

CSE 200 Spring 2010

Calibration Homework Solutions

Chris Calabro and Russell Impagliazzo

April 11, 2010

1 Regular languages

Prove that Reg_{k+1} is a proper superset of Reg_k .

It is immediate from the definition that Reg_{k+1} is a superset of Reg_k . We need to show that the inclusion is proper.

To do this, we need to exhibit a language L_{k+1} in Reg_{k+1} , i.e., that is accepted by a $k + 1$ state finite automaton, but not in Reg_k , that is, cannot be accepted by any k state finite automaton.

Let $L_{k+1} = \{x \in \{0\}^* \mid |x| \equiv 0 \pmod{k+1}\}$ of strings over a one-symbol alphabet whose length is evenly divisible by $k + 1$. Let A be the automaton with states $0, \dots, k$, with 0 as the start state and only accepting state, and transitions $\delta(q, 0) = q + 1 \pmod{k+1}$, i.e., upon reading the next symbol, we go to the next state, unless we are in the last, in which case we go to the initial state.

We can see by induction that 0^l ends in state $l \pmod{k+1}$. since this is true for $l = 0$, and if 0^l is in state $l \pmod{k+1}$, 0^{l+1} ends in state $(l \pmod{k+1}) + 1 \pmod{k+1} = (l + 1) \pmod{k+1}$.

Since the only accepting state is 0 , then 0^l is accepted if and only if $l \pmod{k+1} = 0$, which is by definition if and only if $0^l \in L_{k+1}$.

Thus, automaton A is a $k + 1$ state machine that accepts L_{k+1} , so $L_{k+1} \in Reg_{k+1}$.

Let A' be any at most k state automaton. Consider the states A' reaches on $\lambda, 0, 00, 000, \dots, 0^k$, where $\lambda = 0^0$ is the empty string. Since there are $k + 1$ such strings, and A' has fewer than $k + 1$ states, two of these end at the same state of A' . Let these two be $0^i, 0^j$, with $0 \leq i < j \leq k$. Then $0^i 0^{k+1-j} = 0^{k+1+i-j}$ must end state as $0^j 0^{k+1-j} = 0^{k+1}$. Now $0^{k+1} \in L_{k+1}$, but $0^{k+1+i-j} \notin L_{k+1}$, since its length is between 1 and k since $0 \leq i < j \leq k$. Thus, if the state reached on both strings is accepting A' accepts a string not in L_{k+1} , and if it is rejecting, A' rejects a string in L_{k+1} . So in neither case does A' recognize L_{k+1} . Since A' was an arbitrary DFA with at most k states, $L_{k+1} \notin Reg_k$ as needed, so $Reg_k \neq Reg_{k+1}$ and the inclusion is proper.

2 Reasoning about order

Let $f(n)$ be a positive, integer-valued function on the natural numbers that is non-decreasing. Show that if $f(2n) \in O(f(n))$, then $f(n) \in O(n^k)$ for some constant k . Is the converse also always true?

First, if $f(2n) \in O(f(n))$, by definition of order, there are constants $n_0 > 0, c > 0$ so that for all $n > n_0$, $f(2n) \leq cf(n)$. (Note this is similar to the recurrence: $T(n) = cT(n/2)$, and the proof that f is polynomial is just another proof of the Master Theorem). We'll first prove that f is polynomial on powers of 2, then use monotonicity to conclude the same thing for other values. Without loss of generality, assume $c > 1$ and that n_0 is a power of 2, $n_0 = 2^i$. Let $c' = f(n_0)/c^i$. We'll prove by induction that for $j \geq i$ for $n = 2^j$, $f(n) = f(2^j) \leq c'c^j = c'(2^{j \log c}) = c'n^{\log c}$. The claim is true for $j = i$, by definition of c' . If $f(2^j) \leq c'c^j$, then $f(2^{j+1}) = f(2 \cdot 2^j) \leq cf(2^j) \leq c'cc^j = c'c^{j+1}$, so the claim holds inductively.

Then for any $n \geq n_0$, let $n' = 2^{\lceil \log n \rceil}$ be the next power of 2; $n \leq n' \leq 2n$, so $f(n) \leq f(n') \leq c'(n')^{\log c} \leq c'(2n)^{\log c} = c'cn^{\log c}$. Picking n_0 and cc' in the definition of O , we have $f(n) \in O(n^{\log c})$.

The converse isn't always true. We give a counter-example Let $f(n) = 2^{2^{\lceil \log \log n \rceil}}$. Then $f(n) \leq 2^{2^{\log \log n + 1}} = 2^{2 \log n} = n^2$, so $f(n) \in O(n^2)$. However, if $n = 2^{2^i}$, $f(n) = n$ and $f(2n) \geq n^2$. Since a gap of n occurs for infinitely many n , $f(2n)$ is NOT in $O(f(n))$.

3 Primes

a

Claim 1. $\forall n \in \mathbb{Z}$, if $n \equiv 3 \pmod{4}$, then n has a prime factor $p \equiv 3 \pmod{4}$.

Proof. We show the contrapositive. Suppose the prime factors of n are each congruent mod 4 to an element of $R = \{0, 1, 2\}$. Since R is closed under multiplication mod 4, it follows by induction on the size of the prime factorization of n that n also is congruent mod 4 to an element of R . (We are implicitly using that the modulus operation is a ring homomorphism: i.e. that the mod of a sum (product) is the sum (product) of the mods.) \square

b

Claim 2. $|\{\text{primes } p \mid p \equiv 3 \pmod{4}\}| = \infty$.

Proof. Let p_i be the i th prime and suppose indirectly that p_k is the largest prime congruent to 3 mod 4. Let $n = p_1 \cdots p_k + 1$. Since the prime 2 occurs exactly once in the list p_1, \dots, p_k , we have $n \equiv 3 \pmod{4}$. From part (a), \exists prime factor p of n with $p \equiv 3 \pmod{4}$. Since none of p_1, \dots, p_k divide n , we have $p > p_k$, a contradiction. \square

4 quine

A *quine* is a program that takes no input and produces its own code as output. The name comes from the philosopher W.V.Quine who crafted the phrase

'yields falsehood when preceded by its quotation' yields falsehood
when preceded by its quotation,

an assertion in english without a truth value and yet which is not explicitly self referential, thus showing that the problem with the liar paradox ('this sentence is false') is not the allowance of explicit self reference. (We may see more of this idea in the recursion theorem later.)

A simple quine in C, adapted from code on the quine page (<http://www.nyx.net/~gthomps/quine.htm>) follows.

```
#include <stdio.h>
char*f="#include <stdio.h>%cchar*f=%c%s%c;%cmain(){printf(f,10,34,f,34,10,10);}%" ;
main(){printf(f,10,34,f,34,10,10);};
```

To see why quines are possible in nearly any programming language (and to see how to generate them somewhat automatically, albeit inelegantly), consider a programming language in which it is possible to construct a program A that takes as input a program B and produces a program C that runs the program B on B . Then A with A hardcoded as input is a quine. I.e. A has the property that for any B

$$\text{run } \text{hardcode}(A, B) = \text{hardcode}(B, B),$$

and so

$$\text{run } \text{hardcode}(A, A) = \text{hardcode}(A, A).$$