Computer lab 6

Måns Magnusson

September 12, 2016

Instructions

- This lab should be conducted by students **two by two**.
- The lab constists of writing a package that is version controlled on github.com.
- Both student should **contribute equally much** to the package.
- Commit continously your addition and changes.
- Collaborations should be done using github (ie you should commit using your own github account).
- In the lab some functions can be marked with an *. These parts is only mandatory for students taking the advanced course or students working together in groups of three.
- The deadline for the lab can be found on the **webpage**
- The lab should be turned in as a url to the repository containing the package on github using **LISAM**. This should also include name, github user names and liuid of the students behind the project.

Contents

1 Writing fast R code **3**
1.1 The knapsack package . . . 3
1.1.1 The package vignette . . . 3
1.1.2 Brute force search . . . 3
1.1.3 Dynamic programming . . . 4
1.1.4 Greedy heuristic . . . 4
1.1.5 Implement test suite for your functions . . . 4
1.1.6 Profile your code and optimize your code . . . 4
1.1.7 (*) Implentation in `Rcpp` . . . 5
1.1.8 (*) Parallelize brute force search . . . 5
1.1.9 Document your package using `roxygen2` . . . 5
1.2 (*) Profile and improve your existing API package . . . 5
1.3 Seminar and examination . . . 5
1.3.1 Examination . . . 5

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 a new package on github, see lab 3 **here** 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. To solve this problem is NP which means that (most researchers belive) there are no polynomial time algorithm to solve this problem.

For a more detailed background of the knapsack problem see **this page**.

The data we will use is generated in the following way. To create larger datasets, just set `n` to a larger number.

```
set.seed(42)
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 `vignette(''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 cx` 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 is 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)

Error in eval(expr, envir, enclos):  could not find function "brute_force_knapsack"

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

Error in eval(expr, envir, enclos):  could not find function "brute_force_knapsack"

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

Error in eval(expr, envir, enclos):  could not find function "brute_force_knapsack"

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

Error in eval(expr, envir, enclos):  could not find function "brute_force_knapsack"
```

Question How long 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**. 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 long 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**. Below is an example on how the function should work.

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

Error in eval(expr, envir, enclos):  could not find function "greedy_knapsack"

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

Error in eval(expr, envir, enclos):  could not find function "greedy_knapsack"
```

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

1.1.5 Implement test suite for your functions

Implement a test suite for your functions so that you now know that is working correctly.

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**. Use the package `lineprof` to identify bottlenecks, see if you can write this code any faster.

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

1.1.7 (*) Implentation 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**.

In the function you choose to improve by adding 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 How large 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 will be platform dependent and only work with MacOS/Linux or Windows.

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

1.1.9 Document your package using roxygen2

Document all your function and package using `roxygen2`.

1.2 (*) Profile and improve your existing API package

Use the package lineprof 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 is 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 adress to your github repo with the package using LISAM.

The packages will be assigned to other groups to try your package out and return eventual problems as bugs using the issue tracker. The teacher will the decide if there are any bugs or corrections that is needed to correct to get the lab to pass.

Bibliography