1 KuDoS questions

1. For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, what does:

   - $f = \mathcal{O}(g)$ mean?

     Informally: f is eventually bounded above by g.

     Formally:
     $$f = \mathcal{O}(g) \iff \exists n_0, c. \forall n \geq n_0 . f(n) \leq c \cdot g(n)$$

   - $g = \Omega(g)$ mean?

     Informally: f is eventually bounded below by g.

     Formally:
     $$f = \Omega(g(n)) \iff \exists n_0, c. \forall n \geq n_0 . f(n) \geq c \cdot g(n)$$

2. What is:

   - A Turing machine?

     A Turing machine is a theoretical model of computation based on the idea of a machine which is in one of a finite internal states uses a head to read and write from a tape which is "unbounded to the right" and upon which the input is encoded.

     Formally, a Turing machine can be represented by a quadruple (Q, Σ, δ, s), where Q is the set of states, Σ is the set of symbols which can occur on the tapes ($\{\triangleright, \sqcup\} \subset \Sigma$ are special symbols), $\delta \in Q \times \Sigma \to (Q \cup \{acc, rej\}) \times \Sigma \times \{L, S, R\}$ is the transition function and s is the initial state.

   - A configuration of the Turing machine?

     A configuration of a Turing machine is a triple (q, u, v) where q is the state of the Turing machine, u is the string of symbols to the left and under the tape head and v is the string of symbols to the right which stretches as far as the last non-blank symbol.

   - A computation of a Turing machine?

     A computation of a Turing machine is a finite or infinite sequence of configurations $c_0 \to_M c_1 \to_M^\star \ldots$. Where, for all i, $c_i \to_M c_{i+1}$ is a valid step.

     Without loss of generality, let $c_0 = (q, ua, v)$ and $c_1 = (q', u', v')$. A transition is valid if and only if $\delta(q, a) = (q', b, D)$ and one of the following cases holds:

     $$\begin{cases} D = L, u' = u, v' = bv \\ D = S, u' = ub, v' = v \\ D = R, v = cv'', u = ubc, v' = v'' \\ D = R, v = \varepsilon, u' = ub\sqcup, v' = \varepsilon \end{cases}$$

     If the computation halts, then it halts in either an accepting state acc or a rejecting state rej.

3. What does it mean if a language is:

   - Recursively Enumerable?

     A language L is Recursively Enumerable if there exists a Turing machine M such that $L = L(M)$ – a Turing machine which (when run with $\triangleright x$ on its input tape) halts in an accepting state if and only if $x \in L$. Note that the Turing machine may not halt on strings which are not members of L – and the number of steps before it halts is neither computable, nor can it be given an upper bound.

- Semi-decidable?

  Semi-decidable is a simile for Recursively Enumerable.

- Decidable?

  A language L is decidable if and only if there exists a Turing machine M, which (when ran with $\triangleright x$ on its input tape) will reach an accepting state if and only if $x \in L$; and will reach a rejecting state if $x \notin L$.

  The difference between decidable and semi-decidable is that the machine M halts on all inputs.

- What does it mean if a function is computable?

  A function f is computable if and only if there exists a Turing machine M which, when started with $\triangleright x$ on the tape, will terminate in an accepting state *acc* with $\triangleright y$ on the tape if and only if $f(x) = y$.

4. Define the running time of a turing machine M.

   The running time of a Turing machine M is a function $r : \mathbb{N} \to \mathbb{N}$ such that $r(n)$ is the most steps required in any halting computation on an input of length n. If *no* computation on an input of length n halts, then $r(n) = 0$.

5. For any function $f : \mathbb{N} \to \mathbb{N}$, what does it mean that a language L is in $\mathsf{TIME}(f)$?

   $L \in \mathsf{TIME}(f)$ means that there exists a Turing Machine M such that $L = L(M)$ and the running time of M is in $\mathcal{O}(f)$.

   What does it mean that it is in $\mathsf{SPACE}(f)$?

   $L \in \mathsf{SPACE}(f)$ means that there exists a Turing Machine M with a read-only input table and a mutable work-tape such that M goes no more than $\mathcal{O}(f)$ cells to the right on the work tape.

6. Give an alternative definition of decidability in terms of the computability of a function and the $\mathsf{TIME}(\,\cdot\,)$ function.

   A language L is decidable if and only if there exists a function f such that $L \in \mathsf{TIME}(f(n))$.

7. Define the complexity class P

   P is the class of languages which are accepted by a deterministic Turing Machine in polynomial time. This is formally described below:

   $$\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathsf{TIME}(n^k)$$

8. What is the SAT language?

   SAT is the language consisting of boolean expressions which are satisfiable.

9. What is a verifier V for a language L?

   A verifier V for a language L is a Turing machine which, when started with $\triangleright x, c$ will accept if and only if $c$is a certificate of membership for x being a member of L; and will reject the input if c is not a certificate for x being in L.

   This does not *determine membership of L* – intuitively it checks whether an encoding of a proof of membership (a certificate c) is valid.

10. What is a non-deterministic Turing machine?

    A non-deterministic Turing machine is a Turing machine for which the transition function is not a function; but a relation – $\delta \in Q \times \Sigma \rightharpoonup (Q \cup \{acc, rej\}) \times \Sigma \times \{L, U, R\}$. This means the machine could be in one of an exponential number of possible configurations. The non-deterministic Turing machine is said to accept the string if *any* of those possible configurations are accepting.

11. Wha is a reduction of a language L_1 to a langauge L_2?

    Informally, a reduction is a function f which maps strings in the language L_1 to strings in the language; and strings *not* in the language L_1 to strings which are not in the language L_2.

    A reduction from a language $L_1 \subseteq \Sigma^*$ to a langauge $L_2 \subseteq \Sigma^*$ is a function $f : \Sigma^* \to \Sigma^*$ which satisfies the property:

$$\begin{cases} f(x) \in L_2 & \text{if } x \in L_1 \\ f(x) \notin L_2 & \text{if } x \notin L_1 \end{cases}$$

12. What does it mean for a language L to be NP-hard?

    Informally: a language $L$is NP-hard if it is harder than all languages in NP.

    Formally: a language L is NP-hard if and only if, $\forall A \in \mathsf{NP}. A \leq_P L$ – "for all A in NP, there is a polynomial time reduction from A to L.

    What does it mean for a language L to be NP-complete?

    A language L is NP-complete if and only if it is both in NP and NP-hard.

2 2018 Paper 6 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2018p6q3.pdf

(a) Give a precise definition of each of the complexity classes NP and $\mathsf{co-NP}$.

A language L is in NP if and only if there exists a nondeterministic Turing machine M such that $L(M) = L$.

A language L is in $\mathsf{co-NP}$if and only if there exists a nondeterministic Turing machine M such that $L(M) = \bar{L} \triangleq \{x | x \notin L\}$.

(b) Give an example of the following, in each case giving a precise statement of the decision problem involved.

(i) an NP-complete language

SAT is an NP-complete language. Given a boolean formula (with free variables), does it hold under some interpretations?

(ii) a $\mathsf{co-NP}$-complete language

VAL is a $\mathsf{co-NP}$-complete language. Given a boolean formula (with some free variables), does it hold under all interpretations?

(c) If A and B are the two languages identified in Part (b), give an example of a language that is polynomial-time reducible to both A and B. Justify your answer.

I give Σ^* – the language that accepts every string. I present a reduction from Σ^* to SAT and from Σ^* to VAL.

In polynomial time, we can form the boolean expression **true**. This meets the criteria for a polynomial-time reduction from $\Sigma^* \leq_P \mathsf{SAT}$.

$$\begin{array}{ll} x \in \Sigma^* \implies f(x) \in \mathsf{SAT} & x \notin \Sigma^* \implies f(x) \notin \mathsf{SAT} \\ \quad \mathbf{true} \implies \mathbf{true} \in \mathsf{SAT} & \quad \mathbf{false} \implies \dots \\ \quad \mathbf{true} \implies \mathbf{true} & \quad \mathbf{true} \\ \quad \mathbf{true} & \end{array}$$

In polynomial time, we can form the boolean expression **true**. This meets the criteria for a polynomial-time reduction from $\Sigma^* \leq_P \mathsf{VAL}$.

$$\begin{aligned} x \in \Sigma^* &\implies f(x) \in \mathsf{VAL} & x \notin \Sigma^* &\implies f(x) \notin \mathsf{VAL} \\ \textbf{true} &\implies \textbf{true} \in \mathsf{VAL} & \textbf{false} &\implies \ldots \\ \textbf{true} &\implies \textbf{true} & \textbf{true} & \\ \textbf{true} & & & \end{aligned}$$

So Σ^* is polynomially reducible to both SAT and VAL. Any language $L \in \mathsf{P}$ is polynomially reducible to both SAT and VAL by the algorithm which decides the language and maps $x \in L$ to **true** and $x \notin L$ to **false**.

3 2014 Paper 6 Question 1

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p6q1.pdf

(a) Give two definitions of the complexity class NP, one using the term Turing machine and one using the term verifier.

- Turing machine definition

  NP is the class of languages which can be accepted in polynomial time by a nondeterministic Turing machine: $\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k)$

- Verifier definition

  NP is the class of languages which can be polynomially verified (by a deterministic Turing machine). Polynomially verifiable means there exists a verifier V for language L in time polynomial in x.

(b) For each of the following statements, state whether it is true, false or unknown. In each case, give justification for your answer. In particular, if the truth statement is unknown, state any implications that might follow from it being true or false.

(i) $\mathsf{3SAT} \leq_P \mathsf{CLIQUE}$

This is true. I provide an indirect reduction. From $\mathsf{3SAT}$ to IND and from IND to CLIQUE.

A polynomial-time reduction from $\mathsf{3SAT}$ to IND was given in the lecture notes.

A polynomial-time reduction from IND to CLIQUE takes the complement of the graph – this is linear and hence is in P.

(ii) $\mathsf{TSP} \in \mathsf{P}$

This is unknown. Since the travelling salesman problem is **not** known to be either NP-complete or NP-hard, this would not prove $\mathsf{P} = \mathsf{NP}$.

(iii) $\mathsf{NL} \subseteq \mathsf{P}$

This is true. It's possible to simulate an algorithm which runs in $\mathsf{NSPACE}(f(n))$ on a deterministic Turing machine which in $\mathsf{TIME}(k^{\lg n + f(n)})$ for some k. NL is the set of problems which are solvable in $\mathsf{NTIME}(\lg n)$. Therefore, they can be simulated by a deterministic Turing machine in $\mathsf{TIME}(k^{\lg n + \lg n}) = \mathsf{TIME}(\mathsf{k}^{\lg \mathsf{n}}) = \mathsf{TIME}(n^{\lg k})$. This is in P. Therefore, all problems which are in NL are also in P. So $\mathsf{NL} \subseteq \mathsf{P}$.

(iv) $\mathsf{PSPACE} \neq \mathsf{NPSPACE}$

This is false.

Using Savitch's Theorem, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$. Using the definitions of NPSPACE and PSPACE, we have:

$$\begin{aligned}\mathsf{NPSPACE} &= \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)\\ &\subseteq \bigcup_{k\in\mathbb{N}} \mathsf{SPACE}(n^{2k})\\ &\subseteq \bigcup_{k\in\mathbb{N}} \mathsf{SPACE}(n^k)\\ &\subseteq \mathsf{PSPACE}\end{aligned}$$

We also have from the space hierarchy theorem that $\mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$. Therefore $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

(c) Let $\Sigma = \{0, 1\}$. Prove that $\emptyset$ and $\{0, 1\}^*$ are the only languages in P which are not complete for P with respect to polynomial time reductions.

- Assume that $L \in \mathsf{P}$ and $\exists x, y.x \in L, y \notin L$. This is the set of languages $L \in \mathsf{P} \setminus \{\emptyset, \{0, 1\}^*\}$.

  To prove that L is complete for P, we must prove that for any arbitrary language $L' \in \mathsf{P}$, $L' \leq_P L$.

  Define f as follows:
  $$f(z) = \begin{cases} x & \text{if } z \in L' \\ y & \text{if } z \notin L' \end{cases}$$

  Since $L' \in \mathsf{P}$, we have that L' can be recognised in polynomial time. So f performs a polynomial-time computation then a constant amount of work (writing x or y onto the tape) – $f \in \mathsf{P}$. Since f is a reduction and L' was arbitrary, we can conclude that $\forall L' \in \mathsf{P}.L' \leq_P L$.

  Therefore, all langauges $L \in \mathsf{P}$ except for $\emptyset$ and $\{0, 1\}$ are complete for P with respect to polynomial-time reductions.

- Let $L = \emptyset$

  The language $\{1\}$ is in P.

  Assume that there is a reduction from $\{1\}$ to $\emptyset$. So there exists some f such that $f(1) \in \emptyset$. This is a contradiction! There are no elements in the empty set – so there can be no reduction from $\{1\}$ to $\emptyset$. Hence $\emptyset$ is not P-complete.

- Let $L = \{0, 1\}^*$

  The language $\{1\}$ is in P.

  Assume there is a reduction from $\{1\}$ to $\{0, 1\}^*$. So there exists some f such that $f(0) \notin \{0, 1\}^*$. This is a contradiction – by definition, $\forall x \in \Sigma^*.x \in \{0, 1\}^*$. Therefore, the assumption that there exists a polynomial-time reduction from $\{1\}$ to $\{0, 1\}$ must be false. Hence $\{0, 1\}$ is not P-complete.

Hence, all languages in P except for $\emptyset$ and $\{0, 1\}^*$ are complete for P with respect to polynomial-time reductions. As required.

4 2012 Paper 6 Question 1

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2012p6q1.pdf

(a) Suppose L_1 and L_2 are languages in P. What can you say about the complexity of each of the following? Justify your answer in each case.

(i) $L_1 \cup L_2$

$L_1 \cup L_2 \in \mathsf{P}$. A Turing machine which recognises $L_1 \cup L_2$ in polynomial time can be formed by starting with the machine which recognises L_1, then running the machine which recognises L_2. If **either** machines reach an accepting state then accept. We run two polynomial-time algorithms, the new machine also terminates in polynomial-time.

(ii) $L_1 \cap L_2$

$L_1 \cap L_2 \in \mathsf{P}$. A Turing machine which recognises $L_1 \cap L_2$ in polynomial time can be formed by starting with the machine which recognises L_1, then running the machine which recognises L_2. If **both** machines reach accepting states, then accept. Since we run two polynomial-time algorithms, the new machine also terminates in polynomial-time.

(iii) The complement of L_1

$\bar{L_1} \in \mathsf{P}$. The complexity class P is closed under complementation. Given a machine (Q, Σ, δ, s) which recognises L_1 in polynomial time, the machine M' defined as (Q, Σ, δ', s) will recognise $\bar{L}$ in polynomial time; where δ' swaps transitions into rej with transitions into acc; and swaps transitions into acc with transitions into rej. δ' is defined as:

$$\begin{aligned}\delta' =& \{((q,a),(q',b,d)) \mid ((q,a),(q',b,d)) \in \delta, q' \notin \{acc, rej\}\} \\ & \cup\{((q,a),(acc,b,d)) \mid ((q,a),(rej,b,d)) \in \delta\} \\ & \cup\{((q,a),(rej,b,d)) \mid ((q,a),(acc,b,d)) \in \delta\}\end{aligned}$$

(b) Suppose L_1 and L_2 are languages in NP. What can you say about the complexity of each of the following? Justify your answer in each case.

(i) $L_1 \cup L_2$

This language is also in NP. Given machines $M_1 = (Q_1, \Sigma_1, \delta_1, s)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, s)$ such that $L(M_1) = L_1$ and $L(M_2) = L_2$; the machine $M' = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, s)$ recognises the language $L_1 \cup L_2$.

(ii) $L_1 \cap L_2$

This language is also in NP. Given machines $M_1 = (Q_1, \Sigma_1, \delta_1, s)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, s)$ such that $L(M_1) = L_1$ and $L(M_2) = L_2$; we can form a Turing machine which recognises the language $L_1 \cap L_2$ by running them sequentially and accepting if and only if both machines accept.

(iii) The complement of L_1

$\bar{L_1} \in \mathsf{co} - \mathsf{NP}$. By definition, the complexity class $\mathsf{co} - \mathsf{NP}$is the class of languages which are polynomially falsifiable. So $\bar{L_1} \in \mathsf{co} - \mathsf{NP}$.

(c) Give an example of a language in NP that is *not* NP-complete and prove that it is not.

I give the language $\emptyset$, which accepts no strings. Clearly, this is decidable in polynomial time by the nondeterministic Turing machine which immediately accepts: $(\{s\}, \Sigma, \{((\triangleright, s), (acc, \triangleright, R)), s\})$. Therefore, $\emptyset \in \mathsf{NP}$.

I prove by contradiction that $\emptyset \notin \mathsf{NP}$-complete.

Assume for contradiction that $\emptyset \in \mathsf{NP}$-complete

So for every $A \in \mathsf{NP}$, $A \leq_P \emptyset$.

Clearly $\Sigma^* \in \mathsf{NP}$. So by the definition of NP-completeness, $\Sigma^* \leq_P \emptyset$.

$$\begin{aligned}
\Sigma^* \leq_P \emptyset &\implies \\
\exists f.\forall x.(x \in \Sigma^* \iff f(x) \in \emptyset) &\implies \\
\exists f.\forall x.(x \in \Sigma^* \implies f(x) \in \emptyset) &\implies \\
\exists f.\forall x.(\mathbf{true} \implies f(x) \in \emptyset) &\implies \\
\exists f.\forall x.f(x) \in \emptyset
\end{aligned}$$

This is a contradiction! By the definition of $\emptyset$, $\forall x.x \notin \emptyset$. Since we have reached a contradiction, our original assumption that $\emptyset$ was NP-complete must have been wrong and therefore $\emptyset \notin \mathsf{NP}$-complete