A (Far Too) Short Introduction to Computational Complexity

Fabrice Rossi

CEREMADE
Université Paris Dauphine

2021

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
  - hand 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

Python illustration: maximum

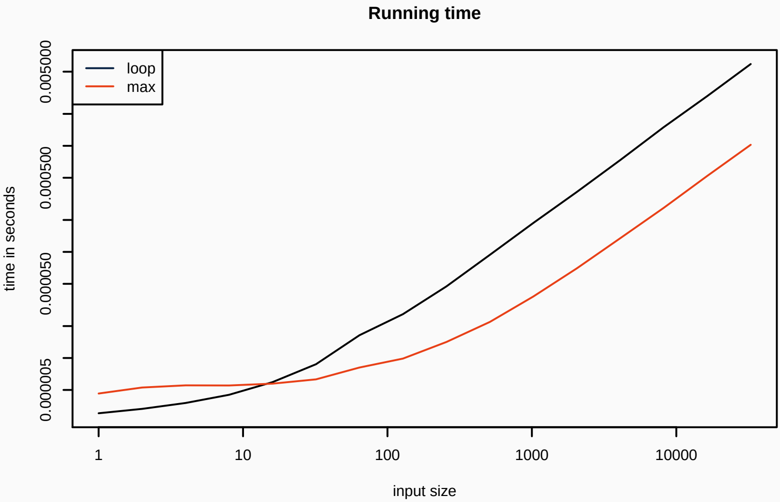
```
import numpy as np
x = np.random.normal(size=(16,))
y = x[0]
for k in range(1,x.shape[0]):
    if x[k]>y:
        y = x[k]
print(y)
```

- very sub-optimal Python code (use `x.max()`!)
- 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 in Python

- time: `timeit` module
- memory: `memory_profile` module

Example



Running time
loop
max
time in seconds
0.000005
0.000050
0.000500
0.005000
1
10
100
1000
10000
input size

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!

Theoretical analysis

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)

Principles

Main components

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

Asbtract 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

```
import numpy as np
x = np.random.normal(size=(16,))
y = x[0]
for k in range(1,x.shape[0]):
    if x[k]>y:
        y = x[k]
print(y)
```

- we disregard the first two lines: import and input
- we disregard the last line: output
- 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

```
import numpy as np
x = np.random.normal(size=(16,))
y = x[0]
for k in range(1,x.shape[0]):
    if x[k]>y:
        y = x[k]
print(y)
```

- 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 Python

- `range(a,b)`
  - creates an iterable
  - the creation cost should be constant
- `k 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, typically 2: checking if the end is reached and reading a value

Basic example

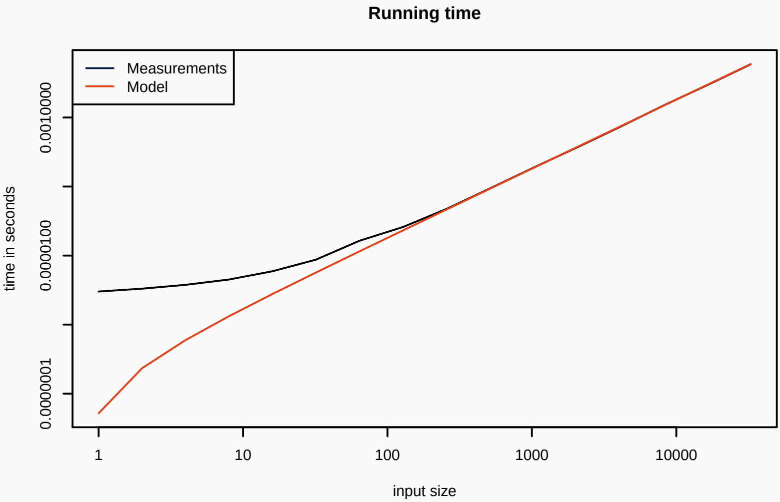
Find the maximum

```
import numpy as np
x = np.random.normal(size=(16,))
y = x[0]
for k in range(1,x.shape[0]):
    if x[k]>y:
        y = x[k]
print(y)
```

- 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)

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

Example



Running time
Measurements
Model
0.0010000
0.0000100
0.0000001
time in seconds
1
10
100
1000
10000
input size

Asymptotic analysis

Principle

Calculate resource usage formulaes 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

Big O notations

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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ (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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

Big O notations

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 $f = \Theta\left(n^d\right)$ (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 $f + h = \Theta(g)$

Asymptotic analysis

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

```
import numpy as np
x = np.random.normal(size=(16,))
y = x[0]
for k in range(1,x.shape[0]):
    if x[k]>y:
        y = x[k]
print(y)
```

- 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)

Total: $\Theta(N)$

Complexity hierarchy

Important complexity levels

Complexity	Name
$\Theta(1)$	constant
$\Theta(\log N)$	logarithmic
$\Theta\left(N^{\frac{1}{c}}\right)$ for $c > 1$	fractional
$\Theta(N)$	linear
$\Theta(N \log N)$	quasilinear
$\Theta\left(N^2\right)$	quadratic
$\Theta\left(N^3\right)$	cubic
$\Theta(N^c)$ for $c > 1$	polynomial
$\Theta\left(c^N\right)$ 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

```
import numpy as np
x = np.random.normal(size=(10,10))
y = x[0,0]
for i in range(0,x.shape[0]):
    for j in range(0,x.shape[1]):
        if x[i,j] > y:
            y = x[i,j]
print(y)
```

- 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: $\Theta\left(N^2\right)$

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

More complex programs

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	$\Theta\left(N^3\right)$
Eigenvalue decomposition of a $N \times N$ dense matrix	$\Theta\left(N^3\right)$
Singular value decomposition of a $M \times N$ matrix ($M \geq N$)	$\Theta\left(MN^2\right)$

Example

Power method

```
import numpy as np

def power_method(A, prec=1e-8):
    x = np.random.random(A.shape[1])
    iterate = True
    while(iterate):
        nx = A@x
        nx /= np.linalg.norm(nx)
        delta = np.linalg.norm(x - nx)
        x = nx
        iterate = delta > prec
    eigenvalue = np.dot(x, A@x)
    eigenvalue /= np.dot(x, x)
    return eigenvalue,x
```

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

Example

Power method

```
import numpy as np

def power_method(A, prec=1e-8):
    x = np.random.random(A.shape[1])
    iterate = True
    while(iterate):
        nx = A@x
        nx /= np.linalg.norm(nx)
        delta = np.linalg.norm(x - nx)
        x = nx
        iterate = delta > prec
    eigenvalue = np.dot(x, A@x)
    eigenvalue /= np.dot(x, x)
    return eigenvalue,x
```

- problem characteristics: N ($N \times N$ matrix)
- initialization: $\Theta\left(N\right)$
- inside the inner loop: $\Theta\left(N^2\right)$
- 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 $\mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\right)$

NP problems

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 $\mathcal{O}\left(N^k\right)$ 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

Concluding remarks

What about memory consumption?

- in general this is straightforward
- but in practice one might run into problems, especially with NumPy
- semantics of `x = 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

Licence

Version

Last git commit: 2021-01-19
By: Fabrice Rossi (Fabrice.Rossi@apiacoa.org)
Git hash: 97cfd0a9975cf193f5790845c00e476c1572a327

Changelog

- Mars 2020: initial Python version