1 Machines & Models

7 February 2022

Sebastian Wild

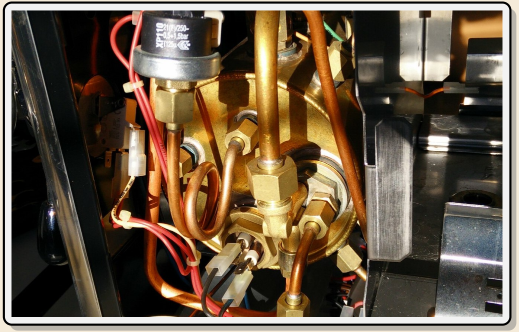
COMP526 (Spring 2022)
University of Liverpool

version `2022-01-31 22:14 H`

Learning Outcomes

1. Understand the difference between empirical *running time* and algorithm *analysis*.
2. Understand *worst / best / average case* models for input data.
3. Know the *RAM machine* model.
4. Know the definitions of *asymptotic notation* (Big-Oh classes and relatives).
5. Understand the reasons to make *asymptotic approximations*.
6. Be able to *analyze* simple *algorithms*.

Unit 1: *Machines & Models*



Outline

What is an algorithm?

An algorithm is a sequence of instructions.

think: recipe

More precisely:

e. g. Java program

1. mechanically executable
   $\rightsquigarrow$ no "common sense" needed
2. finite description $\neq$ finite computation!
3. solves a problem, i. e., a class of problem instances

$x + y$, not only $17 + 4$

typical example: *bubblesort*

not a specific program but underlying idea

What is a data structure?

A data structure is

1. a rule for encoding data
   (in computer memory), plus
2. algorithms to work with it
   (queries, updates, etc.)

typical example: binary search tree

1.1 Algorithm analysis

Good algorithms

Our goal: Find good (best?) algorithms and data structures for a task.

Good "usually" means

can be complicated in distributed systems

- fast running *time*
- moderate memory *space* usage

Algorithm analysis is a way to

- compare different algorithms,
- predict their performance in an application

Running time experiment

Why not simply run and time it?

- results only apply to
  - single *test* machine
  - tested inputs
  - tested implementation
  - ...
  - ≠ ***universal truths***
- instead: consider and analyze algorithms on an abstract machine (survives Pentium 4)
  - ⇝ provable statements for model
  - ⇝ testable model hypotheses

⇝ Need precise model of machine (costs), input data and algorithms.

Data Models

Algorithm analysis typically uses one of the following simple data models:

- **worst-case performance:**
  consider the *worst* of all inputs as our cost metric
- **best-case performance:**
  consider the *best* of all inputs as our cost metric
- **average-case performance:**
  consider the average/expectation of a *random* input as our cost metric

Usually, we apply the above for *inputs of same size* n.

$\rightsquigarrow$ performance is only a **function of** n.

1.2 The RAM Model

Machine models

The machine model decides

- what algorithms are possible
- how they are described (= programming language)
- what an execution *costs*

Goal: Machine model should be
detailed and powerful enough to reflect actual machines,
abstract enough to unify architectures,
simple enough to analyze.

Random Access Machines

Random access machine (RAM)

more detail in §2.2 of *Sequential and Parallel Algorithms and Data Structures* by Sanders, Mehlhorn, Dietzfelbinger, Dementiev

- unlimited *memory* $\texttt{MEM}[0], \texttt{MEM}[1], \texttt{MEM}[2], \ldots$
- fixed number of *registers* $R_1, \ldots, R_r$ (say $r = 100$)
- memory cells $\texttt{MEM}[i]$ and registers R_i store w-bit integers, i. e., numbers in $[0..2^w - 1]$
  w is the word width/size; typically $w \propto \lg n \quad \rightsquigarrow \quad 2^w \approx n$
- Instructions:
  - load & store: $R_i := \texttt{MEM}[R_j] \quad \texttt{MEM}[R_j] := R_i$
  - operations on registers: $R_k := R_i + R_j$ (arithmetic is *modulo* 2^w!)
    also $R_i - R_j$, $R_i \cdot R_j$, $R_i \operatorname{div} R_j$, $R_i \bmod R_j$
    C-style operations (bitwise and/or/xor, left/right shift)
  - conditional and unconditional jumps
- cost: number of executed instructions

we will see further models later

$\rightsquigarrow$ The RAM is the standard model for sequential computation.

Pseudocode

Typical simplifications for convenience:

- more abstract *pseudocode* to specify algorithms
  ↖ code that humans understand (easily)
- count *dominant operations* (e. g. array accesses) instead of all operations

In both cases: can go to full detail if needed.

1.3 Asymptotics & Big-Oh

Why asymptotics?

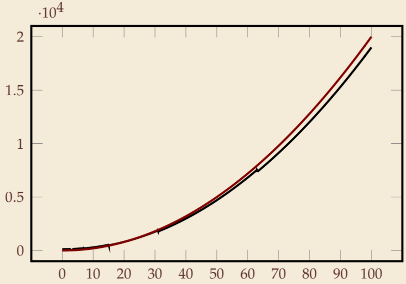
Algorithm analysis focuses on (the limiting behavior for infinitely) **large** inputs.

- abstracts from unnecessary detail
- simplifies analysis
- often necessary for sensible comparison

Asymptotics = approximation around ∞

Example: Consider a function $f(n)$ given by

$$2n^2 - 3n\lfloor\log_2(n+1)\rfloor + 7n - 3\lfloor\log_2(n+1)\rfloor + 120 \sim \mathbf{2n^2}$$



Asymptotic tools – Formal & definitive definition

- **"Tilde Notation:"** $f(n) \sim g(n)$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$ (iff: if, and only if)

  „f and g are *asymptotically equivalent*"

- **"Big-Oh Notation:"** $f(n) \in O(g(n))$ (also write '=' instead) iff $\left|\frac{f(n)}{g(n)}\right|$ is bounded for $n \ge n_0$

  iff $\limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$ (need supremum since limit might not exist!)

  Variants:

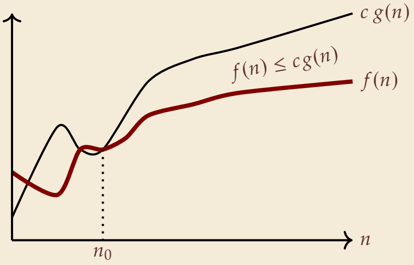
  - $f(n) = \Omega(g(n))$ ("Big-Omega") iff $g(n) = O(f(n))$
  - $f(n) = \Theta(g(n))$ ("Big-Theta") iff $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$

- **"Little-Oh Notation:"** $f(n) = o(g(n))$ iff $\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = 0$

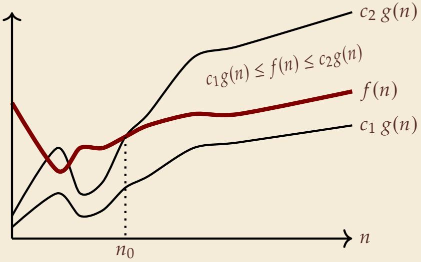
  $f(n) = \omega(g(n))$ if lim $= \infty$

Asymptotic tools – Intuition

- $f(n) = O(g(n))$: $f(n)$ is **at most** $g(n)$ up to constant factors and for sufficiently large n



- $f(n) = \Theta(g(n))$: $f(n)$ is **equal to** $g(n)$ up to constant factors and for sufficiently large n



Plots can be misleading!

Example

Asymptotics – Example 1

Basic examples:

- $20n^3 + 10n\ln(n) + 5 \;\sim\; 20n^3 \;=\; \Theta(n^3)$
- $3\lg(n^2) + \lg(\lg(n)) \;=\; \Theta(\log n)$
- $10^{100} \;=\; O(1)$

Use *wolfram alpha* to compute/check limits.

Asymptotics – Frequently used facts

- Rules:
  - $c \cdot f(n) = \Theta(f(n))$ for constant $c \neq 0$
  - $\Theta(f + g) = \Theta(\max\{f, g\})$ largest summand determines order of growth

- Frequently used orders of growth:
  - logarithmic $\Theta(\log n)$ Note: $a, b > 0$ constants $\rightsquigarrow$ $\Theta(\log_a(n)) = \Theta(\log_b(n))$
  - linear $\Theta(n)$
  - linearithmic $\Theta(n \log n)$
  - quadratic $\Theta(n^2)$
  - polynomial $O(n^c)$ for constant c
  - exponential $O(c^n)$ for constant c Note: $a > b > 0$ constants $\rightsquigarrow$ $b^n = o(a^n)$

Asymptotics – Example 2

Square-and-multiply algorithm
for computing x^m with $m \in \mathbb{N}$

Inputs:

- m as binary number (array of bits)
- n = #bits in m
- x a floating-point number

```
1 double pow(double base, boolean[] exponentBits) {
2     double res = 1;
3     for (boolean bit : exponentBits) {
4         res *= res;
5         if (bit) res *= base;
6     }
7     return res;
8 }
```

- Cost: C = # multiplications
- $C = n$ (line 4) + #one-bits binary representation of m (line 5)

$\rightsquigarrow$ $n \le C \le 2n$