Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, A. R. Molla, O. Saukh January 7, 2016

Algorithm Theory, Winter Term 2015/16
Problem Set 10

hand in (hard copy or electronically) by 10:15, Thursday January 14, 2016, tutorial session will be on January 18, 2016

Exercise 1: Linear-Time Contention Resolution (8 points)

In class, we looked at the following simple contention resolution problem. There are n processes that need to access a shared resource. Time is divided into time slots and in each time slot, a process i can access the resource if and only if i is the only process trying to access the resource. We have shown that if each process independently tries to access the resource with probability $1/n$ in each time slot, in time $O(n \log n)$, all processes can access the resource at least once with high probability. The goal of the exercise is to improve the algorithm and to get an $O(n)$ time algorithm under the following assumptions.

- As in the lecture, all the processes know n (the number of processes). In the algorithm of the lecture, this is needed because the probability $1/n$ for accessing the resource depends on n. As in the lecture, we also assume that all processes start together in the first time slot.
- If a process tries to access the resource in a time slot, the process afterwards knows whether the access was successful or not. Also, we assume that a process only needs to succeed once, i.e., once a process has been successful, it stops trying to access the resource.

The goal of this exercise is to give and analyze a randomized algorithm which guarantees that for some given constant $c > 0$ with probability at least $1 - 1/n^c$, during the first $O(n)$ time slots, each of the n processes can access the resource at least once.

(a) (2 points) Let us first assume that in each time slot at most $n/\ln n$ processes (among n processes) need to access the resource. Adapt the algorithm of the lecture such that all processes succeed in accessing the channel in $O(n)$ rounds with probability at least $1 - 1/n^{c+1}$.

(b) (1 point) Let us now assume that we are given an algorithm which guarantees that after $T(n)$ time slots, the number of processes which have not yet succeeded is at most $n/\ln n$ with probability at least $1 - 1/n^{c+1}$. What is the probability that all n processes succeed when combining this algorithm with the adapted algorithm of the lecture from question (a). Define the appropriate probability events to analyze this probability.

(c) (5 points) It remains to give an algorithm to which manages to get rid of all except $n/\ln n$ of the processes with probability at least $1 - 1/n^{c+1}$. Show that this can be achieved by an algorithm which runs in multiple stages. You can use the following hint.

Hint: *You can make use of the following fact. Consider a time interval consisting of at least $e^2 k$ time slots. During the time interval, there are at most k processes trying to access the resource and in each time slot, each of the at most k processes tries to access the resource with probability $1/k$. Then, with probability at least $1 - e^{-k}$, at the end of the interval, at most $k/2$ of the processes have not succeeded to access the resource.*

Exercise 2: Comparing Two Polynomials (4 points)

Assume that your are given two integer polynomials p and q of degree n. However, you are not given the polynomials in an explicit form. Your only way to access the polynomials is to evaluate them at some integer value $x \in \{1, \ldots, 2n\}$ (i.e., you can compute $p(x)$ and $q(x)$ for values $x \in \{1, \ldots, 2n\}$). You want to find out whether the two polynomials are identical. Give an efficient[1] randomized algorithm which tests whether the two polynomials are identical! If $p = q$, your algorithm should always return "yes", if $p \neq q$, your algorithm is allowed to err with constant probability. How can you get an algorithm which gives the correct answer with probability at least $1 - \epsilon$ for some (arbitrary) given value $\epsilon > 0$?

[1]The complexity of an algorithm is measured by the number of polynomial evaluations it needs to perform.