Welcome

Raveen de Silva (K17 202)
Course Admins: cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2024

Table of Contents

Why should you study algorithms?

- This course is about *problem solving* and *communication*.
- Two pillars:
  - *algorithm design* is the task of designing a process which when executed solves a particular problem;
    - This process might be a way of calculating something, selecting something, etc.
  - *algorithm analysis* is the task of assessing such a process (either your own work or someone else's) with regards to the problem.
    - Does it always give the intended result?
    - How quickly does it run?

Algorithm analysis: why is correctness important?

- A program which fails a test case is wrong, but a program which passes a test case isn't *necessarily* correct.
- Failure on an obscure case can be catastrophic!
- Testing is good, good testing is better, proof is best.
- What makes a strong test?
- How do you prove (or at least justify *in general terms*) correctness and explain it to a peer?

Algorithm analysis: why is efficiency important?

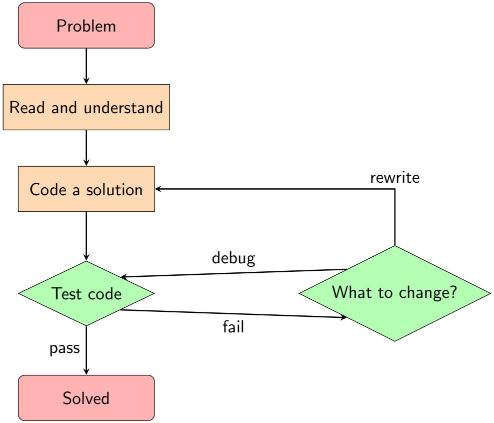
- Using better hardware might save a small (or even large) percentage of the running time of a program.
- However a more efficient algorithm can make staggering improvements!
- Sorting one million numbers could take an hour using bubble sort, or a second using merge sort.
- For more sophisticated algorithms, details such as the choice of data structures are also important.

Why is algorithm design important?

- Algorithm design is the ‘planning stage’, where we decide what our program should do.
- We have all tried to code programs only to find a fundamental flaw in our approach later on.
- A systematic approach to algorithm design separates engineers from “code monkeys”.

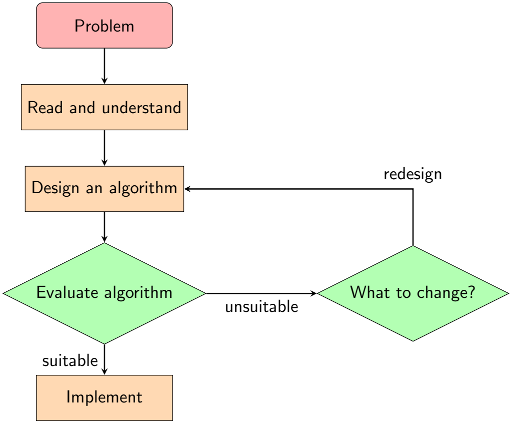
Code-first approach

Solving the problem as you code can lead you into a “debugging vortex”, applying patches without fixing the underlying issue, or to accept incorrect code that passes weak tests.



Design-first approach

Separating design from implementation allows high-level reasoning to confirm that the algorithm meets the requirements of the problem before making implementation decisions.



Separating design from implementation

- We will talk about algorithms in *plain English*, instead of pseudocode or especially code in any programming language.
- Implementation is outside the scope of this course.
- We'll focus on the big picture, to prepare you to:
    - work in teams on large projects
    - communicate clearly and persuasively with stakeholders including colleagues, clients and the public
    - compile design briefs and documentation,

  which are all important skills for the modern computing professional.

Course Learning Outcomes

1. Explain how standard design techniques are used to develop algorithms
2. Solve problems by creatively applying algorithm design techniques
3. Communicate algorithmic ideas at different abstraction levels
4. Evaluate the efficiency of algorithms and justify their correctness
5. Apply the LaTeX typesetting system to produce high-quality technical documents

What is an algorithm?

- An algorithm is a collection of precisely defined steps that can be executed *mechanically*, i.e. without intelligent decision-making.
- Designing a recipe involves creativity, executing it does not; the same is true of algorithms.
- The word "algorithm" comes by corruption of the name of Muhammad ibn Musa al-Khwarizmi, a Persian scientist 780–850, who wrote an important book on algebra, *"Al-kitab al-mukhtasar fi hisab al-gabr wa'l-muqabala"*.

Scope

- In this course we will deal only with algorithms that are:
  - *sequential*: comprised by a sequence of steps, which can only be executed one at a time, and
  - *deterministic*: each step always gives the same result for the same input.
- Parallel algorithms and randomised algorithms are interesting and powerful, but beyond the scope of this course.

Problem structure

- Problems may have ‘flavour text’: a brief story for context. Abstracting this information away is a key first step.

- Algorithm design tasks may also include an input and output specification if it is not otherwise clear.

- In this course, we are usually not concerned with the format in which this information is presented, nor with the minute details of how an implementation of the algorithm accesses input or produces output.

- However it is crucially important that our algorithm *always* transforms valid input into the appropriate output *efficiently*.

Problem structure

- We'll usually measure the efficiency of algorithms using *asymptotic analysis*, rather than the exact time or number of operations taken.
- You will have already encountered big-Oh notation.
    - For example, a process consisting of two nested loops "for each i from 1 to n, for each j from 1 to n" takes $O(n^2)$ time.
    - We'll formalise these ideas in the next lecture.
- We'll usually use the *unit-cost* model of computation, where each machine operation ($+$, $-$, $\times$, $\div$ and accessing memory) takes constant time.
    - The only exception is when we discuss algorithms to multiply large integers in Module 2: there we'll use the *logarithmic-cost* model, where these operations take time proportional to the width (number of bits) of the input numbers.

Example problem: Geese in a Trenchcoat

Problem

There are n geese in a gaggle, of heights $h_1, \ldots, h_n$. Each h_i is a positive integer.

They need to select two of them to wear a trenchcoat together, with one standing on top of the other. The trenchcoat has a fixed size H, so it will only fit if the two geese have total height exactly equal to H.

Design an algorithm which runs in $O(n \log n)$ time and determines whether there is a pair of geese who can wear the trenchcoat together.

Example problem: Geese in a Trenchcoat

Abstraction

Design an algorithm which runs in $O(n \log n)$ time and, given an array A consisting of n positive integers and a positive integer X, determines whether there are two indices $i < j$ such that $A[i] + A[j] = X$.

- If we fix index i (one of the geese), then we know that a matching element must contribute the rest of the total (make up the rest of the required height).
- Let's try all values of i.

Example problem: Geese in a Trenchcoat

Attempt 1

For each goose i, check whether a matching goose (whose height is $H - h_i$) exists, and answer yes if one is found.

- This is not an algorithm!
- It does not describe how to actually perform this check.
- Why does this matter?
    - For a small number of geese, a human could follow this process, checking by eye. If there were say one million geese, it's not at all clear how to do the check.
    - Perhaps this check takes time?

Example problem: Geese in a Trenchcoat

Attempt 2

For each goose i, a matching goose must have height $H - h_i$. For each other goose j, check whether h_j equals this quantity, and answer yes if one is found.

- This is actually an algorithm.
- It is also a correct algorithm.
  - Does it correctly answer all 'yes' instances?
  - Does it correctly answer all 'no' instances?
- Unfortunately it's too slow: "for each i (of which there are n), for each j (of which there are n), do something" is already n^2 iterations.
- Can we search efficiently for the correct j, rather than trying them all? Perhaps if the heights are well organised.

Example problem: Geese in a Trenchcoat

Algorithm

Sort the geese in ascending order of height, using merge sort. For each goose i, a matching goose must have height $H - h_i$. Since the heights are now nondecreasing, we can search for this required length using binary search, and answer yes if it is found.

- Is this algorithm correct?
  - Does it correctly answer all 'yes' instances?
  - Does it correctly answer all 'no' instances?
- What is the time complexity of this algorithm?
  - How long does the sort take?
  - How many times is the binary search run?
  - How long does each binary search take?
- You will usually have to justify that your algorithm is correct and that it runs within the specified time complexity.

Table of Contents

Modules

1. Foundations (week 1)
    - Foundations of algorithm design (data structures, sorting and searching)
    - Foundations of algorithm analysis (correctness, time complexity)
2. Divide and conquer (week 2)
    - Recursion with *independent* subproblems
    - Applications: big integer multiplication, signal processing
3. Greedy algorithms (weeks 3–4)
    - When local considerations align with global
    - Applications: file compression, graphs

Modules

4. Flow networks (week 5)
   - Routing ‘flow’ through a graph
   - Applications: allocation and matching problems
5. Dynamic programming (weeks 7–8)
   - Recursion with *overlapping* subproblems
   - Applications: graphs, string matching
6. Intractable problems (weeks 9–10)
   - Recognising, analysing and relating ‘hard’ problems

Textbooks

Recommended textbook

Kleinberg and Tardos: *Algorithm Design*
paperback edition available at UNSW Bookshop

- excellent: very readable textbook (and very pleasant to read!);
- not so good: as a reference manual for later use.

An alternative textbook

Cormen, Leiserson, Rivest and Stein: *Introduction to Algorithms*
4th edition now available at UNSW Bookshop, 3rd edition also useful

- excellent: to be used later as a reference manual;
- not so good: somewhat formalistic and dry.

Prerequisites

- COMP2521 or COMP9024
  - Fundamental data structures
  - Fundamental algorithms
- MATH1081 or COMP9020
  - Proofs
  - Written communication skills

Extended courses

- The extended courses COMP3821/9801 run in T1 only
- Differences in content and assessment
- Can transfer until at least week 2, and until census date with approval

Courses to consider

- COMP4121 Advanced Algorithms
  - Real-world algorithms, including randomisation (and therefore probability and statistics)
- COMP4128 Programming Challenges
  - Algorithms and C++ implementation for small, self-contained problems, as in contests.
- COMP2111 System Modelling and Design, COMP3153 Algorithmic Verification
  - Formal and rigorous methods for verifying software
- COMP4141 Theory of Computation, COMP6741 Algorithms for Intractable Problems
  - More on intractable problems, from theoretical and practical perspectives.
- Various MATH courses also build on the content covered in this course.

Table of Contents

Portfolio

- Tasks available on Formatif
- In each task, incorporate tutor's feedback until task complete
- Contract grading
  - Choose the grade you are aiming for (PS/CR/DN/HD)
  - To achieve this grade, complete all tasks of this level *and below*
  - DN threshold is 80, not 75
  - Outstanding performance in HD tasks required for top marks
- Weighted 60% of course mark

Formatif

- Demo video:
  https://www.youtube.com/watch?v=KH_855VU0MQ
- Task types: (D), (M), (R), (L), (C)
- Task sheets on Formatif
- Task resources on Moodle, suitable for import to Overleaf

Final Exam

- Multiple choice, short answer and algorithm design problems
- Easier than comparable portfolio tasks
- Hurdle: 40% required to pass the course
- INSPERA (on campus): more info here
- Weighted 40% of course mark

Bonus Marks

- Reward *contributions to other students' learning*, in classes and on the forum
- About 5% of students will get at least 1 mark, half as many for each extra mark up to a maximum of 5

Lectures

- Weeks 1–5, 7–10 (no lectures in flex week)
- Face to face
  - WEB stream students are welcome to attend
- Recording on Echo360
- Slides on Moodle

Consultation

- Mon and Tue 14:00 – 15:00 at K17 202, weeks 1–9
- Fri 14:00 – 15:00 on MS Teams, weeks 1–8
- Weeks 9 and 10, exam consultation: TBA

Tutorials

- Weeks 1–5, 7–10 (no tutorials in flex week)
- Two unofficial streams:
  - Tortoise (M14A, M18A, T12B): revise recent lecture content and solve one problem
  - Hare (M16A, T10A, T13A, T18A): solve two or more problems assuming familiarity with lecture content
- Attend any, provided there is space in the room; M18A and T18A are online
- Prepare by reading the tutorial sheet before attending

‘Labs’

- Weeks 1–5, 7–10 (no labs in flex week)
- Free time to:
  - work on new Formatif tasks
  - clarify feedback on existing tasks
  - get help from the lab demonstrators
  - get marked off for completed tasks

Support

- In addition to classes, we provide the following support services and resources:
  - Ed forum
  - Drop-in sessions at K17 G05:
    - Mon 15:00 – 16:00
    - Tue 11:00 – 12:00
    - Wed 11:00 – 12:00
    - Fri 11:00 – 12:00
    - Fri 15:00 – 16:00

Equitable Learning Services

- Equitable Learning Services is a free and confidential service that provides practical support to ensure that your health condition doesn't adversely affect your studies.

- If you have an Equitable Learning Plan, please provide it to us by email within the first two weeks of the term.

Student welfare resources

How We Help You at UNSW

General Health and Wellbeing	Educational Adjustments
Feelings and Mental Health	Academic and Study Skills
Sexual Health & Relationships	Special Consideration
Uni and Life Pressures	Aboriginal & Torres Strait Islander Community
Reporting Harassment and Gendered Violence	Legal & Advocacy Services

Feedback

- Changes from last term:
  - scheduled labs
  - assignments replaced by portfolio assessment
- Feedback is always welcome, e.g.
  - myExperience survey
  - feedback post on Ed (can post anonymously)
  - email

Table of Contents

Puzzle

Problem

Tom and his wife Mary went to a party where nine more couples were present.

Not everyone knew everyone else, so people who did not know each other introduced themselves and shook hands. People who knew each other from before did not shake hands.

Later that evening Tom got bored, so he walked around and asked all other guests (including his wife) how many hands they had shaken that evening, and got 19 different answers.

- How many hands did Mary shake?
- How many hands did Tom shake?

That's All, Folks!!