Complexity and reductions recitation 2022

Complexity theory is a field which tries to precisely characterise (i.e. upper and lower bound) the computational resources which are required to solve certain problems. 'Computational resources' can mean time, memory (space), and many other more exotic things. Broadly speaking, all results in complexity theory take one of two forms:

- **Upper bound:** a proof that problem X can be solved using Y amount of Z computational resource.
  - E.g.: a proof that the problem of deciding whether a graph $G = (V, E)$ is bipartite can be solved in $O(|V| + |E|)$ time.
  - This is shown by exhibiting an algorithm that solves the problem (e.g. BFS).
  - Can be more exotic than this basic example: for example, the historic results $\mathsf{MIP} = \mathsf{NEXP}$, $\mathsf{NP} \subseteq \mathsf{PCP}$, $\mathsf{PSPACE} = \mathsf{IP}$, $\mathsf{NL} = \mathsf{coNL}$ are all examples of upper bounds. In order to show that two complexity classes are equal, you have to show that all the problems which can be solved by one computational model can also be solved by the other, and vice versa. For example, $\mathsf{PSPACE} \subseteq \mathsf{IP}$ involves showing that any problem which can be solved by a Turing machine in polynomial space can also be solved by a probabilistic polynomial-time verifier interacting with an unbounded prover.
- **Lower bound**: a proof that problem X *cannot* be solved using Y amount of Z computational resource.
  - E.g. a proof that SAT *cannot* be solved in less than exponential time.
  - Lower bounds are usually much harder than upper bounds, although upper bounds can already be very nontrivial to prove.
  - We don't have the aforementioned lower bound, of course—if we did, we'd have $\mathsf{P} \neq \mathsf{NP}$!

In order to prove either lower or upper bounds—in order to prove statements of the form 'problem X can/cannot be solved using Y amount of Z computational resource'—we first have to formally define what a 'problem' is.

What is a language?

- Also called a *decision problem*.
- Informally, a language (or decision problem) is yes/no problem to be solved.
  - Examples of questions that can be captured as languages (some easy, some hard):
    - 'I give you two numbers (x, y). Is x strictly bigger than y?'
    - 'I give you the algebraic expression for a certain polynomial, e.g. $f(x) = (x + y)(x - y) + (x^2 - y^2)$. Is the polynomial equivalent to 0?'
    - 'I give you the description of a position on a chessboard. Does the position I gave you have a winning sequence of moves (i.e. a sequence of moves guaranteed to win) for white?'
- Formally, a language is a subset of $\{0, 1\}^*$, where $\{0, 1\}^*$ is the set of all possible bitstrings (of any length, hence the star). We usually denote a language by L, and so we can write $L \subseteq \{0, 1\}^*$.
- Which bitstrings are in the subset? We interpret the bitstrings as 'problem statements', and whether or not the answer to a given 'problem statement' is YES determines whether or not it's in a given language L.
  - Example: 'I give you two numbers (x, y). Is x strictly bigger than y?' The language here would be the set of all bitstrings that encode two integers x and y such that $x > y$. In other words:

    $$L = \{\langle x, y \rangle : x > y\}.$$

    We often use the notation $\langle x \rangle$ to denote the binary encoding of x.

What is a Turing machine?

- Informally: a Turing machine is an 'algorithm'. It's a piece of code that reads an input x and writes an output y (in the context of decision problems, the output y is typically a single bit). If you could write a Python program to do that computation which takes in x and spits out y, then there's a Turing machine representation of that computation.
- Formally: you can look at the 'Description' and 'Formal definition' sections of the Wikipedia article on the subject, which is actually pretty good. I'm not going to bother to copy it out here, because, unfortunately, the details are slightly

tedious.

- You should know, however, that people refer to Turing machines as having an 'input tape', a 'work tape' and an 'output tape'. You can imagine all three as infinite arrays.
  - The 'input tape' is read only, and on it is written the input x.
  - The 'output tape' is write only and one-way (can't erase from it once you've written or read symbols that you wrote previously), and the Turing machine is supposed to write its output on that tape.
  - The 'work tape' is read/write, and the Turing machine can write any data it wants to keep track of on the work tape.
- You should also know what a **'timestep'** is. Basically, the Turing machine has a 'head' that at any given time is hovering over some symbol on one of its tapes. It does computations by moving the head around and reading and writing symbols on its tapes; at any given time, it can only read or write to the cell which lies directly under its head. Every time it wants to move left one cell, move right one cell, write to the cell that it's hovering over, or change tapes, it needs to expend one **timestep**.
  - (How does it know what it 'wants to' do next? At any given time, the head has a constant size 'state' that it's in. We assume the head has memorised a constant sized 'state table' that tells it what action to execute next from its repertory of legal actions—move left, move right, change tapes, write something to current cell, change 'states'—depending on what's in the cell that it's hovering over and depending on its current 'state'.)
- The **running time** of a Turing machine is the number of timesteps it takes to halt, *as a function of the length of its input* x. (A Turing machine has a well-defined behaviour on any x, and may of course have different behaviour, including running for different lengths of time, on different inputs.)
- The **space usage** of a Turing machine is the number of cells on its work tape that it touches during computation, once again as a function of the length of its input x.
- We say that a Turing machine *decides* L (for some language L) in time $t(n)$ and space $s(n)$ if:
  - for any $x \in L$, the Turing machine on input x runs for at most $t(|x|)$ timesteps, touches at most $s(|x|)$ cells on its work tape, and finally writes (a symbol meaning) 'ACCEPT' on its output tape;
  - for any $x \notin L$, the Turing machine on input x runs for at most $t(|x|)$ timesteps, touches at most $s(|x|)$ cells on its work tape, and finally writes (a

symbol meaning) 'REJECT' on its output tape.

What are P and NP?

- All complexity classes are *sets of languages*.
- P is the set of languages that can be decided by a Turing machine in time at most kn^k for some constant k.
- One definition of NP is as the set of languages that can be *verified* by a Turing machine in time at most kn^k for some constant k. Formally: L is in NP if there exists a polynomial-time computable relation V and a constant k such that

$$x \in L \iff \exists y \in \{0,1\}^{k|x|^k}, \text{ s.t. } V(x,y) = 1;$$

$$x \notin L \iff \forall y \in \{0,1\}^{k|x|^k}, V(x,y) = 0.$$

- The string y above is called the *witness*.
- Examples of problems in NP:
  - The quintessential example, SAT.
    - Input: a Boolean formula (an expression containing Boolean variables along with AND ($\wedge$), OR ($\vee$), NOT ($\neg$) symbols, e.g.

      $$(x \text{ or } y) \text{ and } ((\text{not } x) \text{ and } y)$$

      where x and y are bits).
    - Question to decide: is the formula *satisfiable*? That is, is there an assignment to all the variables in the formula such that the formula evaluates to 1? (The formula above is satisfiable: set $x = 0, y = 1$.)
    - Witness: a satisfying assignment.
    - Verification algorithm: plug in the satisfying assignment and check that the formula evaluates to 1 (evaluating a Boolean formula can be done in polynomial time in the size of the formula).
  - 3SAT: a special case of SAT.
    - Input: a Boolean formula which is an OR of AND clauses, such that each AND clause contains at most three variables. For example:

      $$(x_1 \text{ or } (\text{not } x_2) \text{ or } x_3) \text{ and } (x_2 \text{ or } (\text{not } x_3) \text{ or } (\text{not } x_4))$$

    - Question to decide: is the formula satisfiable?

    - Witness: a satisfying assignment.
    - Verification algorithm: plug in the satisfying assignment and check that the formula evaluates to 1.
  - 3-colouring.
    - Input: a graph $G = (V, E)$.
    - Question to decide: is the graph 3-colourable? That is, does there exist a colouring of the vertices which uses no more than 3 distinct colours such that no two endpoints of an edge share the same colour?
    - Witness: a valid 3-colouring C for the graph.
    - Verification algorithm: loop through all edges in the graph and check that C assigns different colours to all of them.
- All of these problems (SAT, 3SAT, 3COL) are *NP-complete*.

Karp reductions and NP-completeness

- Intuitively, some problems in NP are harder than others. (For example, NP contains P, but we don't expect problems in P to be 'as hard as' some other problems in NP, assuming P $\neq$ NP.)
- The notion of NP-*completeness* formally captures the idea of 'the hardest problems in NP'. A problem is NP-complete only if it is 'as hard as any other problem in NP'.
- We formalise this intuition using the notion of a *reduction*. If problem A can be reduced to problem B, then we write $A \leq B$, and we may say in English that 'if you can solve B, then you can also solve A'.
- In complexity theory, we like to use a relatively weak kind of reduction called a *Karp reduction*. (We like Karp reductions in complexity because they preserve distinctions between classes like NP and coNP. Problems in NP *cannot*, in general, be Karp reduced to problems in coNP, but NP and coNP would be considered equal under the stronger sorts of reductions which are usually found in cryptography.)
- A Karp reduction from language A to language B is a function f such that:

$$x \in A \implies f(x) \in B;$$

$$x \notin A \implies f(x) \notin B.$$

- A language $L \in \mathsf{NP}$ is NP-*complete* only if, for *any* other language $L' \in \mathsf{NP}$, L' can be Karp reduced to L, and the function f involved in the Karp reduction can be computed in polynomial time (in the length of its input x).
- In particular, this means that, if someone finds a polynomial time algorithm for any NP-complete language, then every language in NP has a polynomial-time algorithm (why?), and so $\mathsf{P} = \mathsf{NP}$.
- Note that a Karp reduction is only allowed to look at the 'problem statement' or input. If you have an algorithm that decides B and you're trying to decide whether some input x is in A, and a Karp reduction exists from A to B, then you'd literally just look at the input x, compute f on it, and run your algorithm for B on $f(x)$.
- We could imagine a more general kind of reduction that is not only allowed to transform the input, but is also allowed to look at the output of the algorithm for B on the input $f(x)$ and do some computation on that before outputting its decision on $x \overset{?}{\in} A$. This sort of reduction is called a *Turing reduction*.

Turing reductions

- A *Turing reduction* from problem A to problem B works as follows: you are trying to decide whether $x \in A$, and you are allowed access to an *oracle* which tells you, for any y, whether $y \in B$. If you can, using your oracle, successfully decide A, then there is a Turing reduction from A to B.
- Note that Karp reductions are a special case of Turing reductions. If there's a function f such that

$$x \in A \implies f(x) \in B;$$

$$x \notin A \implies f(x) \notin B,$$

and you are able to compute f, then you (the machine which is trying to decide A on input x) can use your B oracle as follows: compute $f(x)$, ask the B oracle about the input $y = f(x)$, and output whatever the B oracle outputs.

- In cryptography, we often consider Turing reductions. In fact, we usually consider even more general reductions than Turing reductions: in a Turing reduction, we assume that the B oracle is right all the time, but in cryptography, the algorithm we're reducing to doesn't need to be right all the time.

Reductions in cryptography

- The vast majority of (classical) cryptography is built on *assumptions*.
- This is because we can't prove unconditionally that, e.g., secure encryption exists without proving first that $\mathsf{P} \neq \mathsf{NP}$.
- So we just assume that $\mathsf{P} \neq \mathsf{NP}$ (in fact, we usually assume somewhat stronger things, e.g. 'one way functions exist'), and from such assumptions we can build all of cryptography (we can prove that secure encryption exists, that secure digital signatures exist, that secure zero-knowledge proofs exist, etc.).
- How do we do that? In order to show that assumption A (e.g. 'unpickable Chubb locks exist') implies assumption B (e.g. 'my bullion safe is secure'), we show the contrapositive: if not B, then not A. If our safe is *not* secure, i.e. if there exists a probabilistic polynomial-time adversary $\mathcal{B}$ that can break into our safe, then there exists a probabilistic polynomial-time adversary $\mathcal{A}$ that can pick a Chubb lock. (Note, in particular, that this requires us to build our safe in such a way that the *only* efficient means of entry is through picking a Chubb lock.)
- **In other words, we want a reduction from $\mathcal{A}$ to $\mathcal{B}$**: we want to reduce breaking the thing we assumed is impossible to break to breaking the thing we want to prove is impossible to break. We want to show $\mathcal{A} \leq \mathcal{B}$, i.e., that picking the Chubb lock (assumed intractable) is actually easier than breaking into our safe (which we want to be intractable), or that breaking a PRG (assumed intractable) is actually easier than breaking our encryption scheme $\mathscr{E}$ (which we want to be intractable).
- 'Breaking' some assumption, in a cryptographic context, usually means winning some security game with non-negligible probability / non-negligible advantage (which one depends on the game).

A simple example of a cryptographic reduction

- Let $G : \{0,1\}^n \to \{0,1\}^{2n}$ be a secure PRG. Show that the PRG $G' : \{0,1\}^{n+1} \to \{0,1\}^{2n+1}$ is a secure PRG, where

$$G'(s\|b) := G(s)\|b.$$

- We want to show that any adversary for G' can be used to construct an adversary for G. So, we firstly assume that we have a generic adversary $\mathcal{B}$ for G', and then we construct one ($\mathcal{A}$) for G which uses $\mathcal{B}$ to break G.
- The key here is to:
  - Write down everything that $\mathcal{B}$ expects to receive from its challenger:
    - A sample d_α, $\alpha \in \{0,1\}$, so that

$$d_\alpha = \begin{cases} G(s)\|b & \alpha = 0 \\ r^{2n+1} & \alpha = 1. \end{cases}$$

  - Write down what $\mathcal{A}$ (= us!) receives from its challenger:
    - A sample c_α, $\alpha \in \{0,1\}$, so that

$$c_\alpha = \begin{cases} G(s) & \alpha = 0 \\ r^{2n} & \alpha = 1. \end{cases}$$

  - Figure out how $\mathcal{A}$ can use what it gets from its challenger, and whatever public knowledge is available to it, in order to 'simulate' $\mathcal{B}$'s challenger.
    - $\mathcal{A}$ can output $c_\alpha\|b$, for a b that it chooses uniformly at random from $\{0,1\}$.
    - Notice that, in both cases of α, $c_\alpha\|b = d_\alpha$.
    - $\mathcal{A}$ outputs whatever $\mathcal{B}$ outputs, and if $\mathcal{B}$ guesses correctly, then so does $\mathcal{A}$. Therefore, $\mathcal{A}$ has the same advantage as $\mathcal{B}$.
  - Note that, in this very simple case, we actually did a 'Karp reduction'. There are no languages here, and the reduction was *probabilistic* (we used randomness to compute the input we gave to $\mathcal{B}$), so it's not a real Karp reduction—but, morally speaking, we simply identified a single transformation which would both transform a yes-input of $\mathcal{A}$ into a yes-input for $\mathcal{B}$, and transform a no-input of $\mathcal{A}$ into a no-input for $\mathcal{B}$. (It needs to be a single transformation because we of course don't know which is which *a*

priori.) Then we (as $\mathcal{A}$) outputted whatever $\mathcal{B}$ output.

- In more sophisticated analyses we will need to use 'Turing reductions' (quote marks mean: don't take that literally!), where we as $\mathcal{A}$ not only transform the input that we receive but also do computations on the output that $\mathcal{B}$ returns to us.

Bonus problem

- Assume that $G : \{0,1\}^n \to \{0,1\}^{2n}$ is a secure PRG. Construct a PRG $G' : \{0,1\}^{n+1} \to \{0,1\}^{2n+1}$ which is secure, but which becomes insecure if the *parity* of its seed is always chosen to be zero.
- Step 1: show that $G' : \{0,1\}^{n+1} \to \{0,1\}^{2n+1}$ defined as $G'(s||b) = G(s)||b$ is insecure if the last bit of its seed is always chosen to be 0. (Easy!)
- Step 2: how can we leverage step 1? Hint: use the fact that any bijective function maps the uniform distribution to itself.