University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithms and Datastructures
Sample Solution Exercise Sheet 2

Exercise 1: $\mathcal{O}$-notation *(9 Points)*

Prove or disprove the following statements. Use the *set definition* of the $\mathcal{O}$-notation (lecture slides week 2, slides 11 and 12).

(a) $2n^3 + 4n^2 + 7\sqrt{n} \in \mathcal{O}(n^3)$ *(1 Point)*

(b) $n \cdot \log_3(n) \in \omega(n \cdot \log_5(n))$ *(2 Points)*

(c) $2^n \in o(n!)$ *(2 Points)*

(d) $2\log_2\left(n^2\right) \in \Omega((\log_2 n)^2)$ *(2 Points)*

(e) $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$ for non-negative functions f and g. *(2 Points)*

Sample Solution

(a) True. Choose $n_0 = 1$ and $c = 13$. For all $n \geq n_0$ we have $n^3 \geq n^2 \geq \sqrt{n}$ and hence $2n^3 + 4n^2 + 7\sqrt{n} \leq 13n^3 = cn^3$.

(b) False. Consider some $c > \frac{1}{\log_5(3)}$. Then for all n we have $n \cdot \log_3(n) = n \cdot \frac{\log_5(n)}{\log_5(3)} < c \cdot n \cdot \log_5(n)$.

(c) True. For $n \geq 2$ we have

$$(n-1)! = (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \geq 2^{n-2}$$

and hence

$$4(n-1)! \geq 2^n$$

Let $c > 0$. Choose $n_0 = \max\left\{2, \left\lceil \frac{4}{c} \right\rceil\right\}$. Then for all $n \geq n_0$ we have

$$2^n \leq 4(n-1)! \leq c \cdot n \cdot (n-1)! = c \cdot n!$$

(d) False. Let $c > 0$. We have

$$\begin{array}{rccc} & 2\log(n^2) & \geq & c(\log n)^2 \\ \Leftrightarrow & 4\log(n) & \geq & c(\log n)^2 \\ \Leftrightarrow & 4 & \geq & c\log n \\ \Leftrightarrow & \frac{4}{c} & \geq & \log n \\ \Leftrightarrow & 16^{\frac{1}{c}} & \geq & n \end{array}$$

So for a given $n_0 \geq 1$ choose $n = \max\{n_0, 16^{\frac{1}{c}}\} + 1$. For this n we have $n > n_0$ but $2\log(n^2) < c(\log n)^2$.

(e) True. Choose $n_0 = 1$, $c_1 = \frac{1}{2}$ and $c_2 = 1$. For $n \geq n_0$ we have

$$c_1 \cdot (f(n) + g(n)) \leq \max\{f(n), g(n)\} \overset{f,g \geq 0}{\leq} c_2(f(n) + g(n))$$

Exercise 2: Sorting by asymptotic growth *(4 Points)*

Sort the following functions by their asymptotic growth. Write $g <_{\mathcal{O}} f$ if $g \in \mathcal{O}(f)$ *and* $f \notin \mathcal{O}(g)$. Write $g =_{\mathcal{O}} f$ if $f \in \mathcal{O}(g)$ *and* $g \in \mathcal{O}(f)$ (no proof needed).

$\sqrt{n}$	2^n	$n!$	$\log(n^3)$
3^n	n^{100}	$\log(\sqrt{n})$	$(\log n)^2$
$\log n$	$10^{100}n$	$(n+1)!$	$n \log n$
$2^{(n^2)}$	n^n	$\sqrt{\log n}$	$(2^n)^2$

Sample Solution

$$\sqrt{\log n} <_{\mathcal{O}} \log(\sqrt{n}) =_{\mathcal{O}} \log n =_{\mathcal{O}} \log(n^3) <_{\mathcal{O}} (\log n)^2 <_{\mathcal{O}} \sqrt{n} <_{\mathcal{O}} 10^{100}n <_{\mathcal{O}} n \log n$$
$$<_{\mathcal{O}} n^{100} <_{\mathcal{O}} 2^n <_{\mathcal{O}} 3^n <_{\mathcal{O}} (2^n)^2 <_{\mathcal{O}} n! <_{\mathcal{O}} (n+1)! <_{\mathcal{O}} n^n <_{\mathcal{O}} 2^{(n^2)}$$

Exercise 3: Event Scheduling *(7 Points)*

There are n events $e_1, \ldots, e_n$, each described by their starting time $s_1, \ldots, s_n$ and their ending times $t_1, \ldots, t_n$. You can only attend one event at a time and if you started an event, you must attend the entire event. We ignore traveltimes between events. The goal is to attend as many events as possible.

Devise an Algorithm that computes the maximum number of events you can attend. Why is your algorithm correct? What is the asymptotic complexity of the algorithm? Can you devise an algorithm that is $o(n^2)$.

Sample Solution

Algorithm: We first sort the events by their ending times $t_1, \ldots, t_n$, with t_1 being the event that ends as early as possible. We then greedily pick events from this sequence, that is we start with t_1 and then pick the next event in the sequence, for which the starting time $s_i \geq t_1$. We then repeat this until we can no longer attend any new event.

Correctness: Consider the first event o_1 an optimal solution $o_1, \ldots, o_n$ attends and the solution our algorithm produces $a_1, \ldots, a_n$. Its ending time t_{o_1} is at least $t_{a_1} = t_1$, since t_1 is the event e_1 with the earliest ending time. So by replacing o_1 with e_1, the sequence $a_1, o_2, \ldots, o_n$ is still a valid solution. Since $t_{a_1} \leq t_{o_1}$, we have that $s_{o_2} \geq s_{a_2}$. As a result and by the properties of our algorithm, we get $t_{o_2} \geq t_{a_2}$, so we can repeat the same reasoning also to replace o_2 with a_2, and so on and so forth. At the end we have transformed $o_1, \ldots, o_n$ into $a_1, \ldots, a_n$, by replacing each event with one other. So the two solutions contain the same number of events.

Runtime: We need to first sort the events, which can be done in time $O(n \cdot \log n)$. Then we need to iterate through the sorted list, where we only consider every element once. We either ignore it, because its starting time is earlier then the last added events ending time, or we add it to the solution sequence. This takes $O(n)$ time. As a result our algorithm runs in time $O(n \cdot \log n) \subset o(n^2)$.