Obligatory Exercises

This document contains the list of obligatory exercises of the course Efficient Algorithms and Problem Complexity. Please prepare your solutions as individual pdf documents, one per exercise, and submit them [here](#). If corrections are required, the deadline for the second submission is 2 weeks after the ordinary deadline (unless something else has explicitly been announced).

**Exercise 1 (due 2013-09-09)**

**Background.** The Cantor set is a mathematically highly interesting set of points in the unit interval \([0, 1]\). It is obtained by starting with the line segment \([0, 1]\), taking away the open interval of the middle third, so that the two line segments \([0, 1/3]\) and \([2/3, 1]\) remain, and repeating this process over and over again, always removing the middle thirds of the 2^n line segments obtained in the previous step. The Cantor set is the set of points that remain in the limit. It is a one-dimensional fractal.

**The exercise.** Instead of intervals in \(\mathbb{R}\), we want to consider lists of integers. For simplicity, we only take nonempty lists into account. Let us denote a list of integers \(a_0, \ldots, a_n\) as \([a_0, \ldots, a_n]\). Given such a list \(l\), we want to keep only the elements belonging to the left and right thirds, but remove those that are strictly in between. (Imagine that \(a_0, \ldots, a_n\) are spread out evenly on the unit interval \([0, 1]\), with \(a_0\) and \(a_n\) being placed on the two opposite ends of the interval. Then the items to be removed are those whose positions would be removed in the original Cantor construction.) As in the definition of the Cantor set, we want to repeat this recursively, i.e., take away the middle thirds of the two remaining sublists, and so on. However, because of the finiteness of \(l\), we do not have to take this limit, but can stop when arriving at single integers. Let us denote the resulting list by \(C(l)\).

(1) Give a precise formal definition of the function \(C\). As above, you should denote a list consisting of the items \(a_0, \ldots, a_n\) by \([a_0, \ldots, a_n]\). Be careful to make sure that your definition handles the border cases like \([2, 4, 3, 1]\) correctly.

(2) Write down the pseudocode for an algorithm that takes \(l\) as input and returns the list obtained by sorting \(C(l)\). For this, identify a suitable standard sorting algorithm and modify it slightly.

(3) Determine the running time of this algorithm by means of the Main Recurrence Theorem (the Master Theorem).

**Exercise 2 (due 2013-09-23)**

Explain Prim’s algorithm by applying it to an example graph. The graph should be small, yet interesting enough to result in a non-trivial run of the algorithm. Show the contents of the adjacency lists, the ‘parent’ array, and the minheap.
• directly after initialization,
• at some point during the execution of the algorithm, illustrating how a node is added to the MST by showing the situation before and after, and
• upon termination.

Explain briefly what is going on.

Exercise 3 (due 2013-10-03)

Give a version of Floyd’s algorithm (written down in pseudocode) that works on digraphs \( G = (V, E) \) and computes, for all pairs \((u, v)\) of nodes, the set

\[
OnAll(u, v) = \{ w \in V \mid w \text{ occurs on every path from } u \text{ to } v \}.
\]

Make plausible that the algorithm works correctly.

As an example, in the graph

![Graph diagram]

we have

\[
\begin{align*}
OnAll(5, 5) & = \{5\}, \\
OnAll(5, 6) & = \{5, 3, 6\}, \\
OnAll(6, 1) & = \{1, 6\}, \text{ and} \\
OnAll(1, 6) & = \{1, 2, 3, 4, 5, 6\}.
\end{align*}
\]

Note especially the last one! Logically, if there is no path at all between two nodes, then every node lies on each of these (nonexisting) paths.

Hints:

1. While the original algorithm starts out with the \( n \times n \) adjacency matrix \( A \) of \( G \), your algorithm will have to use \( A \) to initialize another \( n \times n \) matrix, which it then iteratively modifies similarly to the way in which Floyd’s algorithm modifies \( A \). Make sure to choose the initial values carefully.

   (Have a look at the example above and ask yourself what are the nodes that lie on all those paths from \( u \) to \( v \) that do not pass any other node.)

2. Apart from the initialization, the only change required concerns the update

   \[
   \]

   in the body of the main loop. Find suitable replacements for the operations \( \min \) and \( + \).

Exercise 4 (due 2013-10-14)

(1) Write down a definition of P-completeness analogous to the definition of NP-completeness, i.e., using polynomial-time reductions.

(2) Which problems are P-complete in this sense, and why?

(3) Thus, what is the problem, i.e., what must be changed if the goal is to come up with a more interesting notion of P-completeness?
Note: What you are asked to write down in (1) is not the correct definition of P-completeness you find in the literature. This one is intentionally flawed. The answer to (2) is simple and should not take more than a few lines, but it is not entirely trivial. Make sure that you use the relevant definitions in your answer. In (3), a rather general statement is sufficient, i.e., no concrete suggestion of a better notion of P-completeness needs to be made.

**Exercise 5 (due 2013-10-21)**

Let \( M \) be a deterministic random access machine \( M \) that decides a problem \( A \). For every input \( x \), let \( \text{used}_M(x) \) denote the set of addresses of those registers of \( M \) that occur at least once as the target of some write instruction during the computation that processes \( x \). If the computation consists of \( l \) steps, it can be represented as a sequence of \( l + 1 \) configurations \( C_0, \ldots, C_l \), where

- \( C_0 \) is the initial configuration,
- \( C_l \) is the final one, and
- each configuration is of the form \( C_i = (\kappa_i, R_i) \), where \( \kappa_i \) is the value of the program counter (the number of the next instruction to be executed), and a mapping \( R_i: \text{used}_M(x) \to \mathbb{N} \), where \( R_i(j) \) is the contents of register \( R_j \) at that point in time.

Now, for \( m \in \mathbb{N} \), let \( |m| \) denote the number of bits of the binary representation of \( m \). (Thus, \( |0| = 1 \) and \( |m| = \lfloor \log m \rfloor + 1 \) for \( m > 0 \).) We say that the memory space consumed by \( M \) with input \( x \) is

\[
\text{space}_M(x) = \max_{0 \leq i \leq l} \left\{ \sum_{j \in \text{used}_M(x)} |R_i(j)| \right\}.
\]

Thus, \( \text{space}_M(x) \) is the maximum number of bits ever stored in memory during the computation, not counting the bits that are only required for storing the input \( x \). We say that \( M \) runs in logarithmic space if \( \text{space}_M(x) \in O(\log |x|) \) (where \(|x|\) denotes the size of \( x \)). The class \( L \) is the set of all decision problems that can be solved in logarithmic space.

Prove that \( M \) runs in polynomial time if it runs in logarithmic space. Thus, \( L \subseteq P \). (Hint: Ask yourself how many configurations \( C_0, \ldots, C_l \) a computation of \( M \) may consist of, i.e., how many steps there can be in the a computation.)

**Exercise 6 (due 2013-10-31)**

Choose one (!) of the problems below and show that it is NP-complete.

**TWICE-3SAT**

**Input:** A propositional formula \( \varphi \) in conjunctive normal form, such that each clause consists of exactly three literals (as in 3SAT).

**Question:** Does \( \varphi \) have at least two different satisfying assignments?

**OMNI-SAT**

**Input:** A propositional formula \( \varphi \) in conjunctive normal form (as in SAT) such that there is at least one variable that occurs in every clause of \( \varphi \).

**Question:** Is \( \varphi \) satisfiable?

---

1 For example, \((x_1 \lor x_2 \lor \neg x_4) \land (\neg x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor \neg x_7)\) is an allowed input because \( x_2 \) occurs in all three clauses.