7. Spośród aut z silnikiem diesla wyprodukowanych w 2007 roku która marka jest najdroższa?
8. Ile jest aut z klimatyzacją?
9. Gdy ograniczyć się tylko do aut z silnikiem ponad 100 KM, która Marka występuje najczęściej w zbiorze danych `auta2012`?
10. Spośród aut marki Toyota, który model ma największą różnicę cen gdy porównać silniki benzynowe a diesel?
11. Spośród aut z silnikiem diesla wyprodukowanych w 2007 roku która marka jest najtańsza?
12. W jakiej marce klimatyzacja jest najczęściej obecna?
13. Gdy ograniczyć się tylko do aut o cenie ponad 50 000 PLN, która Marka występuje najczęściej w zbiorze danych `auta2012`?
14. Spośród aut marki Toyota, który model ma największy medianowy przebieg?
15. Spośród aut z silnikiem diesla wyprodukowanych w 2007 roku który model jest najdroższy?
16. W jakim modelu klimatyzacja jest najczęściej obecna?
17. Gdy ograniczyć się tylko do aut o przebiegu poniżej 50 000 km o silniku diesla, która Marka występuje najczęściej w zbiorze danych `auta2012`?
18. Spośród aut marki Toyota wyprodukowanych w 2007 roku, który model jest średnio najdroższy?
19. Spośród aut z silnikiem diesla wyprodukowanych w 2007 roku który model jest najtańszy?
20. Jakiego koloru auta mają największy medianowy przebieg?