Assignment 4: Computational Complexity

Due date: Thursday 10 April *before* the class starts
(i.e., it *cannot* be returned during class)

1. (5) Problem 6.8 from the book.

2. 10: In class we saw that: If we have a poly-time (algorithm M for language L such that $x \in L \Rightarrow \Pr[M(x) = 1] \geq 1/2 + c$ and $x \notin L \Rightarrow \Pr[M(x) = 1] \leq 1/2 - c$ for a constant c, then we can get another polynomial time algorithm M' such that $x \in L \Rightarrow \Pr[M(x) = 1] \geq 1 - 2^{-n}$ and $x \notin L \Rightarrow \Pr[M(x) = 0] \geq 1 - 2^{-n}$. Prove that we can still do so, if we only assume that M is a polynomial time algorithm M such that: $x \in L \Rightarrow \Pr[M(x) = 1] \geq t + c$ and $x \notin L \Rightarrow \Pr[M(x) = 1] \leq t - c$ for constants $0 < t < 1$ and $c < 1$.

   Hint: Note that when $t = 1/2$ is just we did solve as follows: we run M "many" times and look at the majority function. Now that t is not necessarily $1/2$, what is the most reasonable way to judge the final decision based on many executions of M ? The next step is to just apply the Chernoff bound (use the version we saw in class, not the one in the book).

3. 10: Problem 7.4 from the book. Hint: we solved this in class, but we worked with constant instead of small inverse of polynomials; so you have to use better parameters in your repetition. You might also find this fact useful: $(1 - q)^{1/q} < 1/e$ for any number $q < 1$.

4. 10: Read Lemma 7.13 from the book and sketch its proof.

   Note: This is a very nice lemma showing that if we use a source of randomness that gives us one bit each time, but is not uniformly distributed, can still be used, even if our algorithm is designed to use perfectly uniform random bits.

5. 10: Problem 7.7 from the book. Hint: for that you essentially need to use the argument we saw in class that shows how to eliminate randomness when we can use non-uniformity (i.e. non-uniform advice about "good randomness). But be careful about all the definitions.

6. (25) Problem 6.12 from the book. In order to ignore the "finite field" part of the question, for simplicity suppose our circuits can have two types of gates: addition and multiplication, and the wires of the circuit can carry integers (not just Boolean values). So, we can have a circuit that takes three inputs a, b, c and computes $a \cdot b + a \cdot c$ using only two gates (by computing $a \cdot (b + c)$). In this language, if we talk about $NC0$ circuits, it would be circuits of constant depth (and polynomial size) that can use the above mentioned wires and gates. Also note that the class NC (not $NC0$) allows the depth be $(\log n)^c$ for any constant c.

   Grades: part a: 5, Part b: 10, Part c: 10. Part c is extra credit.

   Hint: part (a) can be solved based on what we saw in class (but you still have to write it fully). For part (b) you just need a smart way of computing a^b for large b that uses much smaller than b multiplications. For part (c) first check what happens if we compute A^2 where A is an adjacency matrix of a graph G. Double hint: the entries of A^2 tell us if there is a path of length 2 between the nodes. Prove a generalized version of this observation by induction and then apply Part 2.