Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, A. R. Molla, O. Saukh January 22, 2016

Algorithm Theory, Winter Term 2015/16
Problem Set 10 - Sample Solution

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.*

Solution:

(a) Suppose in each time slot $k \leq n/\ln n$ processes want to access the resource. We show that if each of them broadcasts with probability $\ln n/n$, after $O(n)$ steps, all of them have succeeded with probability at least $1 - 1/n^{c+1}$. By choosing $p = \ln n/n$ and using the analysis of the lecture, here we show that all the n processes can succeed with high probability.

In the similar way, we define the following events.

$\mathcal{A}_{i,t}$: process i tries to access the resource in time slot t.

$\mathcal{S}_{i,t}$: process i is successful in time slot t.

$\mathcal{F}_{i,t}$: process i does not succeed in time slots $1, 2, \ldots, t$.

By our assumption, $\Pr(\mathcal{A}_{i,t}) = p = \ln n/n$. Therefore,

$$\begin{aligned}\Pr(\mathcal{S}_{i,t}) &= p(1-p)^{k-1} \qquad \text{[since in each time slot } k \text{ processes try to access the resource]} \\ &\geq \frac{\ln n}{n}\left(1 - \frac{\ln n}{n}\right)^{\frac{n}{\ln n}-1} \qquad \left[\text{since } k \leq \frac{n}{\ln n}\right] \\ &> \frac{\ln n}{en}\end{aligned}$$

Then $\Pr(\mathcal{F}_{i,t}) = \big(1 - \Pr(\mathcal{S}_{i,t})\big)^t$, since $\mathcal{S}_{i,t}$ are independent for different t. We get

$$\Pr(\mathcal{F}_{i,t}) < \left(1 - \frac{\ln n}{en}\right)^t < e^{-\frac{t \ln n}{en}}$$

Hence, probability of no success by time t is less than $e^{-\frac{t \ln n}{en}}$ for a particular process. Setting $t = \lceil en \cdot (c+2) \rceil$ gives $\Pr(\mathcal{F}_{i,t}) < e^{-(c+2)\ln n} = \frac{1}{n^{c+2}}$, which is the failure probability of a particular process i by time t. Therefore the probability of success of the process i by time $t = \lceil en \cdot (c+2) \rceil$ is at least $\big(1 - \frac{1}{n^{c+2}}\big)$. Then taking the union bound on the failure probability over all the n processes (as in the lecture), we show that all processes succeed in time $t = O(n)$ with probability at least $1 - \frac{1}{n^{c+1}}$. For this, we define $\mathcal{F}_t$: some process has not succeeded by time t. Then $\mathcal{F}_t = \cup_{i=1}^n \mathcal{F}_{i,t}$. Therefore, the probability that not all processes have succeeded by time t is:

$$\Pr(\mathcal{F}_t) = \Pr\left(\cup_{i=1}^n \mathcal{F}_{i,t}\right) \overset{\text{union bound}}{\leq} \sum_{i=1}^n \Pr\left(\mathcal{F}_{i,t}\right) < \frac{n}{n^{c+2}} = \frac{1}{n^{c+1}}.$$

Hence, all processes succeed with probability at least $1 - \frac{1}{n^{c+1}}$ in time $O(n)$.

(b) Note that we first run the algorithm which guarantees the number of unsuccessful processes is at most $n/\ln n$. Let us call this algorithm as $\mathcal{A}_1$. Then we run the above algorithm in question (a), say the algorithm is $\mathcal{A}_2$. Because after the first algorithm all the successful processes stop trying to access the resource and hence there are at most $n/\ln n$ processes trying to access it. So the algorithm in question (a) guarantees that the remaining processes (which is at most $n/\ln n$) are successful with high probability.

Define two events:
$\mathcal{E}_1$: algorithm $\mathcal{A}_1$ is successful in $T(n)$ time
$\mathcal{E}_2$: algorithm $\mathcal{A}_2$ is successful in $O(n)$ time
From the given assumption, $\Pr(\bar{\mathcal{E}_1}) \leq \frac{1}{n^{c+1}}$ and $\Pr(\bar{\mathcal{E}_2}) \leq \frac{1}{n^{c+1}}$.
Hence, the probability that algorithm A and algorithm B together give some process has not succeed after time $T(n) + O(n)$ is:

$$\Pr(\bar{\mathcal{E}_1} \cup \bar{\mathcal{E}_2}) \overset{\text{union bound}}{\leq} \Pr(\bar{\mathcal{E}_1}) + \Pr(\bar{\mathcal{E}_2}) \leq \frac{2}{n^{c+1}} < \frac{1}{n^c}$$

That is with probability at least $\big(1 - \frac{1}{n^c}\big)$ all n processes succeed after $T(n) + O(n)$ time slots.

(c) The goal is to devise a randomized algorithm which under the above assumptions guarantees that with probability at least $1 - 1/n^{c+1}$, all except at most $n/\ln n$ of the processes can successfully access the shared resource once by $O(n)$ time slots. To do so we split the $O(n)$ available time slots into phases, which shorten in length and participating processes increase their access probabilities.

Phase: The first phase starts with all n processes and ends after $e^2 n$ time slots. Each subsequent phase, up to some point, lasts only half as long as the previous phase. More precisely, the j^{th} ($j \geq 2$) phase starts after $\sum_{i=2}^{j} \frac{e^2 n}{2^{i-2}}$ time slots and lasts for $\frac{e^2 n}{2^{j-1}}$ time slots.

It follows from the 'hint' that at the end of the first phase, at least $n/2$ processes successfully access the shared resource with probability at least $1 - e^{-n}$. With respect to the given assumptions[1], therefore, at most $n/2$ processes remain active (i.e., unsuccessful) and try to access the shared resource (after the first phase). We assume that each phase halves the number of remaining processes and under that assumption we repeat our 'phases scheme' until the number of remaining processes is less than $cn/\ln n$ where c is a constant yet to be determined. We thus define $\ell := \log(\ln n)$ and for simplicity we assume this is a natural number (otherwise take the ceiling of it).

Assume for now that after every phase at most half of the processes are left. Thus the running time of the algorithm is:

$$\sum_k e^2 k = \sum_{i=0}^{\ell} \frac{e^2 n}{2^i} = e^2 n \sum_{i=0}^{\ell} \frac{1}{2^i} \overset{\forall \ell}{<} 2e^2 n = O(n).$$

So far we only *assumed* that after each phase the number of remaining processes is halved; let us prove this. Define phase j is successful if and only if at most $n/2^j$ processes remain active by the end of phase j.

Event $\jmath_i$: Phase i is unsuccessful and it is the first phase for which this is the case.

Due to the hint, $\Pr(\jmath_i) \leq e^{-k} = e^{\frac{-n}{2^i}}$ where $i \in \{0, 1, \cdots, \ell\}$. Note that applying the hint here is correct, since if i is the first phase in which something can fail, it means that phase i started with a proper amount of processes.

Event $\jmath$: The algorithm has failed, in other words some phase $i \leq \ell$ has not been successful.

$$\begin{aligned}
\Pr(\jmath) &= \Pr(\cup_{i=0}^{\ell} \jmath_i) \\
&\overset{\text{union bound}}{\leq} \sum_{i=0}^{\ell} \Pr(\jmath_i) \\
&\overset{\text{hint}}{\leq} \sum_{i=0}^{\ell} e^{-\frac{n}{2^i}} \\
&\leq \ell e^{-\frac{n}{2^\ell}} \\
&= \log(\ln n) e^{-\frac{n}{\ln n}} \\
&= \frac{\log(\ln n)}{e^{\frac{n}{\ln n}}} \\
&< 1/n^{c+1} \qquad \text{[for any constant } c > 0\text{]}
\end{aligned}$$

[1] Once a process has been successful, it stops trying to access the resource further.

Consequently, for $c > 0$ and large enough n ($n \geq 2$), the algorithm succeeds with probability at least $1 - 1/n^{c+1}$. That is the algorithm which runs in multiple phases can get rid of all except $n/\ln n$ of the processes with probability at least $1 - 1/n^{c+1}$.

Remark 1: You may think that why we can not use the above algorithm untill all n processes succeed. The reason behind this is that with $k = o(c \ln n)$ the probability that at most $k/2$ (halve) processes remain is less than $1 - e^{-o(c \ln n)} = 1 - \omega(1/n)$, i.e., not with high probability.

Remark 2: To complete the original goal of the exercise, that is to present 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, we combine the above algorithms as follows:

After the above algorithm in question (c) when less than $n/\ln n$ processes remained unsuccessful or active, we use the simple algorithm of question (a). The complete randomized algorithm is given in Algorithm 1 and is executed by every process. Note that the Algorithm 1 consists of two parts: the first part (where $k \geq n/\ln n$) corresponding to the algorithm in question (c) and the second part (where less than $n/\ln n$ processes remained unsuccessful) corresponding to the algorithm in question (a). Therefore, the combined algorithm guarantees that for some given constant $c > 0$ with probability at least $1 - 1/n^c$, each of the n processes can access the resource at least once, in linear $O(n)$ time.

```
Input: n processes, one shared resource, and time slots 1, 2, ···
Output: All processes successfully access to the shared resource once by O(n) time slots with
        probability at least 1 − 1/n
for i ← 1 to n do
    k := n;
    repeat
        for t ← 1 to e^2 k do
            Try to access the resource with probability 1/k;
            if access was successful then
                Exit ;
            end
        end
        k := k/2;
    until k < n/ ln n;
    while True do
        Try to access the resource with probability (ln n)/n;
        if access was successful then
            Exit ;
        end
    end
end
```

Algorithm 1: Linear-time contention resolution

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[2] randomized algorithm

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

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 $\epsilon > 0$?

Solution:

A simple deterministic solution is: transform both $p(x)$ and $q(x)$ into a "canonical" form i.e., transform $p(x) = \sum_{i=0}^{n} c_i x^i$ and $q(x) = \sum_{i=0}^{n} d_i x^i$. Then compare the coefficients of all monomials i.e., $p(x) = q(x)$ iff $c_i = d_i$ for all i. However, this will take at least $O(n)$ time to transform them into canonical forms and verifying the coefficients. The following randomized algorithm is simple and decide whether $p(x) = q(x)$ much faster than $O(n)$ time.

Choose a random integer r in the set $\{1, \ldots, 2n\}$. Compute $p(r)$ and $q(r)$. If $p(r) = q(r)$ output *YES* (i.e, $p = q$), else output *NO*. Note that we assume that choosing a random number uniformly from the set $\{1, \ldots, 2n\}$ takes 1 time step.
It is easy to see that if $p = q$, the above algorithm always return "correct" answer. If $p \neq q$, then the algorithm might give wrong answer. As for example, the two polynomials $p(x) = x^2 + 7x + 1$ and $q(x) = (x + 2)^2$ are not identical, but for $r = 1$, $p(1) = q(1) = 9$. That is, a bad choice of r can give wrong answer. Now we calculate the probability of error.
Assume $f(x) = p(x) - q(x)$. Hence, there could be at most n roots of the polynomial $f(x)$ in the set $\{1, \ldots, 2n\}$, since the degree of $f(x)$ is at most n. Define a *bad* event as the random choice r is a root of the polynomial $f(x)$. The probability of the bad event is at most $n/2n = 1/2$. In other words, the probability of wrong answer is $\leq 1/2$. Now we repeat the above algorithm k times. If all the repetitions return 'yes', we output "YES", else output "NO". Hence, the probability that all the k repetitions gives wrong answer is $\leq (1/2)^k$, since each repetition is independent. Assume $k = c \log n$ for some constant $c > 1$. Then the probability of wrong answer is $\leq (1/2)^{c \log n} = 1/n^c$. Therefore, the probability of giving correct answer is $\geq (1 - 1/n^c)$. You can make this error probability as small as you want by choosing an appropriate large value of c. The randomized algorithm takes $O(\log n)$ time and output correct answer with probability at least $(1 - 1/n^c)$ for any constant $c > 1$.

Input: Two polynomials p and q
Output: YES or NO
Choose a set of $k = c \log n$ integers independently and uniformly at random from $\{1, \ldots, 2n\}$
for $r \leftarrow 1$ **to** k **do**
  if $p(r) \neq q(r)$ **then**
    Output: NO ;
    Exit ;
  end
end
Output: YES ;

Algorithm 2: Polynomial Identity