Algorithms (COMP3600/6466)
Complexity Theory

Pascal Bercher

(working in the Intelligent Systems Cluster)

School of Computing
The Australian National University

Tuesday & Wednesday, 24. & 25.10.2023

Australian National University

Introduction

Outline of this Week

- We investigate the computational hardness of *decision problems*:
  - What's the "performance" of the best-known algorithm for solving the respective problem?
  - Which problems are equally hard?
  - Which ones are harder than others?
  - We look at how problems can be "turned into each other".
- We measure "performance" in terms of a Turing Machine's:
  - Time requirement (number of operations/transitions)
  - Space requirement (number of cells that can be read/written)

Motivation

Why would we do that? Given a new problem to solve, we:

- . . . know performance bounds for our yet-to-be-designed solver
  - → *No need to look for an "efficient" algorithm if not a single genius so far was able to do that! (Well, don't let that stop you necessarily, see last point!) But it makes a great excuse. :)*
- . . . can use existing solvers instead of designing new ones,
  - → *Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?*
- . . . understand the problems we solve much better.
  - → *If you know that your (new) problem is equivalent to an existing (established) one, that surely helps... (Imagine, you take a course twice! The second time it's much easier...)*

Motivation, cont'd

Some other reasons:

- It's a lot of fun! (It's basically "solving puzzles".)
  Extremely simple reduction:
  - Assume you have an algorithm that checks whether a number is *even*, even(n).
  - But you want to know whether a number is odd... Will you devise a new algorithm?

  → No! Turn your problem into the existing one! Define odd(n) := even($n+1$).
- You can get famous by solving open problems! (E.g., $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.)
  Seriously: *somebody* will be the first... Could be you!
  Interested? Take COMP3630, Theory of Computation.
  (You find all slides on Pascal's webpage.)

Basics

Recap: What do we recap?

Everybody should have taken COMP1600 and thus know already:

- Deterministic Finite Automata (DFAs) and its extension to Non-deterministic Finite Automata (NFAs).
  → We won't use them here, but it won't hurt re-familiarizing yourself with them since Turing Machines can be regarded an extension.
- Formal Languages and language of an automaton (or grammar).
  → We directly build on them, and thus recap them here.
- Turing Machines
  → We directly build on them, and thus recap them here. Slides are mostly taken from COMP3630, so definitions might slightly differ from COMP1600 (but are semantically equivalent).

Recap: Formal languages

- *Alphabet* Σ: A finite set of *symbols*, e.g.,
  - $\Sigma = \{0, 1\}$ (*binary* alphabet)
  - $\Sigma = \{a, b, \ldots, z\}$ (*Roman* alphabet)
- A *String* (or *word*) is a finite sequence of symbols. Strings are usually represented without commas, e.g., 0011 instead of $(0, 0, 1, 1)$
- *Concatenation* $x \cdot y$ of strings x and y is the string xy.
  - ϵ is the identity element for concatenation, i.e., $\epsilon \cdot x = x \cdot \epsilon = x$.
  - Concatenation of sets of strings: $A \cdot B = \{a \cdot b \mid a \in A, b \in B\}$
  - Concatenation of the same set: $A^2 = AA$; $A^3 = (AA)A$, etc.

  (We often elide the concatenation operator and write AB for $A \cdot B$)
- Kleene-$*$ or closure: $A^* = \{\epsilon\} \cup A \cup A^2 \cup A^3 \cdots = \bigcup_{n \geq 0} A^n$
- Any subset of Σ^* is called a *(formal) language*.
- Heads-up: We will see that any language is a decision problem!

Recap: Language of an Automaton

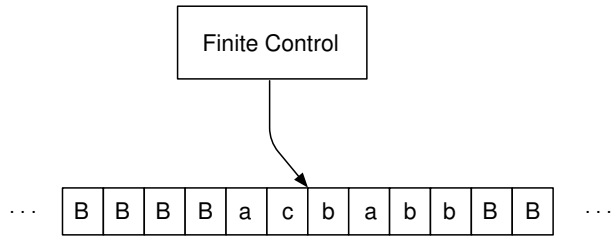
Definition:

- Let A be an automaton (this week: a Turing Machine, ...)
- Then $L(A)$, the language of A, is the set of words "accepted" by A.
- We will define more formally what "accepting" means, but *informally* that's the set of words for which A enters (or *may* enter, in case of a non-deterministic automaton) an "accepting state".

Why is this relevant?

- Decision problems are defined (later) via languages, and
- we will ask whether an algorithm (= Turing Machine) exists that accepts exactly that language – and hence implements what we'd like to do (we'll see that languages describe our problems).

Recap: Turing Machine: Informal Definition

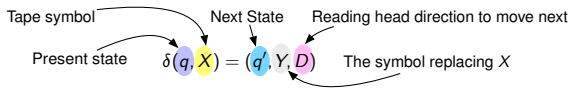


- An tape extending infinitely in both sides
- A reading head that can edit tape, move right or left.
- A finite control.
- A string is accepted if finite control reaches a final/accepting state

Recap: Turing Machine: Formal Definition

A deterministic Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ consists of:

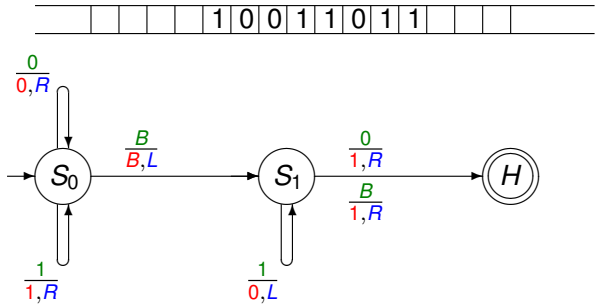
- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ: (deterministic) transition function. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a *partial function* over $Q \times \Gamma$, where the first component is viewed as the present state, and the second is viewed as the tape symbol read. If $\delta(q, X)$ is defined, then



- $B \in \Gamma \setminus \Sigma$ is the blank symbol. All but a finite number of tape symbols are Bs.
- q_0: the initial state of the TM.
- F: the set of final/accepting states of the TM.

Recap: (Deterministic) Turing Machine: Example

Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



(Double-circled states are final states.)

1. What does this TM do? It increments any number by 1!
2. What language does it accept? Σ^*

Recap: Language, Acceptance, and Rejection of Turing Machines

Most important definitions:

- A TM ...
  - *accepts* a word if there is a sequence of transitions from q_0 to some state $f \in F$.
  - *halts* if no state transition exists for the currently read symbol in the current state.
  - *rejects* a word if it is not accepted and all state transitions halt.
- The *language of a TM M*, $L(M)$, is the set of words accepted by it. Note: If $L(M) = L$, then there exists a TM M' with $L(M') = L$ and M' halts on all $w \in L$ (but not necessarily on those not in L).
- A language L is *decided* by a TM M if $L(M) = L$ and for all $w \notin L$, M halts (and thus rejects). A language L is called *decidable* if there exists a TM M, such that M decides L.

We now restrict ourselves to decidable languages and check how "complex" those languages (now: called decision problems) are.

New Basics: Non-deterministic Turing Machines

Definition of non-deterministic TMs:

- Non-deterministic TMs are defined exactly as deterministic ones, with the exception that the state transition function now allows multiple state transitions given the same symbol, $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$ is a *partial function*.
- Even the definition of acceptance, halting, and rejection is the same! (And thus also for the language of a TM.)

The main difference is that now for each word we need to check whether some state transitions (also called: computation path) exist, whereas previously such a path was unique due to determinism.

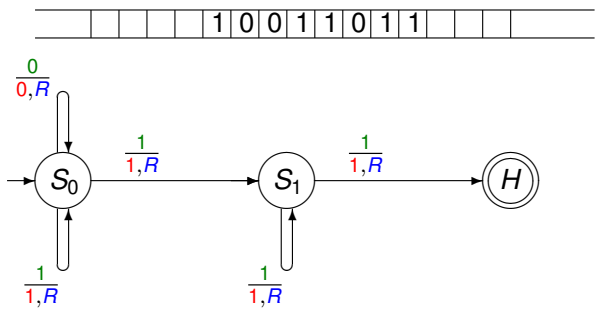
New Basics: Some additional Notes on Turing Machines

The following notes are "redundant" in the sense that they do follow from the previous definitions – but worth pointing out anyway.

- There's an important difference between the acceptance of a word w by finite automata (NFAs, and ϵ-NFAs) and TMs:
  - NFAs/ϵ-NFAs can only accept a word if it's fully read!
  - This is *not* the case for both TMs: They do *not* have to "read" the entire input word. In other words, for a TM to accept a word w it suffices for it to reach any goal state, even if it halts before reading the entire initial tape content.
- Halting neither implies acceptance nor rejection. But rejection implies halting (and non-acceptance), and furthermore on non-deterministic TMs it means that *all transitions* must halt. So if there is a *possibility* to run into an infinite loop, even if all other transitions halt and do not accept, the respective word would still not be rejected (but of course also not accepted).

New Basics: Non-Deterministic Turing Machine: Example

Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



1. What does this TM do? Checking for the "right" input.
2. What language does it accept? $\{w \mid w \text{ contains} \geq 2 \text{ consecutive 1s}\}$
   $= \{w11w' \mid w, w' \in \Sigma^*\}$

New Basics: Decision Problems

Every language is a decision problem. Examples:

- Given two sets A, B, is $A \subseteq B$?
  - → $Subset := \{(A, B) \mid A \subseteq B\}$
  - → Then, checking whether $A_1 \subseteq B_1$ for some concrete A_1, B_1, means checking $(A_1, B_1) \in Subset$.
  - → Can easily be done in **P**.
- Does the SAT formula ϕ a satisfying valuation?
  - → $SAT := \{\phi \mid \phi$ is a SAT formula that has a satisfying valuation $\}$
  - → Then, checking whether some concrete ϕ is satisfiable or not corresponds to checking whether $\phi \in SAT$.
  - → The most famous problem known to be **NP**-complete.
- Does Graph G have a vertex cover of size at most k? A vertex cover is a set of vertices that "covers" all edges.
  - → $VC := \{(\langle V, E\rangle, k) \mid \langle V, E\rangle$ has a node cover $\leq k$, $k \in \mathbb{N}\}$
  - → Then, checking whether a concrete graph G_1 has a node cover, e.g., of size 42, means checking whether $(G_1, 42) \in VC$.
  - → We will also see later that this is **NP**-complete.

"Algorithm" Complexity

Complexity of TMs

We have two kinds of TMs:

- Deterministic Turing Machines and
- Non-deterministic Turing Machines.

For both, we measure their "complexity" both in terms of:

- their (worst-case) runtime
- → number of transitions depending on the input's size.
- their (worst-case) space consumption
- → number of tape cells read depending on the input's size.

Time and Space Consumption, Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if anything not in $\{B, 0, 1\}$ or 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

1. How much "time" does M need, as a function f of w's length? f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $O(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\lvert w\rvert)$	2	8	19	34	53	76

(*exact* numbers depend on implementation details)

2. How much "space" does M need? $O(n)$

So in total, M has polynomial time and space restriction!

Time and Space Consumption in Practice

Our formal theory defines runtime based on Turing Machines, but in practice, we can restrict to pseudo code! Two examples:

Given a CNF formula over n variables, e.g., $(a \vee \neg b) \wedge (b \vee \neg c) \wedge c$, check whether we can make the formula true:

1. Deterministic algorithm: Iterate over all 2^n truth assignments, accept if one makes it true, otherwise reject.
   → polynomial space (re-use variables) and exponential time.
2. Non-Deterministic algorithm: Guess a variable assignment and check it. Accept if one makes it true, otherwise reject.
   → polynomial space and polynomial time with guessing.

So usually we don't have to bother about details of TMs! :)

Complexity Classes

First some auxiliary definitions:

- $\mathbf{DTIME}(t(n)) = \{L \mid \text{det. TM decides } L \text{ in } O(t(n)) \text{ time}\}$
- $\mathbf{NTIME}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ in } O(t(n)) \text{ time}\}$
- $\mathbf{DSPACE}(t(n)) = \{L \mid \text{det. TM decides } L \text{ with } O(t(n)) \text{ space}\}$
- $\mathbf{NSPACE}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ with } O(t(n)) \text{ space}\}$

Now we can define some basic complexity classes:

- $\mathbf{P} = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(n^k)$ $\quad$ $\mathbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathbf{DSPACE}(n^k)$
- $\mathbf{NP} = \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$ $\quad$ $\mathbf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \mathbf{NSPACE}(n^k)$
- $\mathbf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(2^{n^k})$ $\quad$ $\mathbf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathbf{DSPACE}(2^{n^k})$
- $\mathbf{NEXPTIME} = \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(2^{n^k})$ $\quad$ $\mathbf{NEXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathbf{NSPACE}(2^{n^k})$

We focus on classes **P** vs. **NP** vs. **EXPTIME**!
(The remaining ones are just listed for the sake of completeness.)

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - $\mathbf{P} \subseteq \mathbf{PSPACE}$ and $\mathbf{NP} \subseteq \mathbf{NPSPACE}$
  - $\mathbf{EXPTIME} \subseteq \mathbf{EXPSPACE}$ and $\mathbf{NEXPTIME} \subseteq \mathbf{NEXPSPACE}$

How do deterministic and non-deterministic classes relate?

- Deterministic TMs are a special case of non-deterministic TMs.
- Thus, non-deterministic classes can potentially contain more problems than their corresponding deterministic classes:
  - $\mathbf{P} \subseteq \mathbf{NP}$ and $\mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$
  - $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$ and $\mathbf{EXPSPACE} \subseteq \mathbf{NEXPSPACE}$

Relationships among Complexity Classes: What's known

What's also known to the literature:

- **PSPACE** $=$ **NPSPACE** and **EXPSPACE** $=$ **NEXPSPACE** (Savitch's Theorem, 1970)
- **P** $\subsetneq$ **EXPTIME** (We know problems in **EXPTIME** which are provably not in **P**)

So in total, we get:

- (**NPSPACE** and **NEXPSPACE** not shown due to the above.)
- $\underbrace{\textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} \subseteq \textbf{EXPTIME}}_{\neq} \subseteq \textbf{NEXPTIME} \subseteq \textbf{EXPSPACE}$

Reductions

Reductions: Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via *reduction*.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least as hard; otherwise that's not possible).

Definition

$f : \Sigma^* \to \Sigma^*$ is a *polytime-computable* function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input $w \in \Sigma^*$.

Definition

$A \subseteq \Sigma_1^*$ is *polynomial time mapping-reducible* to $B \subseteq \Sigma_2^*$, written $A \leq_P B$, if a polytime-computable function $f : \Sigma_1^* \to \Sigma_2^*$ exists that is also a reduction (from A to B).

Reductions: Main Definition

Definition

- A reduction is a polynomial-time translation of the problem, say r.
- More precisely:
  1. $r(w)$ can be computed in time polynomial in $|w|$.
  2. $w \in A$ if and only if $r(w) \in B$ (so it "preserves the answer").

Example:

- EVEN $:= \{n \mid n \bmod 2 = 0\}$, ODD $:= \{n \mid n \bmod 2 = 1\}$
- Reduction from ODD to EVEN:
  - $r(k) = k + 1$, so we get $k \in$ ODD iff $r(k) \in$ EVEN
  - So essentially we can define odd(n):=even($r(n)$) now.
- If however our goal would have been to show that the 'new' problem ODD is at least as hard as EVEN, then we would have had to reduce from EVEN to ODD (though r would have been the same). Check this statement after "hardness" was introduced!

Reductions: Independent Set

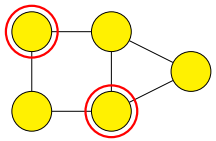
The Independent Set Problem:
Assume you want to throw a party. But you know that some of your friends don't get along. You only want to invite people that *do* get along.

Formalized as graph.

- vertices are your mates
- draw an edge between two vertices if people don't get along

Problem:
Given a graph and a $k \geq 0$, is there an *independent set*, i.e., a subset I of $\geq k$ vertices so that

- no two elements of I are connected with an edge.
- i.e., everybody in I gets along



Example of an independent set of size 2 (*just* the red-circled vertices)

Reductions: Solving the Independent Set Problem

Naive Implementation:

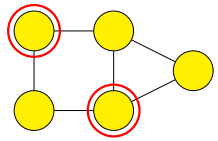
- loop through all subsets of size $\geq k$ (exponentially many!)
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**

Using Non-deterministic Turing Machines:

- *guess* a subset of vertices of size $\geq k$
- *check* whether it is an independent set

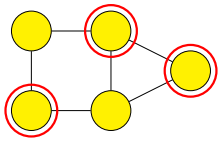
$\rightarrow$ Proves membership in **NP**



Question: Can we do better? Is there a **P** algorithm?
Answer: We don't know! But "hardness" helps figuring this out.

Reductions: Vertex Cover

Given a graph $G = \langle V, E \rangle$, a *vertex cover* is a set C of vertices such that every edge in G has at least one vertex in C.



Example vertex cover:
The red-circled vertices.

Vertex Cover (Decision) Problem.

- Given graph $\langle V, E \rangle$ and $k \geq 0$, is there a vertex cover of size $\leq k$?
- $VC := \{(\langle V, E \rangle, k) \mid \langle V, E \rangle$ has a node cover $\leq k, k \in \mathbb{N}\}$

Naive Algorithm:

- search through all subsets of size $\leq k$ (this is exponential)
- check whether it's a vertex cover

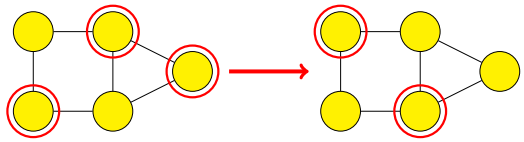
$\rightarrow$ This proves $VC \in$ **EXPTIME**, but we can do better!
(I.e., we could also guess and verify as before, giving $VC \in$ **NP**.)

Reductions: From Independent Set to Vertex Cover

Reductions. Use solutions of one problem to solve another.

Observation. Let G be a graph with n vertices and $k \geq 0$.

- G has a VC of size $\leq k$ iff G has an IS of size $\geq n - k$



- Why?
  - VC with $\leq k$ vertices needs to cover *all* edges.
  - IS with $\geq n - k$ vertices can't cover *any* edge.

What's the reduction? Vertex cover to independent set:

- $\langle G, k \rangle \in VC$ iff $r(\langle G, n \rangle) \in IS$, where $= r(\langle G, n \rangle) = \langle G, n - k \rangle$.
- H the reduction r here only changes the number, but nothing else. But for most reductions, we will have to "translate problems", e.g., when turning a SAT problem into a VC problem!

Reductions: Important Note on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover! (Reducing a problem in **P** to a problem that's **NP**-hard.)
  - You should be able to reduce Vertex Cover to Rush Hour. (Reducing an **NP**-complete problem to one that's **PSPACE**-hard.)

(Hardness and completeness are explained in the next section...)

Completeness

Membership, Hardness, and Completeness

Definition (NP completeness, NP membership, NP hardness)

A language B is ***NP-complete*** if

1. $B \in$ **NP** = **NP** membership
2. *every* $A \in$ **NP** is polytime-reducible to B. = **NP** hardness

- So we have "for all A holds $A \leq_P B$", and therefore we know that B is "hard/expressive enough" to solve all other problems in **NP**. (Because we solve these other A-problems using our B-problem!)
- Therefore, **NP**-complete problems are the hardest ones in **NP**. (In particular they may be harder than those in **P**!)
- Hardness is the opposite of "practical exploitation of reductions": For hardness, reduce *from* a known problem rather than *to* one!

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - It doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove **NP**-completeness, then at least we know that nobody before you found an **NP** procedure. (And maybe none even exists, which follows directly once somebody proves **P**$\neq$**NP**.)
- Why **NP**-completeness? Why not just showing **NP**-hardness?
  - Since the problem could be even harder! (E.g., **PSPACE**-hard, **EXPTIME**-hard, **NEXPTIME**-hard, . . . , and *infinitely* more!)
  - Each problem class has specific "properties". E.g., "**NP**-complete looks like Logic", "**PSPACE**-complete looks like planning", etc.

NP-Hardness

Theorem

*If B is **NP**-hard and $B \leq_P C$, then C is **NP**-hard.*

Corollary

*If B is **NP**-complete and $B \leq_P C$ for $C \in$ **NP**, then C is **NP**-complete.*

Proof.

Polynomial time reductions compose. □

Important! This Corollary is of *major* importance!! Why?
→ It gives us a convenient procedure to show **NP**-completeness!

- First, show **NP** membership. (That's almost always very easy.)
- Then, show hardness by grabbing an **NP**-complete problem and reduce it to yours!

Known NP-Complete Problems

List of known **NP**-complete problems:

- SAT (first problem proved **NP**-hard) and 3-SAT (see tutorials)
- Graph-Colourability and 3-Graph-Colourability (see tutorials)
- Independent Set and Vertex Cover (these slides)
- Hamiltonian path (not covered)
- Traveling Salesman Problem (not covered)
- Many more!

Summary

Summary

- We focused on complexity classes **P**, **NP**, and **EXPTIME**.
- Runtime is measured in terms of number of TM transitions, depending on the encoding size of *input word* for which we want to judge whether it's a member of our language/problem.
- Reductions, which turn one decision problem into another using only polynomial time. Can be used for:
  - Exploiting existing algorithms (reduce *to* known problem)
  - Prove hardness of your problem (reduce *from* known problem)
- Hardness and completeness of problems.

Some concluding words:

- Liked this week? Look into COMP3630, Theory of Computation.
- I hope you enjoyed weeks 6 to 8 and 12! :)
- Good luck in the exam! (And the others you still have.)