Programming Fundamentals

Analysis of Algorithms

João Correia Lopes

INESC TEC, FEUP

22 November 2018

GOALS

By the end of this class, the student should be able to:

- Describe why algorithm analysis is important
- Use “Big-O” to describe execution time
- Describe the “Big-O” execution time of common operations on Python lists and dictionaries

BIBLIOGRAPHY

- Allen Downey, Think Python — How to Think Like a Computer Scientist, 2nd Edition, Version 2.2.23, Green Tea Press, 2015 (Annex B) [HTML] [PDF]
- Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures using Python (Section 5.3, Section 5.4) [HTML]
- Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures using Python (Chapter 2) [HTML]

TIPS

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

CONTENTS

Analysis of Algorithms

- Analysis of algorithms is a branch of computer science that studies the performance of algorithms, especially their run time and space requirements (Wikipedia)
- The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions
- Eric Schmidt jokingly asked Obama for “the most efficient way to sort a million 32-bit integers” and he quickly replied: “I think the bubble sort would be the wrong way to go” (YouTube)

Problems when Comparing Algorithms

The goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

- The relative performance of the algorithms might depend on characteristics of the hardware
  - the general solution to this problem is to specify a machine model and analyze the number of steps, or operations, an algorithm requires under a given model
- Relative performance might depend on the details of the dataset
  - a common way to avoid this problem is to analyze the **worst case scenario**
- Relative performance also depends on the size of the problem
  - the usual solution to this problem is to express run time (or number of operations) as a function of problem size, and group functions into categories depending on how quickly they grow as problem size increases

Run time

- Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input:
  - Algorithm A takes $T(n) = 100n + 1$ steps to solve a problem with size n
  - Algorithm B takes $T(n) = n^2 + n + 1$ steps to solve a problem with size n
- The following table shows the run time of these algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

ORDER OF GROWTH

- The **leading term** is the term with the highest exponent
- There will always be some value of n where $an^2 > bn$, for any values of a and b
- For algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients
- An order of growth is a set of functions whose growth behaviour is considered equivalent
  - For example, $2n$, $100n$ and $n + 1$ belong to the same order of growth
  - They are all linear

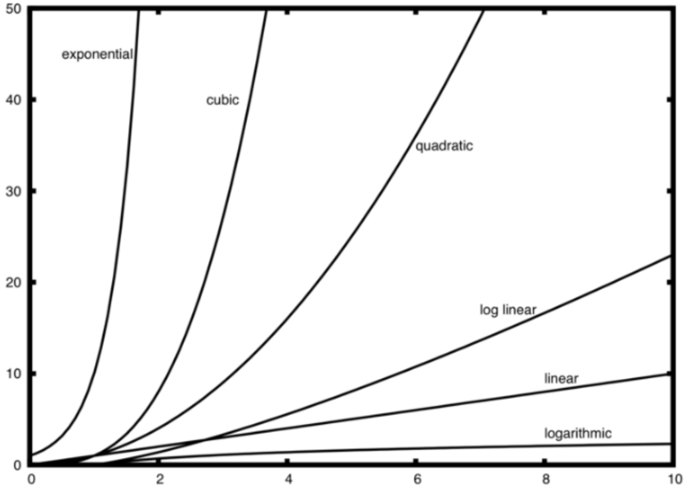
BIG-O NOTATION

- $T(n)$ is the time it takes to solve a problem of size n
- The **order of magnitude** function describes the part of $T(n)$ that increases the fastest as the value of n increases
- Order of magnitude is often called **Big-O notation** (for "order") and written as $O(f(n))$
- It provides a useful approximation to the actual number of steps in the computation

Common Order of Magnitude Functions

f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
$2n$	Exponential

Common Order of Magnitude Functions



Compute $T(n)$

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$$

$$O(n^2)$$

COMPUTE $T(n)$

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$$

$$O(n^2)$$

COMPUTE $T(n)$

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$$

$$O(n^2)$$

PERFORMANCE OF PYTHON DATA STRUCTURES

- Now that you have a general idea of Big-O notation and the differences between the different functions
- Let's talk about the Big-O performance for the operations on Python lists and dictionaries
- It is important for you to understand the efficiency of these Python data structures because they are the building blocks we will use as we implement other data structures
- The designers of Python had many choices to make when they implemented data structures

⇒ https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython

LISTS

Operation	Big-O Efficiency
index []	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Dictionaries

As you probably recall, dictionaries differ from lists in that you can access items in a dictionary by a key rather than a position

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

SUMMARY

- Algorithm analysis is an implementation-independent way of measuring an algorithm
- Big-O notation allows algorithms to be classified by their dominant process with respect to the size of the problem.

EXERCISES

- Moodle activity at: LE16: Analysis of Algorithms