

A (Far Too) Short Introduction to Computational Complexity

Fabrice Rossi

SAMM
Université Paris 1 Panthéon Sorbonne

2018

Analysis of algorithms

Resources

- ▶ running a program uses *resources*
- ▶ two most obvious ones:
 1. time
 2. memory (as in volatile one)
- ▶ less obvious ones:
 - ▶ permanent memory
 - ▶ hard drive bandwidth
 - ▶ network bandwidth
 - ▶ etc.

Algorithm analysis

- ▶ abstract analysis of the resource consumption of an algorithm
- ▶ predicts the typical behavior of a program that implements the algorithm given the characteristics of its inputs

Basic example

R illustration : maximum

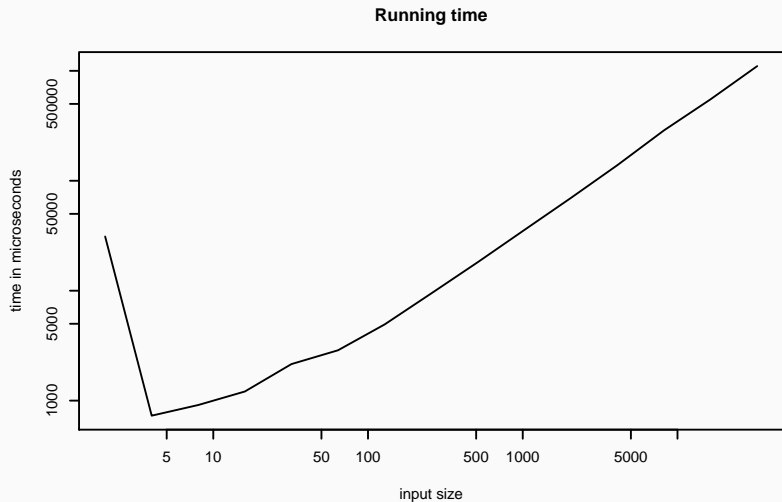
```
x <- rnorm(16)
y <- x[1]
for(i in 2:length(x)) {
  if(x[i] > y) {
    y <- x[i]
  }
}
```

- ▶ very sub-optimal R code (use `max(x)` !)
- ▶ input: the `x` vector
- ▶ output: the `y` value
- ▶ questions:
 - ▶ how long will this code run given the length of `x`?
 - ▶ how much memory will it use?

Experimental measurements

- ▶ time: [microbenchmark](#) package
- ▶ memory: [profmem](#) package

Example



Experimental measurements

Use

- ▶ evaluate the platform, the implementation and the algorithm
- ▶ profiling:
 - ▶ validating formal models
 - ▶ finding hot spots for further optimization

Difficulties

- ▶ data size
- ▶ measurement precision (especially for small input)
- ▶ resource consumption
- ▶ environment

Must be done after programming!

Advantages

- ▶ generic analysis (algorithmic level)
- ▶ asymptotic behavior: predicts the complexity for large scale input
- ▶ no implementation needed

Limitations

- ▶ a bit too abstract in some situations (e.g. most analysis disregard the memory hierarchy)
- ▶ very difficult to conduct in some cases
- ▶ mismatch between observed behavior and predicted ones in complex cases (e.g. simplex algorithm under simple analyses)

Main components

- ▶ abstract model of the computer
- ▶ worst-case or average-case analysis
- ▶ asymptotic analysis

Abstract model

- ▶ theoretical level: **Turing machine**
- ▶ practical level:
 - ▶ uniform cost model: each instruction has the same cost (one!)
 - ▶ instructions:
 - ▶ reading or writing a single value in a variable
 - ▶ comparing two values
 - ▶ standard arithmetic operations
- ▶ variations: taking into account only floating point operations, taking care of transcendental functions (e.g. \exp), etc.

Basic example

Find the maximum

```
x <- rnorm(16)
y <- x[1]
for(i in 2:length(x)) {
  if(x[i] > y) {
    y <- x[i]
  }
}
```

- ▶ we disregard the first line: this is the input
- ▶ outside of the loop: 2 instructions (one assignment, one read)
- ▶ inside the loop: **everything depends on the values!**

How to handle this difficulty?

Worst-case analysis

Principle

- ▶ in general, the exact instructions performed by an algorithm depend on the input
- ▶ this renders the analysis very difficult
- ▶ simple solution:
 - ▶ always consider the worst case: **worst-case analysis**
 - ▶ in tests, always chose the most complex branch
 - ▶ in loops, always assume the loop will run for the maximum time

Average-case analysis

- ▶ principle:
 - ▶ chose a probabilistic distribution on the input space
 - ▶ compute the cost for each possible input
 - ▶ average the costs using the distribution
- ▶ frequently more realistic but very difficult

Basic example

Find the maximum

```
x <- rnorm(16)
y <- x[1]
for(i in 2:length(x)) {
  if(x[i] > y) {
    y <- x[i]
  }
}
```

- ▶ outside of the loop: 2 instructions (1 assignment, 1 read)
- ▶ inside the loop:
 - ▶ always 3 instructions (2 reads, 1 comparison)
 - ▶ 2 additional ones in some cases
- ▶ the loop runs $N - 1$ times for an input of length N

What about the `for` itself?

High level constructs

Problem

- ▶ most programming languages feature high level instructions and data structures
- ▶ those might seem opaque on a cost point of view
- ▶ specifications and/or documentations are needed to make a proper cost analysis

In R

- ▶ `a:b`
 - ▶ creates a vector of length $b-a+1$
 - ▶ the creation cost should be proportional to the length
- ▶ `i in z`
 - ▶ access to all the content: a number of access equal to length z
 - ▶ moving from one cell to another might take only a fix number of operations

Basic example

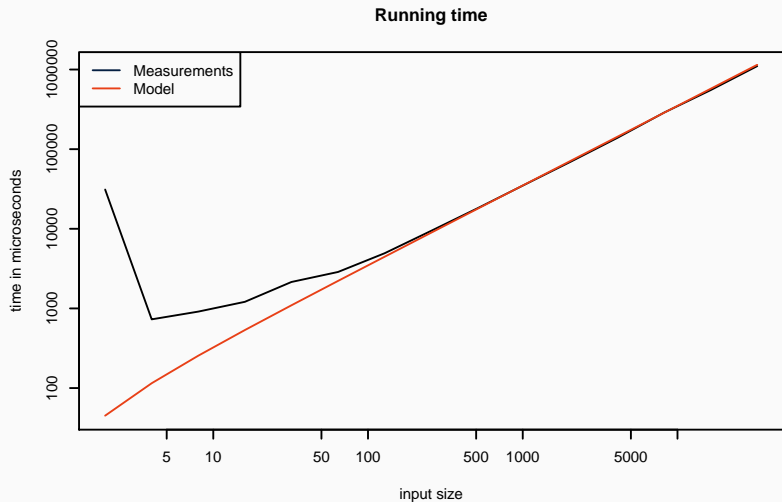
Find the maximum

```
x <- rnorm(16)
y <- x[1]
for(i in 2:length(x)) {
  if(x[i] > y) {
    y <- x[i]
  }
}
```

Total: $2 + 7(N - 1)$

- ▶ outside of the loop: 2 instructions (1 assignment, 1 read)
- ▶ inside the loop (worst-case): 5 instructions per iteration
- ▶ the loop runs $N - 1$ times for an input of length N
- ▶ the loop costs $2(N - 1)$ operations (creating the index and browsing it)

Example



Asymptotic analysis

Principle

Calculate resource usage formulae of an algorithm that are valid when the size of the input goes to infinity.

Motivations

- ▶ practical:
 - ▶ small size inputs drive implementations into very complex zones with problems of overheads and caches
 - ▶ benchmarking is easy for small size inputs not for large ones!
- ▶ theoretical:
 - ▶ eases a lot the analysis
 - ▶ enables one to define classes of comparable algorithm

Definitions

Let f and g be functions from \mathbb{N} to \mathbb{R}

- ▶ f is $\mathcal{O}(g)$ ($f = \mathcal{O}(g)$) if there are M and n_0 such that for all $n \geq n_0$,
 $|f(n)| \leq M|g(n)|$
- ▶ f is $o(g)$ ($f = o(g)$) if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ (with a natural extension to g that can take 0 values)
- ▶ f is $\Theta(g)$ ($f = \Theta(g)$) if there are m, M and n_0 such that for all $n \geq n_0$, $m|g(n)| \leq |f(n)| \leq M|g(n)|$
- ▶ $f \sim g$ if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$

Properties

Numerous interesting properties, such as

- ▶ $f = \Theta(g)$ if and only if $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$
- ▶ if f is a polynomial of degree d , then $f = \Theta(n^d)$ (with $n^0 = 1$)
- ▶ if λ is a non zero constant and $f = \Theta(g)$, then $\lambda f = \Theta(g)$
- ▶ if $f_1 = \mathcal{O}(g_1)$ and $f_2 = \mathcal{O}(g_2)$, then

$$f_1 + f_2 = \mathcal{O}(|g_1| + |g_2|)$$

$$f_1 f_2 = \mathcal{O}(g_1 g_2)$$

- ▶ if $f = \Theta(g)$ and $h = o(g)$ then $f + h = \Theta(g)$

Principle revisited

Given an algorithm with an input of size N , find a function $g(N)$ such that true resource usage of the algorithm f is $\mathcal{O}(g)$ (or better $\Theta(g)$)

Practical consequences

- ▶ precise instruction counting is generally useless
- ▶ on the fly approximation can be used to analyze complex structures
- ▶ documentation/specification need only to give asymptotic guarantees
- ▶ any program with only basic instructions and no loop is $\Theta(1)$ in time!

Basic example

Find the maximum

```
x <- rnorm(16)
y <- x[1]
for(i in 2:length(x)) {
  if(x[i] > y) {
    y <- x[i]
  }
}
```

Total: $\Theta(N)$

- ▶ outside of the loop: do not care!
- ▶ inside the loop (worst-case): $\Theta(1)$ instruction
- ▶ the loop runs $N - 1$ times for an input of length N
- ▶ the loop costs $\Theta(N)$ operations (creating the index and browsing it)

Complexity hierarchy

Important complexity levels

| Complexity | Name |
|---------------------------------------|-------------|
| $\Theta(1)$ | constant |
| $\Theta(\log N)$ | logarithmic |
| $\Theta(N^{\frac{1}{c}})$ for $c > 1$ | fractional |
| $\Theta(N)$ | linear |
| $\Theta(N \log N)$ | quasilinear |
| $\Theta(N^2)$ | quadratic |
| $\Theta(N^3)$ | cubic |
| $\Theta(N^c)$ for $c > 1$ | polynomial |
| $\Theta(c^N)$ for $c > 1$ | exponential |
| $\Theta(N!)$ | factorial |

Analysing an algorithm

Simple cases

- ▶ when:
 - ▶ no high level operations are called
 - ▶ no recursion is used
- ▶ identify the loops
- ▶ determine their worst case number of iterations
- ▶ for nested loops multiply the costs

Remarks

- ▶ mechanisms that handle loops are generally accounted for implicitly by considering each iteration has a constant bookkeeping cost associated to those mechanisms
- ▶ the input size might be characterized by several parameters (e.g., rows and columns for a matrix)

Example

Find the maximum

```
X <- matrix(rnorm(10*10),
            ncol=10, nrow=10)
y <- -Inf
for(i in 1:nrow(X)) {
  for(j in 1:ncol(X)) {
    if(X[i,j] > y) {
      y <- X[i,j]
    }
  }
}
```

- ▶ input size N^2 (or N depending on the point of view)
- ▶ nested loops with N iteration each: $\Theta(N \times N)$
- ▶ inside the inner most loop: $\Theta(1)$ (as always!)
- ▶ the loop costs are automatically taken care off

Total: $\Theta(N^2)$

- ▶ quadratic with respect to N
- ▶ but in fact linear with respect to the input size!

Recursion

- ▶ difficult case
- ▶ leads in general to recursive definition of $f(N)$ the resource usage function
- ▶ general theorems help expressing f in closed form (the so-called **Master theorem**)
- ▶ outside the scope of this introduction

High level operations and API calls

- ▶ use documentation/specification for API calls
- ▶ rely on general complexity theory results (and hope for the best!)

Well known results

| Problem | Complexity |
|---|----------------------------|
| Finding a value in a hash table of size N | $\Theta(1)$ or $\Theta(N)$ |
| Finding a value in a sorted table of size N | $\Theta(\log N)$ |
| Sorting N values | $\Theta(N \log N)$ |
| Multiplying a matrix $N \times P$ by a vector P | $\Theta(NP)$ |
| Multiplying two matrices of size $N \times P$ and $P \times Q$ | $\Theta(NPQ)$ |
| Inverting a $N \times N$ matrix | $\Theta(N^3)$ |
| Eigenvalue decomposition of a $N \times N$ dense matrix | $\Theta(N^3)$ |
| Singular value decomposition of a $M \times N$ matrix ($M \geq N$) | $\Theta(MN^2)$ |

Power method

```
X <- matrix(rnorm(10*10),  
           ncol=10, nrow=10)  
X <- X+diag(1:10)  
X <- X + t(X)  
y <- rnorm(nrow(X))  
y <- y/sqrt(sum(y^2))  
repeat {  
  ny <- X %*% y  
  ny <- ny/sqrt(sum(ny^2))  
  delta <- sum((ny-y)^2)  
  y <- ny  
  if(delta < 1e-8) {  
    break  
  }  
}
```

- ▶ problem characteristics: N ($N \times N$ matrix)
- ▶ initialization: $\Theta(N^2)$
- ▶ inside the inner loop: $\Theta(N^2)$
- ▶ how many iterations?

Power method

```
X <- matrix(rnorm(10*10),
            ncol=10, nrow=10)
X <- X+diag(1:10)
X <- X + t(X)
y <- rnorm(nrow(X))
y <- y/sqrt(sum(y^2))
repeat {
  ny <- X %*% y
  ny <- ny/sqrt(sum(ny^2))
  delta <- sum((ny-y)^2)
  y <- ny
  if(delta < 1e-8) {
    break
  }
}
```

- ▶ problem characteristics: N ($N \times N$ matrix)
- ▶ initialization: $\Theta(N^2)$
- ▶ inside the inner loop: $\Theta(N^2)$
- ▶ **how many iterations?**
- ▶ need some advanced mathematical results
- ▶ here the convergence is linear: the precision is multiplied by a fixed quantity at each iteration
- ▶ loop number $\mathcal{O}(\log(\frac{1}{\epsilon}))$

Decision problems

- ▶ decision problem: a recognition problem in which given an input the answer is yes or no
- ▶ solving the problem consists in building a program that associate the correct answer to any input
- ▶ P class: problems for which an algorithm in $\mathcal{O}(N^k)$ is known
- ▶ NP problems:
 - ▶ NP stands nondeterministic polynomial (for complex reasons)
 - ▶ a problem is NP if a proof that the correct answer is yes can be verified in polynomial time

Examples

P is $A = BC$? for A , B and C matrices

NP does a given graph possess a Hamiltonian cycle?

NP-complete and NP-hard

Reduction

- ▶ A and B two problems
- ▶ A reduces to B if any input for A can be transformed into an input for B such that the answer for this transformed input is the correct one for original input

NP-hard

B is NP-hard if any NP problem is reducible to B in polynomial time.

NP-complete

A NP-complete problem is a NP problem that is also NP-hard.

NP-hard problems

A complicated class

- ▶ NP-hard problem include strictly NP-complete problem
- ▶ some problems in NP-hard are not in NP and not even in the class of decidable problems (e.g. the halting problem)

Optimization problems

- ▶ optimization problems are more general than decision problems
- ▶ translation to decision problems is straightforward: given an optimization problem T one can ask a series of yes/no questions of the form “is there a solution to T with cost below t ?”
- ▶ iconic NP-hard problems are optimization ones, for instance the travelling salesman problem

P versus NP

In a nutshell

See [the wikipedia](#) for details

- ▶ in practice, we only know exponential time algorithms for solving NP-complete problems
- ▶ can we either prove either that there are effectively no polynomial time solutions for NP-complete problem or that $P = NP$?
- ▶ this is one million price problem...

In practice

- ▶ if a problem is NP-hard, we cannot currently solve it *exactly* in reasonable time
- ▶ but many of NP-hard optimization problems admit fast algorithms that provide approximate results with reasonable quality guarantees

What about memory consumption?

- ▶ in general this is straightforward
- ▶ but in practice one might run into problems, especially with R
- ▶ semantics of $x \leftarrow y$?

Complexity and machine learning

- ▶ machine learning is strongly related to optimization
- ▶ many ML optimization problems are NP-hard:
 - ▶ empirical risk minimization for the binary cost
 - ▶ k-means criterion optimization
 - ▶ etc.
- ▶ strong reliance on approximate algorithms



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

<http://creativecommons.org/licenses/by-sa/4.0/>

Last modification: 2018-02-05

By: Fabrice Rossi (Fabrice.Rossi@apiacoa.org)

Git hash: 1b0849e751c9b992d0eb957563be19636769feaa