Computational Complexity

Exercise Session 1

Note: these solutions are (often) merely pointers to the right idea that is needed to solve the problems. These are not fully worked-out solutions. So please do not take these solutions as an example for how to write up your solutions for, e.g., the homework assignments. :-)

Definition 1. Take two functions $f, g : \mathbb{N} \to \mathbb{N}$.

f is $O(g)$ iff there exist $c, n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ $(\leq)$

f is $o(g)$ iff for every $\epsilon > 0$ there exists $n_0 \in \mathbb{N}$ such that $f(n) \leq \epsilon \cdot g(n)$ for all $n \geq n_0$ $(<)$

or equivalently, iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

f is $\Omega(g)$ iff g is $O(f)$ $(\geq)$

f is $\omega(g)$ iff g is $o(f)$ $(>)$

f is $\Theta(g)$ iff f is $O(g)$ and f is $\Omega(g)$ $(=)$

Exercise 1.

1. Show that $(\log n)$ is $o(n)$.[1]
   - *Hint:* show by induction that for arbitrary d it holds that: $d \log n \leq n$ for sufficiently large n. (You may use the fact that for each d it holds that $n + 1 < 2^{1/d} \cdot n$ for sufficiently large n.)
   - *Hint:* alternatively, use the limit characterization, and use L'Hôpital's Rule.[2]
2. Show that the constant function $f(n) = 1$ is $o(n)$.
3. Show that n is $o(n^2)$.
4. Show that n^d is $O(2^n)$ for each $d \in \mathbb{N}$.
   - *Hint:* use the fact that for arbitrary d it holds that: $d \log n \leq n$ for sufficiently large n.
5. Show that $n/2$ is $\Theta(n)$.

Solutions:

1. Take an arbitrary $d \in \mathbb{N}$ such that $d \geq 3$. We show by induction that $d \log n < n$ for sufficiently large n. As base case, take $n = 2^d$. Then $d \log n = d \log 2^d = d^2 < 2^d = n$. For the inductive case, we suppose that $d \log n < n$, and we will show that $d \log(n + 1) < n + 1$. By using the fact that $n + 1 < 2^{1/d} \cdot n$ for sufficiently large n, we know that $d \log(n + 1) < d(\log(2^{1/d} \cdot n)) = d(\log n + 1/d) = d \log n + 1$.

   To now show that $(\log n)$ is $o(n)$, take an arbitrary $\epsilon > 0$. Then take some d such that $1/d \leq \epsilon$. We know that $d \log n < n$ for sufficiently large n. Multiplying both sides with $1/d$, we get $\log n < 1/d \cdot n \leq \epsilon \cdot n$, for sufficiently large n. Since ϵ was arbitrary, this shows that $(\log n)$ is $o(n)$.

   To show that $(\log n)$ is $o(n)$ using the limit characterization, we will show that $\lim_{n\to\infty} \frac{\log n}{n} = 0$. Since $\lim_{n\to\infty} \log n = \infty$ and $\lim_{n\to\infty} n = \infty$, L'Hôpital's Rule applies. Then, $(\log n)' = 1/n \ln 2$ and $(n)' = 1$, and thus $\lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{1/n \ln 2}{1} = \lim_{n\to\infty} 1/n \ln 2 = 0$. Thus $(\log n)$ is $o(n)$.

[1] As is typical in computer science, log refers to the binary logarithm ($\log_2$).
[2] See, e.g., `https://en.wikipedia.org/wiki/L%27H%C3%B4pital%27s_rule`

2. Immediate from (1) and the fact that $1 \leq \log n$ for $n \geq 2$.

3. Since 1 is $o(n)$, we know that for every $\epsilon > 0$ it holds that $1 \leq \epsilon \cdot n$ for sufficiently large n. Then, multiplying both sides with n, we get that for every $\epsilon > 0$ it holds that $n \leq \epsilon \cdot n^2$ for sufficiently large n, showing that n is $o(n^2)$.

   Put differently, if $f(n)$ is $o(g(n))$, then $f(n) \cdot h(n)$ is $o(g(n) \cdot h(n))$. Taking $f(n) = 1$ and $g(n) = h(n) = n$, we get that n is $o(n)$.

4. Take an arbitrary $d \in \mathbb{N}$. Using the fact that $d \log n \leq n$ for sufficiently large n, we get that $n^d = 2^{d \log n} \leq 2^n$ for sufficiently large n. So, n^d is $O(2^n)$.

5. Since $n/2 \leq n$ for all n, it follows that $n/2$ is $O(n)$. Taking $c = 1/2$, we directly get that n is $O(n/2)$, and thus that $n/2$ is $\Omega(n)$.

Exercise 2 (Inspired by exercise 1.14(b) from the book (Arora & Barak, 2009))**.**

In this exercise you will work with various levels of description for algorithms. For the remainder of the course, you may use any level of description for algorithms—whichever is most convenient in the context—as long as the details that you are omitting can be filled in straightforwardly.[3]

1. An undirected graph $G = (V, E)$ consists of a finite set V of vertices and a finite set $E \subseteq \binom{V}{2}$ of edges (subsets of V of size exactly 2). Describe how one can describe any graph as a string using the alphabet $\Sigma_{\text{graphs}} = \{\texttt{(}, \texttt{)}, \texttt{,}, \texttt{0}, \ldots, \texttt{9}\}$. The symbols in Σ are written in typewriter font to differentiate them from symbols in the meta-language.

   - The names of vertices do not matter, so you may give each vertex an arbitrary name in your encoding.
   - *Hint:* give the vertices an arbitrary numbering and represent each edge as a pair of numbers. (*Note:* there are multiple possible suitable representations—so yours may differ from this suggestion.)

2. Describe an algorithm that checks whether a given string over the alphabet Σ_{graphs} is a valid representation of a graph G. Does this algorithm run in polynomial time? If so, explain why, and if not, explain why not.

3. Give a high-level description of an algorithm A that computes whether a given undirected graph G contains a triangle—that is, a triple of vertices v_1, v_2, v_3 such that $\{v_1, v_2\} \in E$, $\{v_1, v_3\} \in E$ and $\{v_2, v_3\} \in E$.
   Use descriptions at the level of detail of *"iterate over all vertices v_1," "check if $\{v_1, v_2\} \in E$," "if such-and-such-easy-to-check-condition is the case, return 1"* and similar statements.

4. Describe your algorithm A at a somewhat more detailed level of description.
   Use descriptions at the level of detail of *"let x_1 be a variable, and set $x_1 := 1$," "read the largest number in the input string, and set x_2 to this value,", "while $x_1 < x_2$, do ...," "if (x1,x2) occurs in the input, do ...,"* and similar statements. In other words, use something like pseudocode.[4]

5. If you were to spell out your algorithm A in full detail as a Turing machine, how would you approach this? For example, how many tapes would you use? What would you intend to write on these tapes? Roughly, what groups of states $q \in Q$ would you create, and for what purpose?

   - *Note:* spelling out the Turing machine corresponding to A is tedious (and typically little fun). So please don't go into full detail, but reflect on what you would have to do if you *were to*.

6. Does your algorithm A run in polynomial time? If so, explain why, and if not, explain why not.

Solutions:

1. Idea: number the vertices (arbitrarily) from 1 to $|V|$. Let $[\![v]\!] \in \{1, \ldots, |V|\}$ be the number assigned to $v \in V$. Represent V with $[\![V]\!] = \texttt{(}1, \ldots, [\![|V|]\!]\texttt{)}$. Represent each edge $e = \{v, v'\}$ with $[\![e]\!] = \texttt{(}[\![v]\!]\texttt{,}[\![v']\!]\texttt{)}$, where v has a lower number than v'. Represent $E = \{e_1, \ldots, e_m\}$ with $[\![E]\!] = \texttt{(}[\![e_1]\!]\texttt{,} \ldots \texttt{,}[\![e_m]\!]\texttt{)}$. Represent $G = (V, E)$ with $[\![G]\!] = \texttt{(}[\![V]\!]\texttt{,}[\![E]\!]\texttt{)}$.

2. Roughly, something along the following lines—depending on the graph encoding used.
   (1) Check if the bracketing (and comma'ing, etc) is right—basically, this you can do with a single run-through of the string, even with a finite-state machine.

[3] This can be a delicate balance, and unfortunately there is no clear-cut recipe for this, so use your own judgment.
[4] See, e.g., `https://en.wikipedia.org/wiki/Pseudocode`

(2) Check if the vertices are mentioned from 1 to $|V|$, and read off the value of $|V|$, and store this in memory.
(3) Check for each edge representation $[\![e]\!]$ it holds that both numbers are different, are between 1 and $|V|$, and are in increasing order. For this, you need to run through $[\![E]\!]$ once, comparing three numbers for each $[\![e]\!]$.

This runs in polynomial time. Step (1) can be done with a single run through the input string. Step (2) takes a polynomial amount of time. Step (3) also takes polynomial time, because you need to run through $[\![E]\!]$ once, and along the way you need to compare the numbers in $[\![e]\!]$ for each e with each other and with $|V|$ (that you stored).

3. Roughly, something along the following lines. Iterate over all triples (v_1, v_2, v_3) of vertices. For each triple (v_1, v_2, v_3), check if $\{v_1, v_2\} \in E$, $\{v_1, v_3\} \in E$ and $\{v_2, v_3\} \in E$. If so, return 1. After iterating over all triples, and for each triple the check failed, return 0.

4. Roughly, something along the following lines—depending on the graph encoding used. Let x_1, x_2, x_3 be variables, initialized at 1. Read the largest number in the input string, and set x_4 to this value. Use three nested loops of the form *"while $x \leq x_4$, do ... and then set $x := x + 1$,"* for $x \in \{x_1, x_2, x_3\}$. Inside these nested loops, use: *"if (x_1, x_2), (x_1, x_3) and (x_2, x_3) all three appear in the input string, return* 1*."* After the nested loops, use: *"return* 0*."*

5. Roughly, something along the following lines—depending on the graph encoding used and the exact algorithm A chosen. Use, say, four working tapes, to store the values of $x_1, \ldots, x_4$. Use a set $Q_{\text{control}} \subseteq Q$ of states for entering and leaving the main phases of the algorithm: starting iteration, start code inside the nested loops, go to next iteration, etc. Use sets $Q_{\text{init}}, Q_{\text{incr}}, Q_{\text{final}} \subseteq Q$ of states to perform the different phases of the iteration. Use a set $Q_{\text{check}} \subseteq Q$ of states to perform the check inside the nested loops. Etcetera.
(This answer can be in an arbitrary level of detail. The main point is to realize that each simple operation has to be 'programmed out' using states and tapes.)

6. Yes, it runs in polynomial time. The value $m = |V|$ is upper bounded by the input size. We do m^3 iterations, and each iteration takes linear time, so overall the algorithm runs in time $O(|x|^3)$, where x is the input string.