Computational complexity exam – problem part. 6.6.2011

1. We say that two functions are friendly if they assume the same value for some argument. Two Boolean circuits (with the same number of input gates) are friendly if so are the functions they compute. Show that deciding whether two circuits are friendly is NP-complete.

2. Construct a deterministic Turing machine working in logarithmic space, which takes as an input a deterministic finite automaton over one letter alphabet and outputs a regular expression generating the language recognized by the automaton.

   Note that the output will be produced on a special *output tape*, whereas the logarithmic bound concerns only the working tapes.

   You may use any encoding of an automaton whose size is polynomial in the number of states.

3. Let f and g be functions computed in polynomial and logarithmic space, respectively. Show that the functions $f \circ g$ and $g \circ f$ can be computed in polynomial space.

   Could we claim this if we only knew that both functions were computed in polynomial space?

4. *Theoretical problem.* Show that if $P = NP$ then factorization could be solved in polynomial time.

   Factorization is a functional problem, which takes as an input a natural number (in binary) and outputs its decomposition into prime factors.

Computational complexity exam – theory part. 6.6.2011

Please show and explain the errors in the following reasoning. The answer should fit on the backward of this sheet. You may also mark the passages you claim erroneous directly in the text.

1. To verify if a number m divides n, it is enough to successively add m to itself and check whether we eventually reach n or go above it. For this, $\mathcal{O}(\log n)$ space is enough. Hence, by testing the successive $m < n$, we can test if n is prime using (only) logarithmic space, but we know that $L \subseteq P$. So why so much ado about the AKS polynomial primality test (from 2002) ?

2. Let $\varphi = \alpha_1 \wedge \ldots \wedge \alpha_k$, where each α_i is a clause with *two* literals. Consider

$$\begin{aligned}\varphi' &= (\alpha_1 \vee \neg x_1) \wedge \ldots \wedge (\alpha_k \vee \neg x_k) \wedge \\ &\quad (x_1 \vee x_1 \vee x_1) \wedge \ldots \wedge (x_k \vee x_k \vee x_k)\end{aligned}$$

It is easy to see that φ is satisfiable iff so is φ', but φ' has already 3 literals in each clause, and we know that the problem 3-CNF SAT is *NP*-complete. But we have just described a reduction $\varphi \mapsto \varphi'$, hence the problem 2-CNF SAT is *NP*-complete as well. However, we learned at the tutorials that the last problem is in P. Therefore we have proved $P = NP$.

3. At the lecture, they showed an algorithm which, for a Turing machine M and a number n, constructs a Boolean circuit with n input gates, recognizing precisely the words of length n accepted by M. But any Boolean function $\{0,1\}^n \to \{0,1\}$ can be computed by a circuit with $\mathcal{O}(2^n)$ gates. The evaluation of a Boolean circuit is possible in time polynomial w.r.t. the size of the circuit. Hence, any Turing machine (provided it always halts) can be simulated by a machine working in exponential time. So what is this cheating with the *Time hierarchy theorem*?

4. Suppose a probabilistic algorithm recognizes a language L with the probability of error $p \leq \frac{1}{4}$, by giving answers *yes* or *no*. Thus if, for an input x, we get an answer *yes* then with probability $1-p$ it holds that $x \in L$, and if we get an answer *no* then with probability $1-p$ it holds that $x \notin L$.

(Suppose, for example, that algorithm decides if $1 = 1$, tossing a coin 5 times. If head occurs all the time, it says *no*, otherwise says *yes*. Suppose the answer is: *no*. Shall we start to doubt if $1 = 1$?)

5. On the lecture, they showed a polynomial interactive proof, in which Mathematician convinces Algorithm[1] that two graphs are *not* isomorphic. We now adapt this protocol to verify that two Turing machines M_1 and M_2 are non-equivalent.

More precisely, Mathematician convinces Algorithm that $L(M_1) \neq L(M_2)$. To this end, Algorithm randomly chooses $i \in \{1,2\}$ and next, using few further random choices, it modifies M_i, so that the machine computes the same, but "looks" quite different than M_i. Algorithm then presents the so-obtained machine N to Mathematician, asking: *what is it ?* (i.e., M_1, or M_2).

If M_1 and M_2 are non-equivalent, Mathematician gives the unique correct answer. However, if they are equivalent then Mathematician (who doesn't know the random bits of the Algorithm) can only guess the expected i with probability $\frac{1}{2}$.

Hence, we have obtained an interactive proof deciding in polynomial time if two Turing machines are non-equivalent, which, as we know, is an *undecidable* problem. Still, they claimed at the lecture (giving even a suitably obscured "proof") that languages recognized by interactive proofs are not only decidable, but even *PSPACE* !

[1] In the English language literature, these figures are usually called *Prover* and *Verifier*.