Designing Inf2-IADS

John Longley

May 9, 2022

IADS (Introduction to Algorithms and Data Structures):

- Core Year 2 course, both semesters.
- Designed and taught by Mary Cryan and John Longley.
- Run in 2019-20 (in-person), 20-21 (online), 21-22 (hybrid).

This talk: Mostly specific to IADS, but will highlight points that may be more widely applicable.

Remit (arising from curriculum re-vamp)

- Topics to cover (many from previous courses):
  - Basic asymptotics, searching/sorting, classic data structures (Inf2B)
  - Graphs and graph algorithms (DMMR)
  - Dynamic programming (touched on in Inf2A)
  - Brief look at language processing/parsing (Inf2A)
  - Brief look at P vs. NP (new for Year 2) and computability (Inf2A)
- Integrate theory and practice:
  - Students to implement (some of) the algorithms
  - Evaluate them empirically as well as theoretically
  - Selecting suitable algs/DSs for a given purpose
  - Native data structures in typical programming languages (e.g. how do Python lists work 'under the hood'?)
- Introduce Python (needed for Year 3).

Main course components

- Lectures, tutorials: Traditional style, but making frequent contact with practical programming issues.
- Python lab sheets: side activity in selected weeks, introducing Python and reinforcing lecture material by small practical exercises (e.g. algorithm implementations and timing experiments). Self-study but supported by optional lab sessions.
- Courseworks (2 practical, 1 pen-and-paper): carefully designed to cover several topics each.

Course content: selected points

Asymptotics

At the core of the course is the following Gang of Five:

$$o \quad O \quad \Theta \quad \Omega \quad \omega$$

(E.g. $n \lg n = o(n^2)$.)

Can all seem very theoretical. To motivate it, we start with some examples of efficient and inefficient algorithms, illustrating the runtime growth rates with actual numbers.

- Concrete examples before general definitions seem to go down well (at both macro and micro levels).
- Front-loading a course with the hard concepts allows max time for them to be assimilated.
- Levels of partial understanding acknowledged and affirmed.

Data structures

- Start at the bottom with organization of program data in memory. (Stack and heap, fixed-size vs. extensible arrays, linked lists, reference vs. content equality.) Important piece of understanding that sometimes 'fell between courses' in past.
- Discuss runtime characteristics of native operations (e.g. 'append' for Python lists).
- Then show how classic data structures can be implemented on top of this: stacks, queues, hash tables, balanced trees, max heaps.

Graph algorithms

Fairly standard: BFS and DFS, topological sorting, Dijkstra's shortest path algorithm.
Shows what earlier material can be used for (queues, stacks, max heaps).

Dynamic programming

General technique illustrated by examples of increasing complexity (Fast Fibonacci function, coin changing, seam carving, edit distance, Viterbi algorithm.)

Language processing

3 lectures: Context-free grammars, CYK parsing (general, $\Theta(n^3)$ time – another DP algorithm), LL(1) parsing (more special, $\Theta(n)$ time).

Complexity and computability

Brief glimpse of these topics ('what should all students have seen at least once?').
Trajectory of course:

- Gradually increasing runtimes: $\Theta(n^2)$ bad in Sem 1, good in Sem 2.
- From specific/concrete to general/abstract.

So exploring the *limits* of what algorithms can do (feasibly, or in principle) serves as a natural endpoint.
Many students found these later topics very interesting.

Courseworks

Each coursework tries to tick several boxes, and to show how different topics (may) connect up. Details vary each year, but outlines have remained largely the same:

1. Python practical: Search engine for a corpus of large textfiles. Involves:
   - file-based mergesort (for compiling index)
   - priority queues (for efficient searches involving multiple terms)
   - state-of-the-art perfect hashing method (to reduce size of meta-index)

   [If you put a lot of work into a coursework and think it has high educational value, seriously consider making it credit-bearing!]

2. Pen-and-paper exercises. Two halves: garbage collection, knapsack problem. Involves:
    - Organization of program memory (stack/heap)
    - BFS graph traversal
    - Asymptotics, including amortized analysis (under various conditions, what proportion of total execution time is spend on GC?)
    - GC itself as a topic worth learning about
    - Greedy algorithms: when they do and don't work
    - Dynamic programming
3. Python practical: Travelling Salesman Problem. After doing a few simple exercises, they get to research, implement and evaluate any TSP algorithm of their choice.
    - Heuristic approaches to hard problems
    - Empirical evaluation
    - Scope for creativity!

Reception by students

- Many students say they find the course interesting and rewarding, but challenging (the theoretical concepts are very new to them).
- They like the courseworks (high learning value, and the fact they're all different). Still some concerns over their length.
- Especially popular in 2020-21: exceptional 'buzz' and rapport with students.
- Motivation enhanced by music and songs in video lectures.

Final point

If designing a course, great if you can do it with someone else. Thank you Mary!