

Algorithms and Data Structures

or, Classical Algorithms of the 50s, 60s, 70s

Mary Cryan

School of Informatics
University of Edinburgh

Syllabus

Introductory Review of Inf2B basics. Time and space complexity; upper and lower bounds; $O(\cdot)$, $\Omega(\cdot)$ and $\Theta(\cdot)$ notation; average and worst case analysis.

Algebraic algorithms Matrix multiplication: Strassen's algorithm.
Polynomial arithmetic: the Discrete Fourier transform (DFT), the Fast Fourier transform (FFT); recurrence relations for recursive algorithms.

Sorting Analysis of Quicksort; best-case, worst-case and average-case analysis.
Sorting for restricted-values; counting sort, radix sort.

Algorithms and Data Structures

- ▶ Emphasis is “Algorithms” rather than “Data Structures”.
- ▶ More **proving** in ADS than in Inf2B (ADS part).
- ▶ Most of the algorithms we study were *breakthroughs* at the time when they were discovered (50s, 60s, and 70s).
- ▶ Two concerns in ADS:
 1. *Designing* clever algorithms to solve problems.
 2. *Proving* that our algorithms are *correct*, and satisfy certain *bounds on running time*.
- ▶ We use three main techniques to design algorithms:
 1. Divide-and-Conquer
 2. Greedy approach (also called “hill climbing”)
 3. Dynamic programming
- ▶ 2 courseworks: due at NOON, Fridays of **week 5** and of **week 9**.

Syllabus cont.

Dynamic programming: Introduction to the technique; matrix-chain multiplication.

Advanced data structures: Data structures for disjoint sets; Union-by-rank, path-compression, etc., “heuristics”.

Minimum spanning trees: Prim's algorithm (using priority queues); Kruskal's algorithm (using disjoint sets).

Graph/Network algorithms Network flows; Ford-Fulkerson algorithm for finding max flow.

Geometric algorithms: Convex hull of a set of points in two dimensions; Graham's scan algorithm.

Course Book

- ▶ T. H. Cormen, C. E. Leiserson and R. L. Rivest, C. Stein *Introduction to Algorithms (2nd Edition)*, MIT Press 2001.
- ▶ Called **[CLRS]** from now on.
- ▶ *Essential* for the course.

*It will be possible to work with the 1st edition, by authors Cormen, Leiserson and Rivest (without Stein). We will refer to the first edition as **[CLR]** in these slides. I'll usually reference both books, but in general it is your responsibility to find the right sections in **[CLR]** (the numbering has changed).*

Pre-requisites

Official pre-requisites:

- ▶ Passes in Inf2B & Maths-for-Inf-3/4 (or year 2 Hons Maths).

Un-official recommendation:

- ▶ Should have 50% at least at **first attempt** in Inf2B and at **first attempt** for all your 2nd year Maths courses.
- ▶ If one mark is less than 50%, would expect others to closer to 60%.
- ▶ If you had to resit one, I would expect 60% or more.
- ▶ At the end, it is your call (talk to me if you're not sure).

If you didn't take Inf2B, but have *excellent* Maths, talk to me . . .

this course is quite mathematical. I'll rely on some things that you've done in maths classes.

Other References

- ▶ Kleinberg and Tardos: *Algorithm Design*. Addison-Wesley, 2005. (Nice book - but doesn't cover many of our topics).
- ▶ Sedgewick: *Algorithms in C (Part 1-5)*, Addison Wesley, 2001.

Course Webpage
(with transparencies and lecture log)

<http://www.inf.ed.ac.uk/teaching/courses/ads/>

Math Pre-requisites

You should know:

- ▶ how to multiply matrices or polynomials,
- ▶ some probability theory,
- ▶ some graph theory,
- ▶ **what it means to prove a theorem** (induction, proof by contradiction, . . .) and to be **confident** in your ability to do this.

The appendices of [CLRS] might be useful for reviewing your math.

Coursework

- ▶ A set of problems each week via a Tutorial sheet. It is very important that you attempt these BEFORE tutorials! Preparing for tutorials will make a huge difference in what you get out of the tutorial - *it will massively improve your final grade for the course.*
- ▶ 2 Assessed Courseworks due (on Friday) in weeks 5 and 9, respectively (will be marked). In total worth 25% of the course mark.
- ▶ Also ... it's a good idea to try coding-up a few of the algorithms :)

Analysis of algorithms

Determine the amount of some **resource** required by an algorithm (usually depending on the **size** of the input).

The resource might be:

- ▶ *running time*
- ▶ memory usage (*space*)
- ▶ network traffic
- ▶ number of accesses to secondary storage
- ▶ number of *basic arithmetic operations* such as $+$, $-$, $*$, $/$.

Basic Notions

Model of Computation: An abstract sequential computer, called a *Random Access Machine* or *RAM*. Uniform cost model.

Computational Problem: A specification in general terms of *inputs* and *outputs* and the desired input/output relationship.

Problem Instance: A particular collection of inputs for a given problem.

Algorithm: A method of solving a problem which can be implemented on a computer. Usually there are many algorithms for a given problem.

Program: Particular implementation of some algorithm.

Running time and number of basic operations

- ▶ Formally, we define the **running time** of an algorithm on a particular input instance to be the *number of computation steps performed by the algorithm on this instance.*
- ▶ This depends on our *machine model* - need the algorithm to be written as a **program** for such a machine.
- ▶ **number of basic arithmetic operations** - abstract way of only counting the *essential* computation steps.
- ▶ Both notions are abstractions of the *actual running time*, which also depends on factors like
 - ▶ Quality of the implementation
 - ▶ Quality of the code generated by the compiler
 - ▶ The machine used to execute the program.

Worst-Case Running Time

Assign a *size* to each possible input (this will be proportional to the length of the input, in some *reasonable* encoding).

Definition

The (*worst-case*) *running time* of an algorithm A is the function $T_A : \mathbb{N} \rightarrow \mathbb{N}$ where $T_A(n)$ is the maximum number of computation steps performed by A on an input of size n .

- ▶ A similar definition applies to other measures of resource.

Bounds

Given a *problem*, a function $T(n)$ is an:

Upper Bound: If there is an algorithm which solves the problem and has worst-case running time at most $T(n)$.

Average-case bound: If there is an algorithm which solves the problem and has average-case running time at most $T(n)$.

Lower Bound: If *every* algorithm which solves the problem must use at least $T(n)$ time on some instance of size n for infinitely many n .

Average-Case Running Time

Definition

The *average-case running time* of an algorithm A is the function $AVT_A : \mathbb{N} \rightarrow \mathbb{N}$ where $AVT_A(n)$ is the *average* number of computation steps performed by A on an input of size n .

For a genuine average-case analysis we need to know for each n the probability with which each input turns up. Usually we assume that all inputs of size n are equally likely.

A little thought goes a long way

Problem: Remainder of a power.

Input: Integers a, n, m with $n \geq 1, m > 1$.

Output: The remainder of a^n divided by m , i.e., $a^n \bmod m$.

Algorithm POWER-REM1(a, n, m)

1. $r \leftarrow a$
2. **for** $j \leftarrow 2$ **to** n **do**
3. $r \leftarrow r \cdot a$
4. **return** $r \bmod m$

- ▶ Real world: integer overflow even for small a, m and moderate n .
- ▶ Even without overflow numbers become needlessly large.

A little thought ... (cont'd)

Algorithm POWER-REM2(a, n, m)

1. $x \leftarrow a \bmod m$
2. $r \leftarrow x$
3. **for** $j \leftarrow 2$ **to** n **do**
4. $r \leftarrow r \cdot x \bmod m$
5. **return** r

Much better than POWER-REM1.

- ▶ No integer overflow (unless m large).
- ▶ Arithmetic more efficient — numbers kept small.

Reading Assignment

[CLRS] Chapters 1 and 2.1-2.2 (pp. 1–27) (all this material should be familiar from Inf2B).

If you did not take Inf2B ... read **all of the ADS part of Inf-2B**.

Problems

1. Analyse the asymptotic worst-case running time of the three POWER-REM algorithms.
Hint: The worst-case running time of POWER-REM1 and POWER-REM2 is $\Theta(n)$, and the worst-case running time of POWER-REM3 is $\Theta(\lg n)$
2. Exercise 1.2-2, p. 13 of [CLRS] (*Ex 1.4-1, p 17 in [CLR]*).
3. Exercise 1.2-3, p. 13 of [CLRS] (*Ex 1.4-2, p 17 in [CLR]*).

A little thought ... (cont'd)

Algorithm POWER-REM3(a, n, m)

1. **if** $n = 1$ **then**
2. **return** $a \bmod m$
3. **else if** n even **then**
4. $r \leftarrow \text{POWER-REM3}(a, n/2, m)$
5. **return** $r^2 \bmod m$
6. **else**
7. $r \leftarrow \text{POWER-REM3}(a, (n-1)/2, m)$
8. **return** $(r^2 \bmod m) \cdot a \bmod m$

Even better.

- ▶ No integer overflow (unless a, m large), nums kept small.
- ▶ Number of arithmetic operations even less.