Optimizing Code

Lasse Letager Hansen
Email: lasse@letager.dk

Aarhus University

September 12, 2020

What is this talk?

Looking at improving readability, speed and the feel of programming.

Overview

Should you optimize?

- Small investment, but huge benefit
- Learn when not to optimize (can be a time-sink)
- Takes time to learn
- More “thinking” than “putting in the effort”
- Extends to everything (not just coding)
- You can optimize: Time, complexitity, readability, memory usage, size, ...

Optimization process

Can something be done better?

- Is there an issue? (run-time, precision, memory, etc.)
- Find the bottleneck (analysis tools, debugging, code profiler, etc.)
- Find solutions (from memory, trial and error, stack overflow, etc.)
- Repeat if necessary

Example

Count uses of words - 0

```
1 d ← scan("100-0.txt", "character", sep="\n")
2
3 word_uses ← list ()
4
5 for (i in 1:length(d)) {
6     line ← gsub("\n", " ", d [[i]])
7     words ← strsplit(line," ")[[1]]
8
9     for (j in 1:length(words)) {
10        word ← tolower(words [j])
11
12        if (word == "") next
13        if (word %in% names(word_uses)) {
14            word_uses [[word]] ← word_uses [[word]] + 1
15        }
16        else {
17            word_uses[[word]] ← 1
18        }
19    }
20 }
```

Time: real 8m28s, user 8m27s

Improving code

First step is finding out where we can improve, so code cleanup goes a long way

- Make code easy to understand
- Indexes are hard to read, use iterators when possible
- Add comments if needed, less is more

Example

Count uses of words - 1

```
21 d <- scan("100-0.txt", "character", sep="\n")
22
23 word_uses <- list ()
24
25 # Split lines on space and newline, to get lists of words
26 lines <- gsub("\n", " ", d)
27 for (line in lines) {
28     words <- strsplit(line," ")[[1]]
29     words <- tolower (words)
30
31     ## Increment count for each word in line
32     for (word in words) {
33         if (word == "") next
34         if (word %in% names(word_uses)) {
35             word_uses [[word]] <- word_uses [[word]] + 1
36         }
37         else {
38             word_uses[[word]] <- 1
39         }
40     }
41 }
```

Time: real 8m10s, user 8m9s

Parallelization

If you process a lot of data in a non-dependent manner, you can parallelize it.

- Use all computation power, instead of parts of it.
- Often adds complexity, and a small overhead to get started
- Improves run-time, but not the run-time complexity

Example

Count uses of words - 2

```
42 library(parallel)
43 library(foreach)
44 library(doParallel)
45 library(iterators)
46 library(itertools)
47
48 d <- scan("100-0.txt", "character", sep="\n")
49
50 words <- tolower(unlist(strsplit(gsub("\n", " ", d)," ")))
51
52 numCores <- detectCores()
53 cl <- makeCluster(numCores)
54 registerDoParallel(cl)
55
56 word_uses <- foreach(words2=isplitVector(words,chunks=numCores*2),
57   .combine=function(a,b) {
58     b[names(a)] <- Map("+", ifelse(Map(is.null, b[names(a)]), 0, b
       [names(a)]), a)
59     b
60 }) %dopar% {
```

Example

Count uses of words - 2 (cont.)

```
61      word_uses ← list ()
62      for (word in words2) {
63          if (word == "") next
64
65          if (word %in% names(word_uses)) {
66              word_uses[[word]] ← word_uses[[word]] + 1
67          }
68          else {
69              word_uses[[word]] ← 1
70          }
71      }
72      word_uses
73 }
74
75 stopCluster(cl)
```

Time*: real 0m36,644s, user 0m4,169s

*Run on 8 cores using parallelization libraries

$\mathcal{O}$-notation / Time complexity

A way to analyze code run-time complexity

- Linear complexity $\mathcal{O}(n)$

```
76  for i in 1:n:
77    # do something
```

- Quadratic complexity $\mathcal{O}(n^2)$

```
78  for i in 1:n:
79    for j in 1:n:
80      # do something
```

- Logarithmic complexity $\mathcal{O}(\log n)$

```
81  while i < n:
82    # do something
83    i ← i * 2
```

- Constant complexity $\mathcal{O}(\log n)$

```
84  x ← 5
```

Data structures

- Thinking instead of brute forcing is the essence of optimization.
- Parallelization only effects the constant, however changing data structures and how we process data can effect the time complexitiy.
- Eg. lists in R take $\mathcal{O}(n)$ time for a lookup by name, while a dictionary (in python) / hashmap takes $\mathcal{O}(1)$ for named lookup[†]. Environments are implemented using hashmaps.
- Instead of getting the computer to use all its resources on inefficient computations, we can instead focus on improving the way we process the data.

[†] https://www.refsmmat.com/posts/2016-09-12-r-lists.html

Example

Count of words - 3

If we use an environment instead of a list, we get a runtime of $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$.

```
d ← scan("100-0.txt", "character", sep="\n")

words ← tolower(unlist(strsplit(gsub("\n", " ", d)," ")))

word_uses ← new.env()

## Increment count for each word in line
for (word in words) {
    if (word == "") next
    if (!is.null(word_uses[[word]])) {
        word_uses [[word]] ← word_uses [[word]] + 1
    }
    else {
        word_uses[[word]] ← 1
    }
}

word_uses ← as.list(word_uses)
```

Time: real 0m3s, user 0m2s

Do not reinvent the wheel

Use the internet

- Try searching for solutions to your problem, and look in the standard library solutions.
- Use StackOverflow, however be mindful to understand the solutions, since copy pasting bad code can make things worse, and make your code hard to understand.
- For example a table counts the number of occurrences of elements in our data directly, meaning the code could just be

```
103 d ← scan("100-0.txt", "character", sep="\n")
104
105 words ← tolower(unlist(strsplit(gsub("\n", " ", d)," ")))
106 word_uses ← as.list(table(words, exclude=""))
```

Time: real 0m2s, user 0m2s

Memory allocation in R

What is the problem with this for loop?

```
107 j ← 1
108 for (i in 1:100000000) {
109   j[i] ← 1
110 }
```

Time: real 28m50s, user 3m18s

It has to resize the vector repeatedly, which is slow, instead resize before the loop:

```
111 j ← rep(NA, 100000000)
112 for (i in 1:100000000) {
113   j[i] = 1
114 }
```

Time: real 0m4s, user 0m4s

Using apply does such optimizations for you, so it is faster, if replacing un-optimized code.

Memory deallocation in R

Similarly if you have big amounts of data that you do not use anymore, clean it up. If you have too much in ram, you will begin using slower types of memory, decreasing operation speed.

Vectorization in R

Instead of doing the same operation on each element in your data

```
115 j ← rep(NA, 400000000)
116 for (i in 1:400000000) {
117   j[i] ← exp(i)
118 }
```

Time: real 0m27s, user 0m26s

you can vectorize the operation, and apply it to everything at once

```
119 j ← rep(NA, 400000000)
120 j ← exp(1:400000000)
```

Time: real 0m10s, user 0m5s

Learning Classical examples

- Searching through a list (Linear search vs Binary search vs Dictionary)
- Sorting a list (Insertion sort, bubble sort, merge sort)
- Dynamic Programming (Memory vs Computation trade-off)

Sources for learning optimization

Good sources for information include:

- www.stackoverflow.com
- www.r-bloggers.com
- www.google.com

Conclusion / Main Points

- Optimization can give a huge payoff
- Doing too much gives a small payoff, find a balance
- Experience is a great help, so try reading up on some optimization
- Optimization is not confined to run-time, think about readability and environment you work in.

Questions?