COMP20007 Design of Algorithms

Introduction and Welcome

Lars Kulik

Lecture 1

Semester 1, 2023

Welcome to COMP20007

- Data structures, including stacks, queues, trees, priority queues and graphs.
- Algorithms for various problems, including sorting, searching, string manipulation, graph manipulation, and more.
- Algorithmic techniques, including including brute force, decrease-and-conquer, divide-and-conquer, dynamic programming and greedy approaches.
- Analytical and empirical assessment of algorithms.
- Complexity classes.

Anany Levitin. *Introduction to the Design and Analysis of Algorithms* Pearson, 2012.
Steven S Skiena. *The Algorithm Design Manual* Springer, 2008.

Staff; Learning Management System

Lecturer and subject coordinators: Daniel Beck and Lars Kulik

Head tutor: Grady Fitzpatrick

Grady will have a weekly time for consultation. Exact time (and later venue) to be announced.

Other support is provided by your classmates, for example via the Ed Discussion Board.

The LMS is our notice board, repository, and discussion forum. Please note though that we will be mostly using Ed for all content. The only exception are lecture recordings, which we will make available via the standard LMS as usual.

Subject Delivery

We will offer all lectures face to face.

We use lecture capture which is useful for revisiting points from a lecture or provide pre-recorded lectures. We also provide handouts in PDF format for every lecture.

Please note that all our workshops are on campus!

All workshops start in Week 2.

Time Commitment

For the 12 weeks of semester, expect

- 22 hours of lectures,
- 22 hours of workshops,
- 36 hours of reading and workshop preparation,
- 48 hours on assignments.

That is roughly an average of 10-12 hours per week.

The commitment is well worth it: Knowledge of algorithms is essential for any computing professional, it expands your mind, improves complexion, and contains all the minerals and vitamins essential for developing boundless wisdom.

Assessment

- ✓ Assignment 1, due around Week 5-6, worth 10%.
- Mid-semester test in Week 5-6, worth 10%.
- ✓ Assignment 2, due around 12, worth 20%.
- 3-hour exam, worth 60%.

To pass the subject you must obtain at least

- 50% in assignments (total $\geq 15/30$); and
- 50% in the mid-semester test and in the exam (total $\geq 35/70$).

Expectations

You need to catch up on any "assumed background knowledge" that you may not have:

- An understanding of sets and relations.
- A grasp of recursion and recurrence relations; a short tutorial on the latter is in Levitin's book, Appendix B.
- Knowledge of basic data structures, such as arrays, records, linked lists, sets and dictionaries.
- Knowledge of some programming language that has a concept of "pointer".

C++/C, Rust

How to Succeed

Understand the material, don't just memorize it (apart, perhaps, from the formulas in Levitin's Appendix A).

If you fall behind, try to catch up as fast as possible.

Don't procrastinate. Start assignments before you are ready. Put in the necessary time.

Attempt the workshop questions every week, before you attend the tutorial, if at all possible.

How to Succeed

Support the learning of your fellow students and expect their support, in class and through the Ed discussion board.

Remember that we are all on the same “learning journey” and have the same goal.

Participate in the discussions on the subject’s Ed site and check regularly for announcements.

Over to You—A Maze Problem

A maze (or labyrinth) is contained in a 10×10 rectangle; rows and columns are numbered from 1 to 10.

It can be traversed along rows and columns: up, down, left, right.

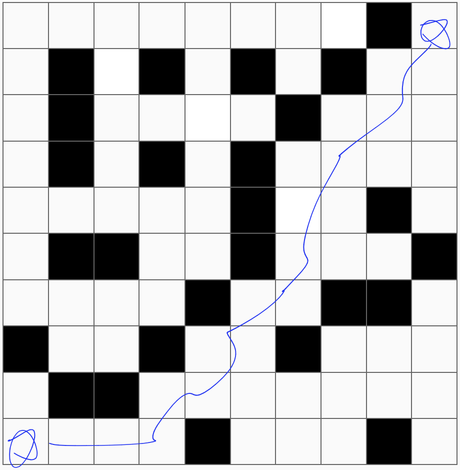
The starting point is (1,1), the goal point is (10,10).

These points are obstacles that you cannot travel through:

(3,2)	(6,6)	(2,8)	(5,9)	(8,4)	(2,4)	(6,3)	(9,3)
(1,9)	(3,7)	(4,2)	(7,8)	(2,2)	(4,5)	(5,6)	(10,5)
(6,2)	(6,10)	(7,5)	(7,9)	(8,1)	(5,7)	(4,4)	(8,7)
(9,2)	(10,9)	(2,6)					

Find a path through the maze.

The Maze Problem ...



What Is a Problem?

My ODE (Oxford Dictionary of English) says: "doubtful or difficult question or task."

In computer science we use the term like that too, but there is a more technical concept of algorithmic problem.

We usually want to find a single generic solution to a bunch of similar questions.

For example, the "maze problem" is to come up with a mechanical solution to any particular maze.

So to us, a "problem" usually has many instances, sometimes infinitely many.

Algorithmic Problems

So a problem in computer science typically means a family of instances of a general problem.

An algorithm for the problem has to work for all possible instances (input).

Example: The sorting problem—an instance is a sequence of items.

can be normalized to UCS problem.

Example: The graph colouring problem—an instance is a graph.

reduce to search problem again.

Example: Equation solving problems—an instance is a set of, say, linear equations.

What Is an Algorithm?

My ODE says: "process or rules for (esp. machine) calculation etc."

A finite sequence of instructions

- No ambiguity, and each step precisely defined
- Should work for all (well-formed) input
- Should finish in a finite (reasonable) amount of time

The (single) description of a process that will transform arbitrary input to the correct output—even when there are infinitely many possible inputs.

What Is an Algorithm?

同义

Not long ago, "algorithm" was synonymous with "numeric algorithm".

Mathematicians had found many clever algorithms for all sorts of numeric problems.

The following algorithm for calculating the greatest common divisor of positive integers m and n is known as "Euclid's Algorithm".

To find $\boxed{gcd(m, n)}$:

$$gcd(n.m) = \begin{cases} m, & n = 0 \\ gcd(m\%n.n) & \end{cases}$$

Step 1: If $n = 0$, return the value of m as the answer and stop.
Step 2: Divide m by n and assign the value of the remainder to r.
Step 3: Assign the value of n to m, and the value of r to n; go to Step 1.

Non-Numeric Algorithms

350 years ago, Thomas Hobbes, in discussing the possibility of automated reasoning, wrote:

> *"We must not think that computations, that is, ratiocination, has place only in numbers."*

Today, numeric algorithms are just a small part of the syllabus in an algorithms course.

The kind of computation that Hobbes was really after was 逻辑形式化 mechanised reasoning, that is, algorithms for logical formalisms, for example, to decide "does this formula follow from that?"

Computability

In 2012 we celebrated Alan M. Turing's 100th birthday.

At the time of Turing's birth, a "computer" was a human employed to do tedious numerical calculations.

Legacy: "Turing machine", the "Church-Turing thesis", "Turing reduction", the "Turing test", the "Turing award"

One of Turing's great accomplishments was to put the concept of an algorithm on a firm foundation and to establish that certain important problems do not have algorithmic solutions.

Abstract Complexity

In a course like this, we are only interested in problems that do have algorithmic solutions.

However, amongst those, there are many that probably ~~provably~~ do not have efficient solutions.

Towards the end of this subject we discuss complexity theory briefly—this theory is concerned with the inherent "hardness" of problems.

Why Study Algorithms?

Computer science is increasingly an enabler for other disciplines, providing useful tools for these.

Algorithmic thinking is relevant in the life sciences, in engineering, in linguistics, in chemistry, etc.

Today computers allow us to solve problems whose size and complexity is vastly greater than what could be done a century ago.

The use of computers has changed the focus of algorithmic study completely, because algorithms that work well for a human (small scale) usually do not work well for a computer (big scale).

Why Study Algorithms and Their Complexity?

To collect a number of useful problem solving tools.

To learn, from examples, strategies for solving computational problems.

To be able to write robust programs whose behaviour we can reason about.

To develop analytical skills.

To learn about the inherent difficulty of some types of problems.

Problem Solving Steps

1. Understand the problem
2. Decide on the computational means (sequential/parallel, exact/approximate)
3. Decide on method to use (algorithm design technique or strategy, use of randomization) genetic, simulation annealing
4. Design the necessary data structures and algorithm
5. Check for correctness, trace example input
6. Evaluate analytically (time, space, worst case, average case)
7. Code it
8. Evaluate empirically pressure test

What we will study

Algorithm analysis

Important algorithms for various problems, primarily

- ✓ Sorting
- ✓ Searching
- String processing
- ✓ Graph algorithms

Approaches to algorithm design

- Brute force
- Decrease and conquer Recursion
- Divide and conquer
- Transform and conquer

Study Tips

Before the lecture, as a minimum make sure you have read the introductory section of the relevant chapter.

Always read (and work) with paper and pencil ready; run algorithms by hand.

Always have a go at the tutorial exercises; this subject is very much about learning-by-doing.

After the lecture, reread and consolidate your notes.

Identify areas not understood and use the Ed Discussion Forum.

Rewrite your notes if that helps.

Things to Do in the First Two Weeks

Read the text, read Chapter 1, and skim Chapter 2.

Make sure you have a unimelb account.

Visit the COMP20007 LMS (Ed) pages and check any announcements.

Use the Ed Discussion Board; for example, if you are interested in forming a study group with like-minded people, the Discussion Board is a useful place to say so.

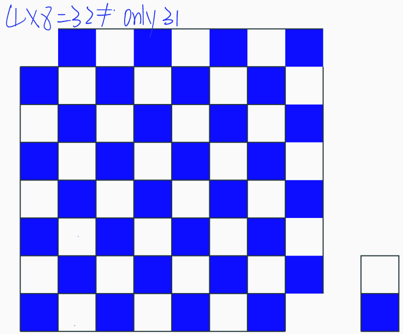
Transform and Conquer? Use Abstraction?

Can we cover this board with 31 tiles of the form shown?

Why can we quickly determine that the answer is no?

Hint: Using the way the squares are coloured helps.

✎



Next Up

Algorithm analysis - how to reason about an algorithm's resource consumption.

Other Data Structures

We shall meet many other (abstract) data structures, such as

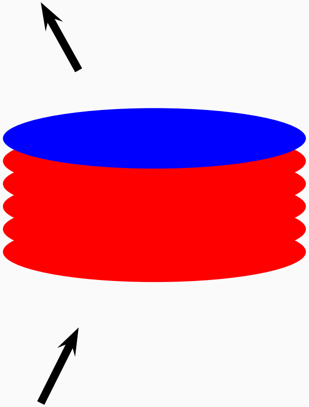
- The priority queue
- Various types of “tree”
- Various types of “graph”

Fundamental Data Structures: The Queue

First-in-first-out (FIFO).

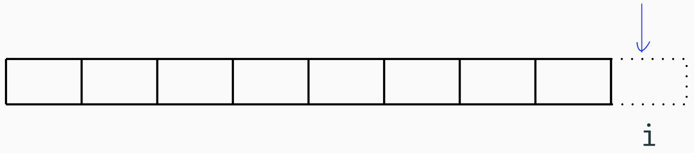
Operations:

- CreateQueue
- Enqueue
- Dequeue
- Head
- EmptyQueue?
- ...

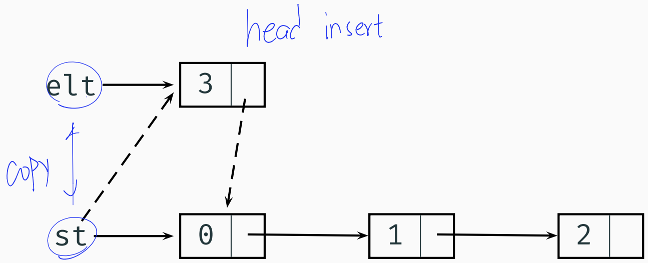


Stack Implementation

By array:



By linked list (push):



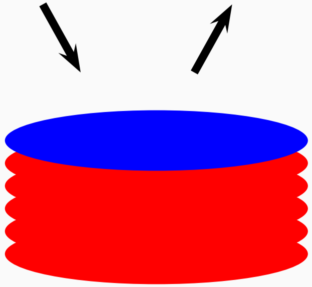
Fundamental Data Structures: The Stack

Last-in-first-out (LIFO).

Operations:

- CreateStack
- Push
- Pop
- Top
- EmptyStack?
- ...

Usually implemented as an ADT.



Abstract Data Types

A collection of data items, and a family of operations that operate on that data.

Think of an ADT as a set of promises, or contracts.

We must still implement these promises, but it is an advantage to separate the implementation of the ADT from the “concept”.

Good programming practice is to support this separation: Nothing outside of the definitions of the ADT should refer to anything inside, except through function calls for the basic operations.

Recursive Processing

Solve the problem on a smaller collection and use that solution to solve on the full collection.

```
function find(A,x,lo,hi)
  if lo > hi
    return null
  else if A[lo] == x
    return lo
  else
    return find(A,x,lo+1,hi)
```

Initial call: `find(A,x,0,last)`

```
function find(p,x):
  if p == null
    return p
  else if p.val == x
    return p
  else
    return find(p.next,x
```

Initial call: `find(head,x)`

We return to recursion in more depth later.

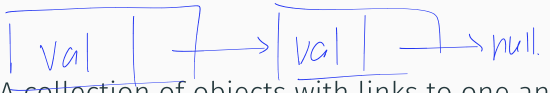
Iterative Processing

Walk through the array or linked list. For example, to locate an item.

```
j := 0
while j < last
  if A[j] == x
    return j        add    subprifex
  j := j+1
return null
```

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next   go to next
                node
return null
```

Primitive Data Structures: The Linked List



A collection of objects with links to one another, possibly in different parts of the computer's memory.

Often we use a dummy head node that points to the first object, or to a special `null` object that represents an empty list.

Inserting and deleting elements is very fast: just move a few links around. $O(1)$

Finding the *i*th element can be time-consuming. $O(n)$

Primitive Data Structures: The Array

An array consists of a sequence of consecutive cells in memory.

Depending on programming language: C/C++. python. java `A[0]` up to `A[n-1]`, or `A[1]` up to `A[n]`. Ruby, Julia.

Locating a cell, and storing or retrieving data at that cell is very fast. $O(1)$

The downside of an array is that maintaining a contiguous bank of cells with information can be difficult and time-consuming.

Exercise: Data Structures

Pick a data structure and describe:

- How to insert an item into the data structure
- How to find an item
- How to handle duplicate items

Algorithms and Data Structures

Algorithms: for solving problems, transforming data.

Data structures: for storing data; arranging data in a way that suits an algorithm.

- Linear data structures: Stacks and queues
- Trees and graphs
- Dictionaries $\Leftrightarrow$ hash tables.

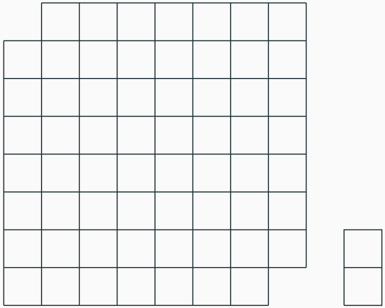
Which data structures are you familiar with?

Approaching a Problem

Can we cover this board with 31 tiles of the form shown?

This is the mutilated checkerboard problem.

There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very $O(31!)$ inefficient) way of solving the problem.



COMP20007 Design of Algorithms

Design of Algorithms

Lars Kulik

Lecture 2

Semester 1, 2023