732A94 Advanced R Programming
Computer lab 6

Krzysztof Bartoszek, Bayu Brahmantio.
(designed by Leif Jonsson, Måns Magnusson and Shashi Nagarajan)
(updated and converted to LaTeX by Arash Haratian)

29 September, 1 October 2025 (U2, U2)

Seminar date: **10 October 10:15** (U2)
Lab deadline: **13 October 23:59**

Contents

Chapter 1

Writing fast R code

In this lab we will will create a package to study the effects of algorithms with different computional complexity and how to speedup R code.

Master students should implement one of the exercises marked with (*).

1.1 The knapsack package

Start out by creating your new package on GitHub with GitHub Actions (or Travis) or on GitLab with GitLab CI. You can choose the option you prefer. See lab 3 for details on how to setup a package.

The package will contain three different functions for solving what is called the knapsack problem. The knapsack problem is a discrete optimization problem where we have a knapsack that can take a limited weight W and we want to fill this knapsack with a number of items $i = 1, ..., n$, each with a weight w_i and a value v_i. The goal is to find the knapsack with the largest value of the elements added to the knapsack.This problem is NP-hard, meaning that it is "at least as hard as the hardest problem in NP" (en.wikipedia.org/wiki/NP-hardness). NP is a (fundamental) class of problems for which there are (currently) no polynomial time algorithms to solve them. It is an open (Millennium Prize) problem, whether it is or is not possible to solve these problemsin polynomial time.

For a more detailed background of the knapsack problem see **this page** (`en.wikipedia.org/wiki/Knapsack_problem`).

The data we will use is generated in the following way. To create larger datasets, just set `n` to a larger number. Be careful with the version of the random number generator!

```
RNGversion(min(as.character(getRversion()),"3.5.3"))

Warning in RNGkind("Mersenne-Twister", "Inversion", "Rounding"): non-uniform 'Rounding'
sampler used

##old sampler used for backward compatibility
## suppressWarnings() can be used so that the above warning is not displayed
set.seed(42, kind = "Mersenne-Twister", normal.kind = "Inversion")
n <- 2000
knapsack_objects <- data.frame(w = sample(1:4000, size = n, replace = TRUE),
                               v = runif(n = n, 0, 10000))
```

1.1.1 The package vignette

The package vignette will be your lab report together with the functions in you package. The vignette should contain the answers to all the questions put below. This vignette should be included with the package and be viewed with `browseVignettes("lab_report_knapsack")`.

1.1.2 Brute force search

The only solution that is guaranteed to give a correct answer in all situations for the knapsack problem is using brute-force search, i.e. going through all possible alternatives and return the maximum value found. This approach is of complexity $O(2^n)$ since all possible combinations 2^n needs to be evaluated.

Implement a function you call `knapsack_brute_force(x, W)` that takes a `data.frame x` with two variables `v` and `w` and returns the maximum knapsack value and which elements (rows in the `data.frame`). The variable `W` is the knapsack size.

The function should check that the inputs are correct (i.e. a `data.frame` with two variables `v` and `w`) with only positive values.

The easiest way to enumerate all different combinations is using a binary representation of the numbers 1 to 2^n and include all elements of that is equal to 1 in the binary representation. A function that can do this for you in R is `intToBits()`. Below shows how the function should work (observe that only the first couple of objects are studied).

```
brute_force_knapsack(x = knapsack_objects[1:8,], W = 3500)

$value
[1] 16770

$elements
[1] 5 8

brute_force_knapsack(x = knapsack_objects[1:12,], W = 3500)

$value
[1] 16770

$elements
[1] 5 8

brute_force_knapsack(x = knapsack_objects[1:8,], W = 2000)

$value
[1] 15428

$elements
[1] 3 8

brute_force_knapsack(x = knapsack_objects[1:12,], W = 2000)

$value
[1] 15428

$elements
[1] 3 8
```

Question How much time does it takes to run the algorithm for $n = 16$ objects?

1.1.3 Dynamic programming

We will now take another approach to the problem. If the weights are actually discrete values (as in our example) we can use this to create an algorithm that can solve the knapsack problem exact by iterating over all possible values of `w`.

The pseudocode for this algorithm can be found **here** (`en.wikipedia.org/wiki/Knapsack_problem#Dynamic_programming_in-advance_algorithm`). Implement this function as `knapsack_dynamic(x, W)`. This function should return the same results as the brute force algorithm, but unlike the brute force it should scale much better since the algorithm will run in $O(Wn)$.

Question How much time does it takes to run the algorithm for $n = 500$ objects?

1.1.4 Greedy heuristic

A last approach is to use the a heuristic or approximation for the problem. This algorithm will not give an exact result (but it can be shown that it will return at least 50% of the true maximum value), but it will reduce the computational complexity considerably (actually to $O(n \log n)$ due to the sorting part of the algorithm). A short description on how to implement the greedy approach can be found **here** (`en.wikipedia.org/wiki/Knapsack_problem#Greedy_approximation_algorithm`). Below is an example on how the function should work.

```
greedy_knapsack(x = knapsack_objects[1:800,], W = 3500)

$value
[1] 192647

$elements
 [1]  92 574 472  80 110 537 332 117  37 776 577 288 234 255 500 794  55 290 436
[20] 346 282 764 599 303 345 300 243  43 747  35  77 229 719 564

greedy_knapsack(x = knapsack_objects[1:1200,], W = 2000)

$value
[1] 212337

$elements
 [1]   92  574  472   80  110  840  537 1000  332  117   37 1197 1152  947  904
[16]  776  577  288 1147 1131  234  255 1006  833 1176 1092  873  828 1059  500
[31] 1090  794 1033
```

Question How much time does it takes to run the algorithm for $n = 1000000$ objects?

1.1.5 Implement a test suite for your package

Add the testsuites for the `greedy_knapsack()` and `brute_force_knapsack()` that are found **here** (`https://github.com/STIMALiU/AdvRCourse/blob/master/Testsuites/`).

Based on these test suites, write your own test suite for `knapsack_dynamic(x, W)` by copying unit test from the other test suites. It is possible that your greedy algorithm will return a better solution, than in the test suites. In such a situation you have to motivate that your algorithm is still a greedy one and has $O(n)$ running time. Try to find the reason why a better solution is found.

1.1.6 Profile your code and optimize your code

Now profile and optimize your code to see if you can increase the speed in any way using any of the techniques described in the lectures and **here** (`https://adv-r.hadley.nz/perf-measure.html`). Use the package `profvis` (or the deprecated `lineprof`, if `profvis` does not help) to identify bottlenecks, see if you can write this code any faster.

Question What performance gain could you get by trying to improving your code?

1.1.7 (*) Implementation in `Rcpp`

Another way of improving your code would be to run some parts of the code using `Rcpp` and writing this part of the code using C++. More details on how to use `Rcpp` can be found **here** (`https://adv-r.hadley.nz/rcpp.html`).

Choose a function you want to improve and add the logical argument `fast`. The argument should be `FALSE` by default (so it works with the test suite where we have not specified the argument `fast`). Implement the fast version of the function using `Rcpp`.

Question What performance gain could you get by using `Rcpp` and C++?

1.1.8 (*) Parallelize brute force search

The brute force algorithm is straight forward to parallelize for computers with multiple cores. Implement an argument `parallel` in `brute_force_knapsack()` that is `FALSE` by default (so it works with the test suite where we have not specified the argument `parallel`). If set to `TRUE`, the function should parallelize over the detected cores.

Note! Your implementation could be platform-dependent, depending on whether you use MacOS/Linux or Windows.

Question What performance gain could you get by parallelizing brute force search?

1.1.9 Document your package using `roxygen2`

Document all your functions and your package using `roxygen2`.

1.2 (*) Profile and improve your existing API package

Use the package `profvis` to identify the bottlenecks in your code from last week. Try to improve this as much as you can, run your test suite continously to check that you do not introduce any new bugs.

1.3 Seminar and examination

During the seminar you will bring your own computer and demonstrate your package and what you found difficult in the project.

We will present as many packages as possible during the seminar and you should

1. Show that the package can be built using R Studio and that all unit tests are passing.
2. Show your vignette/run the examples live.
3. Present the speed of your different algorithms.

1.3.1 Examination

Turn in a the address to your GitHub or GitLab repo with the package using LISAM. To pass the lab you need to:

1. Have the R package up on GitHub with a GitHub Actions (or Travis CI) pass/fail badge, or similarly on GitLab with a GitLab CI pass/fail badge.
2. The test suites for the implemented function(s) should be included in the package.
3. The package should build without warnings (pass) on GitHub Actions (or Travis CI, GitLab CI).
4. All issues raised by GitHub Actions / Travis CI / GitLab CI should be taken care of or justified why they are not a problem or cannot be corrected. Be careful with namespace issues, these you HAVE to take care of.
5. Have the vignette included in your package which contains the demo of your functions and the answers to the questions.