Imperial College London

CO202 – Software Engineering – Algorithms

Introduction

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

Administration

Lectures/Tutorials (Week 2-10)

- **Tuesday, 16:00-18:00, LT 308**
- **Friday, 15:00-16:00, LT 308**

RECORDING
IN PROGRESS

https://imperial.cloud.panopto.eu

Assessment

- **Two courseworks** (groups of three, programming in Python)
- Exam (based on lecture material and courseworks)

Teaching Assistance

- Ghislain Vaillant, Ozan Oktay, Salim Arslan, Daniel Lenton, Andrea Callia D'Iddio, Fahdi Kanavati

Contact us: **doc-staff-202**

Piazza

- piazza.com/imperial.ac.uk/spring2017/202/home
- Online discussion forum
- Students help students, but don’t post solutions

“It starts with students contributing.”

Courseworks

- Groups of three
- Groups will be the same for both courseworks
- **Register your group** on CATE until

Friday, January 27

- Electronic submissions
- **Coursework 1: Feb 3 – Feb 17**
- **Coursework 2: Feb 24 – Mar 10**

Learning Outcomes

Knowledge and Understanding

- Expand your thinking about algorithms and algorithmic design paradigms
- Expose you to several new classes of computational problems and concrete algorithmic solutions
- Give you a sense for general (or commonly useful) approaches to **algorithmic thinking**

Learning Outcomes

Intellectual Skills

- To compare, characterize and evaluate different implementations of basic algorithms
- To design efficient algorithms for practical problems
- To specify which algorithms can be applied to which class of problems

Learning Outcomes

Practical Skills

- To implement efficient algorithms for practical problems
- To perform analysis of algorithms using quantitative evaluation
- To apply basic algorithms to new problems
- *Learn a bit of Python*

Studying Algorithms (& Data Structures)

- Fundamental to any computer science curriculum

- Everyone who uses a computer wants it to **run faster** and/or to **solve larger problems**

- Algorithms are important for… *everything*

Syllabus

- Introduction
- Complexity Analysis
- Divide-and-Conquer
- Dynamic Programming
- Greedy Algorithms
- Randomised Algorithms
- Visualising Algorithms
- String-Matching Algorithms
- Graph Algorithms

References

Books

- [Cormen] **Introduction to Algorithms**
  T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. MIT Press. 2009 (3rd Edition)
- [Sedgewick] **Algorithms**
  R. Sedgewick, K. Wayne. Addison-Wesley. 2011 (4th Edition)
- [Dasgupta] **Algorithms**
  S. Dasgupta, C. Papadimitriou, U. Vazirani. McGraw-Hill Higher Education. 2006

Online

- http://algs4.cs.princeton.edu/lectures/
- https://www.coursera.org/courses?query=algorithms

Word Origin

- **Algorithm** stems from the name of a Persian mathematician, Al-Khwarizmi
- Author of the book 'On the Calculation with Hindu Numerals' in about 825 AD
- Translated into Latin as 'Algoritmi de numero Indorum' (in English 'Al-Khwarizmi on the Hindu Art of Reckoning')

What is an Algorithm? (an informal definition)

- Any well-defined **computational procedure** that takes some value or vector of values as **input** and produces some value or vector of values as **output**

- An algorithm is thus a **sequence of computational steps** that transforms the input into the output

- An algorithm is also a **tool** for solving a well-specified **computational problem**

Example of a Computational Problem

- The *sorting* problem

  input: a sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$

  output: a permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- Some well-known, concrete and **correct** algorithmic solutions to the sorting problem
  - Insertion sort
  - Merge sort
  - Quick sort

What do we mean by ‘correct’?

- An algorithm is considered **correct** if, for every correct input sequence, it halts with the correct output
  - specifications of correct input and output are given by the problem statement
- Easy to say, but hard to do
  - how do we demonstrate that an algorithm is correct?

A Little Thought Exercise

```
RANDOM-SORT(A)
    n = A.length
    sorted = false
    while sorted == false
        A = RANDOM-PERMUTATION(A)
        i = 1
        while i < n and A[i] ≤ A[i+1]
            i = i+1
        if i == n
            sorted = true
```

$$A = \langle 3, 8, 5, 1, 3 \rangle$$

$$A = \langle 1, 3, 3, 5, 8 \rangle$$

How would you establish (in)correctness?

Algorithmic Schemes

- `RANDOM-SORT` embodies a particular algorithmic scheme called **generate and test**

- An **algorithmic scheme** (or **design paradigm**) is a particular way to structure a computational procedure to solve a computational problem

- Examples of other schemes
  - Incremental
  - Divide-and-Conquer
  - Dynamic Programming
  - Greedy

Efficiency

- Correct algorithms are judged by their efficiency at solving a computational problem

  space: amount of (primary) memory

  time: number of CPU cycles

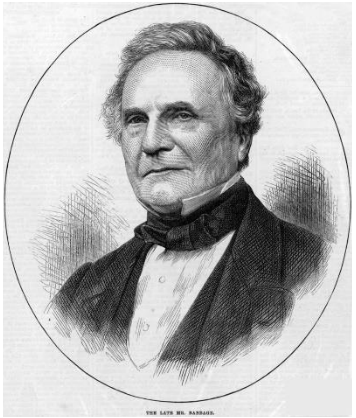
- Efficiency of algorithms (as opposed to programs) is understood in terms of **complexity**

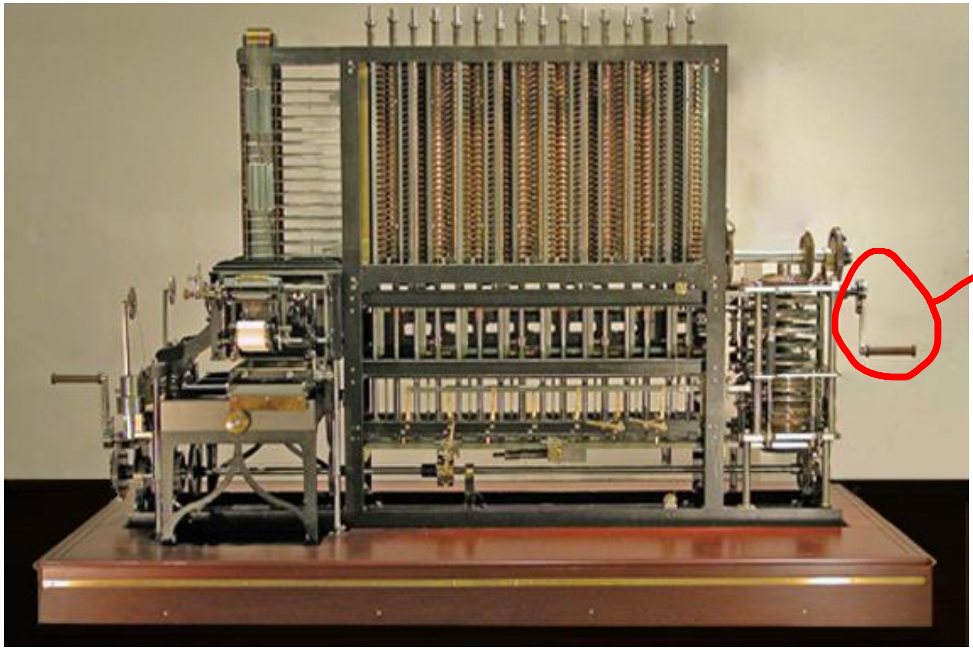
  we focus on **time complexity** (for the purpose of this course), specifically **asymptotic running time**

Efficiency

"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise – By what course of calculation can these results be arrived at by the machine in the shortest time?"

Charles Babbage (1864)





Difference Engine

How often do we need to turn the crank?

Review of Complexity Analysis

Asymptotic running time

- How does the time to compute the output grow as the size of the input grows?
- We answer this question **empirically** for programs, but **analytically** for algorithms
- We need to measure ‘time’

  for programs: observe a wall clock or count CPU cycles

  for algorithms: count abstract **computational steps**
- Random-access machine model of computation

  all memory equally expensive to access

  all basic operations equally expensive to execute

Exercise: Review of Complexity Analysis

```
BUBBLE-SORT(A)                                         cost         times
    n = A.length
    for i = 1 to n-1
        for j = n downto i+1
            if A[j] < A[j-1]
                temp = A[j]
                A[j] = A[j-1]
                A[j-1] = temp
```

Exercise: Review of Complexity Analysis

BUBBLE-SORT(A)	cost	times
n = A.length	c_1	1
for i = 1 **to** n-1	c_2	n
for j = n **downto** i+1	c_3	(n-1)(n-i+1)
if A[j] < A[j-1]	c_4	(n-1)(n-i)
temp = A[j]	c_5	(n-1)(n-i)t_{ij}
A[j] = A[j-1]	c_6	(n-1)(n-i)t_{ij}
A[j-1] = temp	c_7	(n-1)(n-i)t_{ij}

Running time of BUBBLE-SORT

$$T(n) = c_1 + c_2 n + c_3 (n-1)(n-i+1)$$
$$+ c_4 (n-1)(n-i) + (c_5 + c_6 + c_7)(n-1)(n-i)t_{ij}$$

$T(n)$ is a **quadratic function** of n: $T(n) \approx n^2$

Review of Complexity Analysis

Assumption of constant factors

- Does a load/store operation cost more than, say, an arithmetic operation?

`A[j] = A[j-1]` vs. `i+1`

- Specific costs of each basic step are not of concern in algorithmic complexity analysis

  the actual costs are likely to vary significantly, depending on implementation, processor, language, compiler, …

- We assume **all costs are equal** and ignore specific values

$$c_1 = c_2 = \cdots = c_k$$

- In fact, we **ignore every constant factor**

Order of Growth

- We are interested in characterizing the **order of growth** of $T(n)$, not in specific counts of computational steps
- We ignore lower-order terms, since the highest-order term asymptotically dominates
- Example: $T(n) = an^2 + bn + c$

  we consider only the n^2 term (also ignoring factor a), meaning that $T(n)$ is a quadratic function of n

  we write (abusively): $T(n) = \Theta(n^2)$

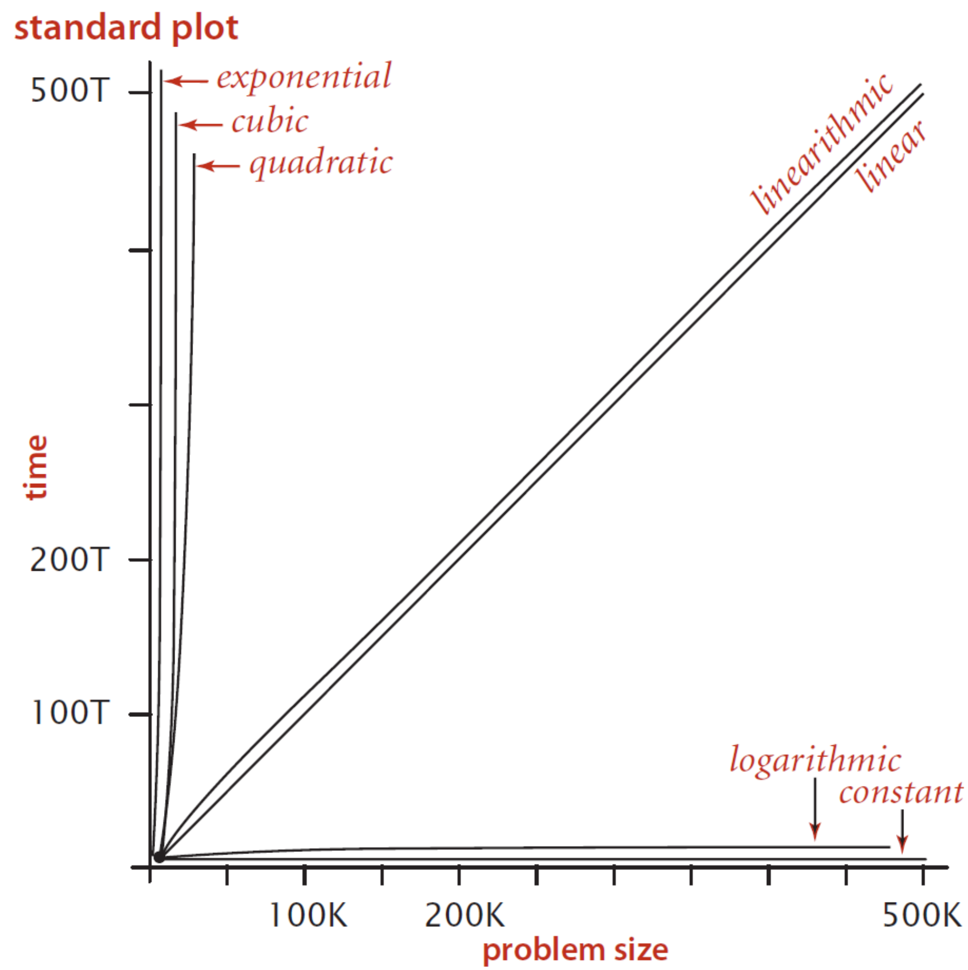
  and say (colloquially): "t of n is big theta of n squared"

Sometimes, you will find $T(n) \in \Theta(n^2)$

Order of Growth

Common order-of-growth hypotheses

$\Theta(1)$	constant
$\Theta(\lg n)$	logarithmic
$\Theta(n)$	linear
$\Theta(n \lg n)$	linearithmic
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$\Theta(2^n)$	exponential

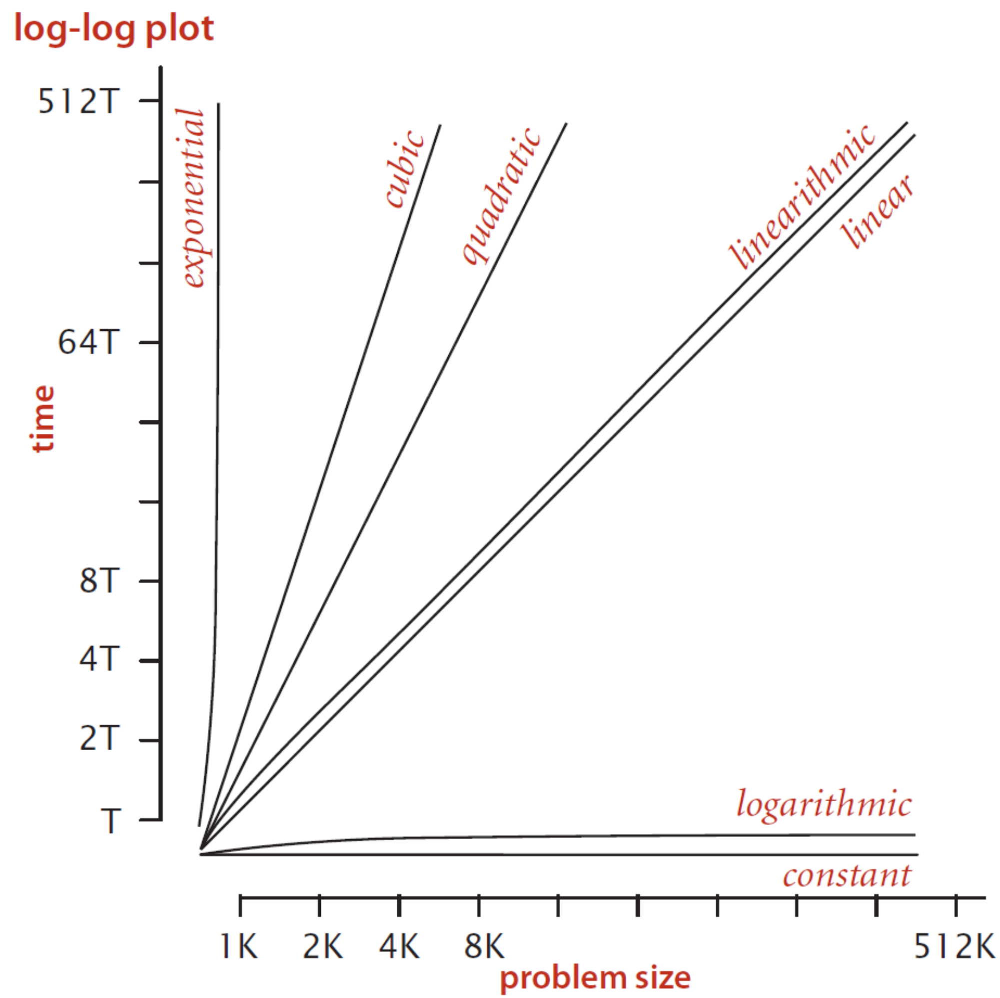


[Sedgewick] p.188

Order of Growth

Common order-of-growth hypotheses

$\Theta(1)$	constant
$\Theta(\lg n)$	logarithmic
$\Theta(n)$	linear
$\Theta(n \lg n)$	linearithmic
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$\Theta(2^n)$	exponential



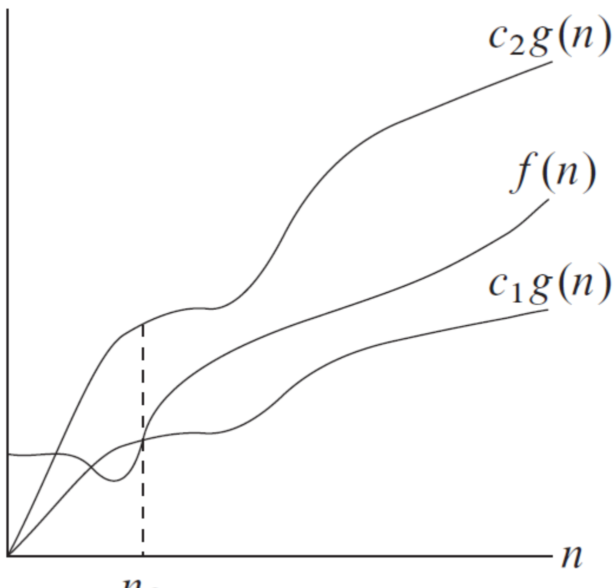
[Sedgewick] p.188

Big Θ-Notation (Theta)

Asymptotic bound

For a given function $g(n)$, we define the set of functions:

$$\Theta\big(g(n)\big) = \{\, f(n) : \ \exists\, c_1, c_2, n_0 > 0 \text{ and } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall\, n \ge n_0 \,\}$$



$f(n) = \Theta(g(n))$

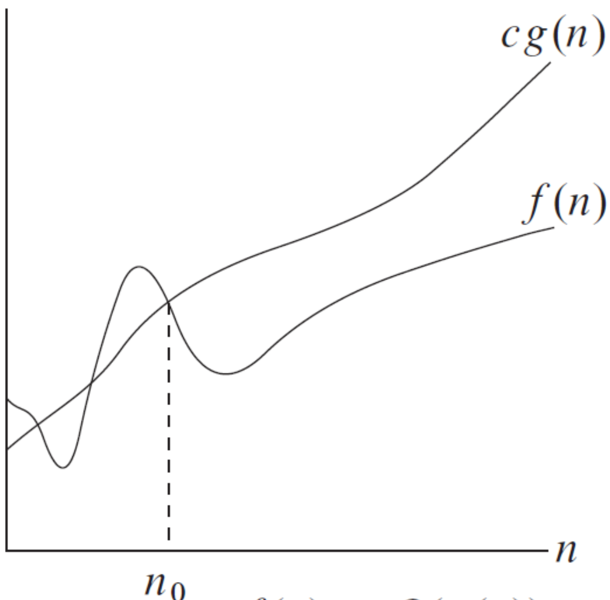
[Cormen] p.45

Big O-Notation

Asymptotic **upper** bound

For a given function $g(n)$, we define the set of functions:

$$O\big(g(n)\big) = \{\, f(n) :\ \exists\, c, n_0 > 0 \text{ and } \\ 0 \le f(n) \le cg(n), \forall\, n \ge n_0 \,\}$$



$f(n) = O(g(n))$

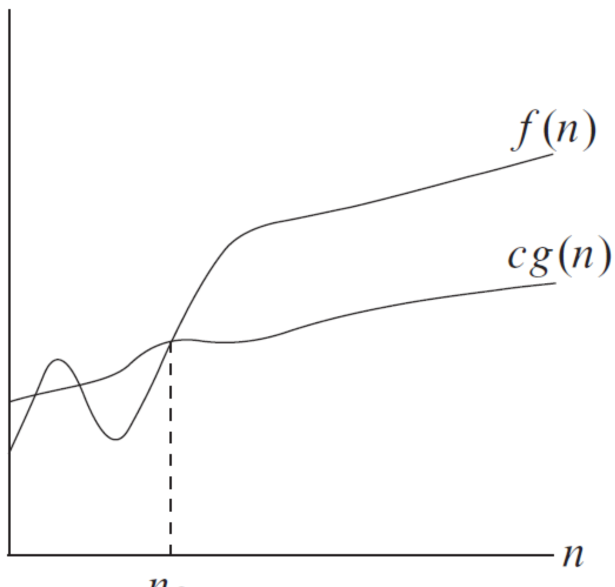
[Cormen] p.45

Big Ω-Notation (Omega)

Asymptotic **lower** bound

For a given function $g(n)$, we define the set of functions:

$$\Omega\big(g(n)\big) = \{\, f(n) : \ \exists\, c, n_0 > 0 \text{ and } \\ 0 \le cg(n) \le f(n), \forall\, n \ge n_0 \,\}$$



$f(n) = \Omega(g(n))$

[Cormen] p.45

Relation of Θ, O, and Ω

Theorem: For any two functions $f(n)$ and $g(n)$

$$f(n) = \Omega\big(g(n)\big) \land f(n) = O\big(g(n)\big) \Leftrightarrow f(n) = \Theta\big(g(n)\big)$$

- The Θ-notation, O-notation, and Ω-notation can be viewed as the asymptotic $=$, $\leq$, and $\geq$ relations for functions

- The theorem above can be interpreted as saying

$$f \geq g \land f \leq g \Leftrightarrow f = g$$

Some Misconceptions Corrected

- Set of functions g is not unique

  General goal of asymptotic complexity analysis is to find the **tightest** bounds, that is, the set of functions g that yields the best characterization of f

- $O(g(n))$ is **not** the ‘worst case bound’
- $\Omega(g(n))$ is **not** the ‘best case bound’
- $\Theta(g(n))$ is **not** the ‘average case bound’

  ‘worst’, ‘best’, and ‘average’ characterize the input data, not the algorithm

What About Data Structures?

- This course focuses (mainly) on algorithms
- Data structures are…
  - used by algorithms to **organize** and **store** input, output, and temporary values during computations
  - and can facilitate **access** to values for retrieving or modifying those values
- Choice of data structure strongly affects the complexity of the algorithm

Assumptions

We make some simplifying assumptions for the purposes of this course

Computational steps are **sequential**

(cf. CO223, Concurrency)

and **centralized**

(cf. CO347, Distributed Algorithms)

Algorithms terminate (and produce a result) after a **finite** number of steps