Advanced R Programming - Lecture 6
Computational complexity

Krzysztof Bartoszek
(slides based on Leif Jonsson's and Måns Magnusson's)

Linköping University

krzysztof.bartoszek@liu.se

29 IX, 1 X 2020 (Zoom)

Today

Optimizing code

Performant Code

Computational complexity

Classes of problems

Big Oh notation

Determining complexity

Questions since last time?

Donald E. Knuth on Optimization

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.
- Donald E. Knuth

Performance

Depends on many things

1. Code
2. Complexity
3. Compiler
4. Hardware
5. Language

If you don't measure, you don't optimize!

How to optimize

0. Choose optimal algorithm
1. Write code that works with accompanying test suite
2. Profile your code for bottlenecks
3. Try to eliminate the bottle necks
4. Redo 2-3 until fast enough

`proc.time()` is a basic starting tool

Profiling (code below is deprecated, see RprofEx.R)

```
Rprof(tmp <- tempfile(),
  line.profiling = TRUE,
  memory.profiling = TRUE)
test_data <- pxweb::get_pxweb_data(
   url =
     "http://api.scb.se/OV0104/v1/doris/sv/ssd/BE/BE0101
                  /BE0101A/BefolkningNy",
   dims = list(Region = c('*'),
     Civilstand = c('*'),
     Alder = c('*'),
     Kon = c('*'),
     ContentsCode = c('*'),
     Tid = as.character(1970)),
   clean = TRUE)
Rprof()
summaryRprof(tmp, lines = "show", memory = "both")
```

Profiling

```
$by.self
                              self.time self.pct total.time total.pct mem.total
get_pxweb_data.R#102               1.96     39.2       1.96      39.2     579.2
get_pxweb_data_internal.R#42       1.16     23.2       1.16      23.2     405.0
get_pxweb_data.R#56                0.52     10.4       0.52      10.4      31.3
get_pxweb_data.R#80                0.38      7.6       0.38       7.6      29.1
get_pxweb_data.R#82                0.32      6.4       0.32       6.4      40.7
get_pxweb_data_internal.R#48       0.26      5.2       0.26       5.2      73.2
get_pxweb_data_internal.R#74       0.26      5.2       0.26       5.2      29.8
get_pxweb_data.R#83                0.08      1.6       0.08       1.6      17.2
api_catalogue.R#75                 0.02      0.4       0.02       0.4       0.0
get_pxweb_data_internal.R#44       0.02      0.4       0.02       0.4      12.6
get_pxweb_data_internal.R#71       0.02      0.4       0.02       0.4      16.0
```

Improvements

0. Optimal data structure and algorithm
1. Look for existing solutions
2. Do less work
3. Vectorise
0. Optimal data structure and algorithm
4. Parallelize
0. Optimal data structure and algorithm
5. Avoid copies

Writing fast code

Speed is important!
(do not forget memory)

Writing fast code

Speed is important!
(do not forget memory)

Time to write code

Writing fast code

Speed is important!
(do not forget memory)

Time to write code
Time to maintain (understand) code

Writing fast code

Speed is important!
(do not forget memory)

Time to write code
Time to maintain (understand) code
Time to execute code

Old Adage About Software

"You can have it Good, Fast, Cheap. Pick any two."

Performance

1. Performance
2. Complexity

Complexity affects performance

Computational complexity

Theoretical worst case
(but what about average case?)

Big-Oh notation

Basic operations

Relationship: operations to problem size

Types of complexity

Time complexity

Space (memory) complexity

Worst case complexity

Average case complexity

(theorerical) Data structures

Matrix (dataframe, list)

List (**NOT** in `R` sense, but with pointers), FIFO, LIFO

Sets (no particular order of elements, cannot index)

Graphs (vertex, edge): vertex adjacency matrix, vertex adjacency list

Classes of problems

Decision problems answer is yes or no, e.g. is x a prime number
Optimization problems find an object that satisfies a certain property, e.g. largest prime number smaller than $x+1$
Non–algorithmic problems cannot be solved by an algorithm, e.g. *halting problem* does a given algorithm end in finite time or fall into an infinite loop?
Presumably nonalgorithmic problems no algorithm is known but we do not know if non–algorithmic e.g. *Collatz problem*

```
repeat{
    if (k%%2==0){k=k/2}else{k=3*k+1}
    if (k==1){break}
}
```

Does it halt for every `k`?

Classes of problems

Non–polynomial problems cannot be solved by an algorithm whose running time is bounded by a polynomial of its input's size e.g. generate all permumations of an n element set, $n!$
Polynomial problems can be solved by an algorithm whose running time is bounded by a polynomial of its input's size e.g. sorting n elements
Non–polynomial problems cannot be solved by an algorithm whose running time is bounded by a polynomial

P class polynomial problems

NP

NP class *Nondeterministic polynomial* class of problems, there exists a polynomial time procedure that verifies if something is an admissable solution, e.g. check if *graph colouring* is admissable

$$\mathbf{P} \subset \mathbf{NP} \quad \text{but} \quad \mathbf{P} \overset{???}{=} \mathbf{NP}$$

NP–complete every problem in NP can be reduced to it in polynomial time
e.g. bin packing, knapsack, longest common subsequence, chromatic number of graph,
TSP ($\mathbb{N}$), multiprocessor scheduling (some)
satisfiability (SAT): is there a way to assign `TRUE`, `FALSE` values so that a logical statement is `TRUE`?
NP–hard: if it can be solved in polynomial time, then SAT$\in\mathbf{P}$

Big Oh

"How fast does a function grow?"

$$f(n) = O(g(n)) \quad \text{or} \quad f(n) \in O(g(n))$$

$$\exists_{C>0}\ \exists_{N_0\in\mathbb{N}}\ \forall_{\mathbb{N}\ni n>N_0}\ |f(n)| \leq C * |g(n)|$$

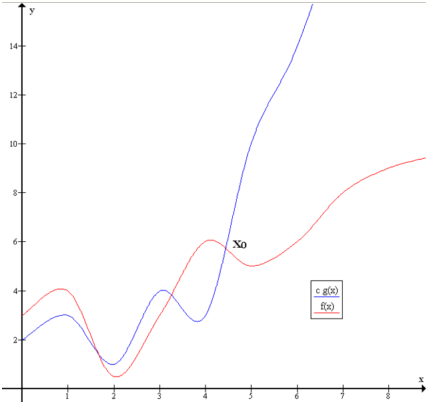
or

$$\limsup_{n\to\infty} \frac{|f(n)|}{|g(n)|} < \infty$$

n number of operations

$f(n)$ does not (up to a scaling constant) grow faster than $g(n)$

Big Oh



https://en.wikipedia.org/wiki/Big_O_notation

Big Oh

Example

$$f(n) = n^2 + 100n + 100$$

Big Oh

Example

$$f(n) = n^2 + 100n + 100$$

$$f(n) = O(n^2)$$

Other Oh

$$f = o(g) \quad \forall_{C>0} \exists_{N_0 \in \mathbb{N}} \forall_{\mathbb{N} \ni n > N_0} |f(n)| \le C|g(n)| \quad \lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = 0$$

$$f = O(g) \quad \exists_{C>0} \exists_{N_0 \in \mathbb{N}} \forall_{\mathbb{N} \ni n > N_0} |f(n)| \le C|g(n)| \quad \limsup_{n\to\infty} \frac{|f(n)|}{|g(n)|} < \infty$$

$$f = \omega(g) \quad \forall_{C>0} \exists_{N_0 \in \mathbb{N}} \forall_{\mathbb{N} \ni n > N_0} |f(n)| \ge C|g(n)| \quad \lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = \infty$$

$$f = \Omega(g) \quad \exists_{C>0} \exists_{N_0 \in \mathbb{N}} \forall_{\mathbb{N} \ni n > N_0} f(n) \ge Cg(n) \quad \liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f = \Theta(g) \quad f = O(g) \text{ and } f = \Omega(g)$$

$$f \sim g \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

Complexities (the data size is a lower bound)

Big Oh	**Name**	**Example, optimal**
$O(1)$	constant	assignments, $O(1)$
$O(\log N)$	logarithmic	binary search (sorted input), $O(\log N)$
$O(N)$	linear	max., $O(N)$
$O(N \log N)$	log–linear	sorting, $O(N \log N)$
$O(N^2)$	quadratic	naive vector-matrix mult., preprocessing
$O(N^3)$	cubic	naive matrix inversion, $O(n^{2.373})$
$O(N^3)$	cubic	naive matrix-matrix mult., $O(n^{2.373})$
$O(N^c)$	polynomial	
$O(c^n)$	exponential	brute force cracking of password, **???**

Quicksort: $O(N^2)$ worst case, but $O(N \log N)$ on average

Determine complexity

```
statement 1
statement 2
...
statement c
```

$O(1)$

Determine complexity

```
if(a)
  statement a
else
  statement b
```

max(O(a),O(b))

Determine complexity

```
for(i in 1:N)
  statement i
```

O(n)

Determine complexity

```
for(i in 1:N)
  for (j in 1:M)
    statement i,j
```

O ?

Determine complexity

```
for(i in 1:N)
  for (j in 1:M)
    statement i,j
```

$O(N * M)$

Determine complexity

```
for(i in 1:N)
  g(i)
```

$g(n) = O(n^2)$

$O(n^3)$

Sorting

naïve sorting: $O(n^2)$
merge sort: $O(n \log n)$ but large number of copies
"merge sorted lists of two into four, then those and so on"

`sort()`
quicksort: average (uniform) $O(n \log n)$, worst $O(n^2)$, low overhead
radix sort: $O(n \cdot k)$, sorts numbers on k digits, by using the digits
shell sort: $O(n^{4/3})$ sorts in–place by swapping elements

Analysis of recursive algorithms (mergesort)

```
mergesort<-function(L){
## assume n=2^k
n<-length(L)
if (n==1){return(L)}
else{
    L1<-mergesort(L[1:(n/2)])
    L2<-mergesort(L[(n/2+1):n])
    ## merge is done in O(n) time
    return(merge(L1,L2))
}}
```

$$T(n) \leq \begin{cases} c_1 & n = 1 \\ 2T(n/2) + c_2 n & n > 1 \end{cases}$$

Analysis of recursive algorithms (Master Theorem)

A function f is multiplicave if $f(xy) = f(x)f(y)$
Let $a, b, c > 0$, $k \in \mathbb{N}$ and $d(n)$ be a multiplicative function. Then the solution to the recurrence equation

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(n/b) + d(n) & n = b^k \end{cases}$$

is

$$T(n) = \Theta(a^k) + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

with asymptotic behaviour

$$T(n) = \begin{cases} \Theta(n^{\log_a d(b)}) & a < d(b) \\ \Theta(n^{\log_b a} \log n) & a = d(b) \\ \Theta(n^{\log_b a}) & a > d(b) \end{cases}$$

Analysis of recursive algorithms (mergesort)

$c_n n$ is not multiplicative so take $T(n) = c_2 \tilde{T}(n)$, then

$\tilde{T}(1) = T(1)/c_2 = c_1/c_2 = c$
$T(n) = 2T(n/2) + c_2 n$ becomes $c_2 \tilde{T}(n) = 2c_2 \tilde{T}(n/2) + c_2 n$

Consider

$$U(n) = \begin{cases} c & n = 1 \\ 2U(n/2) + n & n > 1 \end{cases}$$

n is multiplicative and using the Master Theorem we obtain

$U(n) = \Theta(n \log n)$ and hence $U(n) \geq T(n) = O(n \log n)$.

Actually $T(n) = \Theta(n \log n)$.

Approximate algorithms

If we cannot solve a hard problem let us approximate its solution. Let S_{opt} be the optimal solution and S_{approx} the approximate one

k–absolute approximate algorithm if $|S_{opt} - S_{approx}| \leq k$

k–(relative) approximate algorithm if $s \leq k$, where

$$s = \max(S_{opt}/S_{approx}, S_{approx} - S_{opt})$$

LAB: knapsack problem

The End... for today.
Questions?
See you next time!